University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

AI-powered Ransomware to Stay Hidden

Sandro Padovan
Zurich, Switzerland
Student ID: 17-721-291

Supervisor: Dr. Alberto Huertas Celdrán, Jan von der Assen
Date of Submission: January 1, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

This master thesis explores the usage of Reinforcement Learning (RL) to improve the chances of a ransomware to remain undetected applied on a Raspberry Pi 3 target device. It is extending a framework called RansomAI presented in a previous work [1]. The approach is explored in an Internet of Things (IoT) scenario using radio frequency spectrum sensors as targeted devices with a ransomware configurable at runtime. In the previous work several limitations were identified which this work aims to partially cover. Firstly, a different approach is explored using system calls (syscalls) as the basis of device behavioral fingerprinting. This is used during the training of the RL model to detect anomalies. Secondly, additional benign behaviors are introduced to the scenario in order to test the adaptability of the approach.

The evaluations in this work show that the syscall-based approach has advantages at detecting anomalous behavior compared to the resource usage approach of the previous work. Furthermore, the proposed RL agent is able to select the optimal ransomware configuration with an accuracy of above 90 % after less than two minutes of training, which is slightly faster than in the previous work. Regarding the second extension, it is shown that the additional behavior makes it significantly harder to detect the ransomware. Thus, a syscall-based defense approach in this scenario works better with devices having more uniform behavior patterns. The RL agent is able to learn the optimal configuration also with the additional benign behavior, however with less consistency. Nevertheless, it is able to reach an accuracy of above 90 % after less than eight minutes of training.

To further optimize the RL model and the anomaly detection, as well as to evaluate the generalizability of this approach in other scenarios, future studies are necessary.

Zusammenfassung

Diese Masterarbeit untersucht die Verwendung von Reinforcement Learning (RL) um die Chancen einer Ransomware zu erhöhen unentdeckt zu bleiben, angewendet auf einem Raspberry Pi 3 Zielgerät. Es wird ein Framework namens RansomAI erweitert, welches in einer früheren Arbeit vorgestellt wurde [1]. Der Ansatz wird in einem Internet of Things (IoT) Szenario erforscht, bei dem Radiofrequenz Spektrumsensoren benutzt werden als Zielgeräte einer während Laufzeit konfigurierbaren Ransomware. In der vorhergehenden Arbeit wurden mehrere Einschränkungen identifiziert, welche in dieser Arbeit teilweise adressiert werden. Erstens wird ein anderer Ansatz erforscht, bei dem Systemaufrufe (Syscalls) als Grundlage dienen für die Erstellung von Fingerabdrücken des Geräteverhaltens. Dies wird benutzt während des Trainings der RL-Modells um Anomalien zu erkennen. Zweitens werden zusätzliche gutartige Verhaltensweisen in das Szenario eingeführt, um die Anpassungsfähigkeit des Ansatzes zu testen.

Die Auswertungen dieser Arbeit zeigen, dass der Syscall-basierte Ansatz Vorteile bei der Erkennung von abnormalem Verhalten im Vergleich zum Ansatz basierend auf Ressourcennutzung der vorherigen Arbeit hat. Darüber hinaus ist der vorgeschlagene RL-Agent in der Lage, die optimale Ransomware-Konfiguration mit einer Genauigkeit von über 90 % nach weniger als zwei Minuten Training auszuwählen, was etwas schneller ist als in der vorherigen Arbeit. Bezüglich der zweiten Erweiterung wird gezeigt, dass das zusätzliche Verhalten die Erkennung der Ransomware erheblich erschwert. Daher funktioniert ein Syscall-basierter Verteidigungsansatz in diesem Szenario besser bei Geräten mit einheitlicheren Verhaltensmustern. Der RL-Agent ist in der Lage, die optimale Konfiguration auch mit dem zusätzlichen gutartigen Verhalten zu erlernen, allerdings mit weniger Konsistenz. Dennoch ist er in der Lage, nach weniger als acht Minuten Trainingszeit eine Genauigkeit von über 90 % zu erreichen.

Sowohl um das RL-Modell und die Erkennung von Anomalien weiter zu optimieren, als auch um die Verallgemeinerbarkeit dieses Ansatzes in anderen Szenarien zu evaluieren, sind zukünftige Forschungen erforderlich.

Acknowledgments

I would like to express my sincere gratitude to the supervisors of my thesis, Dr. Alberto Huertas Celdrán and Jan von der Assen. Their suggestions, tips, and feedback during the process of writing my thesis helped a lot. Moreover, the great and fast communication in the last months was greatly appreciated.

Additionally, I would like to thank Prof. Dr. Burkhard Stiller for enabling me to write my master thesis at the CSG. After the many courses and seminars I attended over the last years which were taught by Prof. Stiller and the CSG team, I could apply and connect many of the concepts and insights in this thesis.

Finally, I want to thank my family and significant other for their unconditional support and patience.

Contents

Chapter 1

Introduction

1.1 Motivation

The last 20 years have seen a huge surge in devices connected to the internet with the advent of the Internet of Things (IoT). The number of IoT devices has almost doubled within the last four years and is expected to keep growing to almost 30 billion devices by 2030 [2]. Given the embedded nature of many IoT devices, resources are often constrained which complicates the design of effective security methods. Furthermore, the development of security-by-design mechanisms is hindered by the large heterogeneity of the IoT space [3]. Oftentimes, IoT devices are shipped with low password protection such as default passwords, or even without any passwords making it easy for hackers to gain access to a device. With low security, the user's privacy and critical infrastructure is at risk. Furthermore, an IoT device can become part of a botnet targeting some other service in a large-scale Distributed Denial of Service (DDoS) attack without the device's owner noticing. Adding to the above named reasons for security vulnerabilities, manufacturers often do not have an incentive to build secure devices given the wide availability of low-cost low-security devices on the market. Additionally, usually manufacturers do not manage security patching and system upgrades once the devices are sold [4]. All these factors show the importance and relevance of cyber security in IoT, given the omnipresence of IoT devices in our daily lives.

Another field which has seen a large surge in activity in academia as well as in industry is Artificial Intelligence (AI), leading to the application of AI algorithms in various fields having an increasingly deep impact on human life. Especially in the recent years, AI has become more and more disruptive and is seen as the driver in a new technological revolution [5]. Naturally, AI algorithms have also been applied to cyber security in general and also specifically to the security of IoT devices with promising results. For instance, AI has been used in IoT for device authentication, Denial of Service (DoS) and DDoS attack defense, intrusion detection and malware detection [6].

However, AI has not only been used for cyber attack defense and Intrusion Detection Systems (IDS), but also in attacks and malware itself [7], [8]. Similarly, a previous work

concluded by the Communication Systems Group (CSG) at the University of Zurich used Reinforcement Learning (RL) to minimize the detection of a dynamic defensive system while maximizing the encryption rate of a ransomware targeting IoT devices applied as spectrum sensors. To this end, the authors introduce a RL framework called Ransomware Optimized with AI for Resource-constrained devices (ROAR) [9]. The findings show that it is possible to optimize ransomware attacks using AI, however some shortcomings were identified regarding optimizing the RL model and the ransomware implementation. Specifically, the correct detecting of behaviors was listed as a possible limitation of the proposed solution. Using a different source of data for this detection mechanisms could further improve the performance. Moreover, ideas for potential future work were listed, one of which being the incorporation of additional benign behaviors such as data compression in order to fool the detection mechanisms [1].

While at first sight it might seem unethical to try to improve malware with advanced techniques such as RL, the goal of works such as [1] and also this thesis is to create a better understanding of the threats that are inevitably coming. With this knowledge, new and more sophisticated defense mechanisms can be built, better prepared for malware attacks powered by AI.

1.2 Description of Work

This master thesis extends the above named previous work [1] concluded about adapting a ransomware sample with RL techniques in order to optimize its impact on resource-constrained spectrum sensors applied in a crowdsensing platform called Electrosense [10].

The first extension performed is the implementation of a system call (syscall) based device behavioral fingerprinting approach. Using the syscall data collected on the target device, fingerprints are generated. It is then decided by an Anomaly Detection (AD) based on the fingerprints whether the device is behaving normally or might be infected by a ransomware. This approach is executed on the Electrosense spectrum sensor running on a Raspberry Pi 3 as well as on a separate Command and Control (C&C) server running on a portable laptop. The ransomware used is a publicly available ransomware sample, adapted to be configurable at runtime. It listens for configuration changes during its operation sent by the C&C server. A RL model is trained using behavioral data collected from the target device to learn the optimal configuration that is not detected by a defensive AD assumed to also be based on syscall monitoring. This RL model is trained for different numbers of episodes and then evaluated regarding its accuracy, training time required, rewards received and steps taken per episode. Moreover, to highlight the influence of the AD on the model performance, evaluations are performed using an idealized version of the AD, which manually determines whether a configuration's behavior is detected or not. The results of this experiment are discussed and compared to the previous results. Furthermore, the results obtained are compared to the results of the previous work, to gain a better understanding of the advantages or disadvantages of the different approaches. With this in mind, many variables (*e.g.*, the ransomware configurations) were thus adopted from the previous work, in order to get comparable results.

The second extension focuses on another limitation of the previous work. The addition of other benign behaviors apart form the normal sensor behavior is suspected to have an influence on the detectability of a ransomware and thus also on the performance of a RL model as described above. To explore this, two additional benign behaviors are implemented and executed on the target device. As the scenario is built around the spectrum sensors, the behaviors are chosen be a conceivably realistic behavior used on such a device. A data compression behavior and a Python package installation both fit this narrative, and are thus used for this purpose. As in the previous case, behavioral data from the target device is collected and subsequently evaluated. The AD results regarding the additional behaviors are presented and discussed. Based on these results, the reward function used in the RL model is revised, making sure that the rewards incentivize the desired behavior of the RL agent. Furthermore, the RL agent is evaluated using the additional benign behavior, allowing to compare the new results with the results obtained without any additional behaviors. Finally, conclusions are drawn from the main findings, limitations are listed and discussed, and ideas for future studies given.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 aims to cover the theoretical foundations of the concepts touched upon in this thesis. It first introduces the concept of Spectrum Sensing through the example of Electrosense. Secondly, it introduces ransomware with its history and current relevance. Furthermore, the Chapter gives an introduction to AI and more specifically elaborates on RL. Finally, behavioral fingerprinting is explained with its different purposes and approaches.

In Chapter 3, a survey of existing literature is performed regarding the field of AI applied in malware. After a general overview of the related work, special attention is given to applications of RL in malware. The chapter concludes with an overview of the covered works comparing key dimensions.

Next, in Chapter 4, the scenario of this work is presented, highlighting the main idea and setup of the implementation. The concept of all key components used for the experiments is explained in detail.

While Chapter 4 focuses on the conceptual part, Chapter 5 gives insights into the detailed implementation of the system architecture and components. The technical specifics of each system module are highlighted, how they interact with each other, and what algorithms were explored and decided for.

Subsequently, in Chapter 6, the results and findings of the different experiments are presented, evaluated, and discussed. This includes the evaluation of the implemented prototypes, a comparison of the prototype performance with the previous work, as well as the evaluation of the experiments performed using additional benign behaviors.

Concluding this work, in Chapter 7, the limitations of this work are presented, possible ideas for future work are given in Chapter 8, and finally, in Chapter 9, a summary and conclusion are presented, reiterating the main findings of this work.

Chapter 2

Background

In this Chapter, the fundamental concepts used in this thesis are explained. The intention is to give a comprehensive understanding of the theoretical landscape, necessary for following the practical applications and scenario as well as interpreting the results presented in the subsequent chapters. Section 2.1 gives an introduction to spectrum sensing, exemplified by the description of Electrosense. Next, Section 2.2 delves into the topic of ransomware including an analysis of ransomware in IoT. Moving on, the field of AI is touched upon in Section 2.3, with a more detailed introduction to reinforcement learning. Finally, Section 2.4 covers behavioral device fingerprinting and its applications.

2.1 Spectrum Sensing

As a consequence of the massive increase in mobile communication usage, more Radio Frequency (RF) bands are in use today and the spectrum use is fragmented, bursty and diverse. To efficiently use the spectrum, understanding and having knowledge about the spectrum usage patterns is becoming increasingly important. The solution to this challenge is what spectrum sensing proposes, by continuously monitoring the electromagnetic space and processing the collected data. There are a number of solutions proposed, however this thesis will focus on Electrosense which is introduced in the subsequent section with greater detail [10].

2.1.1 Electrosense

Electrosense is a crowdsourcing solution for efficient, safe, and reliable spectrum monitoring in different regions of the world. To this end and to enable a wide adoption in the crowdsourcing scenario, the framework uses low-cost hardware, such as Raspberry Pi sensors equipped with inexpensive Software-Defined Radio (SDR) front-ends and a general purpose antenna. The authors introduced Open Spectrum Data as a Service (OSDaaS)

by implementing an open API over which the spectrum data can be retrieved, as well as offering spectrum aggregation tools.

The architecture of the Electrosense framework is composed of the following three components [10]:

1. **Sensor**: The sensor consists of simplistic, low-cost hardware and can measure from 20 MHz to 6 GHz. Optionally, the sensor can be equipped with a GPS device for time synchronization between sensors. There are two signal pre-processing pipelines implemented on the sensors: *(i)* the Power Spectral Density (PSD) and *(ii)* the In-phase and Quadrature components of raw signals (IQ). In the PSD mode, the sensors send only squared magnitude Fast Fourier Transforms (FFT) converting the data to the frequency domain, resulting in a lower bandwidth (around 50-100 Kb/s) compared to the IQ mode, where the sensors send compressed raw measurements (up to 50 Mb/s).

2. **Controller**: The controller is used to communicate with the sensors via the MQTT protocol. The controller can influence the scanning strategy of the sensors, *e.g.*, the scanned frequency range or the frequency hopping strategy. Furthermore, the controller offers an interface for administrators to manage measurement campaigns.

3. **Backend**: To collect and process the data, Electrosense uses a four-layered service. In the *ingestion layer*, the data is received and inserted into a distributed queueing system. Next, the data is simultaneously processed in the *batch layer* and the *speed layer*. In the batch layer, the raw data is stored and long-running workloads are executed, whereas in the speed layer continuous computations on a small window of recent data are performed. Finally, in the *serving layer*, an open RESTful API offers query results and data to end-users and other applications.

A number of security and privacy concerns have been addressed in the Electrosense framework, such as using secure Transport Layer Security (TLS) channels for data transmission [10]. However, as [11] shows, spectrum sensors are still vulnerable to some cyber security threats, especially in the context of Internet of Battlefield Things (IoBT), namely identity focused attacks where malicious sensors impersonate legitimate sensors, malware such as rootkits, botnets or ransomware, and Spectrum Sensing Data Falsification (SSDF) attacks where the sensed spectrum data is modified. To mitigate these threats, the authors of [11] introduce a framework to detect heterogeneous attacks on IoBT spectrum sensors using behavioral fingerprinting and Machine Learning (ML) / Deep Learning (DL) and demonstrate that the framework is able to detect malware in Electrosense spectrum sensors. In a further development, the authors of [12] propose a novel approach of using syscall-based behavioral fingerprinting for detecting SSDF attacks using Electrosense devices. Furthermore, in [13], the approach of using device behavioral fingerprinting based on resource usage was extended to classify a broad range of malware samples, including botnets, rootkits, backdoors, ransomware, and cryptojackers. The case of cryptojackers on Electrosense sensors was further explored in [14] and showed that behavior data can be used in order to detect malicious behavior in such a scenario.

2.2 Ransomware

”This section introduces ransomware basics, explaining what it is, how it operates through its lifecycle stages, and the different types it can take. In the context of our increasingly interconnected world, a focused examination is given to how ransomware affects IoT, uncovering potential disruptions to essential services and connected devices. The section emphasizes the changing tactics employed by attackers in the ransomware landscape. To address these challenges, practical countermeasures are discussed, encompassing user training, secure backup practices, and the adoption of advanced cybersecurity solutions.”[1]

2.2.1 Definition and Relevance

Ransomware is a sub-class of malware (*i.e.*, malicious software) designed to acquire revenue, with its name consisting of a combination of the two words ”ransom” and ”ware”, meaning malware that demands some amount of money in exchange for hijacked or inaccessible data [15]–[18]. In general, the evolution of malware can be divided into five phases starting in 1949 with the development of the first worms, viruses or self-propagating pieces of code. Over the years, many new types of malware have been created and malware has developed into a profitable field for cyber-criminals, but is also used by governments *e.g.*, for espionage or sabotage. In the fourth phase of malware evolution starting in 2005, ransomware was introduced [17]. Although the first documented ransomware attack happened in 1989 with a ransomware called AIDS, usage of ransomware attacks remained low, as the cryptography, distribution, and payment methods were rather limited before the invention of the internet and digital currencies [16]. Today, ransomware is a large threat to individuals and corporations with huge sums invested to counter the risk [17]. However, the use of targeted ransomware is still increasing with the numbers of attacks almost doubling in 2022 in comparison to the year before [19]. With the emergence of a new business model in cybercrime called Ransomware-as-a-Service (RaaS), the topic is as relevant as never before [18].

2.2.2 Ransomware Lifecycle

The lifecycle of a ransomware attack can be divided into different steps and there are multiple ways and number of steps to describe such attacks, also depending on the type of ransomware. A ransomware attack starts with the distribution of the malware, *e.g.*, through a malicious email, drive-by downloads or code dropper. Oftentimes, social engineering techniques are employed to lure a victim into downloading and starting a malicious program. After the distribution, the ransomware infects the host system by various different actions such as disabling backup technologies. Furthermore, the ransomware communicates with a C&C server, *e.g.*, to retrieve an encryption key. As a next step, different user-related files of interest are searched. Usually, files with specific file extensions such as `.pdf`, `.jpg`, `.docx`, etc. are selected. Next, the files are locked or encrypted,

[1] This text was generated using generative artificial intelligence

depending on the type of attack. This often includes renaming and moving the files. Finally, after the victim's data is hijacked, a ransom message is displayed demanding for a payment [15].

2.2.3 Types of Ransomware

Ransomware comes in different types and flavors, thus multiple taxonomies were put forward with the intent to classify different ransomware. In general, the literature differentiates between two high-level types of ransomware: *(i)* cryptographic ransomware and *(ii)* locker ransomware. Cryptographic or simply crypto-ransomware encrypts the victim's files and demands a ransom for the decryption. On the contrary, locker ransomware does not use encryption of files, however it prevents the victim from accessing its system or data, *i.e.*, locking it out [18]. Furthermore, there exists a type of malware called scareware, which only mimics the presence of a ransomware attack and tries to exploit the victim's fear for financial gain. In a taxonomy based on severity, crypto-ransomware is the most severe, followed by locker ransomware and finally scareware [15].

Ransomware can also be differentiated by the target of the attack. On the one hand, there are different types of victims, namely end-users vs. organizations (*e.g.*, governments, enterprises, hospitals, etc.). The main differences between an attack on end-users and on an organization are that *(i)* end-users often have a lack of security expertise and resources, *(ii)* the demanded sum is significantly lower for end-users as for organizations and *(iii)* an attack on an organization is more targeted against a single organization, whereas an attack on end-users can affect thousands of victims. On the other hand, there are different platforms on which the attacks take place, *e.g.*, PCs, mobile devices or IoT devices. PCs are the most common target of ransomware with crypto-ransomware being the main threat. As mobile phones have become more and more popular in the last 15 years, they have also become a target for ransomware attacks. Due to the openness of the environment, Android-based devices are much more prone than devices running on iOS with its hard-controlled ecosystem [18]. Regarding ransomware targeting IoT devices, Section 2.2.4 provides a more in-depth analysis of this issue.

Crypto-ransomware can also be distinguished by the method of encryption. In symmetric encryption, only one key is used both for encryption and decryption. This makes the encryption faster and it uses less resources, thus there is a lower chance of early detection and less time to react. However, there is a higher risk of key disclosure which would enable the victim to decrypt the files. The most used algorithm for symmetric encryption ransomware is AES (Advanced Encryption Standard). In asymmetric encryption, a pair consisting of a public and a private key is used. The public key is used for the encryption and can either be generated on the victim's device itself or be provided by the C&C server, in which case a connection to the C&C server would be necessary to start the attack. For the decryption, the private key is necessary. This solves to some degree the problem of key disclosure (unless of course the private key is disclosed). The most used algorithm for asymmetric key ransomware is RSA (Rivest-Shamir-Adleman). In hybrid encryption ransomware, the advantages of symmetric and asymmetric encryption are combined. First, symmetric encryption is used to quickly encrypt the victim's data.

Then, the symmetric key is encrypted using the public key of an asymmetric key pair [15], [18].

2.2.4 Ransomware in IoT

With the enormous increase in IoT devices, the risk of ransomware attacks in the IoT space has also grown in the last years and will likely continue to grow significantly in the future [18]. Although IoT devices are not typically used to store data, they can still be affected by an attack, causing serious harm such as interruptions of manufacturing processes, power outages, losing access to surveillance systems, etc. Wearable devices such as smart watches have also been affected by ransomware, coining the term "ransomwear" [15]. Generally, the focus so far of ransomware attacks in IoT is not on tiny devices, but on mission critical and real-time systems [20]. As discussed in Section 1.1, cyber security in IoT is a major challenge getting more and more important the more omnipresent IoT devices become in out daily lives.

As IoT devices usually have constrained resources, traditional security mechanisms to prevent ransomware attacks will not be applicable due to increased resource requirements. In comparison to regular PCs or mobile devices, IoT devices have several key differences when it comes to executing a ransomware attack. The distribution methods have to be adapted, as for instance malicious email attachments cannot be applied on IoT devices. An attacker thus has to be aware of the topology of the IoT network under attack, such that the attack can be focused on devices that control or manage other devices. A further challenge is determining the owner of IoT devices to whom the ransom demand has to be sent. Moreover, many IoT devices do not have a screen, requiring a different method of communicating the ransom demand [20].

2.2.5 Countermeasures

To counter a ransomware attack, there are several approaches, for instance *(i)* attack prevention, *(ii)* attack detection, or *(iii)* attack recovery [15], [18]. In order to prevent an attack from happening in the first place, there are approaches that try to stop the malicious program from executing the attack (*i.e.*, proactive approaches) and approaches that try to mitigate the effects (*i.e.*, reactive approaches) [15]. In addition, increasing end-user security awareness can prevent ransomware, especially in the distribution phase of an attack. A promising countermeasure against ransomware is the detection of an attack, especially using ML techniques. The ML-based detection of ransomware attacks can be performed using structural features coming from static analysis of the attack binaries, or using behavioral features from dynamic analysis of ransomware [18]. Section 2.4 delves more into the approach of generating a behavioral fingerprint to detect anomalies in a device's behavior. Attack recovery on the other hand tries to recover the hijacked data with three approaches: *(i)* recovery of keys, *(ii)* recovery of files via hardware, or *(iii)* recovery via a (cloud) backup [18].

2.3 Artificial Intelligence

With the term ”artificial intelligence” being coined in 1956, there have since been many definitions and new developments. However at the core, the idea of AI is to simulate, extend and expand human intelligence and behaviors with the help of computers [5], [21]. The development of the research field of AI can be roughly split into multiple phases [22]:

- **1950s - 1970**: Birth of AI

  Although the first neural networks were developed in the 1940s, the 1950s are regarded as the incubation period of AI. Works that still today are important were developed, such as the Turing Test or the Perceptron model [5], [22].

- **1970s**: First AI winter

  After nearly two decades of success in the field of AI, in the 1970s the high spending on research for AI was criticized, also because of the failure to deliver to inflated expectations and over-confidence. This caused a decrease in funding leading to the first AI winter [5].

- **1980s**: New wave of interest

  After the first AI winter, a new wave of interest triggered more active development in the 1980s [5].

- **late 1980s - 1990s**: Second AI winter

  With a shift of interest to other new technologies came a new AI winter with decreased funding and lowered expectations [5].

- **2000s - present**: AI boom

  Since the early 2000s until today, the field of AI has seen many breakthroughs and rapid development. There are multiple catalysts for this, such as the success of ML, the increase in compute power, and the availability of huge amounts of data [5], [21].

Within AI, ML is seen as a vital paradigm which fueled the progress of AI in the last decades [5]. ML can be divided into three different types: *(i)* supervised learning, *(ii)* unsupervised learning, and *(iii)* reinforcement learning (RL). In supervised learning, a model learns from a labeled dataset, *e.g.*, for categorization problems. In unsupervised learning, the goal is typically to find hidden structures and patterns in unlabeled data [23]. The following section will give a more detailed introduction into RL.

2.3.1 Reinforcement Learning

The concept of RL is based on the idea of learning by interacting with an environment, similar to the way humans or other animals learn. Originally, RL was inspired by biological learning systems and the psychology of animal learning during some of the earliest work in AI.

In RL, the central element is an agent which interacts with an environment and learns from its experience. This agent is able to collect information about the state of the environment and thus learn about the effects of his actions without receiving instructions from the outside, making it a closed-loop system. The actions performed by the agent may also have a long-lasting effect, influencing all future situations. Furthermore, the agent has a goal concerning the state of the environment [23].

Typically, a RL system consists of four elements [23]:

- **Policy**

  A policy defines the actions taken by the agent in a given state and by that is the core of the agent as it defines its behavior. It may involve complex computations or be a simple function or lookup table.

- **Reward Signal**

  At each step, the agent receives a number (*i.e.*, the reward signal) from the environment, which defines good or bad events for the agent. The agent wants to maximize the total reward over the long run, making a high reward the sole objective.

- **Value Function**

  In contrast to the reward, the value function defines what is good in a long-term sense. The value of a state is defined as the total reward which is expected taking into account states that are likely to follow. Since values have to be estimated based on observations of the environment, they are much more complex to determine compared to rewards and thus efficient value estimation is a central problem in RL.

- **Environment Model** (optional)

  In model-based methods, the behavior of the environment is modeled which can be used for planning, *i.e.*, taking possible future situations into consideration when making decisions. There are also model-free methods, in which the agent learns in a trial-and-error fashion.

At a given state, the agent is faced with a choice of different actions. For each action, the agent knows the estimated value of the actions, however not the exact value with certainty. In order to decide for one action, there are different strategies, for example the agent could always choose the action with the highest estimated value. This is exploiting the agent's current knowledge and is called *greedy*. However, since the agent does not know the actions with certainty, it could be that some other action has actually a higher value, although its estimate is lower. Therefore, exploring other *nongreedy* actions is improving the agent's knowledge and may lead to a greater total reward long term. There is thus a need to balance between exploitation and exploration. One such strategy is to use a parameter $\epsilon \in [0, 1]$ giving a probability with which nongreedy actions are explored [23].

To formalize the notion of sequential decision making, Markov Decision Processes (MDP) can be used, also to have more mathematical tractability and be able to make precise theoretical statements. In finite MDPs, there is an agent, an environment, and rewards

which the agent tries to maximize, with the set of states, actions and rewards being finite. The probabilities for each state and reward at some point in time depend only on the preceding state and action, but not on earlier ones. This is called the Markov property. As the agent's goal is not to maximize the immediate rewards, but to maximize the expected value of the cumulative reward over the long run, the future rewards have to be discounted to find their present value. This is accomplished with a parameter $\gamma \in [0, 1]$ called the discount rate:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.1}$$

where G_t is the expected return and R_t denotes the reward received at time step t. The closer γ is to 1, the stronger future rewards are valued.

For deciding on which action to take, the value of performing each given action needs to be defined, which is accomplished by a policy π mapping states to probabilities of selecting actions. In so-called Monte Carlo methods, the agent is following a policy and maintains an average of the actual returns received. By the law of large numbers, this sample average will converge to the actual values as the sample size approaches infinity. This is important as the agent can learn directly from experience by interacting with the environment, without the need of a model of the environment. This also makes it possible to learn from simulated samples. In order to improve a policy, a concept called Dynamic Programming (DP) can be applied, assuming there is complete knowledge of the environment. This assumption together with the high computational demand are limiting factors of DP. In DP methods, iterations over the state set are performed with the value of each state being updated based on the values of possible successor states and their probabilities. This concept of updating estimates based on other estimates is called bootstrapping.

Another method which uses bootstrapping, however contrary to DP does not require a perfect model of the environment is Temporal-Difference (TD) learning. It is thus a combination of DP and Monte Carlo methods. Equation 2.2 shows the simplest form of a TD update:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.2}$$

where $V()$ is the value estimate, α is the step-size parameter, R_t is the reward at time t, γ is the discount factor and S_t gives the state at time t.

One example of such a TD algorithm is Q-learning. In the field of RL, Q-learning was an early breakthrough being put forward by [24] in 1989 and is now among the most widely used methods in RL. As the name suggests, the action-value function Q is learned in order to approximate the optimal action-value function $q*$, independent of the policy used. As long as all state-action pairs keep being visited and thus updated, the method will reach correct convergence. The update of the Q function is given by Equation 2.3:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{\alpha} Q(S_{t+1}, \alpha) - Q(S_t, A_t)]. \tag{2.3}$$

Algorithm 2.1 shows the pseudocode representation of the Q-learning algorithm over multiple episodes. The algorithm takes a step-size (or learning rate) parameter α as well as an exploration balancing parameter ϵ as input. After initialization, it loops over the episodes and steps for each episode while applying Equation 2.3 to update its estimation of the Q function [23].

Algorithm 2.1 Q-Learning Algorithm (Pseudocode) [23]

Require: $\alpha \in (0, 1]$, $\gamma \in [0, 1]$, *small* $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$ arbitrarily except that $Q(terminal, \cdot) = 0$
for each episode **do**
    Initialize S
    for each step in episode **do**
        Choose A from S using policy derived from Q (*e.g.*, ϵ-greedy)
        Take action A
        Observe reward R
        Observe next state S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    end for
end for

2.4 Behavioral Fingerprinting

The field of device behavioral fingerprinting has in recent years developed into an increasingly crucial solution to various problems in IoT. On the one hand, behavioral fingerprinting is used to identify device types as well as individual devices. On the other hand, it is also applied to detect anomalies in the device behavior due to faults or cyber attacks. Generally speaking, the goal of device behavioral fingerprinting is to create device behavior patterns (*i.e.*, fingerprints) by monitoring the device's behavior and collecting data. This data is then in a next step processed resulting in a behavior fingerprint ready to be applied as needed. To this end, there are various different data sources, data collection methods, data processing approaches, and application scenarios [25].

2.4.1 Device Identification

Behavioral fingerprinting is used for device type identification (*i.e.*, device category such as general computer, mobile device, IoT sensor, etc.), device model identification (*i.e.*, devices of the same type but different hardware or software configurations) as well as individual

device identification (*i.e.*, differentiate devices of the same model). For device model identification, the data source usually is based on network communication such as packet headers statistics or network flow statistics. For individual device identification on the other hand, the data sources are on a much lower level related to slight hardware variations such as data from clock skew, electromagnetic signals, system processors, system circuits, or resource usage. For instance, [26] uses behavioral device fingerprinting to identify single-board computers of the identical model. The authors identified seven properties that such a fingerprint needs to consider: uniqueness, stability, diversity, scalability, efficiency, robustness and security.

To process and analyze the collected data, most solutions for device identification (both model and individual) rely on some sort of classification algorithm based on ML. These algorithms are trained using a labeled dataset (*i.e.*, a supervised learning algorithm), creating the need for large amounts of data to train as well as extensive compute resources. Overall, using behavior fingerprinting for device identification on various levels has proven successful in contributing among other fields to security, network optimization and network analysis [25].

2.4.2 Anomaly Detection

The application scenarios for AD using device behavior fingerprinting are twofold: *(i)* malfunction and fault detection and *(ii)* attack detection. Generally speaking, for AD, the usual approach is to model a normal behavior of a device using fingerprinting and then detect deviations from that normal behavior.

Assuming that a fault or malfunction of some component affects the behavior of the whole device, such AD techniques can identify these situations. Causes for this can be anything from hardware failures, service or hardware overload or network issues. Depending on the type of application scenario, the data source used for generating fingerprints is different. For instance in systems with more resources such as cloud systems, resource usage and system logs are more prevalent, whereas in IoT and other embedded environments network and sensor-based data sources are more common [25].

Today's IDSs often rely on static features such as previously known attack signatures stored in a repository. This is problematic as it does not consider so-called zero-day attacks, meaning attacks that are unknown or have never been seen before. Signature-based IDSs have proven successful in detecting known attacks, provided that the attack repositories are regularly maintained. In contrast, anomaly-based IDSs are also able to produce strong capabilities for zero-day attack detection [27]. Device behavior fingerprinting can be used to detect anomalies in a device's behavior due to attacks. The most used data source for attack detection is network data, however also sensor data, syscalls, logs, software signatures, hardware events, or resource usage can be used to generate a fingerprint. To process the data, two main approaches have developed: *(i)* modeling normal behavior and detecting anomalies or *(ii)* collecting a labeled dataset of normal and abnormal data and performing classification of input data [25].

The work which this thesis is extending as well as several other works applied behavioral fingerprinting on Electrosense devices with the goal of detecting anomalies inflicted by a ransomware or other types of malware infection. The fingerprint is based on resource usage data using the Linux command `top` to collect data on CPU and memory usage and running tasks. Furthermore, the command `perf` was used to select 75 different Performance Monitor Unit (PMU) events. The data was periodically collected and the fingerprint generated on the device itself, while the data analysis and anomaly detection was performed on a separate system environment [1], [11], [13], [14].

Chapter 3

Related Work

”For this chapter, a literature review was conducted to examine existing research on the incorporation of AI in malware. Google Scholar[1] was employed to search for relevant literature using key terms such as ”AI malware”, ”RL malware”, ”ML malware”, and ”DL malware”.

The subsequent section of the chapter presents an overview of related work in the field, focusing on four surveys that provide a comprehensive understanding of the topic. Additionally, applications that specifically utilize RL in the context of malware are explored in Section 3.1.2.

The section aims to present the findings of the literature review in a clear and concise manner, offering insights into the different aspects of AI’s role in the development and enhancement of malware. By summarizing the surveys and examining applications using RL in malware, readers gain a practical understanding of the advancements in this specialized area.

To conclude the chapter, Section 3.2 serves as a summary of the related work, bringing together key insights, overarching themes, and notable contributions discussed throughout the chapter. This synthesis provides readers with a comprehensive overview of the current state of AI-infused malware research, laying the groundwork for subsequent sections that delve into methodologies, experimental setups, and novel contributions in this evolving field.”[2]

3.1 AI in Malware

Generally, a majority of the available literature uses AI to defend some system or device from malware. Often, AI algorithms, *e.g.*, ML or DL, are employed with the goal of detecting malicious software. This can be performed using static databases of known malware samples where the defensive system compares the software in question with this

[1] https://scholar.google.com/

[2] This text was generated using generative artificial intelligence

database. It is thus crucial that this database is kept up to date. This approach however does not cope well with new and unknown attacks, so-called zero-day attacks [27].

On the contrary, this section focuses on the use of AI for the offensive side to enhance malware or avoid detection. As will be shown, the field of AI-powered malware is fairly new, however rapidly evolving and supported by the immense developments of AI in the last years. In academia, the research field is also gaining more traction and interest, as the number of available publications has risen significantly in the last years. This only shows the increasing importance and threat level of AI-powered malware.

In the next section, a general overview of AI used in malware is presented using works that surveyed the existing literature. To dive deeper into applications of RL in malware, Section 3.1.2 presents some specific works and how and for which purpose RL was applied there.

3.1.1 Surveys

Four different surveys on the topic of AI in malware were selected with the first having been written in 2008 [8], one in 2019 [7], one in 2020 [28], and the most recent one in 2022 [29].

In [8], the authors surveyed the existing literature using AI for malware as well as defense from malware attacks. Four different use cases of AI in malware were defined: *(i)* malware incorporating AI technologies, *(ii)* malware exhibiting intelligent-like behavior, *(iii)* malware behaving like biological equivalents, and *(iv)* malware behaving like humans or intelligent behaviors. Of these, especially categories *(i)*, *(ii)* and *(iv)* are of relevance for this work.

One example of a malware using AI technologies given in [8] is *Zellome*, which was called "an unusual example of self-compiling malware and a novel misapplication of artificial intelligence" [30]. This virus used a genetic algorithm for a polymorphic decryptor, which is not an obvious application for a genetic algorithm, as the same could have been achieved with less complex techniques such as hash tables [30].

Furthermore, the authors of [8] mention in their survey the use of AI techniques to impersonate humans in social engineering attacks. In such attacks, the victim is contacted by the AI bot giving the impression of another person with the intention of infection with a virus by clicking a link.

In a more recent survey, the authors of [7] describe four ways in which malware can take advantage of AI techniques: *(i)* evasion techniques, *(ii)* autonomous operation, *(iii)* AI against AI-powered malware defense and *(iv)* bio-inspired computation and swarm intelligence. Evading detection is important, since the longer it takes to detect malware, the more time it has to perform malicious activities. There are multiple different approaches to avoid detection, such as dodging sandbox detection where the malware recognizes its environment and acts as a benign program in sandbox environments, and only acts maliciously when running on an actual target device. Furthermore, malware can adapt to its environment by taking contextual information into account.

To attack a system using AI-powered defense techniques, adversarial attacks can be used to provoke false predictions. In adversarial attacks, the input into a model is designed to give an incorrect output, *e.g.*, the malware could behave in a way that is wrongly classified as benign [7]. This concept has been applied in several different works such as [31]–[35]

In their survey from 2020, [28] describe five categories of AI-based cyber attacks: *(i)* next-generation malware, *(ii)* voice synthesis, *(iii)* password-based attacks, *(iv)* social bots, and lastly *(v)* adversarial training. For this work, the first category holds the greatest level of interest. One example of such next-generation malware is DeepLocker, a targeted malware which uses the lack of understanding in the decision making of AI models in order to evade detection. The malware identifies its target taking into account the geolocation or voice and facial recognition. If the target was not recognized, the malware remains inactive. DeepLocker uses a Deep Neural Network (DNN) both for concealment and unlocking of the malicious payload. The black-box characteristics of such DNNs makes DeepLocker impossible to detect with traditional techniques [36]. Another example given is a self-learning malware that uses monitoring data of the target system to learn attack strategies with maximal impact. The malicious actions are disguised as accidental failures, reducing the detection likelihood. Furthermore, the malware code is divided into logical modules using independent threads. Each module then removes the traces of the previous module in order to avoid detection [37].

In the most recent survey on the topic from 2022, the authors of [29] aim to show the state of the art of AI-enhanced malware as well as the techniques used to evade and attack defensive systems which also use AI. Five uses of AI in malware are identified: *(i)* Hiding malware from detection, *(ii)* evading traffic detection, *(iii)* attacking defensive AI, *(iv)* attacking authentication factors on mobile devices, and lastly *(v)* other use cases including improved phishing attacks and cyber-physical sabotage. In order to hide malware, multiple different approaches are mentioned, *e.g.*, hiding the malicious code inside of unsuspicious AI models, code perturbation, code generation using Generative Adversarial Networks (GAN), adversarial attacks on detection systems and lastly, sandbox detection. A further relevant use of AI in malware mentioned in the survey is the evasion of network traffic detection. Using unsupervised learning techniques, malware has been shown to evade IDSs while hiding probing, infiltration and C&C network traffic.

3.1.2 Applications of RL in Malware

In this section, related work will be presented with a specific focus on the use of RL in malware. Again, the literature search was performed using Google Scholar using the search term ”RL malware”.

A group of works have been conducted with a similar goal of using RL for modifying Windows Portable Executable (PE) files in order to evade detection by a ML classifier. Usually, to create a more realistic scenario, the classifier used was a binary black-box classifier, *i.e.*, no information about the inner workings was available, only the binary result (malicious / benign). For instance, in [32], [38]–[42] a Deep Reinforcement Learning (DRL) approach such as deep Q-learning was used to evade detection. In such an approach, the RL agent is interacting with a malware sample and selects a sequence of actions, which are

reasonable, functionality-preserving modifications of the malware PE file. One example for such an action would be the addition of a new redundant section. After the training phase, the agent is able to specify the optimal order of actions, such that the malware evades detection by a ML model. This approach has proven successful, with high evasion rates being achieved against different ML classification models and commercial antivirus software.

[43] uses a more classical RL approach, regarding the adversarial attack as a multi-armed bandit problem. When modifying a PE file with a certain action, the content of the modification is as important as the type of action. Thus, the action and its content are treated as a unit and the action-content pair is modeled as an independent slot-machine in the multi-armed bandit problem. It was found that the result was influenced by only a small number of actions and by identifying these essential actions, the root cause of the evasion can be explained. The results show that evasion rates of 74 % - 97 % against ML classifiers and 32 % - 48 % against commercial antivirus software are achieved.

Some RL approaches are limited when it comes to large action-space problems such as opcode level obfuscation. Thus, in a work from 2021, the authors of [31] proposed *AD-VERSARIALuscator* which stands for *Adversarial Deep Reinforcement Learning based obfuscator and Metamorphic Malware Swarm Generator* with the goal of creating metamorphic instances of malware samples by introducing machine language obfuscations and with that evading detection. A Proximal Policy Optimization (PPO) approach is taken in order to deal with the large action-space of opcode obfuscation, possibly including thousands of opcodes. The work demonstrated a 33 % success rate of metamorphic malware evading IDS.

In [44], the authors use another RL approach called Variational Actor-Critic (VAC) for creating adversarial malware that evades detection by ML classifiers. The agent applies actions to the malware which are additive (*i.e.*, adding content to the malware) as well as editing existing content while keeping functionalities intact. VAC performs very well for large action-spaces, however, it cannot be directly applied to discrete action-spaces such as in the case of generating adversarial malware. Therefore, the authors propose an extension which enables the application of VAC for this purpose. Furthermore, the proposed technique allows for further analysis on understanding the weaknesses of the detector.

What is interesting is that no works were found which applied RL during the attack itself, instead RL was applied to modify the malware in some way before the actual attack with the intent of evading detection. On the contrary, this work and the one which this work is based upon use RL to dynamically change the behavior of the malware during the attack, responding to changes of the environment. Also, no works were found which applied RL to malware targeting specifically resource constrained devices such as IoT devices.

3.2 Summary

To summarize the related work presented in the previous sections, Table 3.2 gives an overview over all the covered works. The works are sorted to first list the four surveys,

and otherwise sorted in chronological order. These five different aspects of the works are highlighted:

1. **AI technique**: the employed technique used in the malware, *e.g.*, generative adversarial networks (GAN), deep reinforcement learning (DRL), machine learning (ML), deep learning (DL) or variational actor-critic (VAC).
2. **AI application**: where the AI technique is applied, *e.g.*, Windows OS, PDF files, cyber-physical systems (CPS), or Windows portable executable files (PE).
3. **Evasion**: the type of defensive mechanism trying to detect the malware which in turn is being evaded by the malware, *i.e.*, *static* (without executing the malware) or *dynamic* (the malware is executed by the defensive system *e.g.*, in a sandbox environment) detection.
4. **Adversarial**: whether the AI technique is used to generate an adversarial attack against another AI system on the defensive side.
5. **Phase**: at what point in time the AI technique is employed, *i.e.*, *before* or *during* the attack.

Looking at Table 3.2, it can be observed that only two works published before the year 2016 were covered, one work in 2005 and one survey in 2008. However, since 2016 almost every year there were multiple publications made. This speaks for the fast development and increasing importance of the topic.

Many works employing some form of AI in malware are focused on generating some variation of an existing malware sample using AI. This modification often has the goal of evading detection by some ML/DL classifier, making it an adversarial attack. Of the reviewed works belonging to this group, all are modifying Windows PE files with only one exception modifying the opcode. This can be observed in Table 3.2 by the multiple works applying the AI to PE files, evading a static detection mechanism, and being classified as an adversarial attack.

This process of generating an undetectable variation has to happen before the actual attack, which explains the small number of works that use AI techniques during the attack itself. Apart from this work and its predecessor, only two other works were found applying AI during the attack phase. Firstly, *DeepLocker* acts as a wrapper around some malware payload, which is decrypted if a specific victim has been identified, *i.e.*, the AI is not really active in the damage phase of the attack. Moreover, *DeepLocker* is evading both static and dynamic detection, however it is not adversarial as no ML system is targeted [36]. Secondly, in [37], attack strategies are learned from system monitoring data, injecting strategic failures at critical time and location. This approach is the only work found truly using AI techniques during an attack. Although no specific detection system is described, this work evades dynamic detection by disguising the malware's operation as accidental failures and with approaches such as trace minimization.

Looking at the use of RL in malware, the only applications found were in malware sample modifications against a detection system. Thus, there still exists a research gap regarding the application of RL in a dynamic environment, applied during the attack phase

Table 3.1: Overview and Comparison of Related Work.

Reference	Year	Survey	AI Technique	AI Application	Evasion	Adversarial	Phase
[8]	2008	yes					
[7]	2019	yes					
[28]	2020	yes					
[29]	2022	yes					
[30]	2005		genetic algorithm	Windows OS	-	no	before
[33]	2016		genetic algorithm	PDF files	static	yes	before
[34]	2016		GAN	domain generation	static	yes	before
[39]	2017		DRL	PE	static	yes	before
[35]	2017		GAN	PE	static	yes	before
[32]	2018		DRL	PE	static	yes	before
[36]	2018		DL	malicious payload encryption	static / dynamic	no	during
[38]	2019		DRL	PE	static	yes	before
[37]	2019		ML	CPS	dynamic	no	during
[41]	2021		DRL	PE	static	yes	before
[43]	2021		RL	PE	static	yes	before
[44]	2021		RL (VAC)	PE	static	yes	before
[31]	2021		DRL	opcode	static	yes	before
[40]	2022		DRL	PE	static	yes	before
[42]	2023		DRL	PE	static	yes	before
[1]	2023		RL	ransomware behavior	dynamic	yes	during
this work	2024		RL	ransomware behavior	dynamic	yes	during

directly influencing the behavior of the malware. Furthermore, the application of such AI-enhanced malware in the realm of IoT is also not covered extensively by the existing literature. Especially regarding the rapid rise of IoT, it can be seen that there is a need for further advancements in the research of AI-powered malware used in devices with constrained resources. To this end, this work is aimed at helping to bridge this gap in the research.

Chapter 4

Scenario

In this chapter, the key components and actors of the attack scenario are described, giving context to the idea of the development of this work. First, the overall idea of the scenario is explained, followed by a description of the environment used to execute the scenario, with the main components firstly on the client side and secondly on the server side being covered in detail.

The overall goal of this scenario was to develop a RL agent selecting different ransomware configurations in order to remain undetected. In this scenario an IoT device was targeted by a ransomware. On the device, a defensive system was employed which recognizes anomalous behavior of the device. This defensive system was treated as a black-box making the scenario more realistic. The RL agent also employed such an AD using the device behavior in order to learn which actions result in being detected or remaining hidden. Furthermore, additional benign behaviors were executed on the client device in parallel to the ransomware's encryption, in order to explore the influence on the detection rates of the ransomware.

Figure 4.1 shows the overall system architecture and the setup of the environment used with all the main components. In the remainder of this chapter, the environment and its components are described in more detail, both on the client side as well as the C&C server side.

4.1 Environment

The environment used in this work had two main components:

1. Client: IoT device targeted in a ransomware attack
2. C&C Server: Device taking the role of a command and control server, managing the attack

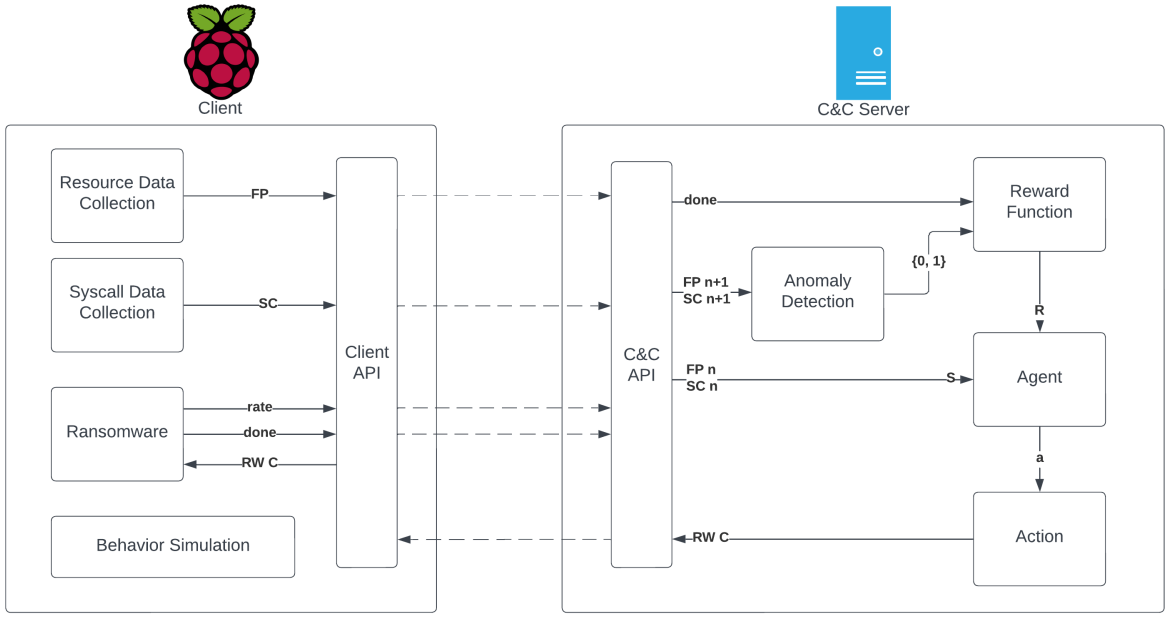

Figure 4.1: System Environment / Architecture

Specifically, for the client device, this work used a Raspberry Pi 3 Model B Rev 1.2 device running the Electrosense software on a Linux based operating system (Raspbian GNU/Linux 9 (stretch)). The client device was equipped with an antenna used for the Electrosense functionalities, as well as a SDR module connected via USB to the device. The device was connected via Ethernet cable to a Local Area Network (LAN).

A device running the C&C software was deployed in the same network as the Raspberry Pi. On this device, everything regarding the fingerprint analysis, RL model, and attack coordination was executed. This work used an Acer Aspire E5 running with 16 GB of memory, an Intel i7-8550 CPU, and a NVIDIA GeForce MX150, using Windows 10 Home.

4.2 Client

On the client side, three main components were active:

1. Ransomware
2. Fingerprint collection
3. Benign behavior execution

4.2.1 Ransomware

The ransomware's main task is, as discussed in Section 2.2, to render unavailable (usually encrypting) the victim's data. In this scenario, the targeted IoT device had already been

infected with the ransomware, which had full access to the device's data storage. Furthermore, the ransomware had an established way of communicating with the C&C server, on the one hand to receive different configuration settings regarding *(i)* the algorithm used for encrypting the files, *(ii)* the rate of encryption, *(iii)* the duration of an encryption burst, and lastly *(iv)* the pause between encryption bursts. On the other hand, the ransomware communicated to the C&C server once the encryption had concluded and successfully encrypted all specified files in the targeted settings. One part of traditional ransomware which was left out of scope in this work is the delivery of the ransom message to the device's owner, as well as everything regarding the anonymous payment of the ransom fee. The focus lied solely on the encryption part of a ransomware attack while not being detected.

The ransomware used in this work was the same as in [1], an extended version of Ransomware_PoC [45]. Implemented in Python, this ransomware sample starts by recursively searching for files with configurable extensions for encryption (*e.g.*, `.pdf`, `.png`, `.docx`, etc.). In a second step, using hard coded keys, these files are then encrypted on a binary level in place in order to not use any additional disk space and are renamed with a custom file extension `.wasted`. For the purpose of [1], the ransomware was extended with several additional capabilities. The main extension involved making the ransomware configurable during runtime, allowing adjustments for using different encryption algorithms, encryption rate, and burst settings including the burst duration and burst pause. The encryption algorithms included *(i)* AES-CTR, *(ii)* Salsa20, and *(iii)* ChaCha20. The encryption burst settings involve a burst duration specified either in seconds (*e.g.*, `s10`) or a number of files (*e.g.*, `f1`), and a burst pause in seconds. Furthermore, the extension of Ransomware_PoC encompasses communication with a C&C server. This enables a C&C server to update the ransomware's configuration during runtime. Additionally, the ransomware reports the achieved encryption rate to the server.

The different configurations of the ransomware used are shown in Table 4.1. The six different configurations were adopted from [1], in order to be able to compare the results. The first configuration (Config 0) has a very low encryption rate while encrypting one file per encryption burst. In practice, such a low encryption rate would not be very effective in the context of a ransomware, as it would take a very long time to encrypt a realistic amount of a victim's data. The next two configurations (Config 1 and Config 2) are the most aggressive configurations with unlimited encryption rates and no pause between bursts. These two configurations form the control group to verify that the AD module is working effectively. The fourth configuration (Config 3) has a rather high encryption rate and short burst pause, making this the fastest configuration apart from the control group. Config 4 and Config 5 have the same encryption rate with different burst settings, Config 4 has a relatively short burst duration of 20 s with a burst pause of 40 s, while Config 5 has a burst duration of 120 s and a burst pause of 30 s.

As a data corpus for the encryption, a dataset of `.pdf` files was used provided by [46]. Analogous as in [1], to select a smaller part of the whole dataset for the purpose of the experiments, the following command was used:

```
find /root/corpus/data/ -size -6 -exec cp "" /root/test_ransomware/data/
```

This selected 11 files with a total size of 32'519 bytes to be used for the encryption process.

Table 4.1: Ransomware Configurations

Config	Encr. Algorithm	Encr. Rate (B/s)	Burst Duration	Burst Pause (s)
Config 0	ChaCha20	16	f1	60
Config 1	AES-CTR	565566*	s0	0
Config 2	Salsa20	632835*	s0	0
Config 3	AES-CTR	500	s10	5
Config 4	ChaCha20	200	s20	40
Config 5	Salsa20	200	s120	30

**Unlimited by configuration, value computed from average rates in [1]*

4.2.2 Fingerprint Collection

Another component which acted in parallel to the ransomware was the fingerprint data collection. Here, data of the device's behavior forming the basis for the creation of fingerprints was collected. Once collected, the data was sent to the C&C server, where the data was further processed. The decision not to preprocess the data on the client side was made since the resources of the IoT device were constrained and the observing of the device's behavior should only influence the behavior itself in a minimal way.

In the previous work, the data was collected from observing the system resources. In this work however, the data collection module was extended to collect data from observing syscalls. Syscalls are the interface of an Operating System (OS) between the user space (*i.e.*, applications) and the kernel space. The specification and thus the available syscalls depend on the OS, however the main idea remains the same [47]. As every action performed using a device is using some specific syscalls, collecting data from the usage of system calls from all processes running on the device should give a very detailed picture of the device behavior. The syscalls were continuously monitored and written to a file. Every 5 seconds (plus some delay from processing the data), the collected data was then sent to the C&C server for further processing, together with the collected resource usage data from [1]. After the request was sent, the next monitoring cycle was started.

4.2.3 Benign Behavior Execution

In order to investigate the adaptability of the model and the influence of the device's normal behavior on the detectability of the ransomware, different benign behaviors were executed on the client device. Until now, the scenario included only the normal behavior of the Electrosense device, which shows very uniform device behavior, as it is a single purpose device. Nevertheless, also on a device like in this scenario different benign behaviors were conceivable and those behaviors should also have been classified as normal by a defensive AD system. To this end, different behaviors were executed in parallel to the other components described in the sections above. The hypothesis here was that depending on the underlying normal device behavior it might make it harder to detect an ongoing ransomware attack. In other words, an AI-powered ransomware could use this as

an advantage to find configurations that remain undetected. Two different benign behaviors were explored and tested: *(i)* a data compression behavior and *(ii)* a Python package installation behavior. These two behaviors were selected as they both are a reasonable behavior in the case of an Electrosense sensor. Furthermore, both behaviors access the file system, while the second behavior also uses the network to download packages, resulting in different behavior patterns. The AD was then trained with the normal sensor behavior plus the additional benign behaviors. The ransomware configurations, also with the same benign behavior running in the background were then evaluated to see whether the ransomware detection changed.

4.3 C&C Server

On the server side, there were the following main components:

1. Fingerprint data feature extraction
2. Anomaly detection
3. RL reward function
4. RL agent

The remainder of this section explains each of the main components in detail, what the objective was and how they interacted with each other.

4.3.1 System Call Feature Extraction

The collected fingerprint data was sent from the client device to the C&C server via a RESTful API. Using a POST request, the attached data files were received and stored on the server. As the amount of data from collecting syscall data was relatively large and text-based, feature extraction was needed to gain relevant data in numerical form in order to be used by the AD.

As the syscall data consisted of text, Natural Language Processing (NLP) techniques had to be used for feature extraction. Furthermore, the input data was of variable length, while algorithms for AD usually expect input of a fixed length. Unlike a text in natural language, in the syscall data the order of the tokens / words was not as critical for extracting useful information. For this reason, the frequencies of the individual word occurrences were used as features, as different behavior was thought to show different frequencies of the used syscalls.

4.3.2 Anomaly Detection

As covered in Section 2.4.2, in device behavior fingerprinting AD can be used for attack detection. In this scenario, the client device was employing such an AD module on the defensive side, to recognize anomalous behavior due to attacks on the basis of syscall data. Furthermore, the ransomware was also using AD on the offensive side in order to train its RL agent. For this work however, the focus lied on the offensive side, thus the defensive AD was not implemented or explored.

Taking the extracted features of the syscall data as input, the AD module produced a binary classification into normal and anomalous. In a later step, the output of the AD was used for the reward computation for the RL model. Depending on the experiment, the AD was trained with different data regarded as normal behavior. First, the AD was trained with the data collected from the device's normal behavior. Secondly, the AD was trained with a training dataset comprising both the normal sensor behavior, as well as additional behaviors executed on the device. It was then explored whether the training data had an influence on the detection rates of infected data.

4.3.3 RL Reward Function

As covered in Section 2.3.1, in RL the learning agent receives a reward signal after each step with the main goal of maximizing the rewards received over the long run. The reward function thus is responsible for determining the reward given to the agent after each step according to the state that the environment is in.

Deciding what comprises a good state or a bad state can be tricky, as there can be multiple variables playing a role and these variables have to be weighted accordingly. In this case, there was the AD output saying whether the previous action had been detected or remained hidden. Thus, in a first prototype, a simple reward function was implemented only using the AD output. In a second enhanced prototype, a reward function was used that in addition to the AD output also used the encryption rate of the selected action for determining the reward. For a detailed description of the used reward functions and their implementation, refer to Section 5.4.

4.3.4 RL Agent

This work used a Q-learning agent to learn the best configuration to select. The concept of Q-learning was covered in Section 2.3.1. For this, an estimation of the Q function had to be implemented, in this case using a simple neural network adopted from [1], depicted in Figure 4.2. This network consisted of three fully-connected layers. The number of input neurons depended on the number of features of the dataset. After that, a logistic activation function was used, followed by the hidden layer with a configurable number of hidden neurons. Finally, a second activation function like Rectified Linear Unit (ReLU) or Sigmoid Linear Unit (SiLU) was used followed by the output layer with its size corresponding to the number of ransomware configurations (*i.e.*, 6). Moreover, the neural

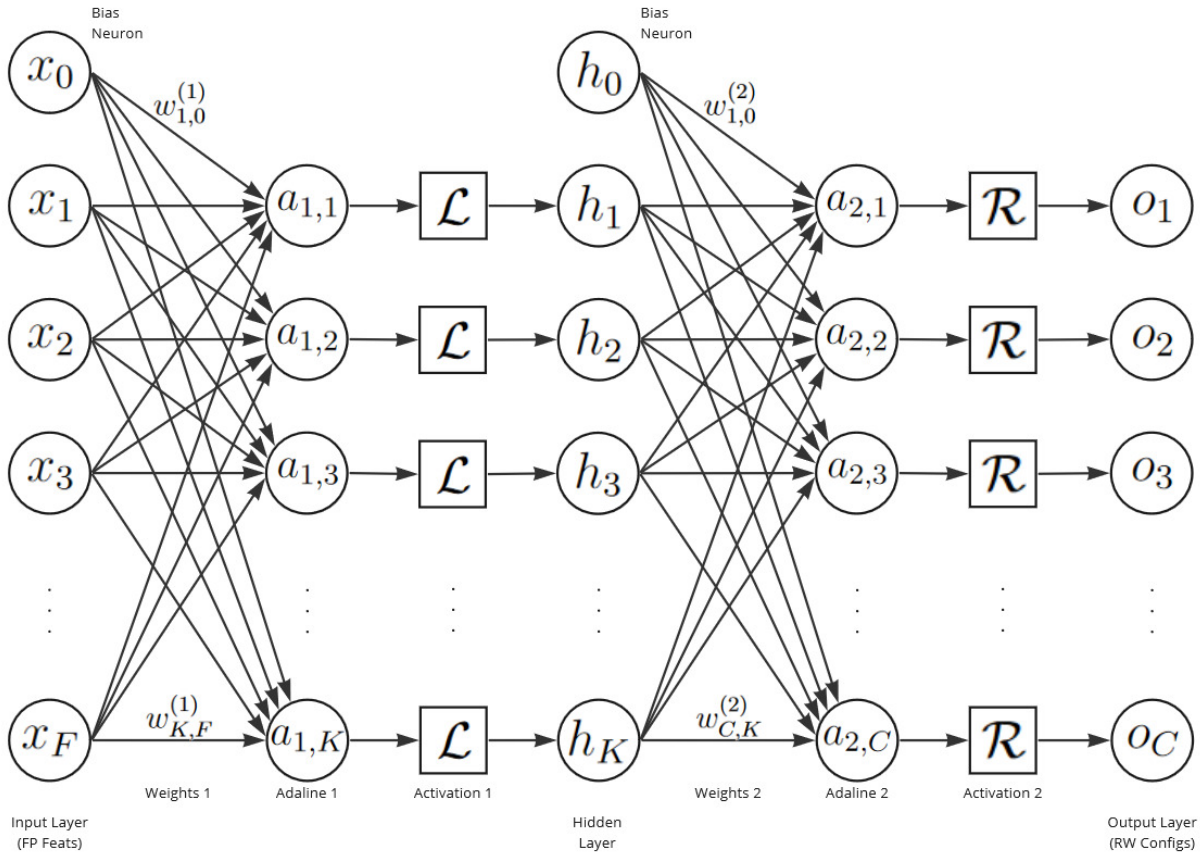

Figure 4.2: Neural Network for Q-Learning from [48] with Logistic and ReLU Activation

network used a learning rate α (also called step-size) to configure the rate at which the agent adapts its learned weights. The higher the learning rate, the faster the weights are learned, however the risk of overshooting the optimum also increases. The lower the learning rate the higher the accuracy, however it also increases the risk of getting stuck at a local maximum as well as requiring more time.

To handle the tradeoff between exploitation and exploration, the agent followed an ϵ-greedy strategy. With a probability of $\epsilon \in [0, 1]$, the agent selected a random action. Over the course of the training, the epsilon was decreased with a decay rate δ to have higher exploration in the beginning and higher exploitation towards the end of the training. Furthermore, a discount factor $\gamma \in [0, 1]$ was used to balance the weights of future rewards versus more immediate rewards.

Chapter 5

System Architecture and Implementation

This chapter introduces the implementation and architecture of the system and scenario described in the previous chapter. It starts with the AD module in Section 5.1, including a comparison of different algorithms and their performances. Subsequently, the different RL components are presented. In Section 5.2, the environment of the RL model is outlined with a detailed description of the feature extraction from the syscall data. Then, in Section 5.3, the implementation of the RL agent including the neural network is presented. Next, Section 5.4 gives insight into the reward function used, as well as other reward functions which were explored during the process of this work. Finally, the chapter concludes with the description of the additional benign behaviors in Section 5.5 and the experiment setup in Section 5.6.

5.1 Anomaly Detection

In order to reliably detect anomalous and benign behavior of the client device, AD techniques were used. In the scenario of this work, there were conflicting objectives regarding the AD: on the one hand, from a defensive point of view, the ransomware's activities should be reliably detected in order to have a robust defensive system. On the other hand however, the ransomware is trying to find configurations which are not being detected, thus an AD system not being able to detect certain malicious behavior would be favoured. To this end, different AD algorithms were tested with different hyperparameter settings. Namely, the following algorithms were evaluated:

- Isolation Forest (IF)
- Local Outlier Factor (LOF)
- One-Class Support Vector Machine (OCSVM)

Table 5.1 shows the True Negative Rate (TNR) and the False Negative Rate (FNR) respectively of different AD algorithms with different hyperparameter settings. Being

classified as normal was considered a negative, which is why for the normal behavior dataset, the values in Table 5.1 correspond to the TNR, while for columns C0 to C5 they show the FNR. OCSVM classifiers were also evaluated with different kernels and hyperparameter settings, but appeared to not be able to capture the differences between normal and infected behavior. Finally, it was decided to use an IF classifier with a contamination factor of 0.03, as it was able to correctly classify normal behavior, the infected control groups C1 and C2, as well as showing detection rates reflecting the settings in the respective configurations. Furthermore, IF classifiers appeared to have a faster training time and detection time on new samples compared to LOF classifiers.

Table 5.1: Performance Comparison of Different AD Algorithms (TNR for Normal, FNR for C0-C5, in Percentage)

Algorithm	**Hyperparameters**	**Normal**	**C0**	**C1**	**C2**	**C3**	**C4**	**C5**
IF	contamination=0.05	93.99	83.24	0.21	0.24	8.23	55.45	13.53
IF	contamination=0.04	95.05	86.70	1.13	0.66	9.62	58.82	15.04
IF	contamination=0.03	96.19	89.78	2.18	1.74	11.79	62.24	18.04
IF	contamination=0.02	97.65	94.14	4.7	4.59	17.5	70.32	25.81
LOF	n_neighbors=1500 contamination=0.05	94.89	81.79	4.81	3.45	17.18	64.31	31.12
LOF	n_neighbors=1500 contamination=0.04	95.37	84.34	5.17	3.77	17.98	66.76	34.42
LOF	n_neighbors=1500 contamination=0.03	96.67	87.14	5.39	4.25	19.41	70.34	38.33

5.2 RL Environment

As discussed in Section 2.3.1, a central part of RL is that the learning agent is able to collect information about the state of the environment. In this scenario, the agent used two pieces of information about the environment: *(i)* the output of the AD module, and *(ii)* the encryption rate reported by the client device.

For the AD module, the syscall data was used. The raw data collected from the IoT device was stored in `.csv` files on the C&C server in the format displayed in Listing 5.1. In order for a RL agent being able to use data as state information from its environment, the AD module has to classify the state information into normal or anomalous. As the syscall data was in text form, there was a need for feature extraction using techniques from the field of NLP.

This work used a Bag-of-Word (BoW) representation of the `syscall` column of the raw data (cf. Listing 5.1), containing the name of the syscall used. Owing to the fact that neither the order of the syscalls used nor the temporal aspects are of importance in this scenario, the `timestamp` as well as the `time_cost` columns were left out. The `pid` column was used during the development and data collection to get insights into what processes were running during the experiment, however it was not used for feature extraction purposes.

Listing 5.1 Example of Raw Syscall Data Collected from the Client Device

```
pid,timestamp,syscall,time_cost
es_sensor/1076,223893.750,ioctl,0.011
es_sensor/1076,223893.778,timerfd_settime,0.014
es_sensor/1076,223893.802,ioctl,0.027
es_sensor/1076,223893.841,poll,0.740
systemd-journa/104,223894.581,getpid,0.009
systemd-journa/104,223894.581,ftruncate64,0.054
systemd-journa/104,223894.652,getpid,0.009
systemd-journa/104,223894.673,getpid,0.009
systemd-journa/104,223894.692,getpid,0.009
systemd-journa/104,223894.716,timerfd_settime,0.014
systemd-journa/104,223894.741,getpid,0.009
es_sensor/1076,223893.841,poll,2.027
systemd-journa/104,223894.761,epoll_wait,1.133
es_sensor/1076,223895.894,ioctl,0.018
es_sensor/1076,223895.924,timerfd_settime,0.013
es_sensor/1076,223895.947,ioctl,0.011
es_sensor/1076,223895.981,timerfd_settime,0.014
es_sensor/1076,223896.006,ioctl,0.033
es_sensor/1076,223896.052,poll,2.073
es_sensor/1076,223898.140,ioctl,0.016
es_sensor/1076,223898.177,timerfd_settime,0.014
es_sensor/1076,223898.201,ioctl,0.011
es_sensor/1076,223898.232,timerfd_settime,0.014
es_sensor/1076,223898.256,ioctl,0.031
es_sensor/1076,223898.301,poll,0.010
SCTP,223889.182,timer/766,(10.083
web-ui/433,223898.312,poll,0.969
es_sensor/1076,223898.301,poll,2.078
SCTP,223899.280,timer/766,1.113
es_sensor/1076,223900.393,ioctl,0.016
es_sensor/1076,223900.431,timerfd_settime,0.013
```

Using the BoW of syscalls, the individual occurrence frequency of each token (*i.e.*, syscall) was used as features. This was achieved using a `CountVectorizer` of the scikit-learn Python library [49]. During the feature extraction, this vectorizer was fitted to the normal dataset, such that the vectorizer built a vocabulary of the occurring words. This vocabulary then determined the number of features extracted. The dataset then had the structure of a matrix with one row per fingerprint and one entry per token giving the number of occurrences. Using this dataset, the AD module could then be trained. If new data had to be classified, the dataset had to be transformed using the same fitted instance of the `CountVectorizer` class, ensuring that the same vocabulary was used and the number of resulting features were thus the same.

Furthermore, the encryption rate was used as state information of the environment. The encryption rate was reported by the client device, together with the resource usage fingerprint. Together with the syscall frequency features, the encryption rate was used as information about the state of the environment in the reward signal computation and the selection process of the agent.

5.3 RL Agent

As outlined in Section 4.3.4, the agent's task was to learn the best ransomware configuration to select based on its interactions with the environment. In order to learn the Q function, a simple three-layer neural network was used. The implementation of the neural network is depicted in Listing 5.2 and was adopted from [1]. It performs a forward pass through the neural network in order to predict the next action to select based on the current state of the environment. Afterwards, the agent performs a backward pass (*i.e.*, backpropagation) in order to update the weights of the model accordingly.

As described in [1], the used neural network encountered the so-called dying ReLU problem when using a ReLU activation function. This is an issue where the learned Q-values become zero and "die out". This problem was resolved by using a SiLU activation function, which in contrast to ReLU allows for negative values. Thus, to avoid this issue from the beginning, the SiLU activation function was used in combination with a Logistic activation function, both of which are depicted in Figure 5.1.

5.4 RL Reward

The main objective of a reward function is to give high rewards to a state which is deemed good for the agent, and penalize bad states. To this end, in a first prototype, a simple reward function was developed giving rewards based on the outcome of the AD. If the selected action of the agent remained undetected, a fixed reward of +20 was given, whereas if it was detected, a negative reward of -20 was given. If full encryption was reached, a bonus reward of +50 was given. Table 5.2 shows the expected rewards for each configuration based on the detection rates of the AD.

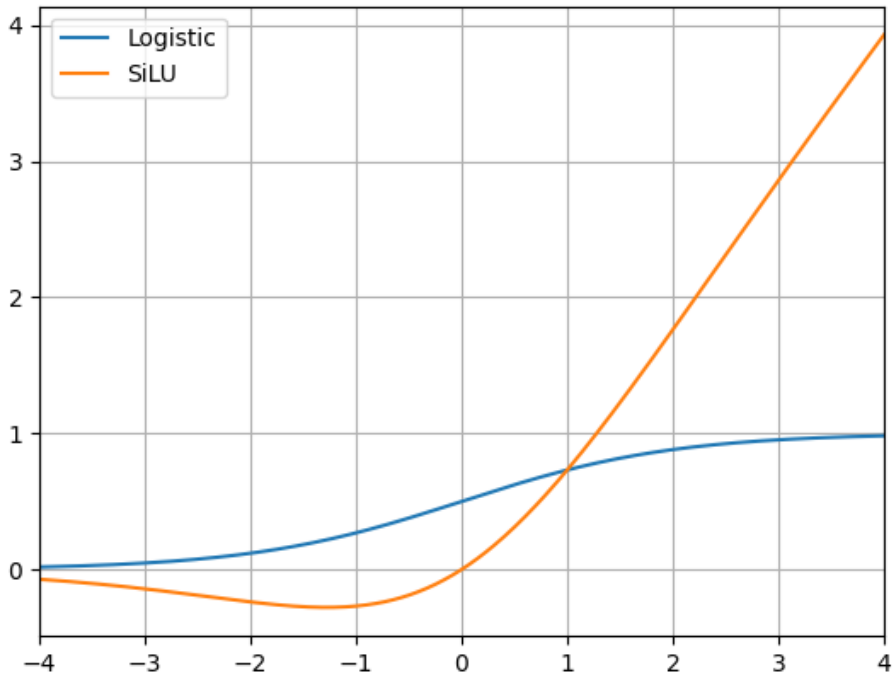

Figure 5.1: Activation Functions Used in Neural Network

Table 5.2: Expected Rewards of Simple Reward Computation

Configuration	**Expected Average Rewards**
Config 0	$0.8978 * 20 + (1 - 0.8978) * (-20) = 15.912$
Config 1	$0.0218 * 20 + (1 - 0.0218) * (-20) = -19.128$
Config 2	$0.0174 * 20 + (1 - 0.0174) * (-20) = -19.304$
Config 3	$0.1179 * 20 + (1 - 0.1179) * (-20) = -15.284$
Config 4	$0.6224 * 20 + (1 - 0.6224) * (-20) = 4.896$
Config 5	$0.1804 * 20 + (1 - 0.1804) * (-20) = -12.784$

As can be seen in Table 5.2, Config 0 has the highest expected reward, followed by Config 4. All other configurations have a negative reward, as they are more often classified as anomalous than as normal. Thus, a RL agent using this reward function is expected to learn to select Config 0 as the best action to take.

In reality however, Config 0 would not be a very effective ransomware, as the encryption rate of 16 B/s is very low and encrypting a realistic amount of data would take very long. For this reason, a second reward function was developed which also included the encryption rate in addition to the AD output. The idea was that a higher encryption rate should also yield a higher reward. Several different variants of reward functions were explored as can be seen in Table 5.3. The expected average rewards were calculated using the following formula, where FNR_C is the AD's FNR of the respective configuration, R_h is the reward function output for the hidden case, and R_d the reward function output for

the detected case:

$$R = \frac{FNR_c}{100} * R_h + (1 - \frac{FNR_c}{100}) * R_d \tag{5.1}$$

When looking at the detection rates and encryption rates of the different configurations, it is obvious that Config 4 should be considered the best, as it is classified as normal with 62.24 % while still having an encryption rate of 200 B/s. All other configurations except Config 0 and Config 4 were detected far too easily. The different variants of the reward functions all contain two functions, one for detected actions and one for hidden actions. Both functions include the encryption rate r, to give higher rewards to higher encryption rates. The constants h and d were used to control the reward given for hidden actions and deducted for detected actions respectively. Furthermore, in variants 4-6 a scaling constant s was introduced to the hidden reward function in order to better distinguish between low and medium encryption rates, such as is the case for Config 0 and Config 4. This is visualized in Figure 5.2, where the reward function variants 1 and 5 are compared. It can be seen that variant 5 gives a lower reward for low encryption rates and more strongly rewards higher encryption rates. On the contrary, variant 1 quickly becomes flat and thus differentiates less between high and low encryption rates. Finally, variant 5 was selected for its ability of rewarding undetected behavior as well as penalizing having an encryption rate which is prohibitively low. As a consequence, Config 4 has the highest expected reward of 60.79 and was thus deemed the best configuration, followed by Config 0 with an expected reward of 11.15. The bonus reward for reaching full encryption was significantly increased compared to the simple reward to +1'000, in order to more strongly incentivize a faster encryption.

5.5 Additional Benign Behaviors

As described in Section 4.2.3, the idea of this experiment was to evaluate the performance of the proposed solution in the presence of additional behaviors on the client device which are classified as benign. For this, two additional behaviors were evaluated: *(i)* a data compression behavior and *(ii)* a libraries installation behavior. The AD module was then trained using collected fingerprints of these behaviors in combination with the normal sensor behavior and then evaluated. In the following, the implementation of the benign behaviors is presented.

Both behaviors were implemented as Bash scripts which were executed in parallel to the data collection process. To run the behavior, the following command could be used as an example:

```
nohup ./benign_behaviors/compression.sh &
```

This command is using `nohup`, a utility to execute commands immune to hangups, *i.e.*, the behavior kept being executed even if the Secure Shell (SSH) connection was terminated [50].

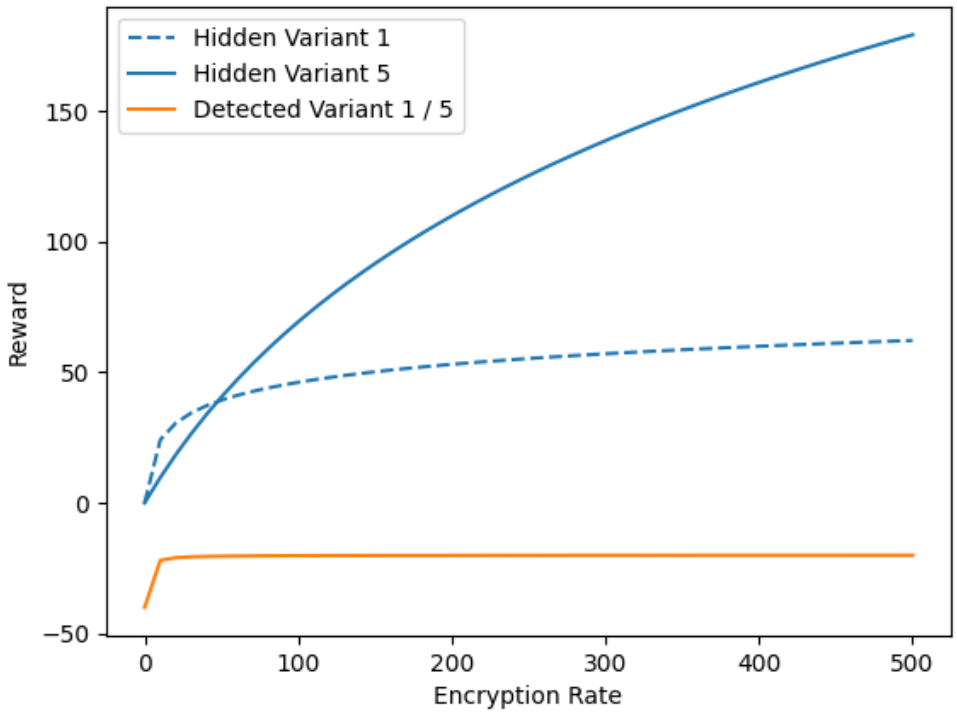

Figure 5.2: Comparison of Performance Reward Functions Variant 1 and Variant 5

5.5.1 Behavior 1: Data Compression

In this behavior, a data corpus is continuously compressed. Afterwards, the compressed files are removed and the loop starts again. For the compression, `tar` (tape archive) is used, which is a command-line utility for compressing and archiving files and directories in linux [51].

The Bash script is listed in Listing 5.3. The `tar` command is used with the options `-czf`, which creates a new archive, uses gzip, and specifies a file name for the archive.

5.5.2 Behavior 2: Libraries Installation

For the second behavior, libraries are being downloaded and installed using `pip`, the package manager for Python [52].

The following five Python libraries were installed for the purpose of this behavior:

- `numpy`
- `pandas`
- `matplotlib`

- `requests`
- `flask`

The decision was made to use five commonly used libraries. With each package requiring other dependencies, the final list of installed packages is shown in Listing 5.5. As can be seen in Listing 5.4, before the packages are installed, a virtual environment is created and activated, such that the behavior does not interfere with the system-wide list of installed packages. Afterwards, the package installation is started using the `requirements.txt` file shown in Listing 5.5. Once the installation terminated, all installed packages are uninstalled and the loop starts again.

5.6 Experiment Setup

In order to run the experiments, behavioral datasets first had to be collected. To this end, the respective behavior was executed on the device, including the ransomware with the corresponding configuration, as well as additional behavior in some cases. The fingerprints were then collected on the client device and periodically sent via REST API to the C&C server, where the data was stored. In this data collection scenario, the C&C server did not run the RL agent, nor did it send any configuration updates. This collection process was performed for the normal behavior datasets over a time frame of 24 hours. Some ransomware configuration datasets were collected over a shorter amount of time as it became clear that less data would also suffice, however at minimum, six hours were collected.

To allow for a faster execution of the experiments and their evaluation, a simulated environment was implemented in which there was no need for a client device. Instead, the operation of the API was simulated to select a random fingerprint from the previously collected fingerprint datasets.

Listing 5.2 Neural Network Implementation Using Log - SiLU Activation Functions

```
import numpy as np

class ModelQLearning(object):
    def __init__(self, learn_rate, num_configs):
        self.learn_rate = learn_rate
        self.allowed_actions = np.asarray(range(num_configs))

    def forward(self, weights1, weights2, bias_weights1,
                bias_weights2, epsilon, inputs):
        inputs = inputs[0].reshape(-1, 1)
        adaline1 = np.dot(weights1.T, inputs) + bias_weights1

        hidden1 = 1 / (1 + np.exp(-adaline1))  # logistic activation

        adaline2 = np.dot(weights2.T, hidden1) + bias_weights2

        q = adaline2 / (1 + np.exp(-adaline2))  # SiLU activation

        possible_a = self.allowed_actions   # epsilon-greedy policy
        q_a = q[possible_a]
        if np.random.random() < epsilon:  # explore randomly
            sel_a = possible_a[np.random.randint(possible_a.size)]
        else:  # exploit greedily
            argmax = np.argmax(q_a)
            sel_a = possible_a[argmax]
        return hidden1, q, sel_a

    def backward(self, q, q_err, hidden, weights1, weights2,
                 bias_weights1, bias_weights2, inputs):
        inputs = inputs[0].reshape(-1, 1)

        delta2 = 1 / (1 + np.exp(-q)) * (1 + q * (1 - (1 /
                (1 + np.exp(-q))))) * q_err  # derivative SiLU
        delta_weights2 = np.outer(hidden, delta2.T)

        delta1 = hidden * (1 - hidden) * np.dot(weights2, delta2)
        delta_weights1 = np.outer(inputs, delta1)

        weights1 += self.learn_rate * delta_weights1
        weights2 += self.learn_rate * delta_weights2
        bias_weights1 += self.learn_rate * delta1
        bias_weights2 += self.learn_rate * delta2
        return weights1, weights2, bias_weights1, bias_weights2
```

Table 5.3: Reward Computation for Performance Rewards

Variant	**Functions**	**Constants**	**Expected Avg. Rewards**
1	Hidden: $10 * \ln(r+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=0 d=20	C0: 23.26 C1: -16.68 C2: -17.33 C3: -10.35 C4: 25.42 C5: -6.91
2	Hidden: $10 * \ln(r+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=0 d=10	C0: 24.35 C1: -6.89 C2: -7.50 C3: -1.51 C4: 29.21 C5: 1.33
3	Hidden: $10 * \ln(r+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=-10 d=0	C0: 16.46 C1: 2.67 C2: 2.15 C3: 6.15 C4: 26.78 C5: 7.76
4	Hidden: $s * \ln(\frac{r}{s}+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=0 d=10 s=10	C0: 7.49 C1: -7.40 C2: -7.90 C3: -4.20 C4: 15.15 C5: -2.74
5	Hidden: $s * \ln(\frac{r}{s}+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=0 d=20 s=100	C0: 11.15 C1: -0.73 C2: -4.42 C3: 3.45 C4: 60.79 C5: 3.35
6	Hidden: $s * \ln(\frac{r}{s}+1) + h$ Detected: $\frac{-d}{max(r,1)} - d$	h=0 d=100 s=100	C0: 2.47 C1: -78.98 C2: -83.03 C3: -67.26 C4: 30.43 C5: -62.55

Listing 5.3 Bash Script for Behavior 1

```
#!/bin/bash -u

path_to_compression_files="./benign_behaviors/compression_files/"
compressed_files="compressed_files.tar.gz"

while true; do

  # compress all files in directory with compression files using tar
  tar -czf "$compressed_files" "$path_to_compression_files"

  if [ $? -ne 0 ]; then
    echo "Compression failed"
  fi

  rm -f $compressed_files

  sleep 0.2

done
```

Listing 5.4 Bash Script for Behavior 2

```
#!/bin/bash

path_to_requirements="./benign_behaviors/requirements.txt"

# create and activate virtual environment
echo "activating venv..."
python3 -m venv ./benign_behaviors/venv
source ./benign_behaviors/venv/bin/activate
echo "done"

while true; do

  # install requirements without caching
  echo "installing requirements..."
  pip install -r $path_to_requirements --no-cache-dir --quiet
  echo "done"

  # uninstall all packages
  echo "uninstalling everything..."
  pip freeze | xargs pip uninstall -y --quiet
  echo "done"

done
```

Listing 5.5 Listed Python Packages Used for Behavior 2 in `requirements.txt` File

```
certifi==2022.5.18
chardet==4.0.0
click==7.1.2
cycler==0.10.0
Flask==1.1.4
idna==2.10
itsdangerous==1.1.0
Jinja2==2.11.3
kiwisolver==1.1.0
MarkupSafe==1.1.1
matplotlib==3.0.3
numpy==1.18.5
pandas==0.25.3
pyparsing==2.4.7
python-dateutil==2.8.2
pytz==2023.3.post1
requests==2.25.1
six==1.16.0
urllib3==1.26.9
Werkzeug==1.0.1
```

Chapter 6

Evaluation

In this chapter, the results and main findings of the experiments performed are presented, compared, and evaluated. Starting with Section 6.1, the performance of the different implemented prototypes is described, including the simple Q-learning prototype, the improved Q-learning prototype, as well as the improved Q-learning prototype with ideal AD. Following, in Section 6.2, the achieved results are compared in detail to the results of experiments performed using resource usage fingerprints from [1]. Finally, in Section 6.3, the results of the experiments using additional benign behaviors are shown and discussed.

6.1 Performance of Prototypes

At first, an experimental prototype (Prototype 11) was implemented in order to show the feasibility of the experiment (*i.e.*, a Proof of Concept (PoC)). The naming and numbering of the prototypes was adopted and continued from [1], thus starting with number 11. Prototype 11 was the first implementation using syscalls and is equivalent to the implementation of Prototype 1 of [1]. The prototype does not implement an intelligent agent yet, the agent merely selects every ransomware configuration once and then terminates. Nevertheless, this prototype was useful to show the feasibility and adaptability of the architecture when used with syscall-based fingerprints. However, the prototype did not produce any results worth evaluating.

In the remainder of this section, the performance results of the subsequent prototypes and prototype configurations are evaluated, namely the simple Q-learning prototype, the improved Q-learning prototype, and finally the improved Q-learning prototype with ideal AD.

6.1.1 Simple Q-Learning Prototype

After the first PoC prototype, the simple Q-learning prototype (Prototype 12) was developed. This was the first prototype implementing the Q-learning techniques described

in Section 2.3.1. It is a progression of Prototype 11, applying the Q-learning algorithm for a single episode over multiple steps using the simple reward computation described in Section 5.4. After the configured number of steps is reached, the execution is terminated and the reward for reaching full encryption is given.

The prototype was evaluated using different settings and values for the following hyperparameters:

- Exploration rate $\epsilon \in [0, 1]$: The agent selects a random action with probability of ϵ
- Learning rate α: Step size parameter for the neural network to learn the Q-function (cf. Section 4.3.4)
- Discount factor $\gamma \in [0, 1]$: Used to handle how much importance is given to rewards that lie far in the future.

Finally, the hyperparameters were set at $\epsilon = 0.2$, $\alpha = 0.0005$, and $\gamma = 0.75$. Using these values, Figure 6.1 and Figure 6.2 show the results of Prototype 12 run over 500 and 1'000 steps respectively. In the plots, the absolute rewards and the Exponential Moving Average (EMA) of the rewards are visualized. As one can see, over time the agent was able to increase the rewards it receives, *i.e.*, it was able to learn which actions to select in order to collect higher rewards. This shows that it is feasible to use syscalls as the basis of a RL agent to learn actions which result in fewer detections by a defensive AD system and thus remain hidden.

In spite of the promising plots, when looking at the accuracy results of the agent trained after 10'000 steps shown in Table 6.1, some issues become visible. First of all, according to the reward function used in this prototype, Configuration 0 had the highest expected rewards and should thus be selected. However, Configuration 4 had the highest accuracy with 83.58 %, while Configuration 0 only achieved 3.44 % after training. One explanation for this is that the agent was merely able to learn which configurations to avoid, rather than finding the most optimal one. When looking at the final Q-values, one can see that Configuration 0 achieved the highest values.

Looking these results, such a prototype using a reward system only incorporating the output of the AD had obvious shortcomings, as was also mentioned in [1]. The fact that the reward system only incentivized not being detected is sub-optimal, since the main goal of a ransomware is encrypting files in a timely manner while not being detected. Thus there was a need for incorporating the encryption rate into the reward system. Moreover, the ϵ parameter should be adjusted using a decay rate parameter, such that the exploration probability is gradually decreased over the course of multiple episodes.

6.1.2 Improved Q-Learning Prototype with Performance Rewards

As discussed in the section above, the simple Q-learning prototype had several shortcomings, therefore, an improved version (Prototype 13) of the simple prototype was developed.

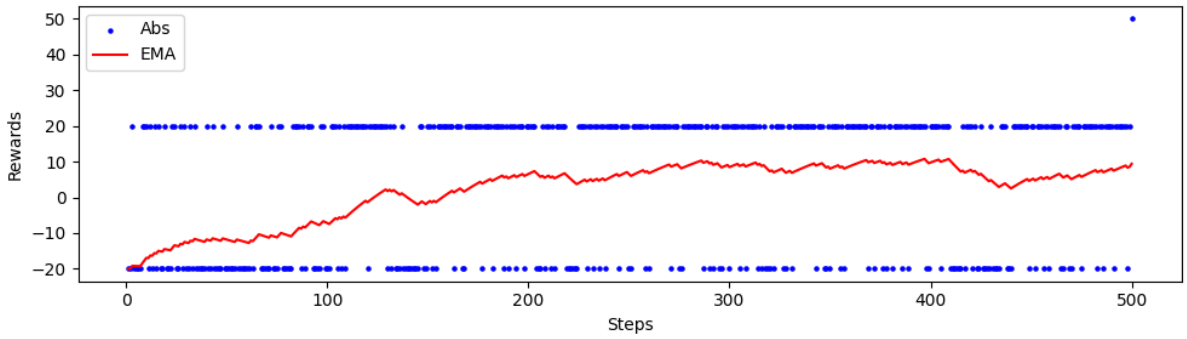

Figure 6.1: Results for Prototype 12 over 500 Steps with $\epsilon = 0.2, \alpha = 0.0005, \gamma = 0.75$

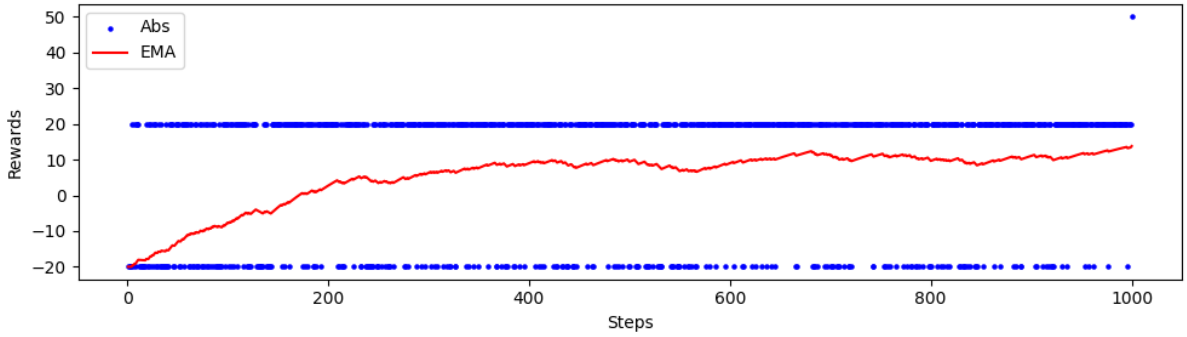

Figure 6.2: Results for Prototype 12 over 1'000 Steps with $\epsilon = 0.2, \alpha = 0.0005, \gamma = 0.75$

Here, the reward function was extended to also incorporate the encryption rate, as described in Section 5.4. With this, the secondary goal of reaching encryption as fast as possible was handled and the different configurations could be differentiated more easily.

Furthermore, Prototype 13 was run over a configurable amount of episodes. In each episode, the encryption process was calculated by using the encryption rate, assuming that one step takes one second. After the artificial corpus was encrypted, the episodes were terminated and the bonus reward was given. A configurable corpus size is used to determine whether the encryption had finished and the final reward should be given. In this case, 2000 bytes was used for the experiments which for instance resulted in 10 steps to full encryption for Configuration 4.

Another improvement made was the introduction of a decay rate for the epsilon parameter, in order to have higher exploration at the beginning and gradually decrease it towards the end. For this, another hyperparameter δ was introduced. The epsilon calculation was done as follows with n denoting the episode number:

$$\epsilon_n = \frac{\epsilon}{1 + \delta * n} \tag{6.1}$$

Figure 6.3 shows the results of running the improved Q-learning prototype (Prototype 13) for 10'000 episodes. Furthermore, Table 6.2 presents the accuracy result of the same prototype. As can be seen in the plots, the agent was able to increase the rewards

Table 6.1: Accuracy for Prototype 12 after 10'000 Episodes with $\epsilon = 0.2, \alpha = 0.0005, \gamma = 0.75$

Configuration	**Initial Accuracy**		**Trained Accuracy**		**Final Q-Values**
	Percentage	# Samples	Percentage	# Samples	
Config 0	03.47 %	297 / 8552	03.44 %	294 / 8552	72.92186305
Config 1	03.64 %	311 / 8552	03.23 %	276 / 8552	5.50346083
Config 2	03.33 %	285 / 8552	03.30 %	282 / 8552	30.36052495
Config 3	83.10 %	7107 / 8552	02.99 %	256 / 8552	38.32739379
Config 4	02.99 %	256 / 8552	83.58 %	7148 / 8552	30.76325477
Config 5	02.99 %	296 / 8552	03.46 %	296 / 8552	13.37773805

it receives, especially in the first approximately 4'000 episodes. When looking at the number of steps taken in each episode, one can see that the average appears to remain almost constant, indicating that there was not really any development throughout the course of the training. Also, the number of steps show high fluctuations. Taking into consideration the detection rates of the AD shown in Table 5.1, it seems that the agent is not able to reach a high number of steps and thus full encryption because the optimal Configuration 4 is still detected with a probability of 37.76 %. This was an issue as the agent would receive negative rewards in more than a third of its decisions, although the most optimal configuration was learned and selected.

In contrast to the previous prototype, this improved prototype was able to learn the best Configuration 4, as is shown in Table 6.2. After 10'000 episodes of training, the agent selected the optimal configuration with an accuracy of 99.61 %. In addition to the high accuracy, the final Q-values also show that Configuration 4 predominated over the others with a value of 156.67, followed by Configuration 0 with a value of 33.36.

Besides the accuracy results presented above, Table 6.3 shows the accuracies of the prototype for the optimal Configuration 4 when trained for different numbers of episodes. As one can see, already after around 50 episodes of training, which corresponds to about 1.5 minutes of training time, the agent reached an accuracy of above 90 %. In less than 10 minutes of training, the agent achieved accuracy levels of above 97 %. These values demonstrate that a system such as the one presented in this work is able to learn a ransomware configuration that evades detection in a short amount of time with promising accuracies.

6.1.3 Improved Q-Learning Prototype with Ideal AD

As the high fluctuations in the AD was identified as a limiting factor in the model's performance, to verify the learning capabilities of the proposed model, an ideal version of the AD was implemented. This ideal AD did not rely on an AD algorithm as before, but manually decided whether an action is detected or not based on prior knowledge. By the use of a configurable set of actions that are not detected, this version computed the

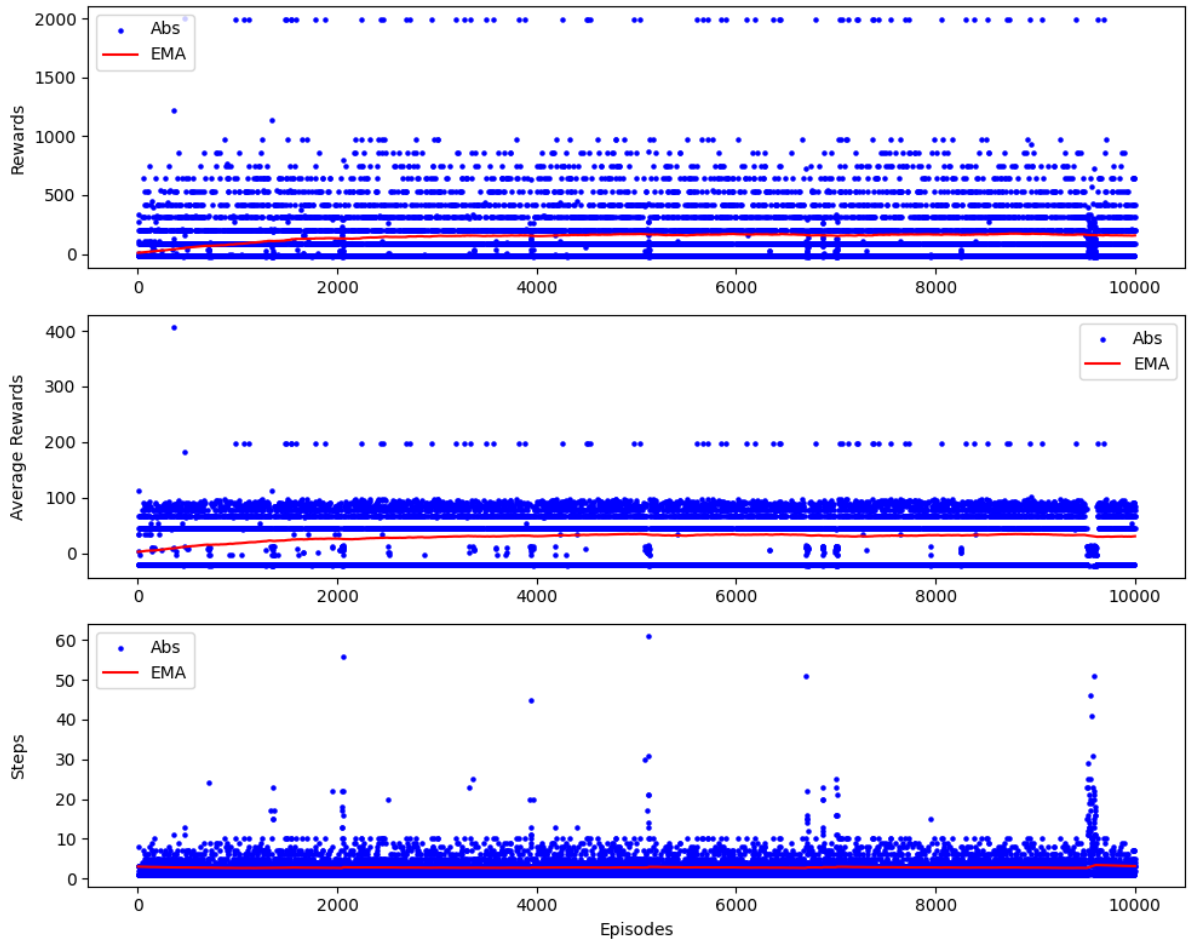

Figure 6.3: Results for Prototype 13 over 10'000 Episodes with $\epsilon = 0.4, \delta = 0.01, \alpha = 0.0005, \gamma = 0.75$

Table 6.2: Accuracy for Prototype 13 after 10'000 Episodes with $\epsilon = 0.4, \delta = 0.01, \alpha = 0.0005, \gamma = 0.75$

Configuration	**Initial Accuracy**		**Trained Accuracy**		**Final Q-Values**
	Percentage	# Samples	Percentage	# Samples	
Config 0	07.32%	626 / 8552	00.07%	6 / 8552	3.33625127e+01
Config 1	06.58%	563 / 8552	00.07%	6 / 8552	-1.10514655e-14
Config 2	65.96%	5641 / 8552	00.05%	4 / 8552	-5.78874988e-37
Config 3	06.50%	556 / 8552	00.12%	10 / 8552	7.38995201e+00
Config 4	06.85%	586 / 8552	99.61%	8519 / 8552	1.56667436e+02
Config 5	06.78%	580 / 8552	00.08%	7 / 8552	-1.71861340e-01

Table 6.3: Accuracies of Selecting Config 4 and Training Times of Improved Q-Learning Prototype

Episodes	**Accuracy (in %)**	**Training Time**
50	90.41 %	1.5 min
100	94.63 %	3.4 min
200	97.32 %	6.9 min
500	98.61 %	17.5 min
1000	99.23 %	35.9 min

reward leading to a consistent reward given for each action. Considering the output of the AD model, Configurations 0 and 4 were classified as hidden, while all other configurations would be detected by the ideal AD. Table 6.4 shows the expected rewards of each configuration using the reward function described in Section 5.4.

Table 6.4: Expected Rewards Using Ideal AD

Configuration	**Detection Rate**	**Expected Reward**
Config 0	0 %	14.84
Config 1	100 %	-20.00
Config 2	100 %	-20.00
Config 3	100 %	-20.04
Config 4	0 %	109.86
Config 5	100 %	-20.10

The results of running the prototype with the ideal AD configuration are shown in Figure 6.4. The other hyperparameters were equal to those in the experiment described in Section 6.1.2. It is visible that the EMA of all plots clearly converges towards an optimal action taken by the agent. When looking at the expected rewards in Table 6.4, the expected optimal path is selecting Configuration 4 for ten steps until full encryption. This

exact pattern is recognizable in Figure 6.4 where the rewards converge towards approximately 2'000 (reward of configuration 4 for ten steps plus the bonus reward for reaching full encryption of 1'000). Also, the number of steps converges towards ten and thus the average reward converges towards approximately 200.

In addition to the plots, Table 6.5 presents the accuracy results for the ideal AD prototype. Configuration 4 has the highest accuracy by far with 99.71 %, confirming the result that the agent learned the optimal configuration.

Due to these results, the conclusion can be made that the performance of the RL agent heavily relies on the performance of the AD model used. When used with an ideal AD, the agent exhibited a much more ideal and consistent learning performance compared to when used with the real AD model.

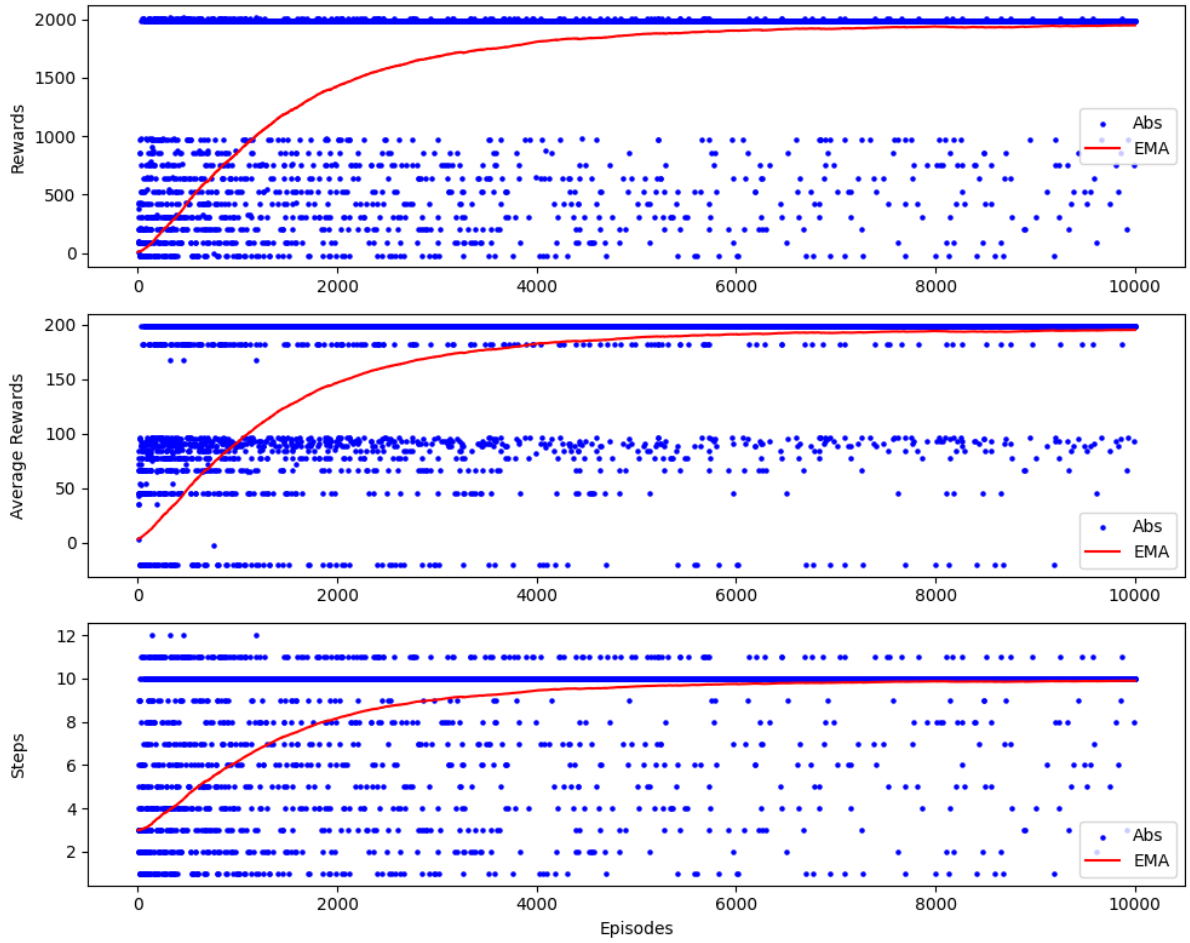

Figure 6.4: Results for Prototype 13 with Ideal AD over 10'000 Episodes with $\epsilon = 0.4, \delta = 0.01, \alpha = 0.0005, \gamma = 0.75$

Table 6.5: Accuracy for Prototype 13 with Ideal AD after 10'000 Episodes with $\epsilon = 0.4, \delta = 0.01, \alpha = 0.0005, \gamma = 0.75$

Configuration	**Initial Accuracy**		**Trained Accuracy**		**Final Q-Values**
	Percentage	# Samples	Percentage	# Samples	
Config 0	07.32%	626 / 8552	00.05%	4 / 8552	4.76353773e+002
Config 1	06.58%	563 / 8552	00.06%	5 / 8552	-4.64321204e-155
Config 2	65.96%	5641 / 8552	00.04%	3 / 8552	-3.83546470e-137
Config 3	06.50%	556 / 8552	00.13%	11 / 8552	-1.24895980e-179
Config 4	06.85%	586 / 8552	99.71%	8527 / 8552	5.86183749e+002
Config 5	06.78%	580 / 8552	00.02%	2 / 8552	-1.07854263e-164

6.2 Comparison with Resource Usage Fingerprints

This section is dedicated to comparing the results presented in the previous section to those obtained in [1], [48], where device behavioral fingerprinting based on resource usage was performed. Key findings and performance metrics will be compared in the following.

To begin with, because the underlying source of behavior data is different, it is natural to expect differences in the detection rates of the different behaviors. Table 6.6 presents the comparison between the TNR and FNR respectively of the previous work and this work for the different behaviors. Although there are some differences, it can be seen that both approaches also share similarities. Unsurprisingly, both classifiers recognize the normal behavior largely as normal, as they were trained with this behavior. Furthermore, both classifiers seemingly do not detect Configuration 0 well, as both show a FNR of around 90 %. The second highest FNR is reached by Configuration 4 in both cases. Moreover, the classifiers seem to consistently detect the control group Configuration 1 and Configuration 2. On the other hand, there are some striking differences when looking at the detection rates of Configuration 3 and Configuration 5. Using fingerprints based on resource usage, the AD classifies Configuration 3 as normal with 80.18 % while with syscall-based fingerprints, only 11.79 % is classified as normal. Similarly, Configuration 5 is classified as normal with 77.38 % for the resource-based case and 18.04 % for the syscall-based case. The encryption rate cannot be the only explanation for this data, as Configuration 4 and 5 both have the same encryption rate but very different detection rates. It seems that the detection rates for resource-based fingerprints correlate more with the encryption rate of the configurations, while the detection rates of syscall-based fingerprints seem to show a stronger influence from the encryption burst settings. One possible explanation for this is that syscalls-based fingerprints capture data on a much lower level compared to resource usage data, and thus are able to capture more differences between the behaviors.

Based on the AD results, the expected rewards were calculated and the optimal configuration was determined. Using the simple reward function which only considers the output of the AD module, Configuration 0 was considered the best for both resource-based fingerprints as well as for syscall-based fingerprints [48]. However, when using a

Table 6.6: TNR and FNR Comparison of Resource-based and Syscall-based Fingerprints

Behavior	**Resource-based FP** [1]	**Syscall-based FP**
Normal	88.94 %	96.19 %
Config 0	91.62 %	89.78 %
Config 1	0.62 %	2.18 %
Config 2	0.21 %	1.74 %
Config 3	80.18 %	11.79 %
Config 4	82.05 %	62.24 %
Config 5	77.38 %	18.04 %

reward function that also considers the encryption rate of the configurations, there is an interesting difference. For resource-based fingerprints, Configuration 3 is considered the most optimal configuration, as it has an encryption rate of 500 B/s while being classified as normal in 80.18 % of cases [48]. Using syscall-based fingerprints however, Configuration 4 is considered the most optimal. Essentially, because more fingerprints are being detected by the AD when using syscall-based data, the ransomware cannot use as fast of a configuration compared to resource-based data.

Table 6.7: Expected Rewards Comparison of Resource-based and Syscall-based Reward Functions

Configuration	**Resource-based Reward** [48]	**Syscall-based Reward**
Config 0	24.18	11.15
Config 1	-19.05	-0.73
Config 2	-19.68	-4.42
Config 3	45.87	3.45
Config 4	39.91	60.79
Config 5	36.49	3.35

The expected rewards of the two approaches are shown in Table 6.7. Since different reward functions were used, the absolute values of the rewards cannot be directly compared with each other. Nevertheless, the expected rewards can be compared in relation to the other configurations of each approach. For instance, it can be seen that Configuration 4 in the syscall-based (*i.e.*, the optimal configuration) has a much higher reward than any other configuration. In contrast, although the optimal configuration in the resource-based case (*i.e.*, Configuration 3) of course also has the highest expected reward, the expected reward is much more similar to other sub-optimal configurations such as Configuration 4 or Configuration 5.

Additionally, the RL agent's performance metrics can be compared. Table 6.8 shows the accuracy values achieved by the resource-usage-based prototype and the training times for different numbers of episodes. When comparing these values to the ones reported in Table 6.3, some performance improvements can be noticed. The prototype presented

in this work achieved an accuracy of above 90 % in just 50 episodes or 1.5 minutes of training. After 100 episodes, an accuracy of 94.63 % was achieved, while the prototype reported in [1] achieved an accuracy of 91.43 %. However, the resource-based prototype reached 100 episodes in 2 minutes, while this work's prototype took 3.4 minutes for 100 episodes. Although the agent proposed in this work takes more time to train to a certain number of episodes, it appears to be learning faster and thus reaches higher accuracy scores in a shorter amount of time.

A possible explanation for the faster learning might lie in the reward functions. As discussed above and shown in Table 6.7, this work used a reward function in which the optimal configuration achieved distinctly higher rewards than any other sub-optimal configuration. On the contrary, the reward function used in [1] did not differentiate as strongly between the rewards given to the optimal configuration and other configurations. This reward function design might be the cause of the differences in learning speed and number of episodes necessary for the agent to achieve a high accuracy.

Table 6.8: Accuracies of Selecting the Optimal Configuration and Training Times of Resource-based Prototype [1]

Episodes	**Accuracy (in %)**	**Training Time**
100	91.43 %	2.0 min
200	94.86 %	5.1 min
300	96.32 %	6.5 min
400	96.21 %	8.1 min
1000	98.61 %	23.9 min
2000	99.07 %	66.4 min
5000	99.71 %	172.2 min

6.3 Evaluation of Additional Benign Behaviors

After successfully showing the feasibility of implementing a RL agent that uses syscall-based behavioral fingerprinting, to test the adaptability of the agent, further developments were necessary. Thus, as described in Section 4.2.3, the scenario was extended to include additional normal, benign behaviors on the client device. Using this extended scenario, the RL model was tested and evaluated, adapting the AD as well as the reward function to the changed behavior monitored on the device. The remainder of this section presents the results and findings of these experiments.

For all normal behaviors described in Section 5.5, a dataset of behavioral data based on syscalls was collected. Furthermore, for each ransomware configuration a dataset was collected with the additional behaviors executed in parallel to the encryption of the ransomware.

Consequently, there were three datasets of behaviors classified as normal:

1. No Additional Behavior (NAB), *i.e.*, the normal behavior of the Electrosense sensor
2. Additional Behavior 1 (AB1), *i.e.*, the data compression behavior
3. Additional Behavior 2 (AB2), *i.e.*, the packages installation behavior

6.3.1 AD with Additional Benign Behaviors

The same IF AD classifier used in the previous experiments was then trained using these three datasets. As the training data consisted of three different concatenated datasets and not one uniform dataset as in the cases before, the training data had to be shuffled. This prevented the classifier from learning any ordering properties of the dataset. However, it was observed that the detection rates showed a rather high variability depending on the shuffling of the training dataset. To counter this, for the AD evaluation, the process was repeated 100 times and the mean was taken. The FNRs of this classifier are shown in Table 6.9.

Table 6.9: FNRs for IF AD Trained with All Three Benign Behaviors

Config	**No Additional Behavior**	**Additional Behavior 1**	**Additional Behavior 2**
Config 0	98.92 %	96.96 %	90.70 %
Config 1	72.10 %	15.77 %	76.61 %
Config 2	70.82 %	28.05 %	71.41 %
Config 3	88.94 %	46.08 %	80.00 %
Config 4	97.39 %	88.09 %	89.10 %
Config 5	94.07 %	71.40 %	75.83 %

These values show some surprising behavior of the AD. Firstly, when looking at the FNRs of the ransomware without any additional behavior, one can see that the AD struggled to detect the ransomware and for most configurations classified it as benign. Secondly, the FNRs of AB1 shows much lower values than NAB however still higher than when the classifier was not trained with additional benign behaviors. Finally, AB2 shows very high FNRs, similar to NAB. This is very surprising, especially that the ransomware without additional behavior (*i.e.*, NAB) was not detected while AB1 apparently was more easily detectable. Additionally, the differences between AB1 and AB2 are remarkable.

Due to the results of the AD, it was suspected that the AB2 dataset introduced a significant amount of noise to the classification. In order to test this hypothesis, the IF classifier was trained with NAB and AB1, leaving out the AB1 dataset. The results of this evaluation is presented in Table 6.10.

Evidently, compared to when the classifier is also trained with AB2, it was able to detect the ransomware's behavior much better here. Especially when looking at the FNRs of NAB, the differences are striking. Configurations 1 and 2 (*i.e.*, the control group with unlimited encryption rate) were both classified as normal with over 70 % when trained

Table 6.10: FNRs for IF AD Trained with No Additional Behavior and Additional Behavior 1

Config	No Additional Behavior	Additional Behavior 1	Additional Behavior 2
Config 0	93.25 %	96.53 %	64.17 %
Config 1	24.01 %	13.73 %	47.00 %
Config 2	22.40 %	21.12 %	43.77 %
Config 3	50.13 %	33.44 %	45.57 %
Config 4	89.01 %	84.99 %	61.98 %
Config 5	77.23 %	63.38 %	42.55 %

with all three behaviors, while they were classified as normal in less than 25 % of cases when AB2 is left out. Unsurprisingly, the classifier was not able to correctly detect the AB2 dataset, as it had not been trained with it. These results support the hypothesis that the AB2 dataset introduced too much noise and thus deteriorated the AD capabilities to detect the ransomware's behavior. As for the reasons why AB2 introduced too much noise while apparently AB1 did not, only speculation can be made without further investigation. As a consequence of this added noise of AB2, only NAB and AB1 were considered going forward.

Moreover, with the new datasets also other AD algorithms were explored other than the IF classifier used for the previous experiments. The same classifiers as in Section 5.1 were explored, namely LOF and OCSVM in addition to IF. It was found that with the additional normal behaviors, a LOF classifier actually works better than the IF classifier. The FNRs of the LOF classifier when trained with NAB and AB1 is shown in Table 6.11.

Table 6.11: FNRs for LOF AD Trained with No Additional Behavior and Additional Behavior 1

Config	No Additional Behavior	Additional Behavior 1
Config 0	98.33 %	89.87 %
Config 1	8.93 %	8.10 %
Config 2	8.72 %	14.14 %
Config 3	32.93 %	19.84 %
Config 4	86.83 %	73.01 %
Config 5	71.32 %	43.69 %

As can be observed, the LOF classifier was much better able to detect the ransomware for most configurations. Especially the control group (*i.e.*, Configurations 1 and 2) were classified as normal in below 9 % of cases. Configuration 0 was still very hard to detect. Interestingly, if the ransomware executed the benign behavior in parallel to the encryption process, the ransomware was actually more detected than if the ransomware does not execute any additional behaviors.

However, it is apparent that the ransomware's behavior was less detected in the presence

Table 6.12: FNRs Comparing the AD Trained only with NAB vs. Trained with NAB and AB1

Config	AD Trained with NAB	AD Trained with NAB and AB1
Config 0	89.78 %	98.33 %
Config 1	2.18 %	8.93 %
Config 2	1.74 %	8.72 %
Config 3	11.79 %	32.93 %
Config 4	62.24 %	86.83 %
Config 5	18.04 %	71.32 %

of additional benign behaviors. This is highlighted in Table 6.12, since the FNRs of all configurations was higher when the AD was trained with additional behaviors. For instance, the FNRs of the control group increased from approximately 2 % to almost 9 %, or Configuration 4 increased from 62.24 % to 86.83 %. Remarkably, Configuration 0 increased from 89.78 % to 98.33 %, making Configuration 0 almost undetectable. It can thus be concluded that the presence of additional benign behaviors drastically improved the ransomware's ability to remain undetected.

Devices with a more uniform behavior pattern are therefore easier to protect in such a scenario. This often is the case for IoT devices, as oftentimes, these are single-purpose devices such as sensors. If such devices are attacked, the behavior is likely to change significantly and this could be detected by an AD based for instance on syscall monitoring. On the other hand, if a device has multiple different purposes such as a PC or smartphone, the normal behavior is much more diverse. A defensive AD system would likely struggle more to detect malicious behaviors in multi-purpose devices compared to devices with a uniform behavior.

6.3.2 Reward Function with Additional Benign Behaviors

Based on the results of the AD presented in the section above, the expected rewards of the reward function could be determined. Recall that the following reward function was used with the constants $s = 100$, $d = 20$, and $h = 0$:

$$\text{Hidden: } s * \ln(\frac{r}{s} + 1) + h \tag{6.2}$$

$$\text{Detected: } \frac{-d}{\max(r, 1)} - d \tag{6.3}$$

Using this reward function and the AD results of the classifier trained with NAB and AB1, the expected rewards are shown in Table 6.13. As one can see, Configuration 4 was still the most optimal here. However, there were some issues with this reward function. Configuration 0 had the lowest expected reward, although it was not detected in more

Table 6.13: Expected Rewards with Additional Behaviors

Config	Expected Rewards
Config 0	14.24
Config 1	58.95
Config 2	58.07
Config 3	45.56
Config 4	92.75
Config 5	72.59

than 98 % of cases. Moreover, the unlimited Configurations 1 and 2 achieved medium to high expected rewards, while still being detected in more than 90 % of cases. Even though Configuration 0 is very slow, in light of the very high evasion rate it could be argued that this should actually be considered the most optimal configuration. Therefore, for these AD results a different reward function had to be used to better represent the effectiveness of the configurations.

After some experimentation, the following revised reward function was used with $h = 50$ and $d = 50$:

$$\text{Hidden: } \ln(r+1) + h \tag{6.4}$$

$$\text{Detected: } \frac{-d}{\max(r,1)} - d \tag{6.5}$$

With this revised reward function, the expected rewards turned out as shown in Table 6.14. Additionally, the revised reward as well as the previous reward functions are depicted in Figure 6.5. As can be observed, the previous reward function had a much steeper curve for the hidden cases, thus higher rates were more heavily rewarded. Furthermore, the penalty for being detected was larger in the revised case. Combining the hidden and the detected case, this reward function put more weight into not being detected, and less into achieving a high encryption rate. Thus, Configuration 0 was the most optimal choice using the revised reward function.

6.3.3 RL Agent Performance with Additional Benign Behaviors

Using the adjusted AD using the LOF classifier as well as the revised reward function, the RL agent could now be evaluated using the additional benign behavior. For this, the same Q-learning prototype as before was used, except for the mentioned adjustments which could be set using options in the configuration file. The remainder of this section presents the results and interpretation of these experiments.

It was quickly discovered that the learning process of this RL agent took more time to reach a certain number of episodes compared to previous experiments. There are three

Table 6.14: Expected Rewards for Revised Reward Function with Additional Behaviors

Config	Expected Rewards
Config 0	$0.9833 * 52.83 + (1 - 0.9833) * -53.13 = 51.06$
Config 1	$0.0893 * 63.25 + (1 - 0.0893) * -50.00 = -39.89$
Config 2	$0.0872 * 63.36 + (1 - 0.0872) * -50.00 = -40.12$
Config 3	$0.3293 * 56.22 + (1 - 0.3293) * -50.10 = -15.09$
Config 4	$0.8683 * 55.30 + (1 - 0.8683) * -50.25 = 41.40$
Config 5	$0.7132 * 55.30 + (1 - 0.7132) * -50.25 = 25.03$

possible explanations for this phenomenon. Firstly, since the classifier was trained with the additional behavior, the feature space was much larger. Before, the `CountVectorizer` extracted 1'071 features from the training data, while with the additional behavior, the number of features reached 2'307, *i.e.*, more than twice as much. Secondly, as Configuration 0 was the optimal choice and the AD usually does not detect this, the number of steps per episode was much higher. Before, the optimal configuration took 10 steps until full encryption of the simulated data corpus of 2'000 bytes, while in this case, 125 steps were needed. The more steps were taken per episode, the longer was the duration of the episode. Thirdly, the LOF classifier was, although it performed better on the additional behavior dataset regarding the detection rates, slightly slower compared to the IF classifier.

Figure 6.6 depicts the result plots of the RL agent trained over 2'000 episodes using the AD trained with NAB and AB1. This training run for 2'000 episodes took 506 minutes, *i.e.*, more than eight hours. Compared to the agent's training runs before, without the additional behavior, this was considerably higher. When looking at the plots, it can be observed that the agent, like before, was able to increase the rewards it received, thus it was able to show learning capabilities. Also, the optimal reward of approximately 7'550 ($= 124 * 52.83 + 1'000$) is more and more often achieved, when selecting Configuration 0 until full encryption, *i.e.*, for 125 steps. The number of steps taken in each episode was also rising to an average of around 40 steps. Compared to the previous experiments this was much higher, as the number of steps without the additional behavior seldomly was higher than 5 steps. This fact also serves as an explanation of the much longer training time needed in this case.

After around 1'800 episodes, one can observe a drop in the rewards achieved, as well as in the number of steps taken. A possible explanation for this behavior is that the agent may have learned a sub-optimal configuration due to some fluctuations in the AD. If the agent chose the optimal action however still was detected multiple times in a row, it may have influenced the agent to choose a sub-optimal configuration instead. In this case, it looks like Configuration 4 was learned, as the rewards achieved correspond to choosing Configuration 4 ten times until full encryption. Supporting this hypothesis, the number of steps are grouped around ten in this part of the training process.

Table 6.15 presents the accuracy evaluation of this experiment of the RL agent tested with additional benign behavior over 2'000 episodes. The optimal Configuration 0 achieves an

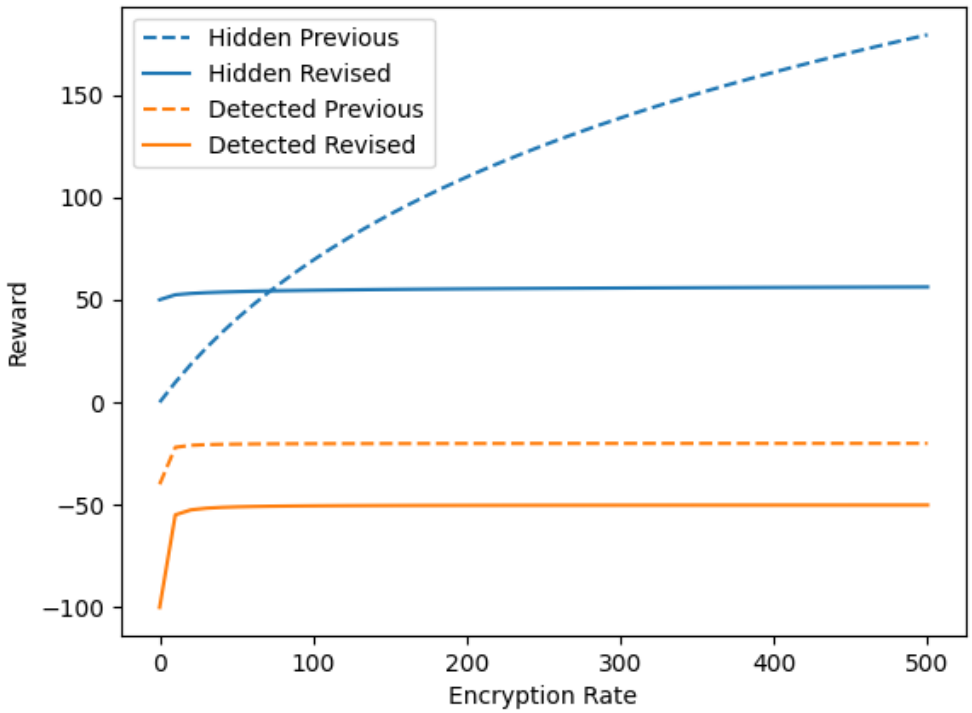

Figure 6.5: Revised Reward Function in Comparison with Previous Reward Function

accuracy of 99.63 %, which substantiates the RL agent's learning capabilities. Furthermore, Configuration 0 has the highest Q-values after training.

Additionally, the experiment was then performed with 5'000 episodes, to test whether more training would further improve the results. The resulting plots are shown in Figure 6.7. The training of the RL agent for 5'000 episodes took 1'385.9 min, *i.e.*, more than 23 h. Overall, the plots seem to show a consistent behavior compared to the agent trained for 2000 episodes. The rewards, the average rewards, as well as the number of steps are rising fast in the beginning and then slowly seem to converge to some value.

However, certain periods during the training are visible where the agent apparently changed its course and diverted from the optimal configuration. One such period is clearly visible at around 2'500 episodes where the rewards and the number of steps take a sudden drop. As described above, the suspected reason for this is the imperfect AD resulting in optimal actions still being detected. The agent then updated its learned weights, possibly ending up giving a sub-optimal configuration too much weight.

This issue is further exposed when looking at the accuracy evaluation of the RL agent trained over 5'000 episodes shown in Table 6.16. The values show that after the training, the agent selected Configuration 4 with an accuracy of 99.82 % while the supposedly optimal Configuration 0 only achieved an accuracy evaluation value of 0.07 %. Going back to the plots in Figure 6.7, looking closely one can see that at the very end of the training the rewards as well as the number of steps take a small drop. Apparently, in this very moment at the end of the 5'000 episodes of training, the agent switched to

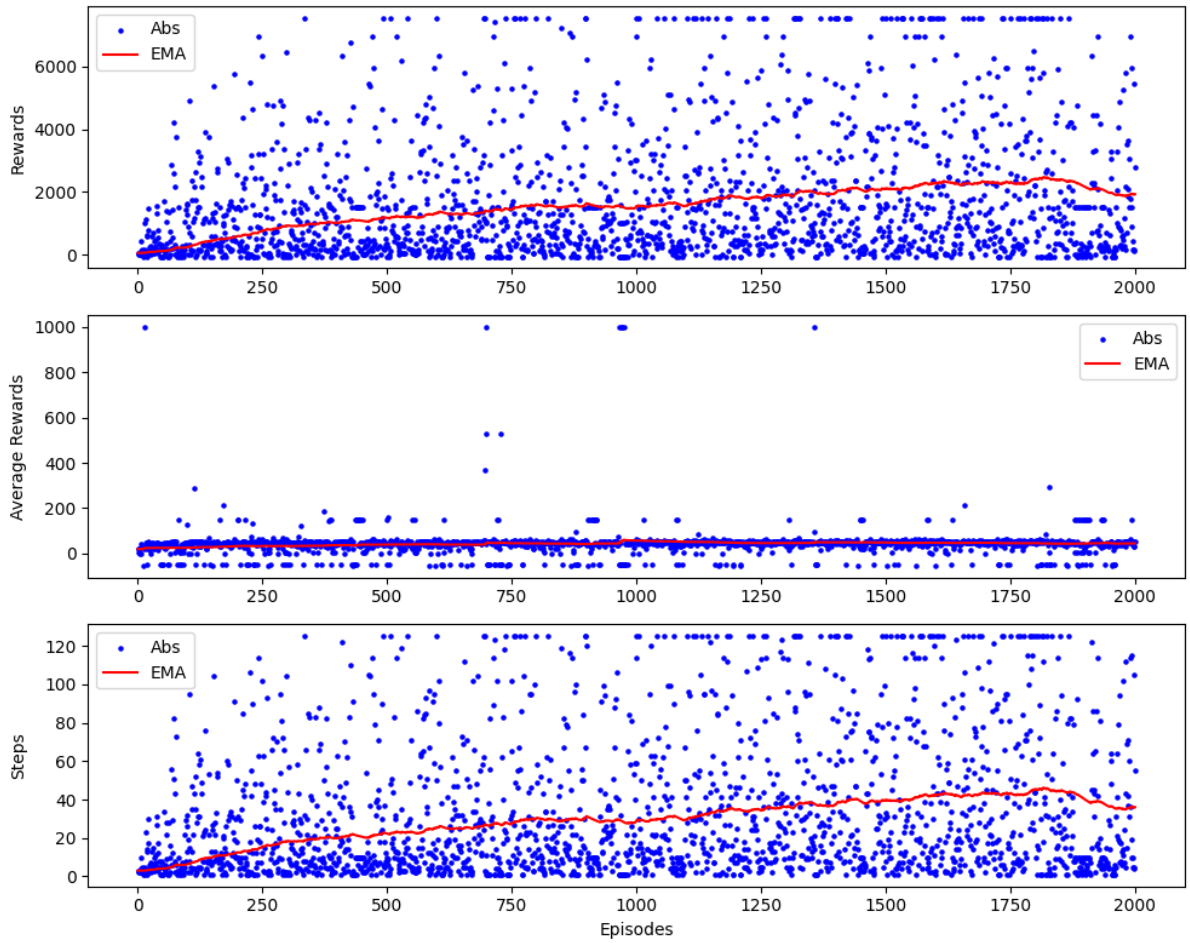

Figure 6.6: Results for Prototype 13 with Additional Benign Behavior over 2'000 Episodes with $\epsilon = 0.4, \delta = 0.05, \alpha = 0.0005, \gamma = 0.75$

Configuration 4 from Configuration 0, resulting in the accuracy results described above. One reason for this could also be the similarity of the expected rewards and the detection rates between Configuration 0 and 4, since both configurations were not well detected. In the previous experiments without additional behavior, the optimal configuration had the highest reward by far, here the gap between the optimal and the second-best configuration was much closer, possibly making it harder for the agent to differentiate the two.

To test the accuracy of the prototype with additional benign behavior for different training times, the experiment was repeated for different numbers of episodes. The resulting accuracies and their training times can be seen in Table 6.17. The agent was able to reach an accuracy of above 90 % in less than 10 minutes. Compared to the agent tested without additional benign behavior, the accuracy levels reached with a certain number of episodes was very similar. However, the training times were much higher as discussed previously, approximately five times longer.

In conclusion, after the experiments run with additional benign behavior, it can be stated that the agent showed similar learning capabilities as without the additional benign behavior. Due to the AD struggling more with detecting the ransomware configurations, some

Table 6.15: Accuracy for Prototype 13 with Additional Benign Behavior after 2'000 Episodes with $\epsilon = 0.4, \delta = 0.05, \alpha = 0.0005, \gamma = 0.75$

Configuration	**Initial Accuracy**		**Trained Accuracy**		**Final Q-Values**
	Percentage	# Samples	Percentage	# Samples	
Config 0	65.77%	5625 / 8552	99.63%	8520 / 8552	2.09840300e+02
Config 1	06.75%	577 / 8552	00.11%	9 / 8552	4.64193451e+00
Config 2	07.30%	624 / 8552	00.05%	4 / 8552	-5.51316332e-43
Config 3	06.38%	546 / 8552	00.06%	5 / 8552	1.40514306e+01
Config 4	06.98%	597 / 8552	00.05%	4 / 8552	8.59115417e+01
Config 5	06.82%	583 / 8552	00.12%	10 / 8552	6.35431035e+01

Table 6.16: Accuracy for Prototype 13 with Additional Benign Behavior after 5'000 Episodes with $\epsilon = 0.4, \delta = 0.05, \alpha = 0.0005, \gamma = 0.75$

Configuration	**Initial Accuracy**		**Trained Accuracy**		**Final Q-Values**
	Percentage	# Samples	Percentage	# Samples	
Config 0	65.77%	5625 / 8552	00.07%	6 / 8552	7.43793487e+01
Config 1	06.75%	577 / 8552	00.01%	1 / 8552	-9.68197237e-26
Config 2	07.30%	624 / 8552	00.04%	3 / 8552	-4.33260042e-92
Config 3	06.38%	546 / 8552	00.02%	2 / 8552	-1.77811110e-17
Config 4	06.98%	597 / 8552	99.82%	8537 / 8552	5.67962778e+01
Config 5	06.82%	583 / 8552	00.04%	3 / 8552	2.22276496e+01

configurations were hardly detectable. This resulted in Configuration 0 being deemed the optimal configuration using the revised reward function. Possibly due to the expected rewards of Configurations 0 and 4 not having a large difference, the agent sometimes struggled to decide between the two.

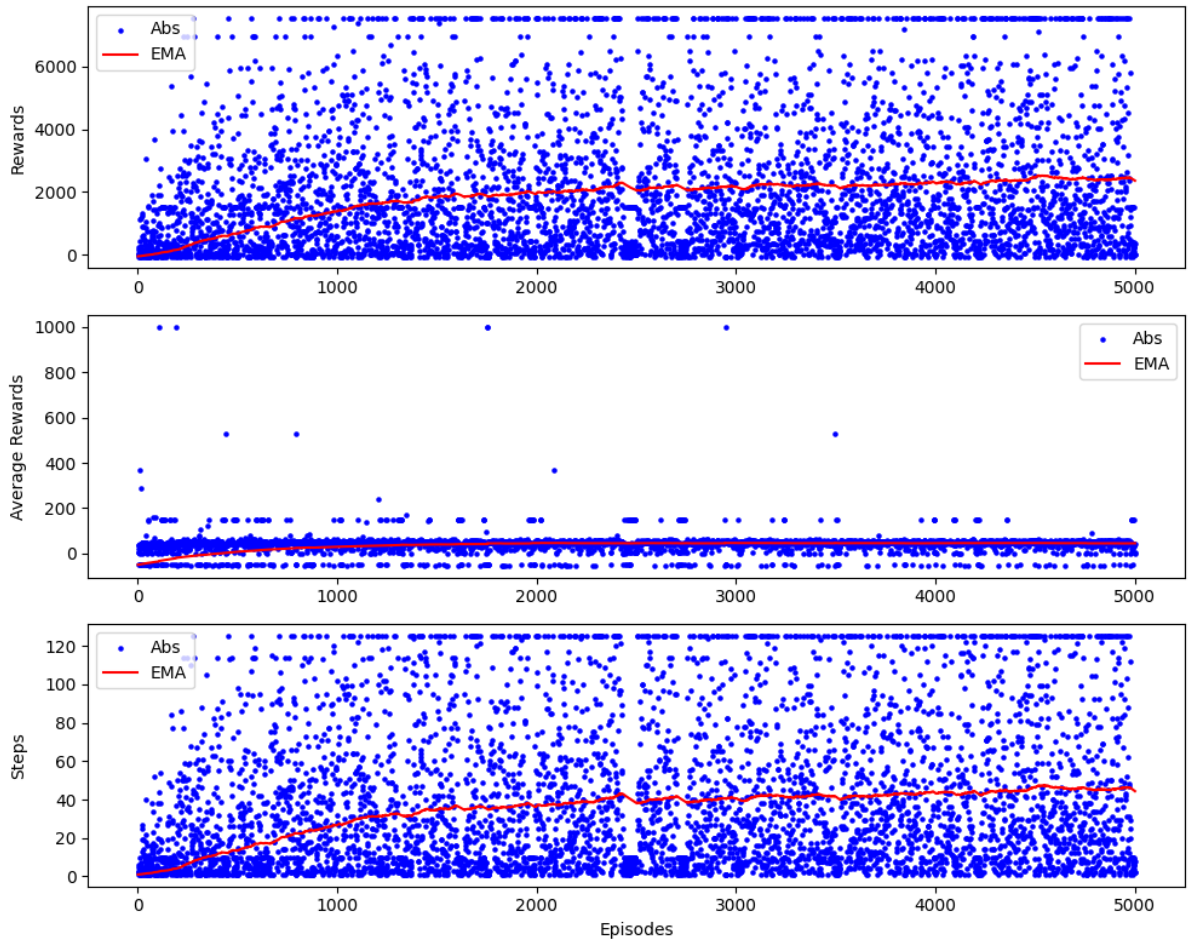

Figure 6.7: Results for Prototype 13 with Additional Benign Behavior over 5'000 Episodes with $\epsilon = 0.4, \delta = 0.05, \alpha = 0.0005, \gamma = 0.75$

Table 6.17: Accuracies of Selecting Config 0 and Training Times of the Improved Q-Learning Prototype with Additional Benign Behavior

Episodes	**Accuracy (in %)**	**Training Time**
50	90.01 %	7.6 min
100	94.46 %	16.2 min
200	95.29 %	35.1 min
500	98.62 %	101 min
1000	99.39 %	238.6 min

Chapter 7

Limitations

”A thorough evaluation of any research endeavor involves acknowledging and addressing potential limitations. This chapter delves into the known constraints of the research presented in this thesis, recognizing their significance in interpreting the findings and determining the generalizability of the results. Despite the substantial contributions made by the research, several areas warrant further exploration to enhance its comprehensiveness and applicability.

The limitations identified in this chapter stem from various aspects of the research process, including study design, fingerprint data collection methods, and analytical techniques. These constraints may influence the interpretation of the results and limit the ability to generalize the findings to broader contexts or environments. Nevertheless, acknowledging these limitations allows for a more informed understanding of the research's strengths and weaknesses, paving the way for future research that addresses these shortcomings and expands the knowledge base in the field.”[1]

As this work is extending previous work, only a select set of its limitations could be addressed in this work. Thus, there are several limitations that still remain present in this work. For instance, the performance of the ransomware as well as the efficiency of fingerprint retrieval was not addressed or further optimized in this work. As mentioned in [48], there might still be room for improvement regarding the speed of operations.

Moreover, this work and its findings are limited to the specific scenario and problem described in previous chapters. Although efforts were made to show the adaptability of the approach by incorporating additional benign behaviors, the scenario was still limited for instance by the device type or the type of malware used. As shown, the normal device behavior has a significant impact on the detectability of a ransomware, however, different devices most likely show different behavior patterns and thus different detection rates of ransomware. Furthermore, as this work did only use ransomware with different configurations, other malware samples such as botnets, rootkits, backdoors, or cryptojackers possibly have different results regarding the AD.

[1] This text was generated using generative artificial intelligence

As discovered, the performance of the RL agent largely depends on how well the AD classifier is able to detect the ransomware. This was shown by using an ideal version of the AD. With the AD used here, the sometimes large fluctuations mean that the RL agent's learning is hindered.

To be able to compare the results achieved with syscall-based fingerprints with the previous approach of resource usage-based fingerprints, the same ransomware configurations were adopted from [1]. This might be a limiting factor, as there might be even more optimal configurations further improving the balance between encryption speed and stealthiness.

Additionally, to explore the adaptability of the RL agent, two different benign behaviors were added. However, one of the two appeared to add a significant amount of noise to the AD, harshly limiting its ability to differentiate between normal and malicious behavior. The cause of this limitation needs to be further explored, *e.g.*, by testing a larger set of additional behaviors.

Finally, the revised reward function described in Section 6.3.2 might be further improved. The revised reward function presented in this work lead the RL agent used with additional benign behavior to sometimes struggle between the two best configurations. Due to this limitation, the agent sometimes randomly changed course and selected the second-best configuration for a period of time, before switching back to the optimal configuration.

Chapter 8

Future Work

”Chapter 7 discussed the limitations of the research presented in this thesis, identifying known constraints that may affect the interpretation and generalizability of the findings. Building upon this foundation, this chapter explores potential future directions for research, aiming to extend the knowledge base and address these limitations.

Beyond refining the research approach, this chapter delves into new research questions and methodologies that could further advance the field. These future directions encompass exploring previously unexamined aspects of the research topic, employing advanced data collection methods, and leveraging advanced analytical techniques. By pursuing these avenues, the field can continuously evolve and provide insightful advancements in our understanding of the research topic.”[1]

In order to further investigate the effects of different normal behaviors, more work is necessary in the future. By exploring a more diverse set of behaviors, one could examine the cause of the phenomenon experienced in this work, where the AD did not respond well to a certain behavior. Furthermore, this could be combined with testing if different fingerprint data sources perform differently when used with different benign behaviors.

For the purpose of testing the generalizability of the work presented, the approach could be used in different environments, *i.e.*, with different types of devices. This would allow to draw conclusions regarding the question whether this approach also works for different classes of devices, possibly not even in the IoT space.

Since the AD is a central component influencing the performance and accuracy of the RL agent, improving the AD's classification could prove worthwhile. To this end, other feature extraction methods from the field of NLP could be explored, such as Term Frequency - Inverse Document Frequency (TF-IDF) [49]. To improve the performance of the AD model, Principle Component Analysis (PCA) could be used to reduce the dimensionality of the features. The two approaches of resource usage fingerprinting and syscall fingerprinting could also be combined such that the positive aspects of both approaches could be leveraged.

[1] This text was generated using generative artificial intelligence

Moreover, other malware types could be explored, rather than focusing only on ransomware. Especially in the field of IoT, malwares such as botnets, backdoors, or cryptojackers could be an interesting use case to explore. It would be compelling to see whether it is feasible to use RL to hide a configurable version of such malware samples.

Finally, instead of training the RL agent to select the most optimal ransomware configuration from a prewritten set of configurations, it could be interesting to test the feasibility of training the agent to write a new optimal configuration. The agent would then test different values for the encryption rates, different encryption burst durations and different burst pauses. By interacting with the environment, the agent could possibly find more optimal configurations than it could when selecting from prewritten configurations.

Chapter 9

Summary and Conclusions

In this thesis, an extension to RansomAI [1] is presented in several aspects. The proposed and extended framework leverages RL to dynamically alter the configuration of a ransomware in order to avoid being detected by a defensive AD. The scenario in which this framework was applied uses a client device targeted by the ransomware and a C&C server, on which the RL agent as well as any other component necessary for the operation is running.

Device behavioral fingerprinting is used to monitor the behavior of the device. The target device is assumed to use an AD system based on the fingerprint data for defensive purposes. On the attacker side, this AD system is imitated and used to train the RL agent to stay undetected in such an environment. Although this attack is targeting a ML model, it is not an adversarial attack in the classical sense, since the input to the ML model (*i.e.*, the behavior data) can only be influenced indirectly via the ransomware configurations. As the defensive AD is treated as a black box, the scenario becomes more realistic.

A configurable version of an existing ransomware sample called Ransomware_PoC [45] was developed in [1] and was adopted in this work. This ransomware can be configured at runtime regarding the encryption algorithm used, the encryption rate, and encryption burst settings including burst duration and burst pause. With this, six different ransomware configurations were used for the evaluation. The ransomware is listening for configuration changes from the C&C server via a RESTful API during its operation.

To measure the state of the environment for the RL agent, device behavioral fingerprinting is used. As a first extension to the previous work, the fingerprints are generated from syscall data. Periodically, the syscalls of all processes running on the client device are collected and sent to the C&C server. There, the occurrence frequency of the individual syscalls is extracted and used as the set of features. This serves as input to the AD, deciding based on the input whether the device is behaving normally or might be infected. As a second measure on the state of the environment, the encryption rate is periodically reported by the ransomware running on the client device to the C&C server. The output of the AD and the encryption rate are fed into a reward function, giving a positive or negative reward signal to the agent depending on how good the state of the environment is.

The scenario was implemented using a device with constrained resources, namely a Raspberry Pi 3 running as an Electrosense sensor [10]. Such a device is used to monitor the radio frequency spectrum as a crowdsourcing solution. Various data sets were then collected using this setup, including normal behavior data and infected behavior data for each ransomware configuration. With these precollected data sets, a simulated environment was implemented to speed up the development and RL agent training. The developed prototypes were also evaluated using this simulated environment.

Initially, a first PoC prototype was implemented, showing the viability of the approach using syscall data for device behavior fingerprinting in combination with a RL agent. This prototype did not apply any real RL concepts yet, thus there are no results available.

Next, a simple prototype was developed which applied Q-learning using a neural network over a single episode. While this prototype showed promising results in that the agent learned to select actions that receive higher rewards, it was not able to find the most optimal configuration yet according to the expected rewards. Also, the reward function only considered the output of the AD and not the encryption rate.

In the next improved version of the prototype, the agent was trained for multiple episodes also using Q-learning. An artificial data corpus was used for a more realistic setting for the ransomware to encrypt in the simulated environment. Furthermore, a reward function was used that also incorporated the encryption rate, thus not only incentivizing the stealthiness, but also the speed of encryption. This prototype was shown to reach an accuracy of above 90 % in less than two minutes of training time. Additionally, the prototype was evaluated using an ideal version of the AD, in which certain ransomware configurations were detected manually, while others remained hidden. This was done in order to test the agent's learning capabilities under ideal conditions, without the fluctuations of the AD. These results showed that indeed the agent is able to learn the optimal configuration with increasing accuracy over time.

In comparison with the prototype presented in [1], there were significant differences found when using syscalls as the basis of fingerprints. Most configurations were more easily detected by the AD when using syscalls, which also had an influence on the configuration that was deemed the most optimal. While in [1] Configuration 3 was the most optimal, here it was Configuration 4 due to the AD being much better able to detect Configuration 3. Furthermore, the reward function used in this work was better able to distinguish between the best configuration and other sub-optimal ones. This serves as a potential explanation of a small decrease in the learning time needed to reach certain accuracy levels. The RL agent using syscall data was able to reach an accuracy of above 90 % slightly faster than the RL agent using resource usage data.

A major contribution of this work was the introduction of additional benign behaviors to the scenario. Two additional behaviors of the client device that should be classified as non-malicious behavior by the AD were implemented. Firstly, a data compression behavior was developed that continuously compresses a data corpus. Secondly, a package installation behavior was implemented, that uses `pip` to install Python packages. Subsequently, behavioral data was collected for the two additional behaviors, including data for the ransomware in conjunction with the additional behaviors. Testing the AD trained with all benign behaviors (*i.e.*, normal device behavior plus the two additional behaviors)

gave some surprising results. It was discovered that apparently the package installation behavior introduced significant noise to the detection, which left the AD unable to differentiate between normal and infected behavior. When the package installation behavior was left out however, the AD was again able to detect the malicious behavior.

It was shown that the presence of additional benign behavior executed on the client device significantly improved the ransomware's stealthiness. All configurations used in this work were less detected, with Configuration 0 only being detected in 1.67 % of cases. This leads to the conclusion that devices with more uniform behavior patterns are easier to protect in such a scenario.

Due to the changed AD results when using the additional benign behavior, a revised reward function was designed, to better reflect the effectiveness of the configurations under these new circumstances. With this, Configuration 0 was deemed the most optimal configuration, despite the fact that it has a very low encryption rate. Next, the RL model was evaluated with additional benign behavior. In general, the agent was able to learn the configuration also with the additional benign behavior. However, possibly due to the reward function design, the agent was less consistent such that in certain periods during the training, the second-best configuration was learned. Moreover, the agent's training took significantly longer compared to without additional behavior. This may in part be explained with the increased number of steps taken per episode as the configurations were much less detected. Nevertheless, the agent achieved an accuracy of above 90 % in less than eight minutes.

Bibliography

[1] J. von der Assen, A. H. Celdrán, J. Luechinger, *et al.*, *RansomAI: AI-powered Ransomware for Stealthy Encryption*, 2023. arXiv: 2306.15559 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2306.15559.

[2] L. S. Vailshery, *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030*, Last Visit 11.07.2023, 2022. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[3] F. Meneghello, M. Calore, D. Zucchetto, M. Polese, and A. Zanella, "IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices", *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8182–8201, 2019. DOI: 10.1109/JIOT.2019.2935189.

[4] R. Ahmad and I. Alsmadi, "Machine learning approaches to IoT security: A systematic literature review", *Internet of Things*, vol. 14, p. 100 365, 2021, ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2021.100365. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660521000093.

[5] Y. Jiang, X. Li, H. Luo, S. Yin, and O. Kaynak, "Quo vadis artificial intelligence?", *Discover Artificial Intelligence*, vol. 2, no. 1, 2022, ISSN: 2731-0809. DOI: 10.1007/s44163-022-00022-8.

[6] H. Wu, H. Han, X. Wang, and S. Sun, "Research on Artificial Intelligence Enhancing Internet of Things Security: A Survey", *IEEE Access*, vol. 8, pp. 153 826–153 848, 2020. DOI: 10.1109/ACCESS.2020.3018170.

[7] C. Thanh and I. Zelinka, "A Survey on Artificial Intelligence in Malware as Next-Generation Threats", *MENDEL*, vol. 25, no. 2, pp. 27–34, 2019. DOI: 10.13164/mendel.2019.2.027.

[8] J. Pan and C. Fung, *Artificial intelligence in malware - cop or culprit?*, University of Western Australia, 2008.

[9] J. Lüchinger, *RansomAI*, Last Visit 16.12.2023, 2023. [Online]. Available: https://github.com/jluech/RansomAI/tree/master.

[10] S. Rajendran, R. Calvo-Palomino, M. Fuchs, *et al.*, "Electrosense: Open and Big Spectrum Data", *CoRR*, vol. abs/1703.09989, 2017. arXiv: 1703.09989. [Online]. Available: http://arxiv.org/abs/1703.09989.

[11] P. M. S. Sánchez, A. H. Celdrán, G. Bovet, G. M. Pérez, and B. Stiller, "SpecForce: A Framework to Secure IoT Spectrum Sensors in the Internet of Battlefield Things", *IEEE Communications Magazine*, vol. 61, no. 5, pp. 174–180, 2023. DOI: 10.1109/MCOM.001.2200349.

[12] A. H. Celdrán, P. M. Sánchez Sánchez, C. Feng, G. Bovet, G. M. Pérez, and B. Stiller, "Privacy-Preserving and Syscall-Based Intrusion Detection System for IoT Spectrum Sensors Affected by Data Falsification Attacks", *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8408–8415, 2023. DOI: 10.1109/JIOT.2022.3213889.

[13] A. H. Celdrán, P. M. S. Sánchez, M. A. Castillo, G. Bovet, G. M. Pérez, and B. Stiller, "Intelligent and Behavioral-Based Detection of Malware in IoT Spectrum Sensors", *International Journal of Information Security*, vol. 22, no. 3, pp. 541–561, Jun. 2023. DOI: 10.1007/s10207-022-00602-w. [Online]. Available: https://doi.org/10.1007/s10207-022-00602-w.

[14] A. H. Celdrán, J. von der Assen, K. Moser, *et al.*, "Early Detection of Cryptojacker Malicious Behaviors on IoT Crowdsensing Devices", in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–8. DOI: 10.1109/NOMS56928.2023.10154392.

[15] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions", *Computers & Security*, vol. 74, pp. 144–166, 2018, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740481830004X.

[16] M. Humayun, N. Jhanjhi, A. Alsayat, and V. Ponnusamy, "Internet of things and ransomware: Evolution, mitigation and prevention", *Egyptian Informatics Journal*, vol. 22, no. 1, pp. 105–117, 2021, ISSN: 1110-8665. DOI: https://doi.org/10.1016/j.eij.2020.05.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1110866520301304.

[17] M. Alenezi, H. Alabdulrazzaq, A. Alshaher, and M. Alkharang, "Evolution of malware threats and techniques: A review", *International Journal of Communication Networks and Information Security*, vol. 12, p. 326, Dec. 2020. DOI: 10.17762/ijcnis.v12i3.4723.

[18] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A Survey on Ransomware: Evolution, Taxonomy, and Defense Solutions", *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022, ISSN: 0360-0300. DOI: 10.1145/3514229. [Online]. Available: https://doi.org/10.1145/3514229.

[19] AO Kaspersky Lab, *Targeted ransomware doubled in 2022, new techniques and groups emerge*, Last Visit 22.07.2023, Dec. 2022. [Online]. Available: https://www.kaspersky.com/about/press-releases/2022_targeted-ransomware-doubled-in-2022-new-techniques-and-groups-emerge.

[20] I. Yaqoob, E. Ahmed, M. H. ur Rehman, *et al.*, "The rise of ransomware and emerging security challenges in the Internet of Things", *Computer Networks*, vol. 129, pp. 444–458, 2017, Special Issue on 5G Wireless Networks for IoT and Body Sensors, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.09.003.

[Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128617303468.

[21] C. Zhang and Y. Lu, "Study on artificial intelligence: The state of the art and future prospects", *Journal of Industrial Information Integration*, vol. 23, p. 100 224, 2021, ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2021.100224. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2452414X21000248.

[22] M. Haenlein and A. Kaplan, "A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence", *California Management Review*, vol. 61, no. 4, pp. 5–14, 2019. DOI: 10.1177/0008125619864925. [Online]. Available: https://doi.org/10.1177/0008125619864925.

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd Ed. The MIT press, 2018, ISBN: 978-0-262-19398-6.

[24] C. J. C. H. Watkins, "Learning from delayed rewards", Ph.D. dissertation, University of Cambridge, 1989.

[25] P. M. S. Sánchez, J. M. J. Valero, A. H. Celdrán, G. Bovet, M. G. Pérez, and G. M. Pérez, "A Survey on Device Behavior Fingerprinting: Data Sources, Techniques, Application Scenarios, and Datasets", *CoRR*, vol. abs/2008.03343, 2020. arXiv: 2008.03343. [Online]. Available: https://arxiv.org/abs/2008.03343.

[26] P. M. Sánchez Sánchez, J. M. Jorquera Valero, A. Huertas Celdrán, G. Bovet, M. Gil Pérez, and G. M. Pérez, "A methodology to identify identical single-board computers based on hardware behavior fingerprinting", *Journal of Network and Computer Applications*, vol. 212, p. 103 579, 2023, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2022.103579. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S108480452200220X.

[27] R. Ahmad, I. Alsmadi, W. Alhamdani, and L. Tawalbeh, "Zero-day attack detection: A systematic literature review", *Artificial Intelligence Review*, 2023, ISSN: 1573-7462. DOI: 10.1007/s10462-023-10437-z. [Online]. Available: https://doi.org/10.1007/s10462-023-10437-z.

[28] N. Kaloudi and J. Li, "The AI-Based Cyber Threat Landscape: A Survey", *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3372823. [Online]. Available: https://doi.org/10.1145/3372823.

[29] L. Fritsch, A. Jaber, and A. Yazidi, "An Overview of Artificial Intelligence Used in Malware", in *Nordic Artificial Intelligence Research and Development*, E. Zouganeli, A. Yazidi, G. Mello, and P. Lind, Eds., Cham: Springer International Publishing, 2022, pp. 41–51, ISBN: 978-3-031-17030-0.

[30] P. Ferrie and H. Shannon, "It's zell(d)ome the one you expect", in *Virus Bulletin*, May 2005, pp. 7–11. [Online]. Available: https://cryptohub.nl/zines/vxheavens/lib/apf50.html.

[31] M. Sewak, S. K. Sahay, and H. Rathore, "ADVERSARIALuscator: An Adversarial-DRL based Obfuscator and Metamorphic Malware Swarm Generator", in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, pp. 1–9. DOI: 10.1109/IJCNN52387.2021.9534016.

[32] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, *Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning*, 2018. arXiv: 1801.08917 [cs.CR]. [Online]. Available: https://arxiv.org/abs/1801.08917.

[33] W. Xu, Y. Qi, and D. Evans, "Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers", *Network and Distributed Systems Symposium 2016*, Feb. 2016. [Online]. Available: https://evademl.org/docs/evademl.pdf.

[34] H. S. Anderson, J. Woodbridge, and B. Filar, "DeepDGA: Adversarially-Tuned Domain Generation and Detection", *CoRR*, vol. abs/1610.01969, 2016. arXiv: 1610.01969. [Online]. Available: http://arxiv.org/abs/1610.01969.

[35] W. Hu and Y. Tan, "Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN", *CoRR*, vol. abs/1702.05983, 2017. arXiv: 1702.05983. [Online]. Available: http://arxiv.org/abs/1702.05983.

[36] D. Kirat, J. Jang, and M. P. Stoecklin, *DeepLocker: Concealing Targeted Attacks with AI Locksmithing*, presented at Black Hat USA 2018, Las Vegas, NV, Aug. 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf.

[37] K. Chung, Z. T. Kalbarczyk, and R. K. Iyer, "Availability Attacks on Computing Systems through Alteration of Environmental Control: Smart Malware Approach", in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 1–12, ISBN: 9781450362856. DOI: 10.1145/3302509.3311041. [Online]. Available: https://doi.org/10.1145/3302509.3311041.

[38] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading Anti-Malware Engines With Deep Reinforcement Learning", *IEEE Access*, vol. 7, pp. 48 867–48 879, 2019. DOI: 10.1109/ACCESS.2019.2908033.

[39] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading Machine Learning Malware Detection", Black Hat USA, White Paper, 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Anderson-Bot-Vs-Bot-Evading-Machine-Learning-Malware-Detection-wp.pdf.

[40] T. Quertier, B. Marais, S. Morucci, and B. Fournel, *MERLIN -- Malware Evasion with Reinforcement LearnINg*, 2022. arXiv: 2203.12980 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2203.12980.

[41] R. Labaca-Castro, S. Franz, and G. D. Rodosek, "AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning", in *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track*, Y. Dong, N. Kourtellis, B. Hammer, and J. A. Lozano, Eds., Cham: Springer International Publishing, 2021, pp. 37–52, ISBN: 978-3-030-86514-6.

[42] S. Pandey, N. Kumar, A. Handa, and S. K. Shukla, "Evading malware classifiers using RL agent with action-mask", *International Journal of Information Security*, Jul. 2023, ISSN: 1615-5270. DOI: 10.1007/s10207-023-00715-w. [Online]. Available: https://doi.org/10.1007/s10207-023-00715-w.

[43] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, *MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers*, 2021. arXiv: 2003.03100 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2003.03100.

[44] M. Ebrahimi, J. Pacheco, W. Li, J. L. Hu, and H. Chen, "Binary Black-Box Attacks Against Static Malware Detectors with Reinforcement Learning in Discrete Action Spaces", in *2021 IEEE Security and Privacy Workshops (SPW)*, 2021, pp. 85–91. DOI: 10.1109/SPW53761.2021.00021.

[45] Jimmy, *Ransomware-PoC*, Last Visit 11.12.2023, 2021. [Online]. Available: https://github.com/jimmy-ly00/Ransomware-PoC.

[46] Library Of Congress Web Archiving Program, *.gov PDF dataset*, Last Visit 18.12.2023, 2019. [Online]. Available: https://www.loc.gov/item/2020445568/.

[47] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Boston, MA: Pearson, 2015, ISBN: 978-0-13-359162-0.

[48] J. Lüchinger, *AI-powered Ransomware to Optimize its Impact on IoT Spectrum Sensors*, University of Zurich, 2023.

[49] scikit-learn developers, *Feature Extraction*, Last Visit 16.12.2023, 2023. [Online]. Available: https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction.

[50] Free Software Foundation, *nohup(1p) — Linux manual page*, Last Visit 18.12.2023, 2010. [Online]. Available: https://linux.die.net/man/1/nohup.

[51] Free Software Foundation, *tar(1) - Linux man page*, Last Visit 18.12.2023, 2010. [Online]. Available: https://linux.die.net/man/1/tar.

[52] The pip developers, *pip*, Last Visit 19.12.2023, 2020. [Online]. Available: https://pip.pypa.io/en/stable/.

Abbreviations

AB1	additional behavior 1
AB2	additional behavior 2
AD	anomaly detection
AES	advanced encryption standard
AI	artificial intelligence
API	application programming interface
BoW	bag-of-word
C&C	command and control
CPS	cyber-physical system
CPU	central processing unit
DDoS	distributed denial of service
DL	deep learning
DNN	deep neural network
DoS	denial of service
DP	dynamic programming
DRL	deep reinforcement learning
EMA	exponential moving average
FFT	fast fourier transform
FNR	false negative rate
GAN	generative adversarial network
IDS	intrusion detection system
IF	isolation forest
IoBT	internet of battlefield things
IoT	internet of things
IQ	in-phase and quadrature
LAN	local area network
LOF	local outlier factor
MDP	Markov decision process
ML	machine learning
NAB	no additional behavior
NLP	natural language processing
OCSVM	one-class support vector machine
OS	operating system
OSDaaS	open spectrum data as a service
PCA	principal component analysis
PE	portable executable

PMU	performance monitor unit
PoC	proof of concept
PPO	proximal policy optimization
PSD	power spectral density
RaaS	ransomware as a service
ReLU	rectified linear unit
REST	representational state transfer
RF	radio frequency
RL	reinforcement learning
ROAR	ransomware optimized with AI for resource-constrained devices
RSA	Rivest Shamir Adleman
SDR	software-defined radio
SiLU	sigmoid linear unit
SSDF	spectrum sensing data falsification
SSH	secure shell
TD	temporal difference
TF-IDF	term frequency - inverse document frequency
TLS	transport layer security
TNR	true negative rate
VAC	variational actor critic

List of Figures

List of Tables

List of Listings

List of Algorithms

Appendix A

Codebase

The codebase used for this master thesis is split into two parts, one intended to be used on the client device and one on the C&C server. Both parts are publicly available including the respective installation guidelines at the following URLs:

- C&C server: `https://github.com/SandroPadovan/extended_roar_server`
- Client device: `https://github.com/SandroPadovan/extended_roar_client`