University of Zurich[UZH]

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

Optimizing MTD Deployment on IoT Devices using Reinforcement Learning

Timo Schenk
Zurich, Switzerland
Student ID: 15-937-337

Supervisor: Dr. Alberto Huertas, Jan von der Assen
Date of Submission: October 4, 2022

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/

Zusammenfassung

Das explosionsartige Wachstum des IoT geht mit einer Zunahme von Cyberangriffen einher, wobei Ransomware, Rootkits und Command-and-Control-Malware besonders häufig vorkommen. Ein vielversprechender Ansatz zur Schadensbegrenzung ist Moving Target Defense (MTD), wo die Angriffsfläche eines Ziels dynamisch verändert wird. Der Stand der IoT-MTD ist jedoch noch unausgereift, und es mangelt an Forschung zur Koordinierung mehrerer MTD-Techniken in realen Anwendungen. Als Mittel zur Optimierung eines solchen Systems erforscht diese Arbeit die Anwendung von Reinforcement Learning (RL), um MTD-Techniken reaktiv gegen die zuvor genannten Malware-Familien einzusetzen in einem realen Crowdsensing-Szenario. Zunächst wird die Aufgabe RL-basierter MTD-Auswahl analysiert und die wichtigsten Systemanforderungen herausgearbeitet. Danach werden drei Simulationen, sowie die Implementierung eines kompletten Online-MTD-Agenten vorgestellt. Da Online-RL kostspielig ist, verlagern sich die Simulationen von einer zunächst eher theoretischen Perspektive hin zur Realität, um die Übertragung von MTD-Strategien auf eine reale Umgebung zu ermöglichen. Die erste Simulation stellt eine Baseline dar und setzt einen Supervisor zur Erzeugung von Belohnungssignalen ein. Die Zweite tauscht diesen Supervisor gegen eine Komponente zur Erkennung von Anomalien aus. Zur Vergleichbarkeit wird in beiden Simulationen ein neu gesammelter Datensatz mit Rohdaten zum Angriffsverhalten verwendet. Die dritte Simulation nutzt ebenfalls die Anomalieerkennung, verwendet jedoch einen zweiten Datensatz von Verhaltensweisen, die von einem echten Online-Agenten aufgezeichnet wurden. Während der Agent der ersten Simulation lernt, MTD-Techniken für alle Angriffe der oben genannten Familien auszuwählen, zeigen die zweite und dritte Simulation, dass die Konvergenz eines realistischen Agenten durch Ungenauigkeiten bei der Anomalieerkennung beeinträchtigt wird, Angriffe jedoch mehrheitlich abgewehrt werden (>91%). Schließlich werden die Auswirkungen des Online-Agenten diskutiert und der Ressourcenverbrauch auf einem Raspberry Pi 3 evaluiert. Mit einem Speicherbedarf von weniger als 1 MB und einer Auslastung von weniger als 80% der verfügbaren CPU und des Arbeitsspeichers stellt die Hardware keine Einschränkung dar. Die Zeit für das Erlernen neuer Angriffe, kann jedoch ein Hindernis darstellen.

Abstract

The explosive growth of the IoT has come along with an increase of cyberattacks with ransomware, rootkits and Command-and-Control malware being particularly common families. One promising approach for mitigation is offered by Moving Target Defense (MTD), which works by dynamically altering a target's attack surface. However, the state of IoT MTD is still immature, especially lacking research dedicated to coordinating multiple MTD techniques in real applications. As a means to optimize such a system, this work explores the application of reinforcement learning (RL) to reactively deploy MTD techniques against the aforementioned malware families in a real crowdsensing scenario. First, the task of RL-based MTD selection is analyzed to distill major system requirements. Thereafter, three training simulations are presented along with the implementation of a complete, online MTD agent. As online RL is costly, the simulations gradually shift from a rather theoretical perspective towards approximating reality to allow policy transfer to a real environment. Using a supervisor to create reward signals, the first simulation marks a baseline. The second exchanges this supervisor for an anomaly detection component. For comparability both simulations use a new dataset of raw attack behaviors. The third simulation also leverages anomaly detection, yet utilizes a second dataset of behaviors monitored by a real online agent. While the agent of the first simulation learns to select MTD techniques against all attacks of the aforementioned families, the second and third simulations show that a realistic agent's convergence is affected by anomaly detection inaccuracies, but generally attacks are effectively mitigated. Finally, implications of the online agent are discussed and its resource consumption is evaluated on a Raspberry Pi 3. Requiring less than 1MB storage and always utilizing below 80% of the available CPU and RAM, hardware poses no limitation. However, the time required to learn new attacks may impair viability

Acknowledgments

First of all, I especially want to thank Dr. Alberto Huertas Celdrán for his continuous support along this project, for taking the time to go through many complex meetings and always giving clear feedback. Further, I want to thank Pedro Miguel Sánchez for his highly appreciated technical advice and Jan von der Assen for promoting this project. Finally, I want to thank Prof. Dr. Burkhard Stiller for giving me the opportunity to pursue this exciting topic at the Communication Systems Group.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) has experienced explosive growth over recent years and forcasts estimate the number of connected devices to continue to grow by billions annually [19], [53]. IoT devices permeate many areas of modern society with applications spreading from smart-homes and cities, over healthcare and industrial tracking to crowdsourcing [19].

A particular IoT crowdsourcing scenario is given by crowdsensing, where large groups of individuals share benefits of data monitored via resource-constrained sensor devices [37]. As an example, the ElectroSense initiative provides a global crowdsensing platform for sharing radiofrequency spectrum data with participants typically employing Raspberry Pi (RP) devices equipped with software-defined radio kits (SDR) [51],[43].

The rapid growth of the IoT, the resource-constrained nature of respective devices and the heterogeneity of application scenarios have accelerated the emergence of related cyberattacks [61]. Against this background, crowdsensing participants are at particular risk, as static sensor devices typically are not subject to the scrutiny and security measures employed at more resourceful machines of everyday usage [46]. This clearly shows the need for novel and automated approaches to security management that take these circumstances into account.

One security approach of great potential is Moving Target Defense (MTD). MTD aims to thwart off adversaries who rely on the static nature of the attack target by proactively or reactively moving certain system parameters [41]. As a security paradigm, MTD follows the philosophy that perfect security is most likely not achievable, which matches the current state of IoT security. Rather than preventing any attacks, the idea is to defend against them in a dynamic manner [14], [19]. Despite being promising as a defense paradigm against cyberattacks on IoT devices, there are many open research questions to achieve desired security benefits. In particular, the optimal deployment of MTD solutions is a frequently-observed limitation of existing MTD-based frameworks [14]. This is even more prevalent if multiple MTD techniques must be coordinated to cope with a series of different attack vectors [14].

A promising solution to manage the MTD deployment control task in an automated and comprehensive manner is Reinforcement Learning (RL). As opposed to other machine

learning (ML) approaches which typically work offline, RL is designed to also operate in a completely online manner. Further, it does not require cumbersome labelling efforts as normal for supervised ML. Instead, RL provides a flexible framework for learning from successes and failures by interacting with an environment. As such, it is tailored to dynamically learn policies for sequential decision problems - like the problem of how to optimally coordinate MTD techniques.

1.1 Motivation

Considering the rapid growth of the IoT and its increasingly complex attack surface, this work aims to explore how Moving Target Defense (MTD) can be optimally applied within a comprehensive framework leveraging Reinforcement Learning (RL).

The combination of RL and MTD is a relatively young and diverse field of research, so the number of open challenges is naturally large. However, based on a literature review, the following key challenges can be distilled:

The current state of MTD on its own is still considered immature with respect to techniques targeted specifically for IoT devices [41],[14]. Further, there is a clear lack of work evaluating MTD solutions in real world scenarios. Thus, the focus should be on novel IoT-MTD techniques validated within real and exemplary IoT applications, such as the ElectroSense crowdsensing platform [41],[14].

Against this background, a number of MTD techniques have been developed at the University of Zurich to thwart off ransomware, rootkits as well as command and control (CnC-based) malware [12]. However, while having been validated in the ElectroSense scenario for their effectiveness, it is unclear how these techniques can be optimally leveraged to maximize IoT defense capabilities.

Besides the general lack of literature on real-world IoT MTD, there is a lack of research dedicated to deploying among multiple MTD techniques [14]. Where single MTD techniques have been utilized and enhanced against specific attacks, it has not yet been investigated how multiple MTD solutions that mitigate a range of attacks can be coordinated and optimized within a well-defined and holistic defense framework [14]. There is a clear need to address issues revolving around the construction of such a framework, including the question of what MTD techniques to deploy, when, and how. Certainly, in order to apply MTD thoroughly as a security paradigm, this is a crucial challenge to solve.

As new attacks arise and MTD techniques are developed, it is imperative to dynamically adapt to these changing circumstances and figure out how mitigating techniques can be mapped to given attacks. RL-based agents are very promising to solve this task as they can learn online and discover optimal MTD choices by trial and error. However, to the best of our knowledge, it has not yet been investigated how multi-purpose MTD can be embedded in the RL framework.

Where RL has been used to optimize single MTD techniques, the resulting agents have not been validated sufficiently in real-world scenarios. Most often, agents are being trained in

artificial simulations, without any transfer of the learnt policies to a real context for validation. Further, generally there are no results available regarding resource requirements that are necessary to judge solution feasibility in the IoT context. Thus, to derive stronger statements about the viability, the effectiveness and the efficiency of an RL-based MTD system targeted for the IoT, it should be constructed based on real environment data and tested for its resource consumption.

1.2 Aims

This work aims to construct an RL-based framework for optimized MTD deployment. As a primary goal, the focus is on deciding WHAT MTD technique to utilize given any possible state of the IoT device. In particular, and based on the open challenges identified in the previous section, this work aims to present the following contributions:

1. An analysis of currently available MTD techniques [12], targeted malware (ransomware, rootkits and CnC), as well as the general RL framework to generate a description of the problem domain. Thereby, the goal is to identify basic requirements and to formulate relevant assumptions for constructing an RL-based multi-MTD system capable of thwarting off a range of attacks in the ElectroSense crowdsensing scenario.

2. The design, implementation and performance evaluation of three different MTD agent prototypes that learn offline in simulated environments. Having the aim of iteratively mimicking reality more and more closely, these environments are constructed at increasing levels of complexity. In essence, the proposed simulation prototypes vary along the dimensions environment data used, as well as the degree of training supervision:

   (a) The first prototype acts as a baseline and aims to show what RL can achieve under ideal conditions, respectively how well an agent can learn to choose MTD techniques when presented an attack behavior state. This simulation uses raw attack behavior data and employs a supervisor to determine the construction of episodes and rewards.

   (b) The second prototype leverages an anomaly detector, which allows to construct episodes in a completely unsupervised manner. Positive or negative reward signals are estimated by the anomaly detector based on the normality of states after MTD execution. Thus, while still sampling from ideal raw behavior data, this simulation aims to measure how much impact imprecisions given by the anomaly detector have on the system's learning capabilities.

   (c) The third prototype also works completely unsupervised, yet considers realistic data, that also an online agent would observe. Essentially, the data used for this simulation environment accounts for noise introduced by an MTD agent controller itself, besides the raw attack behavior used in previous prototypes. Thus, this prototype aspires not only to be useful for analyzing a potential

online learning process, but also for pretraining agents for later online deployment. This further implies that the performance evaluation of the learnt policy aims to be transferrable to the real-world crowdsensing scenario.

3. The design and implementation of a fully functional online MTD controller agent, capable of autonomously learning in a real sensor environment. This comprises the orchestration of tasks like observing behavior states on the sensor, interpreting them, deploying MTD techniques and performing agent learning updates.

4. An evaluation of the resource requirements induced by the online MTD agent on the sensor device to validate the feasibility of the proposed RL-based MTD system.

1.3 Outline

As an initial step towards achieving the aims stated above, Chapter 2 starts off with relevant background regarding malware families of interest, MTD design and techniques, as well as RL. Besides introducing the main concepts in these areas, related work is presented to point out the lack of existing, RL-based MTD solutions and contextualize the present work. Subsequently, Chapter 3 discusses the problem domain of fitting reactive MTD into the RL framework as well as assumptions needed to establish a base for a well-defined training process. Further, the chapter derives the minimally required functional components for an RL-based MTD controller as well as desired security and efficiency properties. This marks the first aim of this work. Next, Chapter 4 dives into data needs for RL and implications for constructing simulated environments. The chapter introduces the features considered for states and explores properties of a collected raw device behavior dataset. Furthermore, the most important aspects of data preprocessing are explained as used for later chapters. The following Chapter 5 seeks to fulfill the goal 2a and 2b of this thesis by presenting the first two offline prototypes. Besides elaborating on the construction of the corresponding simulation environments, the performance of accordingly trained agents is evaluated. After these preliminary considerations of applying RL for MTD selection, Chapter 6 continues with measures undertaken to make the simulated environment as realistic as possible. Thus, the chapter explains the design of a full-fledged, online MTD controller which is required to collect refined data as used for a third and more realistic simulation environment. After discussing the data collection procedure and analyzing an accordingly refined dataset, an agent is trained in the corresponding simulation and evaluated for its performance. This ensures the accomplishment of aim 2c and 3 from the previous section. Next, Chapter 7 discusses implications of running an MTD controller agent online. Moreover and dedicated to the final goal 4 of this thesis, the full MTD controller is evaluated on a RP device for its resource consumption. Finally, Chapter 8 concludes with a summary, points out limitations and proposes directions for future work.

Chapter 2

Background and Related Work

This chapter gives an overview of the background technologies relevant for this work. It starts by shortly describing the crowdsensing platform used as a base. Next, it introduces the general concepts of Moving Target Defense, three different malware families as well as corresponding, selected MTDs that subsequent chapters will build upon. Additionally, the most important theoretical background is covered with regards to RL. After having considered the main aspects thereof, related work is presented to contextualize the present work.

2.1 Crowdsensing: ElectroSense

Frequently, IoT devices are not just deployed in a standalone manner, but work interconnectedly in larger networks, such as in the case of crowdsensing [37]. Crowdsensing denotes a special application of crowdsourcing, that leverages sensor data monitored by a large and diversified group of collaborating individuals [49].

In this work, the ElectroSense Platform is used as an exemplary, open-source real-world representative of an IoT crowdsensing network to develop and test the proposed system [51],[43]. ElectroSense is a crowd-sourcing initiative that uses Raspberry Pi devices equipped with radio sensors to collect spectrum data worldwide. The goal is to make this data available in real-time for analysis by different kinds of stakeholders. As an open initiative, everyone can contribute with sensor measurements and access the commonly collected data.

Setting up a sensor requires a RP 3 or higher, an SDR (USB dongle) receiver, as well as a dipole antenna-set and an internet connection. An image of the RP OS preconfigured for ElectroSense can be burned onto an SD card to boot up the RP. Upon assembly of the RP, the SDR and the antenna, the device can be connected to the internet (typically via ethernet), powered on and registered onto the ElectroSense platform. This is sufficient to start contributing spectrum data [10]. For this thesis a setup with a RP3 of 1GB RAM is considered.

2.2 Malware Threats

IoT devices in general and RPs used for ElectroSense in particular are subject to a large number of potential cyberattacks. Despite this variety, three different families of malware are especially common and thus chosen for this work: Command and Control (CnC-based) malware, rootkits as well as ransomware. This section briefly presents each of these families and provides information about selected examples of malware used in later chapters.

2.2.1 Command and Control

CnC-based malware tries to establish a communication channel between a target machine and a controlling server with the goal of remotely instructing this compromised device to perform some harmful activity [23]. CnC functionality marks a central component for botnets, where a botmaster command and controls an army of infected devices (bots) [32]. IoT devices are frequent targets of CnC malware as they usually lack the security measures leveraged for more resourceful machines and often malware can operate for prolongued periods of time before it is detected. Types of malicious CnC activity include, but are not limited to executing DDoS attacks, installing backdoors, spreading other malware, exfiltrating sensitive data (data leakage) or unwanted digital currency mining [67]. For this work, four different CnC-based malware are considered. First, the tick [42], allows to remotely control a number of bots via a server by means of a remote shell or extract files from victim devices. Next and similarly, jakoritarleite provides a python implementation for client- (victim) and server-side components enabling data leakage and remote control [29]. Besides these two attacks, two further data leakage attacks are utilized as provided by an own proof of concept implementation using shell script and the netcat command [56]. The first, periodically leaks a file of interest from a victim to an attacker machine. The second allows to periodically send commands (ps aux, ls /etc, df -h, free) from an attacker to a victim. The victim then executes these commands and sends the results back to the server. Thus, this second option is further characterized by malicious activity, besides the system information data leakage. In later chapters these attacks will be referred to as `data_leak_1` and `data_leak_2` respectively.

The CnC channel marks a single point of failure for sending commands to the victim. Thus, detection of the CnC channel and disruption of communication is of great importance for finding effective countermeasures [23].

2.2.2 User-Level Rootkits

Another family of malware potentially targeting IoT devices is given by rootkits. While there exist both kernel- and user-level rootkits, the focus will be on the latter here. Linux User-level rootkits operate in user space (ring 3) by abusing the dynamic linker to preload shared libraries specified by an attacker before legitimate libraries can be loaded. Thus,

libraries modified to perform malicious actions may be injected between the kernel and regular, shared libraries [52].

As per normal operation, there are three main ways of specifying libraries to preload: First, via the *LD_PRELOAD* environment variable, secondly by the *–preload* command line option if the dynamic linker is evoked directly, and third by manipulating the file */etc/ld.so.preload* [39]. All three methods define a list of paths to shared libraries that should be loaded before all others and are considered by the dynamic linker as per the order above. The file */etc/ld.so.preload* has a system-wide effect. This means that such specified libraries are preloaded for all programs running on the system [39].

User-level rootkits effectively manipulate *LD_PRELOAD*, */etc/ld.so.preload* or the the dynamic linkers filepath considered for preloading. Linux allows for multiple definitions of symbols in shared libraries, but they are only resolved once. Thus, a user-level rootkit takes precedence. By preloading malicious library implementations it may completely alter the normal flow of execution [52]. The Umbreon rootkit, for instance, may hijack about 100 symbols, including the *chown* command [65]. The beurk rootkit, effectively manipulates */etc/ld.so.preload* by appending malicious libraries [71]. The bdvl rootkit tampers with the location where the dynamic linker checks for preloading shared libraries (in the file */lib/arm-linux-gnueabihf/ld-2.24.so*). Instead of */etc/ld.so.preload* it adds a custom path, which points to a malicious version [21]. This way the rootkit may hide its presence, as */etc/ld.so.preload* remains as usual. Both beurk and bdvl are used for this work.

A further issue is that rootkits often hide relevant files, such as */etc/ld.so.preload* or its replacement [8] [21]. This is an obstacle for sanitizing the system as a hidden file may neither be opened nor adapted. Further, it might also not be possible to create a file with the same name with a clean version. Resource-constrained IoT devices and remotely deployed spectrum sensors are particularly vulnerable to such threats due to more difficult intrusion detection.

2.2.3 Crypto Ransomware

Ransomware has become a lucrative cybercrime business and yields millions of dollars every year [24]. Further, as one of the fastest emerging attack categories for the IoT, crypto ransomware may also be launched against Raspberry Pi devices as employed in ElectroSense [70].

Crypto Ransomware tries to hijack user files and other relevant resources on a target system with the aim of forcing the victim to pay a ransom for the locked data. Typically, this is achieved by using strong cryptography to encrypt named parts of a system and displaying a message demanding the ransom. Upon payment the data can be decrypted at the attackers will [2], [24]. Crypto ransomware usually does not encrypt an entire hard drive, but only targets specific files (text, images, video files) which are not essential for the system to function [70]. Locker ransomware in contrast, locks a user from accessing the system, but leaves data files untouched [70]. In this work however, only crypto ransomware will be considered, specifically the Ransomware-PoC attack by jimmy-ly00 [30].

2.3 Moving Target Defense

This section presents background information about MTD as a security paradigm, as well as MTD design principles necessary to understand the mode of operation of selected MTD techniques explained in subsequent sections.

The security of networked systems is heavily impacted by the asymmetric nature of attack and defense [9]. While attackers need to find and exploit only one vulnerability to compromise a system, defenders must aim to patch the entirety of potential vulnerabilities. Thus, the theoretical effort required by the defender to secure a system is a multitude larger than the attackers effort - alongside the usual maintenance requirements. Attackers have a time and informational advantage as they can analyze a static system for longer durations and launch targeted penetration tests repeatedly. Once a vulnerability has been found, the attacker may exploit it for prolonged periods of time - possibly also in homogeneous, similar systems - until it is patched by the defender [9]. However, even if the defender knows about a security breach, a patch might not be available on short notice, if at all. This all leads to a significant security disadvantage for systems configurations which are deterministic, static and homogeneous [9].

Against this background, a novel cyberdefense paradigm called Moving Target Defense (MTD) was introduced in 2009 [45], [14]. As a proactive means to thwart off attacks, MTD proposes to continuously "move" or alter certain aspects of a given system or device. By targetedly manipulating such system configurations, the attack surface effectively changes, such that the attacker's task becomes increasingly complex and uncertain [14]. Due to the moving attack surface, vulnerable components are harder to identify and exploit as one system state of the past cannot predictably be leveraged for subsequent attacks [14]. Thus, the MTD paradigm aims to reverse parts of the asymmetric nature for attack and defense in traditional, static system settings by limiting time, complicating reconnaissance and thus introducing higher cost for attackers [9].

However, assuming that it is impossible to achieve perfect security in any system, the objective of MTD is not to prevent attacks completely, but to enable normal system operation even if there are malicious actors present [14]. Thus, MTD can be either employed proactively to hamper attacks, or to reactively mitigate or thwart off ongoing attacks.

2.3.1 MTD Design Principles

According to Cai et al. [9] there are three main questions that need to be clarified to design a technique following the MTD paradigm: *What* to move in the system, *When* to move, and *How* to move it [9]. This work, will mainly be concerned with deciding on the *What* to move, respectively with choosing among multiple MTD technique the one which fits best a given situation.

WHAT

An MTD technique changes the attack surface of a system by moving a number of system attributes, properties or features which a potential attack would rely upon. Thus, the range of features to alter becomes a parameter, the so-called "Moving Parameter" (MP) [9]. Based on the system layer to which the MP belongs the following categorization has been proposed for MTD techniques [68]:

1. dynamic data: techniques changing the encoding, the representation or the format of application data [68], [12], [35]. Here, the aim is to adapt data regarding certain aspects, that malware needs to rely upon, such as file format or storage location or directory structures [12]. [35].

2. dynamic software: techniques altering the code of an application dynamically. This comprises modification of program instructions, their order, format or grouping [68]. The goal is to use the software as a moving parameter by creating multiple variants (diversification) that are functionally equivalent, but differ with respect to their behavior out of the specification [9]. Among other approaches this can be achieved compiler-based, by hand, or via computational outsourcing over the web [27], [9].

3. dynamic runtime environment: techniques changing the execution environment, such as address space randomization (changing memory layout and location of code/libraries and functions, i.e. against stack-overflow attacks) [22], [4], or instruction set randomization (changing the interface of the operating system (OS) to an application, i.e. system calls for I/O devices, i.e. against code injection attacks) [68], [60].

4. dynamic platform: techniques changing platform properties. This includes CPU architecture and OS version, virtual machine instance, platform data format etc [68], [41]. Virtualization technologies are especially noteworthy in this category as an application service can be flexibly run from an active virtual environment, or a pool of backup instances with different OS/configurations such as to maximize diversity [47], [6].

5. dynamic networks: techniques modifying network properties like protocols, addresses or topology [68], [41]. Such MTD techniques aim to invalidate knowledge that an attacker may gain from reconnaissance attacks by periodically shuffling potential target addresses (network address shuffling) [9].

WHEN

Besides defining what system features to utilize as the MP, it must be decided *When* the current value of the MP should be altered. This means the points in time, respectively the frequency of moving must be determined [9]. Dependent on the MTD technique, this may heavily influence the performance of the system to protect. If the MP is not changed frequently enough, the attack surface is almost static from an attackers point of view. If the MP is changed within too short intervals however, the system might be harder to

attack, yet a considerable overhead is introduced, possibly deteriorating the availability of the system [9], [11], [12].

Clearly, the effectiveness of moving is related to the time of potential attacks. If there are no attacks, moving a target does not provide any clear benefit (except maybe deterrence). In an ideal setting, the target remains unchanged without any attacks, but moves with high frequency when there is an immediate threat of an attack [9]. Practically, there are three main types of solutions prevalent in the literature to trigger the MP value change: time-based, event-based, or a combination of the two [41]. For instance, time-based approaches alter the MP in fixed or varying intervals or at certain dates [9], [11] and follow therefore a proactive scheme. Event-based solutions may be triggered by an anomalous event and work in collaboration with an intrusion detection system or another attack detection component [14], [15]. These approaches are reactive by nature. A hybrid of time- and event-based schemes may work both proactively and reactively [72], [54].

HOW

Deciding on how the MP is exactly changed implies determining a selection and a replacement operation [9]. This means that in order to change the MP, first a value has to be selected from a defined set of valid MP states [9],[41] and then, the old value of the MP needs to be replaced by the new one [9]. Selecting the new MP value has been classified into three types of methods: diversification, shuffling (randomization), or general redundancy-based approaches [26],[14],[41].

Diversification-based MTD techniques employ system components that have different implementations yet are functionally equivalent [14]. System diversity can be realized based on a variety of domains. Azab et al. for instance, proposed a [5] code diversity MTD mechanism, based on exchanging variants of code components. [33] examines automatic software diversification on instruction-, program- or system level. Further, in [63] diversification of programming languages is used within the context of web applications to avoid code and SQL injection attacks.

Shuffling-based MTD techniques rearrange system or network configurations in a typically randomized fashion. The most prevalent approach is notably network address shuffling, where host IP addresses are altered periodically [28],[3],[11],[4]. Port-hopping marks a further example with the aim to hide service identities and thwart off reconnaissance attacks [38]. Shuffling techniques are not limited to network addresses, but may also include platform properties. In [64] for instance, an OS-Rotation scheme is employed, where virtual machines with different OS distributions and web applications are periodically exchanged for each other.

The remaining approach to select new values of the MP is called redundancy-based as it builds upon identical replicas of system components. In [36], the authors propose to protect cyber-physical system (CPS) sessions by keeping open a number of indistinguishable and redundant network sessions and disseminating messages via a randomized one.

2.4 MTD Techniques

As a starting point for this thesis, four different MTD techniques have been implemented at the University of Zurich, which mitigate malware from the three families explained in Section 2.2. In the following, the mode of operation of each of these MTD techniques is explained and classified according to the previously elaborated MTD design principles.

2.4.1 MTD against CnC - Private IP Address Shuffling

As the CnC channel is a single point of failure for sending commands to the victim and leaking data, parameters that may disrupt this channel are promising to use as the moving parameter in an MTD technique.

An option to break the CnC channel is to migrate the victim's private IP address to a new one such that the control server cannot reach the victim anymore. The proposed MTD technique works as follows [16], [12]: First, it creates a list of all the possible private IP addresses within the local network of the target device. Then, all addresses currently in use (obtained by the arp-scan command) are removed, yielding a list of available IP addresses. The target device then randomly chooses an address from this list and requests to migrate to it via the ifconfig command. Next, the MTD checks if the device can connect to the internet to decide whether the migration was successful. If the internet can be reached, the MTD restarts all required services (in the case of this work the ElectroSense service as this connection is disrupted as well) and the MTD is complete. However, if there is no connectivity, the requested IP is removed from the list of available addresses and a new address is chosen until the IP address migration was successful. There may be no internet connectivity in case that the previously requested IP address is reserved and belongs to an offline device that has not been found using arp-scan. But this just triggers a new migration request. The code for the MTD described can be found in [16]. This MTD technique operates on the network-layer and can either be used proactively, or reactively to disrupt CnC communication.

2.4.2 MTD against Rootkits

In order to sanitize a system from user-level preloading rootkits, all the malicious libraries or the links pointing to them need to be removed. Thus, *ld-2.24.so* has to be checked for whether it contains */etc/ld.so.preload*. If not, this sane file path has to be relinked [12]. Further, */etc/ld.so.preload* has to be replaced by a backup version, in case it has also been tampered with. This is sufficient to deal with the beurk or bdvl rootkits as decribed above, independent of their file hiding efforts. This MTD may be executed either proactively, or reactively upon detection of malicious activity. With using paths of linked libraries as the moving parameter, this MTD technique belongs to the runtime environment layer. The code can be found in [16].

2.4.3 MTD against Ransomware - Encryptor Trapping with Dummy Files

Typically, crypto ransomware encrypts files recursively by stepping through chosen directory structures. This and the following section elaborate on two MTD solutions that can be used against such an encryption approach. Both leverage data files as the moving parameter and therefore belong to data-layer MTD techniques. The code for both techniques can be found in [16], further explanations in [12].

The idea of this MTD technique is to trap the executing ransomware on the victim device by dynamically expanding and collapsing the directory tree with dummy files. Simultaneously the encrypting process is searched for and killed. As a means of mitigation, this MTD works reactively with the goal of keeping as much data safe as possible.

First, a directory is identified, where the ransomware is expected to do damage. The MTD then moves to a random sub-directory and therein creates a new subdirectory with dummy files. After a preset number of dummy files are created, again a new subdirectory is created and the process repeats, always moving one level deeper.

It is important to note that the dummy files are always created in a subdirectory two levels deeper as the current target directory. Because once encryption has already started in a directory, files that are added later cannot be read and encrypted anymore. The goal is to create dummy files at a path which is not yet in memory. At some point the ransomware will step into a dummy directory and gets stuck in there, because of the recursive file creation.

To make sure that no scarce disk space is depleted, the MTD further deletes dummy files which have already been encrypted. To make sure the process terminates, the MTD monitors all processes to find the encryptor and kill it. The monitoring is based on the CPU consumption as the encryption of files is per assumption an extremely CPU-intensive task. All processes below a certain threshold can be filtered out, just as well as whitelisted processes (here, i.e. the ElectroSense es_sense process). Subsequently, the remaining, suspicious processes can be checked for how many files are opened per minute. If the number is above a configurable threshold, the process is killed.

2.4.4 MTD against Ransomware - File Ending Randomization

This MTD technique is based on the fact that malware often targets files with specific extensions. By randomizing the file endings of important files, they can be hidden from this type of malware and kept safe from manipulation. Tested against ransomware encrypting specific files types, the technique works as follows: First pseudo file extensions are generated by randomly sampling alphanumeric strings. Next, selected file extensions are replaced with the random ones, while a directory is maintained, to track the mapping of real to generated file extensions. Clearly, the mapping is ensured to be one to one, such that for any pseudo file extension the original can be found. Finally, once the malware is mitigated, the complete file names can be reconstructed. Clearly, to keep critical data

safe, the file endings need to be changed before an attack happens. Respectively, in case of a recursively encrypting ransomware, file extensions are only worth changing if they are not yet read into memory. Concludingly, this MTD can be deployed both proactively and reactively.

2.5 Reinforcement Learning

RL is a major pillar of ML and is concerned with learning by interaction with an environment in order to achieve long-term goals [62]. Examples for such goals can be to make a robot learn how to walk, how to win a chess game or how to optimally deploy MTD techniques. The core idea of any successful RL method is that actions whose long-term consequences are beneficial with respect to the goal are being reinforced over time, while presumably unfavourable actions are inhibited and exchanged for better actions. First, general terms of RL are clarified, and the concept of policy iteration is introduced. Next, a brief overview of RL methods is given, and finally the combination of neural networks and RL as used in this work is highlighted.

2.5.1 Agent, Environment, Policies and Value

RL does not rely on sample supervision or complete environmental models, but instead follows a systematic trial-and-error learning approach within a specified framework of interaction. The main components of this framework are depicted in Figure 2.1.

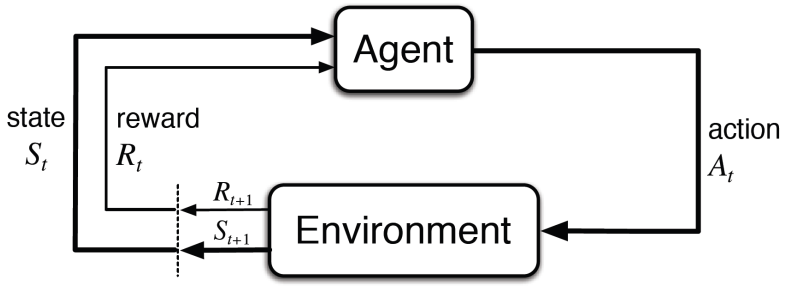

Figure 2.1: Agent-Environment Interaction [62]

The learner and decision-maker is called agent. The environment marks everything outside of the agent and is represented in the form of states. States are vectors of relevant environmental features observable by the agent. The agent cannot directly control the environment, but may indirectly influence it via actions. An action causes an effect in the environment, a state change, which can be translated into a reward, a numerical signal. If the action taken is in line with the long-term goal, a positive reward is yielded. In contrast, if the consequences of an action appear to be negative with respect to the goal, the reward will be minimal. The environment feeds its new state, as well as the reward signal back to the agent. Based on this new information, the agent selects the next action and this interaction loop repeats. If the interaction naturally breaks into sequences, because there

is a terminal state, where there is no more action possible (such as in case of success or failure to achieve the goal, i.e. winning/losing a game), the observed series of states, actions and rewards is called an episode. In contrast, continuing tasks refer to interaction patterns that do not have a clear terminal state and go on continually. In both cases, the states, actions and rewards are correspondingly defined by their time step t at which they occur.

Mathematically speaking, the goal of the agent maps to maximizing the expected return, meaning to maximize the expected cumulative discounted future rewards over all time steps: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} R_{t+k+1}$ Here, R_t denotes the reward at time step t, where γ corresponds to a discount factor.

In order to maximize this expected return G_t, the agent needs to learn a so-called policy. A policy is generally defined as a mapping from states to probabilities of selecting each possible action: $\pi_t(a|s) \forall s \in States, a \in Actions$. With experience these probabilities are shifted towards actions which lead to higher cumulative rewards.

Experience in the form of a sequence of observed rewards can be captured by a so-called value function. A value function maps a state to a value, the estimated expected return to be in that state. Clearly, this value depends on the policy the agent follows. Therefore, the value of state s under policy π is defined by: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ with G_t as defined above. The value of taking action a in state s under policy π is analogously defined as the action-value function: $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$. Phrased differently, value functions are estimates how rewarding it is for the agent to be in a given state, respectively how good it is to perform a certain action in that state.

2.5.2 Finding Optimal Policies

Ideally, the agent always selects those actions that yield the highest value in the long-run, which corresponds to acting upon an optimal policy. Correspondingly, a value function is said to be optimal if it assigns to each state or state-action pair the maximum expected return that is achievable by any policy. Hence, there is an interdependency between value functions and policies.

RL methods generally aim to find the optimal policy by iteratively and alternatingly estimating value functions and improving a current policy. This idea is called generalized policy iteration (GPI) and is shown in Figure 2.2.

Estimating the value function is called policy evaluation, or the prediction problem, as the goal is to predict the state/action-state values taking the current policy as fixed. The way the values are predicted heavily depends on the concrete RL method applied. Policy Improvement on the other hand is achieved by making the policy greedy with respect to the current value function. This means that a new policy is derived by utilizing the equation $\pi_t(s) = \text{argmax}_a\, q(s, a)$. Here the action value function is used, as the action presumably yielding the greatest reward can be directly determined. If only the state-value function is given, a complete transition model of the environment is needed to derive a policy improvement.

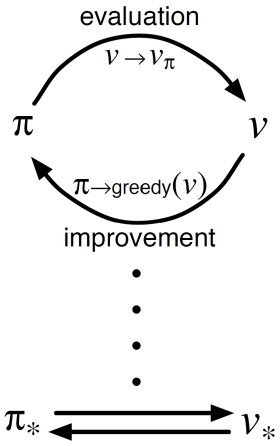

Figure 2.2: Generalized Policy Iteration [62]: π_*, v_* denote the optimal policy and value function. Note that the greedy improvement of the policy may either happen via the state-value function $v_\pi(s)$, or (more commonly) the action-value function $q_\pi(s, a)$

Where policy evaluation changes the current value function, policy improvement changes the current policy, but in the limit, with sufficient experience, alternating these two processes strives towards an optimal policy corresponding to an unchanged optimal value function.

2.5.3 Overview of RL methods

After considering the main components of the RL framework and covering the general concept of policy iteration, different RL methods can be contextualized. Where policy improvement usually does not depend on the concrete method, the means of policy evaluation differs heavily. The manner how an agent estimates the value function determines how it learns from experience. In general, policy evaluation aims to approximate a table of states and actions. This requires to find the value for each state-action pair.

If the state-action space is particularly small, the optimal value function may even be found by calculating the Bellman-Optimality equation[62]. However, this is a rare case and requires a complete model of the environment, as well as the rewards issued for all state-action pairs. Another, purely computational approach, which also requires a complete transition model is Dynamic Programming (DP). In DP, the value function is estimated based on the estimated values of successor states. This is called bootstrapping and is repeated until the calculated state values stabilize.

Another simple, yet effective way of estimating action values is to take the average of rewards received after observing each state-action pair as done by Monte Carlo (MC) Methods. In the limit, if a state-action pair is experienced infinitely many times, the average will converge to the true action value for the evaluated policy. MC Methods only require experience from actual or simulated interaction with the environment, and no model is needed. This is a huge advantage for tackling most RL problems. Compared

to DP, MC methods do not bootstrap from other estimates, but action values cannot be calculated before the end of an episode.

Temporal Difference (TD) Learning is another approach to estimate action values, combining ideas of DP and MC methods. In TD, value estimates can build on estimates of successor states (limited bootstrapping) and use immediately received rewards instead of waiting until the end of an episode (as in MC). This means that, as no model is required of the environment, learning updates may happen during episodes. TD is often faster in practice than MC methods.

If the state-action space is sufficiently small, DP, MC and TD methods (i.e. SARSA, Q-Learning etc.) can be implemented using a table with an entry for each action value. However, in many cases the state-action space is too large such that the value function must be approximated. The next section presents Deep RL as a solution and the relevant algorithm as used for this work.

2.5.4 Deep RL

The combination of Deep Learning (DL) and RL algorithms is referred to as Deep RL. Deep RL has paved the path to a series of impressive results in artificial intelligence, such as the achievement of super human level performance in Atari Games [40], Go [58], or Poker [7] and robot navigation [44]. In Deep RL, the action-value function is approximated via a deep neural network. This network outputs the q-values of available actions given a state, whereby the maximum-valued action is called the greedy action, belonging to the greedy policy. Figure 2.3 shows the accordingly modified agent-environment interaction loop.

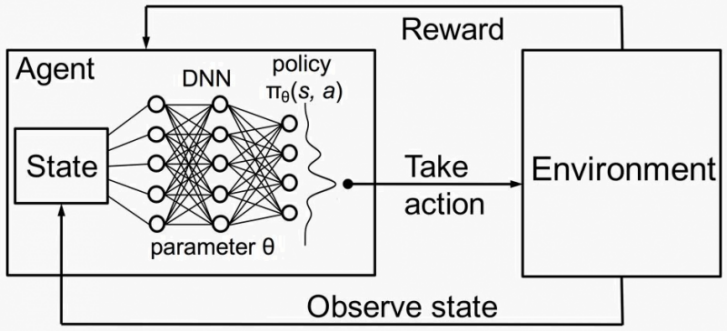

Figure 2.3: RL interaction loop with a neural network as a q-value function approximator. A probability distribution over actions is predicted by the network given a state as input. [66]

In traditional supervised DL, where data is labeled upfront, network parameters are learnt by performing gradient descent on the loss calculated as the distance of the networks predicted labels to the real labels. In Deep RL, however, target labels are not known. According to classic Temporal Difference RL methods, action-values are updated partly based on bootstrapping from other action values, and based on experienced reward. Analogously, the target q-values needed to update the action-value network are set to be based

on immediately observed reward, as well as the network's action-value prediction of the successor states. This poses an inherent difficulty to train a neural network in a stable manner, as the targets are based on the network itself. Notably, the key algorithm to overcome these problems which has first led to reasonably stable convergence results on high-dimensional RL tasks is Deep Q-Learning. Algorithm 1 presents a variant of Deep Q-Learning as it is used to train prototypical RL agents throughout this work.

Algorithm 1 Deep Q-Learning with Experience Replay

1: Initialize replay memory D to capacity N
2: Initialize online and target action-value functions Q^O and Q^T with random weights
3: Initialize exloration factor ϵ close to 1
4: **for** episode 1, M **do**
5: Initialize s_t
6: **for** t = 1, T (max timesteps within an episode) **do**
7: With probability ϵ select a random action a_t
8: Otherwise select $a_t = max_a(Q^O(s_t, a; \theta))$
9: *step*: Execute action a_t and observe reward r_t and state s_{t+1}
10: Store trainsition (s_t, a_t, r_t, s_{t+1}) in D
11: Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D
12: Calculate targets:
13:
$$y_j = \left\{ \begin{array}{ll} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma max_{a'}(Q^T(s_{t+1}, a'; \theta)) & \text{for non-terminal } s_{t+1} \end{array} \right\}$$
14: Perform a batch gradient descent step using $(y_j - Q^O(s_t, a_j; \theta))^2$
15: $s_t \leftarrow s_{t+1}$
16: Perform ϵ-decay to minimize exploration over time
17: **if** $tot\_steps \mod update\_freq == 0$ **then**
18: $Q^T \leftarrow Q^O$, update target net
19: **end if**
20: **end for**
21: **end for**

Algorithm 1 has a few key features. First, as can be observed by the nested for loop, learning happens over a number of episodes, each consisting of a certain number of action choices as well as state and reward observations (transitions). Each transition is stored in a memory buffer (line 10). Now, as the core of the learning procedure, the action-value network is updated based on targets y derived from a random-sample of all transitions stored in this buffer (line 11-14). In fact, in each step, learning does not necessarily happen based on presently observed state-action pairs, but on arbitrary samples replayed from memory. This is of crucial importance for decorrelating sequences of state-action pairs that often occur in reality (i.e. a lot of transitions from state z to state u happen during time window w). Without this decorrelating memory, gradient descent would update the network in an unstable manner making convergence difficult. The next key feature is the temporal difference target (line 13). Structurally, y_j exactly corresponds to the update in classic, tabular Q-learning. The target q-value is calculated based on the current reward r_j as well as the maximum-valued action (instead of the chosen one, determined by lines 7 and 8). The maximum-valued action is further weighted by the so-called discount factor

γ, which, as set between 0 and 1, determines the importance of future rewards. Another specialty in Deep Q-Learning is the use of two networks, a network used for the target calculation Q^T and a network used for choosing the current action and learning updates Q^O (online network). The reason for this is to have more target stability and thus, to have more robustness in the training procedure. After a certain number of update steps (line 17-19, $update\_freq$) the weights of the online network Q^O are copied over to the target network Q^T.

In summary, first there is an action choice given a state s_t. This choice is based on an exploitation-exploration tradeoff (line 7-8). Next, a state, action, next state, reward tuple is observed and stored in the replay memory. Next, a batch is sampled from memory and targets are calculated based on the temporal difference update, using a separate target network. The online network is updated with gradient using these targets as momentary labels. Finally, every certain number of steps the target net is updated with the online net's weights and the next action is taken or the next episode is started.

Note, that this procedure can either be executed in a simulated, offline-manner, or online in a real environment. There, the key difference for the algorithm is given by a single agent-environment interaction step (line 9). In online-learning, the interaction happens for real, whereas in offline-learning, the environment feedback is artificially constructed, given an action.

2.6 Related Work

This section presents related work done in the area of Reinforcement Learning, Moving Target Defense and the IoT with the aim of identifying aspects of interest that have not yet been considered.

2.6.1 Combining MTD and RL techniques

While there exist a number of MTD techniques applicable in the IoT, the current state of IoT MTD is still considered immature with a need to prioritize real-world scenarios [41],[14]. This work thus leverages four novel, and real-application tested IoT MTD techniques as proposed by [12] and aims to utilize them within an RL-based MTD Selection framework.

To the best of our knowledge, the usage of RL to select the correct MTD strategy among multiple MTD techniques as done in this work has not yet been studied in literature. In line with this, Cho et al. found in their 2020 survey on general MTD that it has not been investigated yet how to optimally deploy among multiple MTD techniques [14]. However, there are a number of works considering the application of RL to optimize a single MTD technique. Even so, in general these works do not focus on IoT devices.

For instance, [17] presents an RL-based approach to generate a desirably diverse and secure set of software configurations for general MTD. The authors formulate the MTD strategy

as a single player game using Monte Carlo Prediction and test their system's success of finding secure configurations as per fitness scores derived from STIG [sti], achieving 10-90% of the maximum fitness score.

Next, Chai et al. [13] present DQ-MOTAG, an anti-DDoS system combining Deep RL (DRL) and proactive network address shuffling MTD to block bot-like behavior. Their RL Algorithm adaptively adjusts the shuffling periods, meaning the WHEN of the MTD, and thus reduces network resource consumption, while maintaining defense performance. The system is evaluated in a simulated environment of jikecloud servers by measuring the number of blocked malicious actors in an exemplary setting.

[59] considers the application of MTD in Beyond 5G networks, requirements and a prototypical design incorporating a DRL component but does neither provide an implementation nor an evaluation. However, the author proposes DRL for MTD action selection based on continuous monitoring of the network state. There DRL is sketched to be deployed either in a single-agent setup with an MTD-Controller for proactive defense, or in a multi-agent setup with a game theoretic model including an attacker and a defender for reactive defense.

Multiple works have modeled RL for MTD as an adaptive multi-agent process between an attacker and a defender using game theory. For example, [20] presents such an approach formulated as a two-player general-sum game of an attacker and a defender competing for the control over a set of servers. There, Deep Q-Learning and the Double Oracle Algorithm is used to derive an optimal MTD policy. As by nature of purely game theoretic settings, the results are evaluated in a simulated, computational manner. By finding a mixed-strategy Nash equilibrium of an own utility function, the feasibility and stability of the approach is shown.

Similarly, in [57], the authors pursue a game-theoretic multi-agent RL approach. They leverage Bayesian Stackelberg Markov Games (BSMGs), which allow to model uncertainty over attacker types and MTD specifics and the BSS-Q Algorithm to learn optimal movement policies. [57] evaluates the system in a simulated web application scenario where databases and programming language are being used as the moving parameter. Comparing the rewards achieved by the agent for different approaches, the BSS-Q algorithm appears superior against baselines (i.e. an unadaptive, equi-probabilistic moving strategy).

Yoon et al. [69] also employ multi-agent DRL, yet for proactive, network MTD (IP Shuffling) against reconnaissance attacks in in-vehicle SDNs. Their system aims to minimize security vulnerability while maximizing service availability by choices on link bandwidth allocation and the frequency (WHEN) of triggering IP shuffling. The evaluation is performed on a proof-of-concept, in-vehicle SDN prototype with three agents executing different controlling tasks. The proposed algorithm outperforms different baselines regarding metrics such as rewards received, packet loss rate, etc.

Table 2.1 summarizes the most important aspects of related works described above and contextualizes the own contribution.

Table 2.1: Overview of Related Works (Devices: C-Computers, Operation of MTD P-Proactive, R-Reactive, Env/Environment: S-Simulation, R-Real-world application scenario)

Source / Year	Devices	Attacks	MTD type	RL Approach	Operation	Env	State Data (S) / Actions (A)	Rewards / Metrics	Performance Evaluation
[17], 2021	C	Unspecific	Unspecific	Classic Monte Carlo	R/P	S	S: system config. parameters A: config. change	Parameter (security) fitness scores (acc. to STIG), +1 reward if improved, otherwise -1	10-90% of the max. fitness score
[57], 2020	C	Web App attacks	Platform	Game Theory (Bayesian Stackelberg Markov Game), Multi-Agent RL Attacker-Defender Model, BSS-Q	R	S	S: system config. states (Def), A: config. change (Def) CVE list (Att)	Reward derived from est. impact of attacks (CVSS)	Convergence to max. reward plot. Proposed BSS-Q Algorithm achieves higher rewards over episodes than other algorithms
[20], 2020 [50], 2015	C	Control over servers	Unspecific	Game Theory, Two-player general sum game, Multi-Agent RL, DQ-Learning Double oracle algorithm	R	S	S: set of servers, each having a state indicating the nr of probes launched, who (Def./Att.) controls them and availability (up/down). A: reimage servers (Def.), probe servers (Att)	Reward function dependent on nr of servers under control (confidentiality) / servers up (availability)	Convergence to max. reward plot. Payoff table for different attacker/ defender strategy combinations (relative nrs)
[13], 2020	C / CPS	DDoS	Network	DRL-based adapting shuffle periods (reconnections of users and servers),	P	S	S: network state vector A: adapt shuffling period	Reward sinking with nr of DDoS attacks on proxy servers. Credit scores for user connections + blocking threshold	Exempl. setting of users and spys showing blocked/ non-blocked connections. Reduced nr of shuffling rounds maintaining security level.
[59], 2021	C	5G-attacks, DDoS, Spoofing, MitM	Network	Only Planning Stage: Single Agent (Proactive) Multi-Agent (Reactive)	P/R	None	None	None	None
[69], 2021	In-Vehicle: Sensors, Actuators, Controllers	Reconnaissance	Network	DRL, Multi-Agent recurrent deterministic policy gradient with anomaly detector (MARDPG-AG)	P	S	S: network traffic, packet loss, shuffling overhead and security vulnerability score of network slices A: IP Shuffling	Reward function dependent on bandwidth allocation efficiency, Security vulnerability and Packet Loss	Convergence to max. reward plot, comparison against baseline algorithms, Robustness against adversarial attacks
own, 2022	IoT / C	Rootkits, Ransomware, Command and Control, Backdoors	Data, Runtime Environment, Network	DRL and Anomaly Detection	R	R/S	S: On-device behavior fingerprints based on linux perf events A: MTD Technique Selection	Reward function dependent on correctness/effectiveness of selected MTD technique (binary).	Convergence to max. reward plot. Correct MTD technique selection accuracy up to 100%, Real-world resource consumption evaluation

Most evidently, as can be seen from the "Env"-column, there is a lack of research utilizing RL and MTD in real-world IoT scenarios. All of the works previously considered, evaluate the performance of their systems in a simulated setup without any clear connection to a real and running application. This work thus utilizes data from RP devices linked to ElectroSense to train and test an MTD deployment agent. Yet still there remains a simulation component as detailed in later chapters since complete online learning requires impractically more time and data. Moreover, most related works consider multi-agent setups relying on game-theoretic models that simulate attackers and defenders. The present work thus follows a completely different approach by performing single-agent DQ-Learning on real-world data. As noted at the beginning of this section, there is no existing literature that leverages RL for the deployment selection among multiple MTD techniques against a variety of attacks. This can be read from the columns "Attacks" and "MTD type" in Table 2.1. Only MTD techniques of a single type are considered. Regarding the type of state data and actions, related works often use a defined set of system parameters that the agent learns to move. This is in contrast to this work, that leverages real, on-device behavioral fingerprints as states and a range of different MTD techniques as actions.

Despite the differences, the related works show that Deep RL, and notably DQ-Learning, are feasible learning techniques for MTD-related control tasks. While the reward functions and consequently also the metrics used for the evaluation specifically depend on the RL task at hand, they hint at suitable options that can be made use of for this work. For instance, plotting the convergence to the maximum of an own reward function over a sequence of episodes. Where in most other works, no standardized numerical performance metric is presented, this work also utilizes the accuracy as principal metric for correct action choices.

Chapter 3

Problem Domain, Requirements and Assumptions

This chapter derives a set of requirements and implications of fitting MTD deployment into the RL framework based on a description of the problem domain as well as the findings from the literature review in the previous chapter 2.

3.1 Mapping MTD Techniques to Attacks with RL

Deploying MTDs proactively can be an effective measure against attacks, provided that the timing of the WHEN to move matches the attack execution. Thus, proactive MTD must trade off a higher security level (given by a higher moving frequency) against resource consumption or service quality impact. Certainly, it would be beneficial to launch MTD techniques more targetedly and alongside attacks. This makes reactive MTD deployment particularly appealing to study.

Coordinating MTD techniques in a reactive manner requires at its base that malicious behavior is recognized and matching MTD techniques are deployed for mitigation. As there cannot exist an MTD technique against all kind of attacks this implies, that some kind of mapping from device-, respectively attack behavior to corresponding MTD techniques is needed. In the case of the MTD techniques explained in 2.4 a clear mapping exists from techniques to malware families: The IP shuffling MTD mitigates attacks from the CnC and Backdoor families, the directory trapping and file extension manipulation techniques defend against ransomware, and the library sanitization MTD removes rootkits.

However, it is not straightforward to decide from an observed device behavior, which MTD should be deployed. This mapping problem is the main aspect of the MTD coordination task that RL aims to solve in this work. The deployment agent's primary goal is to learn WHAT MTD technique is correct for any given attack behavior state by interacting with a sensor environment.

Where deciding on a simple mapping from state samples to MTD techniques can in principle be phrased as a supervised learning task, this is not of concern here. The

goal is to create a framework that avoids the need for any labelling supervisor as far as possible. Consequently, and to remain in line with the MTD paradigm's aim to reduce the asymmetric nature of attackers and defenders, reactive, RL-based MTD should also make as few assumptions as possible about specific malware threats and patterns. Thus, the aspired RL-based MTD system should learn in the most unsupervised fashion feasible. This property makes the system promising to also deal with zero-day attacks in a well-defined manner.

The remainder of this chapter aims to provide an intuitive understanding of how to fit MTD deployment selection into the RL framework and is structured as follows: First, the main phases existing in the MTD process are analyzed to have a clear idea of where an agent can make decisions or receive rewards. Next, both malware-specific properties as well as peculiarities and limitations regarding the available MTD techniques must be taken into account as they could influence the RL process. Subsequently, the previous considerations are leveraged to formulate basic conditions for episodes within the RL training loop, and, concludingly, requirements are specified for integrating an MTD agent into a real-world scenario.

3.2 Phases in the MTD Process

On an abstract level, there exist three main phases relevant to RL in the reactive MTD process: Device behavior before, during, and after MTD execution. Figure 3.1 provides an abstract view on these phases as an RL agent may perceive them.

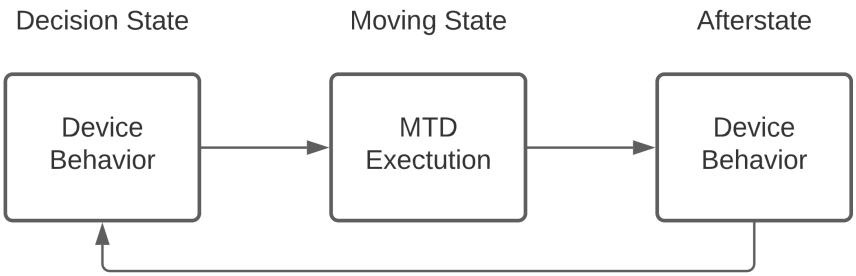

Figure 3.1: Phases in MTD

The first phase indicates a situation in which a decision must be made of whether an MTD technique should be deployed or not, and if so, WHAT technique. For an RL agent, this phase can be considered a `decision state`, meaning a device behavior state that is used for deciding which action to take. This action can be either to deploy a chosen MTD technique autonomously, or to send a deployment decision to another software component handling the execution. The second phase of the MTD process marks the time during which the MTD technique is executed. This `moving phase` is characterized by the time period where the MP actually transitions to a new value. This phase heavily depends on the concrete technique and may take from milliseconds to minutes or even hours. The third phase is characterized by a so-called `afterstate`, denoting device behavior that results after the MTD technique has finished executing, respectively the state after an RL

agent's deployment action. Ideally, this afterstate contains all the relevant information about the effect of the prior moving phase and in particular, whether the MTD was successful or not. As soon as the afterstate becomes available, rewards can, in theory, be calculated to reinforce actions for the MTD deployment agent. To adhere to the RL loop as described in Section 2.5, this afterstate alongside the rewards need to be fed back to an agent which decides upon the next step to take. This implies that the afterstate is turned into a decision state in the next iteration loop of the RL algorithm.

3.3 Environment, State and Reward Considerations

To learn to map attack behaviors to MTD techniques, an agent must not make any attack-specific assumptions, as this could limit its general learning capabilities. Therefore, the *decision states* considered by the agent as per the previous section, must cover a wide range of system features that may be relevant for a variety of potential attacks.

Such system features can either be monitored directly on the sensor device or via an external component that captures in- and outgoing traffic patterns. However, due to simplicity and independence of other devices it is desirable to monitor system behavior on-device. Additionally, features monitored on-device only are sufficient to capture patterns of various malware families, as both communication patterns (i.e. packets sent and received, TCP connections, etc) as well as other low-level device behavior (like cpu consumption, context switches, cache misses, etc) can be recorded at once. For example `Perf` as a lightweight performance monitoring tool included in the Linux Kernel provides exactly such capabilities for a range of different types of low-level device features.

The above described requirements for features provide hints as to how the RL environment must be set up. The low-level features monitored on-device should directly correspond to a decision state presented to the agent. This implies that an agent's environment must comprise the sensor device itself which at first might sound counterintuitive. However, the agent as a logical controlling component must only be capable of interpreting decision states of a sensor and launch corresponding MTD techniques. Therefore, it could in principle reside on-device or off-device. While this is a matter of system design, this work narrows the focus to on-device only agents to remain independent of any other external components that are typically not given in the ElectroSense scenario where individuals contribute flexibly and in a self-governed manner.

3.3.1 Reward Calculation Requirements

The way an MTD deployment agent interacts with the environment must be coupled to the behavior prevailing in the afterstate. The afterstate is the result of the agent's action, more precisely, the state after the execution of an MTD technique. This must be the base for calculating the rewards. In essence there are two cases: First, the case where a selected MTD technique was correct for a given behavior state, and second, the case where the MTD was not correct. Clearly, positive, reinforcing rewards should be fed back

to the agent for a correct deployment decision and penalties, respectively negative reward signals should be yielded for incorrect decisions.

But determining the correctness of an MTD technique is not a trivial task in general as the devices' behavior is not known upfront. However, patterns of normal behavior are known and measurable. Given some malicious behavior, in the case of deployment of the corresponding mitigating MTD technique, the afterstate should exactly correspond to normal behavior. However, in case of an incorrect MTD technique, the afterstate should exactly correspond to the previous attack behavior. Thus, a feedback statement for a deployment action can be made by measuring the distance of the afterstate to normal behavior. If the distance is very small, the MTD must have been correct and the agent should be rewarded. But if it is large, the afterstate deviates heavily from normal behavior and must thus correspond to anomalous behavior indicating an wrongful MTD deployment. In the latter case the agent must be penalized. In summary, the calculation of rewards requires a component that interprets and judges the afterstates for normality.

Note that afterstates only correspond exactly to normal behavior in case of perfectly working MTD techniques. This assumes that defense works completely, killing all parts of a given malware that influence device behavior. To assess whether this assumption is valid, the malware families and MTD techniques considered must be scrutinized more thoroughly. Chapter 4 as well as the next section deal with this matter alongside other concerns.

3.4 Malware and MTD-Specific Properties

This section elaborates on properties of both the malware and MTD techniques considered that might impact the RL process. The MTD technique against rootkits runs only milliseconds and effectively sanitizes the device from malicious libraries. Thus, no special assumptions need to be made to learn how to deploy rootkit MTD with RL. But ransomware as well as CnC/Backdoor malware and the corresponding mitigating MTD techniques must be examined more closely.

3.4.1 CnC

As described in Section 2.2, CnC malware usually employs a client script running on the victim machine as well as a server script that enables to send commands to the client/victim. The IP shuffling MTD technique effectively disrupts the communication channel, but it cannot free the victim device from the client script. Thus, dependent on what this client part does, it might influence the devices' normal behavior. In other words, the afterstate might not correspond exactly to normal behavior even if the MTD was successful. This theoretically breaks the assumption of perfectly working MTDs made in the previous section, but it might not be a problem in practice. First, the remainders of the CnC client scripts cannot accept commands and thus may have negligible effect. And secondly, in order to provide the correct reward signal to an agent, the afterstate

does not necessarily need to correspond exactly to normal behavior. It only needs to be sufficiently close to normal behavior, such that it is not flagged as malicious behavior by an interpreter of the afterstate. Yet clearly, not having to deal with such remainder client scripts simplifies the training process.

3.4.2 Ransomware

Crypto ransomware usually encrypts data of specific target directories instead of locking a user from the entire system. This means that for both the MTD techniques against ransomware explained in Section 2.4, the target directory must be known such that they achieve the promised benefits. The directory must be specified where the file tree is expanded for the directory trap to capture and isolate the encrypting process. Similarly, the file extensions must be changed in the target directory before the ransomware starts encrypting the files. Consequently, the MTD against ransomware is only effective, if it moves the right data at the right time.

This has consequences for training an RL agent. First, if the moving phase starts too late even in case of a correctly deployed MTD, the malware might have encrypted all the data already. Stated differently, it is very challenging to reactively deploy an anti-ransomware MTD with success if the target directory to encrypt is fairly small in size and the encryptor thus runs only shortly. Further, as the runtime of the ransomware is only temporary, the device behavior corresponds exactly to normal behavior once the target directory is encrypted. It can only be distinguished from a normal state in the sense that a number of files or directories are unintentionally encrypted on the target device, but not with respect to any malicious processes being active. While this can be used to penalize the agent, it would be useless to redeploy MTD techniques if the ransomware has already finished executing. Hence, the decision states taken into account by the agent should only consider ransomware that is still actively in the process of encryption. This way it still affects low-level device features and launching an MTD actually helps to mitigate the attack. This assumption that theoretically temporarily executing malware can be dealt with similarly to non-temporarily running malware is necessary for the RL interaction loop to work in a well-defined manner without a-priori knowledge about the malware. Otherwise, additional malware-dependent assumptions regarding affected features would be needed. Anyhow, in the case of ransomware, this assumption is legitimate as usually an encryptor does not just target a very small directory on the victim, but instead tries to encrypt larger parts of the file system implying a much longer execution time.

3.4.3 Summary of MTD and Malware Assumptions for RL

In order to make an MTD deployment system fully autonomous with RL, an interpreter is needed that checks whether the afterstate corresponds to normal behavior, to give positive/negative rewards for correctly/incorrectly deployed MTDs. This system can only work for sure, if the normal system behavior is not altered sustainably by neither the attacks, nor mitigating MTD techniques. This includes the assumption of perfectly working MTD techniques. This assumption does not necessarily hold for all types of attacks (i.e.

mitigating CnC malware with the IP shuffling MTD may keep client parts of the malware active), but it is still valid if the impact to normal behavior is negligible (i.e. no work done by a passive CnC client script) or potential sources of impact are removed (i.e. if the CnC client process is killed by rebooting, or else).

There is malware executing either continuously (i.e. CnC, rootkits) or temporarily on the device (i.e. ransomware). With regards to detecting malicious behavior, the RL agent must assume that malware processes are still actively running to make any useful choice of action. Otherwise, if malicious processes have already finished, the agent cannot observe effects from taking deployment actions.

The MTD techniques explained in 2.4 have different execution durations, meaning that their moving phases are not equally long. The MTD against rootkits takes only milliseconds to finish, the technique against CnC runs for a few seconds at most, and the runtime of the ransomware MTD depends on the target directory size (minutes to hours). This impacts the RL process as afterstates cannot be always observed in the same time window for different MTD techniques. This results in an online agent receiving feedback at different times after its action choice. This is highly relevant for the implementation and application, but is not assumed to represent an obstacle for applying the RL framework itself.

3.5 RL Framework Implications

3.5.1 Need for Deep RL and Temporal Difference

As per Section 3.3, the feature space used for both the decision- and afterstates should be sufficiently large to capture the low-level effects of a variety of different malware. Further, features such as tasks running, packets sent and received or memory information cover a wide range of different values that differ significantly dependent on the processes running on the system. This means, that the whole state space presented to the agent is extremely large. This creates the need for approximating the action-value function instead of learning it exactly. Concludingly, a purely computational method like Dynamic Programming can be excluded right away for an algorithmic approach. Instead, Deep RL, is a promising approach as it allows to approximate the action-value function interactively via a neural network.

In Monte Carlo methods, rewards only become available at the end of episodes. Accordingly, a neural net could only be updated at the end of an episode. This, however, would increase the required learning time for an agent and could make the training process more unstable. Temporal Difference Learning is more desirable in the given setup as the neural network for the action-value function can be updated after every action the agent takes, making the training process faster. Concludingly, the Deep Q-Learning Algorithm (see Algorithm 1) is an adequate choice for the MTD selection task at hand, as it utilizes temporal difference and accounts for large state-spaces. Further, it allows to learn based on randomly sampled transitions from a replay memory, which ensures that frequently ocurring sequences of attacks and actions are decorrelated.

3.5.2 Learning from Experience: Framing Episodes

Such that an MTD deployment agent can learn, it requires experience in the form of episodes. An episode can be defined as a sequence of agent-environment interaction tuples: $E = \{(S_t, A_t, S_{t+1}, R_t, Done)|t \in T\}$. Thus, an episode reflects a series of the agent decisions, successor states and rewards as per the feedback provided by the environment. *Done* is a binary variable indicating whether an episode ends, T is the set of all time steps of the episode. The beginning of an episode is naturally given by the observation of an attack behavior state. This can also be indicated by some interpreter that compares the current device behavior to normal behavior. If there is an attack, the agent should start deploying reactive MTD techniques until it launches the correct one. Thus, an episode ends, if the mitigation was successful and the device behavior is restored to normal. Figure 3.2 depicts a simplified state machine for this episodic pattern.

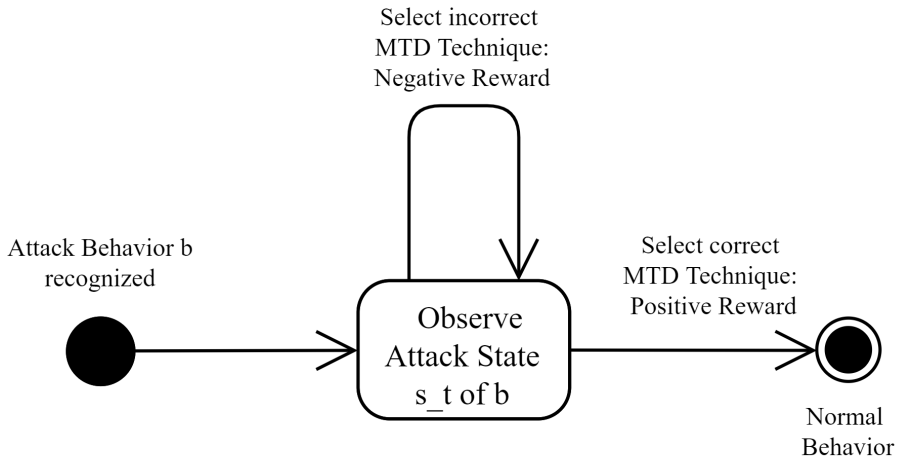

Figure 3.2: Episode State Pattern FSM: Repetitive observation of the same attack until mitigation

This pattern can be repeated over and over, by detecting malicious behavior on the device, deploying MTD techniques until mitigation and learning from the environmental feedback until, ideally, the agent converges to a point where it chooses the correct MTD for any given attack state with high accuracy.

Here it becomes clear, that an episode can only end regularly, if there is actually an MTD that mitigates a given attack. If there is no reactive MTD available, the agent could get stuck deploying useless MTDs. However, this problem can be avoided by simply limiting the number of possible MTD executions within a single episode and upon exceeding them a warning can be issued gracefully. That way, no special considerations need to be made for attacks that cannot be defended against with the current set of existing MTD techniques. For the training process it can simply be assumed that there is always a correct technique available.

Theoretically, it is possible to deploy multiple MTD techniques at once. Certainly, this is a waste of resources in general. But additionally, this would complicate the training process extraordinarily since it is not clear how effects can be attributed to individual

MTD techniques. Therefore, the agent should only learn how to deploy a single correct MTD technique at a time. If there is another one running already it must wait and observe afterstates first.

3.5.3 On- versus Offline Learning: Simulation for Pretraining

In spite of the achievements of temporal difference learning, it still requires thousands of episodes to make an agent's action-value function network converge to practically useful accuracy. This is even more prevalent for the MTD coordination task since the state space is very large. In the early phases of the learning process an agent's action choices are usually not much better than selecting randomly. This implies a very poor performance of an unsufficiently trained agent.

The convergence directly depends on the number of episodes, respectively the number of feedback loops completed. However, observing a very large number of episodes in a real environment is extremely resource-intense and time-costly. As per the previous section, the device would need to be attacked, MTD techniques would need to be launched and afterstates observed until normal behavior is restored, just to start new episodes over and over again. Being attacked this many times in a real environment is neither suitable, nor practical for training an agent. Further, the speed of convergence also depends on the speed of the feedback derived from the afterstate. If MTD techniques only run for a very short duration, fast feedback loops are guaranteed, accelerating the agent's progress. However, in the case of MTD techniques executing for longer periods of time, the feedback only arrives very delayed at the agent. Both the need of frequently observing new attacks as well as the possibly prolongued exectution times of a single episode pose a severe limitation for learning an RL-based MTD deployment agent from scratch in an online manner. Dependent on the speed of the feedback loops it may take weeks or even months of constant episode run-throughs to reach an effective and efficient deployment system.

Certainly, if there are no other options, a poorly trained (close to random) agent is still better than nothing. In the worst case, such an agent just tries out the whole set of available MTD techniques, possibly deciding that the attack cannot be mitigated with the existing set.

However, often online RL in a real-world system is still not feasible from scratch. Especially in the case of IoT devices, it might not be possible to waste this many resources for trial and error MTD deployment. Dependent on the resource constraints of the device it might also severely degrade the service level of a live system. This could also be one of the reasons why related work as studied in Section 2.6 has neither tackled online learning nor online evaluation in real-world live systems. However, the effects of on-device RL should be examined more closely for any given system at hand in order not to make prematurely limiting assumptions.

Nonetheless, a more appealing alternative to full online learning from scratch is to pretrain an agent offline in a simulation to make it achieve a decent performance on a testbed before actually deploying it on a real world system. This way resources can be saved and longer feedback-loops and training time of online learning can be circumvented. However,

in order for the simulated training to produce useful performance for a later real world application, the simulation must be as close as possible to the real setting. The policy learnt in the simulation can only be transferred to the real environment, if the simulated environment data matches the states that an agent would observe in the real setting.

Where in online learning, time and practicality are limiting factors, offline learning in simulations still has huge data requirements. In fact, environmental data, states and episodes must be crafted in a way that makes it transferrable to a realistic online setting. This requires the collection of environment data for the full range of possible states. Stated differently, it requires to monitor data from the different phases in the RL-based MTD process as introduced in Section 3.2. This means that data from decision states, as well as afterstates (and possibly also the moving phase) must be monitored for a sensor device without doing the full online training. Provided that there are a number of different malware types and mitigating MTD techniques available, the sensor devices can be infected with malware (decision state), MTD techniques can be executed (moving state) and results observed (afterstate) all along the execution of a script which monitors low-level device behavior. The subsequent Chapter 4 provides more detail on the environment data and the collection procedures relevant for data used in both simulated, offline- as well as online-learning.

During training the agent must observe a large and balanced range of different types of malwares and MTD technique combinations (state-action pairs) to achieve good convergence results without any bias towards certain attack-mitigation combination patterns. This is a further advantage in offline learning, as there episodes can be sampled randomly, thereby guaranteeing a sufficient degree of attack exploration. In a real online setting an attack might be launched over and over again, whereas other attacks hardly ever occur in the device, which possibly impacts an agent's generalization capabilities. Even though, this problem can be reduced by random sampling from a memory buffer to train a DQN, in offline learning, the problem is less prevailing. More details that must be taken into account for creating effective MTD deployment simulations are postponed to the subsequent Chapter 5 since they depend on the specific goals of the prototypical RL agent at hand.

3.6 Required System Components and Properties

3.6.1 Online Agent Functional Requirements

Striving towards a fully working online, RL-based MTD deployment agent requires to integrate the logical MTD decision-making component onto the platform of interest, which is a RP device equipped with a sensor and linked to ElectroSense in this case.

Based on the preceding analysis of the problem domain in this chapter, the following minimally required functional components can be deduced:

1. A monitoring component, which collects a wide range of system features that are collectively influenced by malicious processes running on the device.

2. An anomaly detection component, which interprets the data collected by the monitoring component and recognizes ongoing suspicious behavior.

3. All MTD techniques, readily available as executables or scripts.

4. The core RL agent itself, which uses the monitored, suspicious data (decision states) to decide upon and launch an MTD technique and learn from monitored, and correspondingly interpreted afterstates.

5. A controlling component, which orchestrates and aligns all the previous components.

These components are sufficient to start learning from scratch in an online manner. However, as explained in the previous Subsection 3.5.3, in order to achieve an actually useful online agent within the given time constraints of this work, there is an additional requirement of a component simulating the environment. Ideally the agent can then be pretrained in the simulation, employed in the real-world and make useful MTD choices right from the beginning. Thus, the construction of suitable environment simulations takes also an important role in this work. However, independent of whether the environment is simulated or not, Figure 3.3 summarizes this work's primary task of learning by interaction how to select MTD techniques to defend against malware of the CnC-, rootkit- and ransomware families.

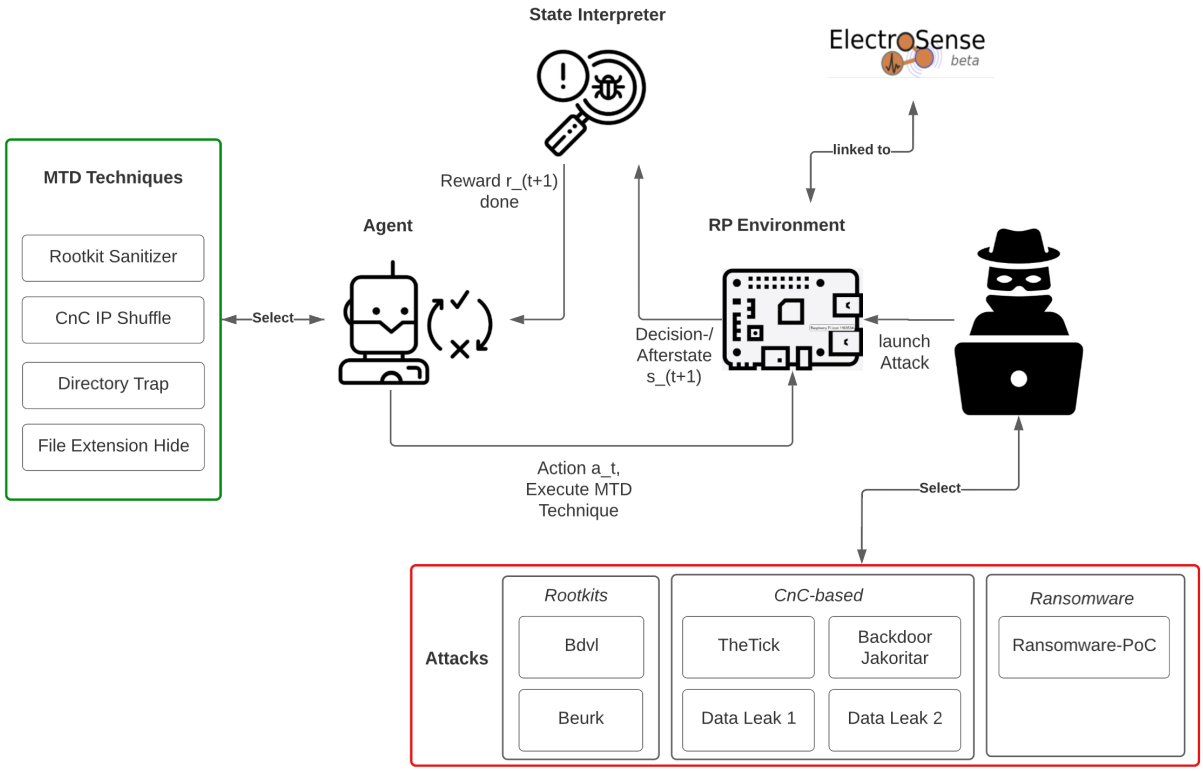

Figure 3.3: Agent-Environment Interaction Loop for RL-based MTD Selection

3.6.2 Security and Efficiency

Clearly, the main security goal of an MTD deployment agent is to effectively deploy MTD techniques which mitigate observed attack behavior.

Thus, the agent's action-value function approximator must learn to generalize from experienced behavior states to new, unseen states. In other words, the function's parameters must learn to capture the patterns relevant for attack classes and consistently map respective sample states to mitigating MTD techniques. The principal metric to assess the quality of this state-action mapping interactively learnt by the agent is the accuracy. This corresponds to measuring the percentage of correctly, selected MTD techniques using the agent's greedy policy, given a series of states of priorly known attack classes. The greedy policy corresponds to choosing the highest valued action (by the function approximator), given a state. The percentage of correct action choices clearly dominates the security properties of the system. Continuation of service, efficiency of mitigation and other properties which are desirable and relevant for the IoT in general and ElectroSense in particular directly depend on the policy learnt by the agent. This work thus mainly focuses on the effectiveness of the policy to assess system security. Certainly, the viability of the whole system, may also be heavily influenced by the anomaly detection component's estimates of the afterstates to be normal behavior. Thus, its accuracy should also be assessed accordingly on both decision and afterstates.

In addition to the malware detection and the MTD technique selection capability, resource consumption should also be considered to assess the feasibility in the IoT scenario. Especially important dimensions are given by processing times, disk space as well as CPU and memory requirements. The processing times by the agent are relevant to assess attack mitigation speeds in a real scenario. Disk space requirements should be validated to show that the code and the ML models actually do fit on the target sensor device. Lastly, it must be shown that the CPU consumption of the RL agent as well as its memory and learning requirements do not exceed any hardware limitations of the sensors.

Chapter 4

Environment Data

In essence, learning what MTD technique to deploy at any given time poses two main challenges: First, to separate normal from attack behavior, and secondly, to distinguish different malware families. Consequently, the quality of states and their range of features considered by the agent are of primal importance for learning to occur. This chapter aims to give an overview of the data leveraged for environment states and its relevant properties.

It heads off by introducing the sensor configuration and the setup of the environment, as well as the exact features considered for states. Next it provides general information about the data collection, with a special focus on data needs of realistic simulations. Subsequently, an initial dataset of raw behaviors is explored in two ways: First, by visualizing the variation of the features over time, in order to validate whether the dataset can be used for RL simulations. Secondly, the features are inspected with respect to their distributions, to point out differences for the range of considered malware and normal behavior. The chapter concludes with procedures applied for data preprocessing that will be relevant for the agent prototypes presented in the following chapters 5 and 6.

4.1 Sensor Configuration and Feature Sources

4.1.1 Sensor Configuration

For the collection of data, a sensor setup as described in Section 2.1 is considered. To facilitate testing, 3 RP devices equipped with an antenna were preconfigured with an image of RP OS 9 (32-bit Stretch, Debian/Linux-based) that runs all ElectroSense services from start up. Concretely, two RP 3 Model B+ with 1GB RAM and one RP 4 Model B with 2GB RAM were used. However, throughout this work, results are only presented for the RP 3 due to the availability of two devices and thus the increased speed of testing. While some experiments have also been conducted for the RP 4, as they generally led to equivalent results, they won't be shown here. All sensors have been installed indoors near Zurich, Switzerland.

4.1.2 Features: perf Events

As per Section 3.3, the data utilized should cover a wide range of system features to cover a variety of potential attacks. In order to collect data with such features the Linux `perf` command is used. `perf` is a powerful and lightweight profiling tool specific to the kernel and can be leveraged to monitor a range of in-device event sources[18], [48]. Thus, it provides adequate flexibility to record behavior patterns on RP spectrum sensors.

Event sources

Figure A.1 gives an overview of the event sources covered by `perf` [25]. There are six types of event sources, of which only two are considered in this work: Software events, as well as kernel tracepoint events. These two types of sources are readily available for monitoring and cover an extremely wide range of events, which make them promising for attack detection. Software events are based on low-level kernel counters. This comprises context switches and CPU migrations. Kernel tracepoint events are static kernel-level instrumentation points that are hardcoded in relevant logical places in the kernel. Examples of such events originate from the virtual memory (kmem, writeback), the block device interface, the system call interface or the scheduler. The `perf stat` command output counts for these events, which can be used as state features.

4.2 Data Collection

The function of gathering data from a sensor environment has two distinct purposes in this work. The first and most straight forward reason for data collection is the provision of state information to an online agent. But, due to learning time constraints, pretraining an agent in an offline manner is desirable (see Subsection 3.5.3). Thus, the second purpose of data monitoring is to create a dataset, suitable for offline-learning. The main goal of such a precollected dataset is to be utilized in a simulation that mimics the online-environment. The idea is to sample behavior states from this dataset in a manner that corresponds to the sequences of states that an online agent would observe. For transferrability reasons, for both cases of data collection for online- and offline-learning, the same procedure is applied. A shell script is launched on the device which records the event count for all the features at a fixed interval of 5 seconds. This series of feature vectors is stored in a CSV file for further processing. Analogously to the work done in [55] by Sanchez et al., 75 different `perf` events are monitored, belonging to 8 different event families. Figure A.2 visualizes all the features and their respective event family. For testing purposes, a few selected features of interest have been added regarding CPU usage, tasks running as well as packets sent and received. Although indirectly they may already be covered by the selected `perf` features, these additional features may help improve the ML/DL models to learn. Table A.1 and A.2 list the full range of features monitored. The full code of the script used for monitoring as well as all collected datasets are available at [56].

4.2.1 Crafting Realistic Simulation Data

From an application perspective, an RL-Agent trained in a simulation is only as useful as it allows to transfer the learnt policy to the real-world setup. If the data used for states in simulated pretraining does not correspond to what a real online agent would observe, inevitably a policy different from the optimum policy of the online setting will be learnt. Thus, as also stated in Chapter 3, the degree of similarity of simulated environment data, to states in a real environment is crucial for policy transfer between the offline- and online setting.

Hence, the question is how the data of a full-fledged online-learning agent looks like. The goal is then to collect a dataset that is as close as possible to such data and to feed it into a kind of episode-generator simulating the environment in offline learning. Here, it becomes critical that the `perf` features explained above are inherently susceptible to influences of processes running on the device. Scripts additionally running next to the default ElectroSense services may substantially alter the behavioral fingerprint of the device given by its `perf` event counts. While this may be beneficial to detect malicious processes, it also poses a difficulty to monitor realistic simulation data. Since the sum of all running processes affects the event counts, a fully functionally-working online agent is required upfront and must be run in parallel to collect a dataset suitable for realistic simulations. Essentially, this means that all the components as per Section 3.6.1 would need to be actively included in a data monitoring setup.

Besides the processes running in total on the device, the environment data is also influenced by the stage of the current processes. An agent that undergoes a cyclic pattern of decision state monitoring, MTD choice and deployment, as well as afterstate monitoring, interpretation and learning can therefore be expected to show slightly different patterns of event counts dependent on the stage of execution and the current stack. So independent of the current device behavior, decision and afterstate data could in theory be slightly different. This remains to be verified by inspecting the data monitored at these two stages. Further, it is not transparent whether and how certain types of malware or MTD techniques alter device behavior for extended periods of time despite mitigation.

The previous considerations show that technically it can be a very difficult task to craft fully realistic simulation data. Without further analysis it is not clear how the features are impacted by different stages of MTD mitigation as well as different kinds of malware or agent processes running on the device. Additionally, it is not per se transparent how the ML/DL components are influenced by these types of variation in the features.

This imposes the need for extensive exploration of both potential simulation data, and of different setups for training the RL agent. Thus, the upcoming prototyping chapters give more insight into the general learning capabilities in different simulated settings. These chapters will also go more into depth of how different kinds of simulation environments can be constructed with the goal of iteratively striving towards a more and more realistic setting.

However, at this stage and as a starting point, the simplest possible way to generate simulation environment data is to attack the sensor with a given malware and to record

the `perf` event counts as explained in the beginning of this section. This can be repeated for all the malware families considered in this work to gather an initial dataset. First, in order to attribute variations in the data to the malicious processes exclusively, no further non-default processes are executed on the device, i.e. no agent controller code is executed in parallel. The next section explores such *raw* data of different malware families regarding their stability over time and compares the distributions of these different types of behaviors.

4.3 Data Exploration: Comparison of Raw Behaviors

4.3.1 Dataset Dimensions

The following sections focus on a dataset created using a RP 3 Model B+ with 1 GB RAM. In addition to normal behavior, it contains data for malware belonging to the families ransomware (ransomware_poc), rootkits (bdvl, beurk) and CnC/Backdoors (backdoor_jakoritar, thetick, 2 data leak attacks) as explained in Section 2.2. Concretely, for each type of behavior, the monitoring has been run for a minimum of 8 hours, resulting in a total number of 59004 unfiltered, raw data points. Table A.3 gives an overview of the dataset and the numbers of samples for each monitored behavior. Analogously, Table A.4 displays the size of a smaller dataset created using a RP 4 exhibiting comparable properties.

4.3.2 Variation over Time and Feature Distributions

In order to assess the usefulness of the data for RL-based MTD, it is analyzed with respect to two main characteristics. First, regarding the stability of the features over time and secondly, regarding the distributions of the features. The code used to generate all the different data visualization plots hereafter can be found in [56].

In theory, RL algorithms can to a certain degree cope with non-stationary data, meaning that the agent may learn to adapt to changing environments given enough training. However, precollecting such changing data to be used in realistic simulations is extremely cumbersome, if even possible. From a practical perspective it is much more desirable to deal with stationary data, which is stable over time. This is an especially beneficial property for creating simulated environments as it allows to randomly sample behaviors for simulated states from a precollected dataset irrespective of any time step. But certainly, random state sampling is only a good approximation of reality, if the dataset used has negligible temporal dependencies. To visualize that this holds true for this work, the event counts of each collected feature is plotted against the timeline. As an exemplary proof of the stability of the features over time, Figure 4.1 shows the counts of the *kmalloc* event belonging to the virtual memory family. It can be observed that the different attacks cover a slightly different range of values and remain constant within that range except for a few outliers.

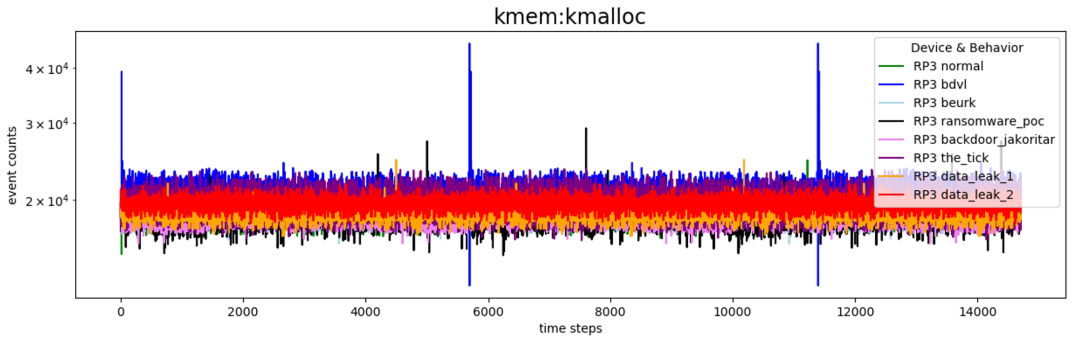

Figure 4.1: Feature timeline for different malware families for the virtual memory event source and kmalloc event

Other features, like the tasks, or RAM currently used (monitored via the Linux `top` command) are not as stable over time, showing cyclic increases and drops (see figures A.3 or A.4). Such features are excluded for building the environment due to their intransparent temporal dependencies. A clear example for this is given by the `tasks` feature, which counts the number of total tasks in the table of processes. This number is influenced by the number of zombie tasks (finished, but not yet removed from the table of processes), which depend on earlier system states. Dependent on previous computations, it is thus possible that normal behavior has a much larger number of tasks than certain malwares, making it an unsuitable environment feature.

However, plotting the timeline of all the features has shown that most of them are stable within a certain range, and that different attack behaviors often occupy slightly different ranges of event counts. Yet, as the variation over time can make it a bit fuzzy to visually distinguish different behaviors, the kernel density is estimated for each feature distribution in the available dataset and plotted accordingly. The distribution of the features is important to assess the overall potential of statistical models to learn a separation boundary. Further, it can help explain performance results of the agent. Certainly, the more distinct the distributions are, the higher are chances for the agent to separate the different behaviors and correctly map MTD techniques.

As a means of comparison to Figure 4.1, Figure 4.2 displays the distributions of the *kmalloc* feature for the different behaviors. The plot shows that for instance the backdoor jakoritar attack and beurk largely overlap with normal behavior. This suggests that these attacks may be harder to learn compared to attacks with more pronounced differences like bdvl or the second data leak attack. However, other features show different patterns as determined by the unique properties of the malware at hand. For instance, Figure 4.3 presents the distributions of the *block_getrq* tracepoint. It can be observed that there, different malware show much more distinct patterns. For this event, ransomware, the first data leak attack and bdvl seem to be the most distinct from normal behavior.

The previous data analysis has shown, that the features are (with few exceptions) sufficiently stable over time such that randomly sampled data points can be used as states in simulated, yet approximately realistic environments. Thus, the collected raw behavior dataset is suitable for prototypical applications of RL to the reactive MTD coordination task. Further, the distribution of the features indicates that normal and attack behav-

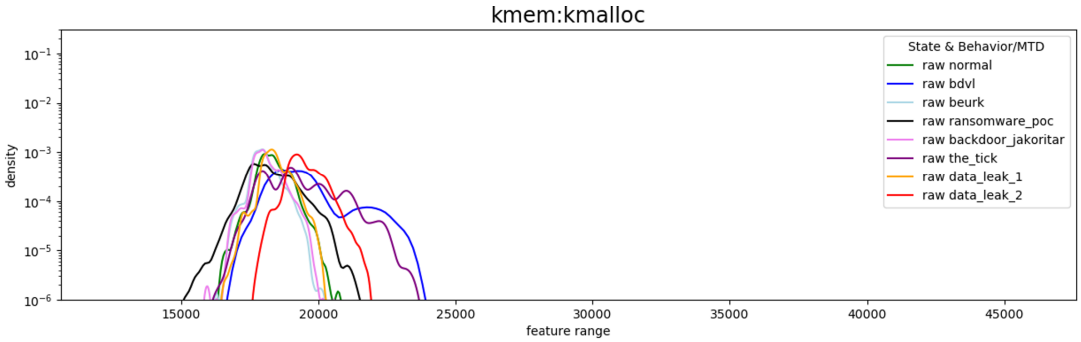

Figure 4.2: Feature Distributions of the *kmem:kmalloc* Event

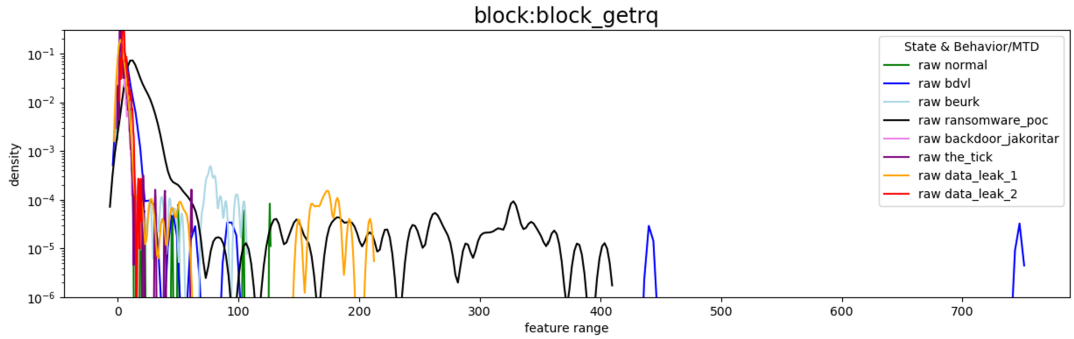

Figure 4.3: Feature Distributions of the *block_getrq* tracepoint

iors are in principle separable, but certain attacks (i.e. beurk/backdoor jakoritar) might be more difficult to learn to defend against compared to others (ransomware, data leak attacks).

4.4 Data Preprocessing

This section provides an overview of the data preprocessing steps that are relevant for all subsequent chapters.

4.4.1 Cleaning and Scaling

Before the datasets collected are fed into any RL process, irrelevant or otherwise inappropriate columns and samples are removed. First, columns that are constant throughout all types of behaviors are dropped as they provide no benefit with respect to learning a separation boundary. Further, columns that only refer to a current status are left out. These include timestamp or connectivity information. Note that these time status columns can be dropped due to the limited variation of the features over time shown above. Table A.1 and A.2 list which concrete feature columns are constant or status information and thus filtered. The tables show a further `Exluded` column, with selected features that are not used as well as explained in more detail in Chapter 6. All other event counts are used

throughout this work as input features for the agent and anomaly detection components as derived in Subsection 3.6.1.

Regarding sample filtering, the goal is to minimize the impact of possible external events and to clean the data from outliers. Therefore, data is only considered if the device is linked to ElectroSense. If the device is not connected, it cannot fully correspond to the behavior that should be considered normal. Next, the samples of the first five minutes as well as the last sample in the monitoring are removed. This is because the monitoring is launched via a secure shell (ssh), and potential noise introduced by logging in and out of the device should be limited. Cleaning the remaining data from further outliers is essential for both RL and anomaly detection training. The reason of these outliers is generally hard to determine, because the device can be influenced by various external or internal anomalous events. For instance, hardware problems to read from the SD card, power failures or network delays and other events might notably impact the features. Independent of any potential reason, outliers are filtered based on the Z-Score [34] throughout this work. The Z-Score is computed as $\frac{x-\mu}{\sigma}$, where μ denotes the mean and σ the standard deviation respectively. Data points of an absolute Z-Score ≥ 3 in any feature are not included in any training or validation set used for model training.

Figures 4.2 and 4.3 show that the feature distributions vary considerably across different behaviors. Thus, to ensure that all features influence the training process by the same scale, a scaling is needed which does not make any limiting assumptions on the feature distribution. *MinMax* Scaling does offer that property and is computed by the following formula for each feature i in a given training set:

$$input_i = \frac{(value_i - min_i)}{(max_i - min_i)}$$

This ensures that all features in the training set are scaled to a range between 0 and 1. However, the scaled feature values in the test set may exceed this range as the features may have different minima or maxima. At this point it should be highlighted that it can be very challenging to perform adequate scaling in Deep RL tasks. If data is observed only interactively from the environment, scaling is very difficult right from the first observations. There is an upfront need to have sufficient behavior data to get minimum and maximum feature values that enable good scaling for later learning processes. Experiments have been conducted with different ways of scaling, such as using only normal behavior to make no assumptions about attack behavior feature values, or scaling based on all behavior data leveraged within an offline learning simulation. However, the exact way of scaling will be indicated specific for the given prototype section.

4.4.2 Principal Component Analysis

In the IoT context, prediction models should be kept smaller than models that can be executed on more resourceful machines. Thus, this work also experimented with PCA

to potentially decorrelate features, reduce the dimensions of the data and to speed up learning and prediction times.

PCA aims to map the features into a lower dimensional space by means of linear combinations and thereby distill components that maximize variance. In order not to distort the principal components towards features of large variance, features should be scaled to contribute equally with unit variance. This is however not the case in MinMax scaling, but is possible via standard scaling using the Z-score formula $z = \frac{x-\mu}{\sigma}$. This poses the problem of not accounting perfectly for the original feature distribution. But another more practical problem for RL-based MTD is that the PCA transformation should be calculated based on the complete training dataset including all behaviors. But in the context of this work, not all attacks can be assumed available, since there is always the possibility of zero-day attacks. Similarly to the difficulty of calculating ideal scaling parameters, it is even harder to derive an optimal PCA transformation in the RL use case at hand. The option to use PCA to reduce the dimensions to a specifiable number of components is available in the code [56]. However, the results presented in subsequent prototyping chapters do not use any PCA due to the aforementioned reasons and in order not to make any limiting assumptions. Despite that comparable results were achieved with PCA and 10-20 principal components only, it is excluded as it may lead to more unstable results in the general case.

Chapter 5

Prototypes for Offline RL-based MTD

This chapter extends the data exploration from the previous chapter and aims to discover more about the application of RL and DL to the problem domain. The goal is to derive the conceptual proof that an agent can learn to find mitigating MTD techniques, for given attack behaviors by interacting with an offline, simulated environment under ideal conditions.

Therefore, two different prototypes are presented and evaluated for their MTD deployment choices. The first starts with the simplest possible scenario, where an agent learns to map MTD actions to behavior states with the help of a supervisor. The second prototype strives towards a more realistic scenario by introducing an anomaly detection component which makes the learning process unsupervised. To facilitate the overview, only samples from the raw behavior dataset will be used for both prototypes as per the previous Section 4.3. Only the following Chapter 6 then goes a step further, by constructing the most realistic simulation environment, making use of data monitored while an agent controller is running in parallel. The code for all simulation environments and the performance results can be found in [56].

5.1 Prototype 1: Supervised Simulation on Raw Data

As a baseline, the convergence of RL-based MTD should be verified under ideal conditions. For this, the environment, and in particular the state and reward signals are constructed in a completely supervised manner. In contrast, the agent's action choices are independent of offline or online learning and just correspond to the output of the online Q-network Q^O or a random action (see Algorithm 1 in Subsection 2.5.4).

5.1.1 Simulation Environment

The core of a simulation environment marks an interface for the agent-environment interaction step, meaning a function that takes a state and action as input and produces

as output the next state and the reward. Figure 5.1 depicts such a single step, including all actors, and components as relevant for this simulation. The attacker with a choice of malware of the families CnC, rootkits and ransomware is abstracted away behind the dataset of raw attack behaviors.

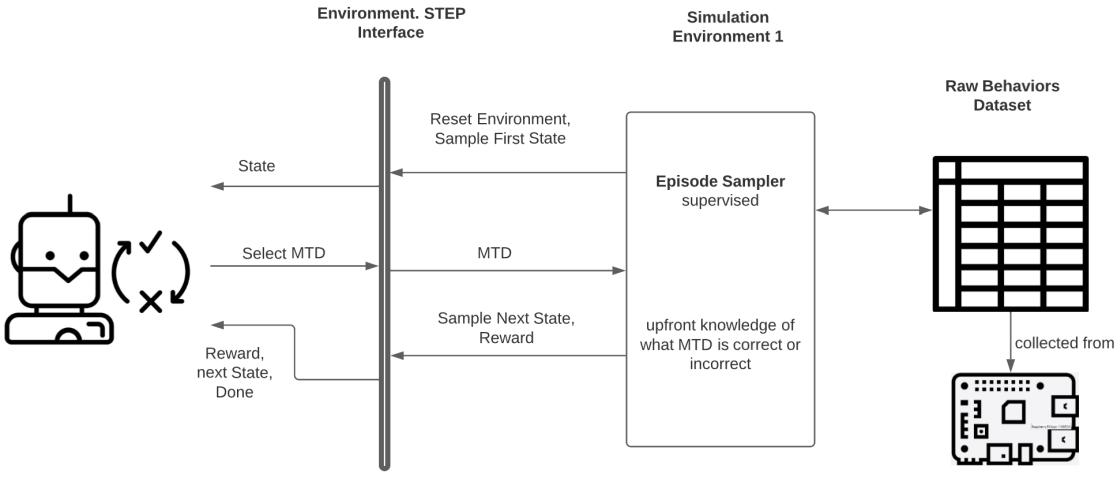

Figure 5.1: Prototype 1: Agent-Environment Interaction Step: Supervised

For this first prototype, all state signals are sampled from previously collected raw attack behavior data. The reward signals are constructed based on a priori knowledge of the mapping of mitigating MTDs to behaviors. A positive reward is given (+1) in case of choosing the correct MTD, and a negative (-1) in case of an incorrect MTD. Figure 5.2 depicts the full simulated training process flow, abstracting away the details of the DQ-Learning algorithm.

Note that in this process flow, the behaviors are randomly sampled from the dataset of raw behaviors. Initially, a random attack is chosen as decision state to start an episode. Thereafter, the sequence of behaviors always matches the one that a real online agent would observe (i.e. bdvl attack - wrong MTD - bdvl attack - wrong MTD - bdvl attack - correct rootkit MTD - normal behavior). However, using this episodic pattern is a simplification regarding two important aspects: First, an episode is always started with an attack. In a real scenario, such an attack must be recognized first to reactively deploy MTDs. An episode may also start wrongfully upon normal behavior. Secondly, episodes are terminated by sampling raw normal behavior after the correct MTD technique is chosen. This does not necessarily correspond to reality as it is possible that an MTD technique does not erase the effects of an attack completely (see Subsection 3.4.3). In fact, this simulation procedure ensures that the assumption of perfectly working MTDs is always valid throughout training. The possibility of device behavior being sustainably altered by an attack or an MTD technique is thus excluded.

As a further important simulation property, it should be highlighted that it is not necessary to actually execute any MTD techniques during the training process. Technically, this implies that for this simulation, there is not even a need be in possession of any MTD technique. It can be done completely time-agnostic of any MTD and on a separate device

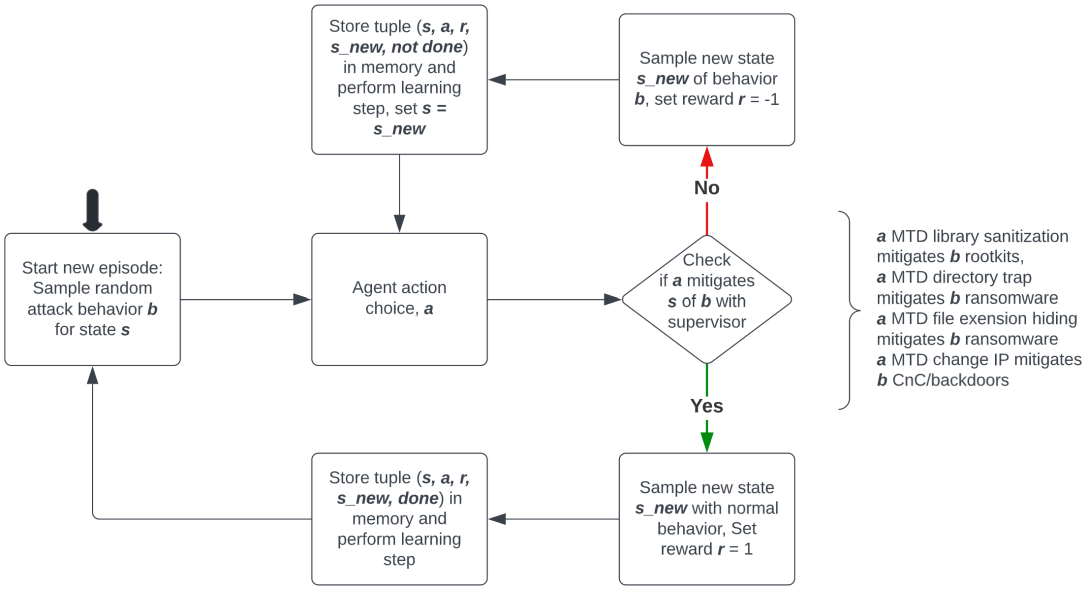

Figure 5.2: Flowchart of the training process used for Prototype 1: Supervised Simulation Environment. Note that rewards are set directly based on the correctness of the MTD choice (+1 if correct - terminate episode, -1 if incorrect - continue episode)

- as long as there is a dataset with known behaviors to sample from. These are all crucial advantages of having a simulation compared to learning online from scratch.

All states are assumed to contain all relevant information for taking an action. This corresponds to modelling the environment as a markov decision process (MDP). By having a supervisor, there is always a correct and time-independent action, reward and successor state, given a state of a certain behavior. The exact state transitions are stochastic. In essence, this simulation corresponds to a sequential game of a non-adaptive adversary and a learning defender on ideal data.

5.1.2 Performance Evaluation

For the evaluation of an agent in the previously described simulation environment, all raw behavior data is read and split into 80% training and 20% test samples. The training set is used for the agent-environment interaction, the test set to assess the performance on unseen state samples after convergence. The data is preprocessed by removing a range of features suspected to be constant, time-related, cyclic or otherwise unstable, and by filtering outliers in the train split (as per Section 4.4). Further, in order to make as few assumptions about the malware as possible, the data is scaled based on minimum and maximum values of normal data only. Next, DQ-Learning as per Algorithm 1 is applied to train an agent interacting with the simulation environment. For this, a number of hyperparameters need to be selected. As input for the DQ-Network 46 features are considered corresponding to the number of state dimensions after excluding all irrelevant features (see feature column in Tables A.1 and A.2). The output size of the DQ-Network corresponds to four actions, one for each available MTD technique. The replay memory is set up as a ring buffer containing 500 transitions at maximum, and is initialized with 100

sample transitions before the agent starts to learn. The batch size for the gradient descent step in the RL loop is accordingly set to 100 transitions. $1e^{-4}$ is chosen as the learning rate and the reward discount factor γ is set to 0.1. Choosing γ close to 0 ensures that immediate rewards are weighted much more than future rewards, which is desirable for correct MTD selection and speeds up the training process. ϵ as the exploration parameter starts at probability 1 and decays with every learning update by $1e^{-4}$ until it reaches a minimum level of 0.01 (ensuring that the agent continues to explore indefinitely). As a final hyperparameter, the frequency of replacing the target network Q^T by the online network Q^O is set to every 100 learning update steps.

The convergence of an agent with the above hyperparameters is shown in Figure 5.3 over 10000 episodes. The maximally achievable score is 1 as measured by a running average of the last 20 average rewards per episode. The average episode reward is calculated as the sum of all rewards received per episode divided by the number of interaction steps per episode. In the best case, there is only one interaction step, meaning the correct MTD technique is chosen at the first observation of an attack state, yielding reward 1.

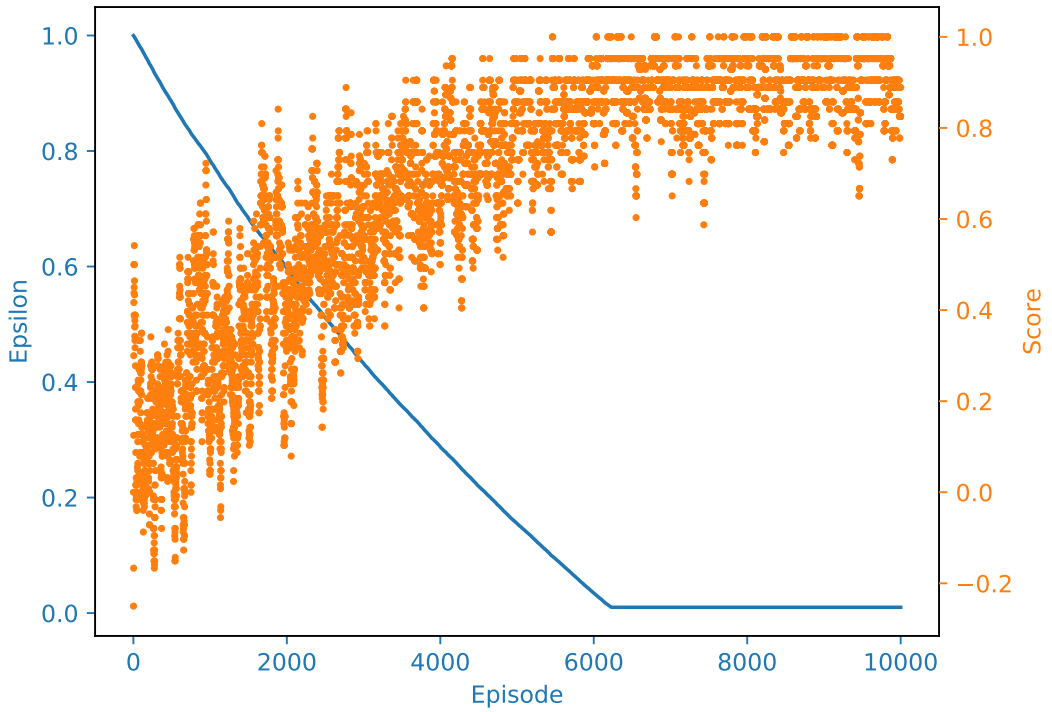

Figure 5.3: Learning process over episodes and epsilon decay

Figure 5.3 shows that the agent's action choices effectively improve over time despite a certain range of fluctuation. Overall, the convergence is relatively stable, reaching a desirable level of accuracy after approximately 6000 episodes. Initially there is a larger variation due to a larger exploration probability given by epsilon, and relatively uncertain action choice. This also explains the slight curve in the epsilon decay as initially a larger number of steps are required per episode.

Table 5.1: Prototype 1: Greedy MTD Selection Accuracy of the Online DQ-Network Q^O.

Behavior	Accuracy	Objective
ransomware_poc	98.93%	ransomware_mtds
bdvl	99.03%	rootkit_sanitizer
beurk	67.28%	rootkit_sanitizer
the_tick	97.65%	cnc_ip_shuffle
backdoor_jakoritar	69.13%	cnc_ip_shuffle
data_leak_1	99.38%	cnc_ip_shuffle
data_leak_2	100.00%	cnc_ip_shuffle

Where Figure 5.3 clearly demonstrates the agent's learning progress, it does not show how well the agent performs on different attack behavior states, respectively how accurately it chooses MTD techniques. Therefore, after the training process is finished, the accuracy of the agent's greedy action choices is evaluated on the separate and unseen test set. This corresponds to feeding test state vectors into the priorly trained online DQ-Network Q^O and comparing the predictions as in a supervised setting to the correct MTD objective. The results are shown in Table 5.1.

As expected from the data analysis in the previous chapter, the beurk and backdoor jakoritar attacks are the hardest to map to an MTD technique as both of them are very close to normal behavior. Our agent selects the correct rootkit_sanitizer MTD technique in only 67% of the cases when observing beurk behavior, for the backdoor jakoritar attack, the IP shuffling objective is achieved with about 69% accuracy. However, all other attacks are correctly mapped to the mitigating MTD technique with accuracies between 97% and 100%. These results serve as a reference to an upper bound of performance for the subsequent prototypes, which aim to approximate the real-world setting by gradually loosening assumptions and making the training process fully unsupervised.

5.2 Prototype 2: Unsupervised Simulation on Raw Data

Clearly, the type of training environment presented in the previous section is an idealized and simplified version, which cannot accurately simulate an online environment. From the agent's learning perspective, this is mainly due to two reasons: First, in reality, rewards cannot be calculated with the help of a supervisor since attacks must be assumed unknown. Secondly, it is not known when an episode should start or be marked as finished.

This section presents a simulation scenario, that does not need to rely on a supervisor for neither the reward calculation nor the start and termination of episodes. To achieve that, an anomaly detection component is utilized as motivated in Chapter 3.

In the context of this work, an autoencoder neural network is used as anomaly detector. An autoencoder is a deep neural network with an equal number of input and output dimensions, but hidden layers of smaller size. Figure B.1 presents a sample architecture. The goal of this network is to learn an encoding for the input features x via a compressed bottleneck, and to reconstruct the original input features as close as possible as output

$\widetilde{x}$. The mean squared error loss, $MSE = ||\widetilde{x} - x||_2^2$, of the input to the output is used for backpropagation. Thus, by training on normal behavior only, an autoencoder effectively learns to reconstruct normal behavior with minimal loss. Upon feeding a different type of behavior to the network, there is a larger MSE between input and output. And if the MSE exceeds a certain threshold, an anomaly can be flagged. For all later experiments, the threshold is calculated based on the mean plus a multiple of the standard deviation of a separate validation set.

5.2.1 Simulation Environment

Analogously to the previous prototype, Figure 5.4, displays a single agent-environment interaction step as used for this second simulation, including all relevant actors and components.

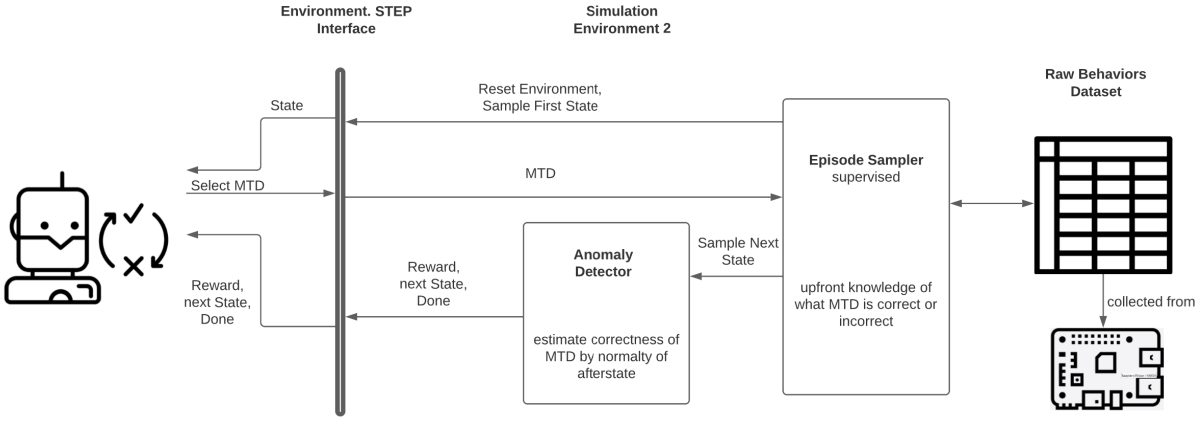

Figure 5.4: Prototype 2: Agent-Environment Interaction Step: Unsupervised

Furthermore, and in more detail Figure 5.5 shows the exact training process for the second simulation. While it also uses raw behavior data as in the previous section, it increases the complexity of the flowchart 5.2 by anomaly detection at two key stages: First, at the beginning, before starting a new episode - at decision states. At this point the aim is to only trigger the MTD deployment agent, if there is actually an ongoing attack on the device. The second stage where anomaly detection is used is after an MTD has finished running. Here, the goal is to interpret such afterstates for whether the previously executed MTD was successful or not. Then, the MTD's estimated success determines rewards for the agent, whether to deploy new MTDs or to terminate an episode.

As indicated by the second branching condition (in the middle of Figure 5.5), the sampling of behavior states still happens by making use of a supervisor. However, this only guarantees that a realistic sequence of behaviors is fed to the agent. If for instance attack behavior b is running on the device, this behavior should still be on the device, if the MTD chosen does not mitigate the attack. In case the MTD mitigates the attack, the simulation should also restore the device behavior to normal. In a real environment, this sequence of behavior states is just given naturally, but in a simulation, this can only be

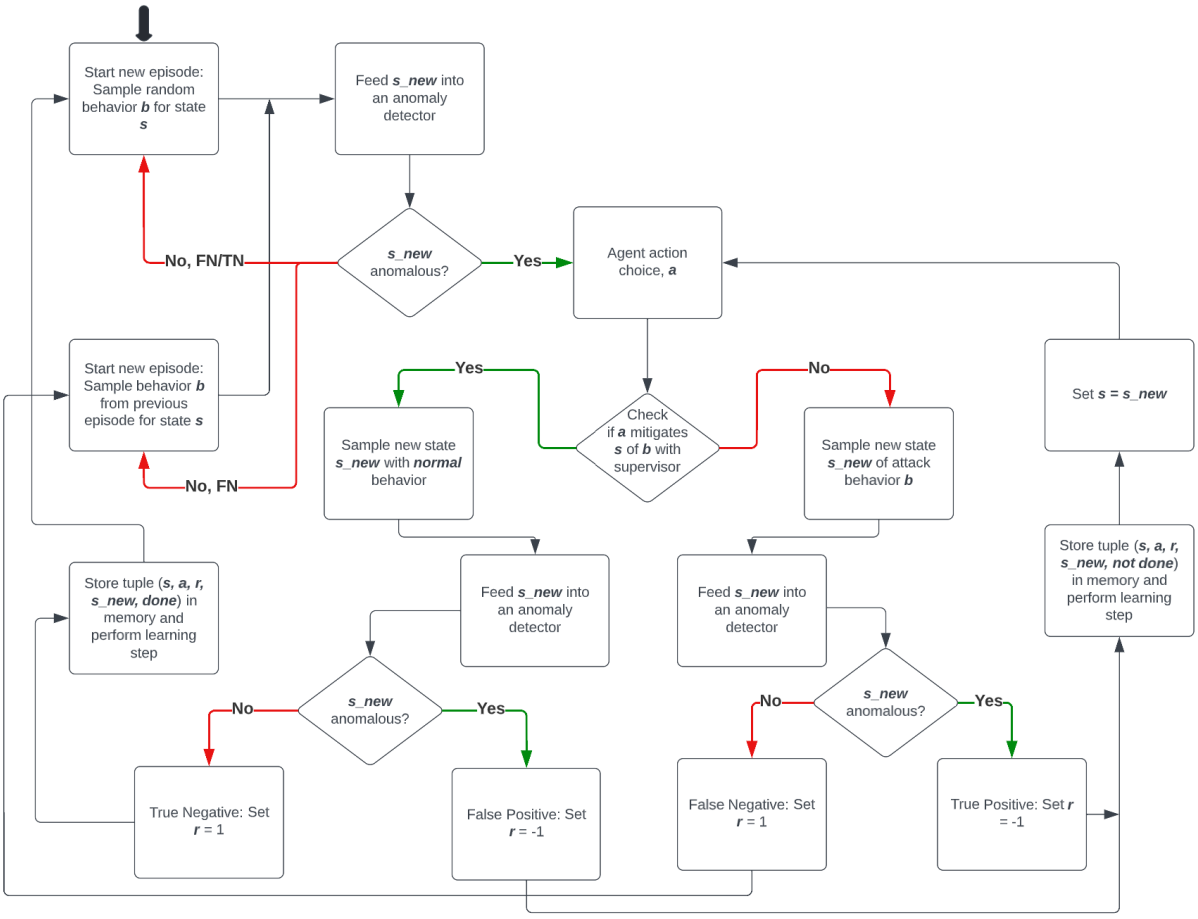

Figure 5.5: Flowchart of the training process used for Prototype 2: Unsupervised anomaly detection for reward calculation and episode framing. Inaccuracy of the autoencoder opens up the possibility of false negatives or false positives resulting in incorrect RL update targets.

done via a supervisor based on known behaviors. However, this poses no problems with respect to the unsupervision of the learning process itself, as it does neither affect rewards, nor the flow within episodes.

Using an anomaly detector adds another conceptual layer to the training process, that makes it completely unsupervised. To see that it is helpful to closely examine all the paths in Figure 5.5, especially at the outcomes of the anomaly detection step on the afterstates (visible by the bottom two branching conditions in the figure). Essentially there are four cases:

1. True Negatives (TN, agent correctly chooses an MTD technique mitigating a given attack b, and the autoencoder correctly flags the normal afterstate as normal)

2. True Positives (TP, agent chooses a wrong MTD technique, which does not mitigate a given attack b, yet the autoencoder correctly flags the attack b afterstate as anomalous)

3. False Negatives (FN, agent chooses a wrong MTD technique, which does not mitigate a given attack b, and the autoencoder flags the attack b afterstate incorrectly as normal)

4. False Positives (FP, agent correctly chooses an MTD technique mitigating a given attack b, yet the autoencoder incorrectly flags the normal afterstate as anomalous)

Note that the cases of TP and TN exactly correspond to what happens in prototype 1 from the previous section. Stated differently, a perfectly working anomaly detector leads to the exact same training process flow as in the supervised simulation. However, FN and FP are generally inevitable and require a special treatment to create a realistic simulation. FN basically mean, that the behavior on the device is malicious (i.e. attack b), but it is not recognized as such. Thus, the reward signal is mistakenly positive ($r = +1$) and a single episode should be terminated. However, obviously the malicious behavior b is still running on the device. Thus, the next episode must start again with this behavior b. This ensures that the simulated training directly mimics a real-world setting.

FP, as a second flaw of the autoencoder, mean that the behavior on the device is normal, but it is recognized as an anomaly. Therefore, the reward signal is mistakenly negative (r = -1) and an episode does not end, even if it should, and another MTD is set to be deployed on normal behavior. If the FP-Rate is large this can lead to a loop where many MTDs are deployed without mitigating any attack. In both cases of FN and FP, the key problem for the RL agent is that incorrect transitions are stored in the memory buffer. This implies that each FN or FP can be considered an adversarial sample in the training data, possibly distorting the training process.

Please note, that the anomaly detection may not only make imperfect predictions at afterstate time (bottom two branching conditions), but also at decision state time, before new episodes are started (first branching condition). For the cases of states predicted normal, the MTD agent is not triggered. This includes both TNs and FNs. In case of a TN, device behavior has been correctly recognized as normal and nothing is done. But in case of a FN, an attack has gone undetected and the simulation will continue to restart the next episode with this attack, until it is detected and the agent is triggered. Certainly, in a real setting, if an attack is very hard to distinguish from normal behavior this may lead to long periods of no MTD deployment without further measures. This clearly marks the worst case in this scenario.

If the decision state is flagged abnormal, the MTD agent is triggered in both cases of TPs and FPs. TPs correspond to the desired process flow, where MTDs are deployed in a useful manner. FPs, on the other hand, trigger the agent without any direct benefit until the normality of behavior is recognized. It is however interesting to note, that MTD techniques launched due to FP can be seen as a means of unintentional proactive defense.

In special cases, it is possible that a behavior is repetitively flagged as malicious by the anomaly detector (both as TP, or FP), such that MTD techniques are launched again and again. This is for instance a problem for attacks that cannot be mitigated with the existing set of techniques. To account for this case, the agent is only allowed to deploy each MTD technique once per episode. Concretely, for each episode an action buffer is

maintained that stores the previously taken actions. When this buffer is exhausted, an episode is terminated with a warning, and a new episode starts with an empty action buffer. Note however, that this only happens in extremely rare cases due to FP in the simulation considered here, as no attacks are injected in the training process that cannot be defended against.

In summary, TP and TN always lead to correct transitions in the agent's replay memory buffer. But FP and FN result in wrong transition entries, possibly distorting the training process. FNs either do not correctly trigger MTD deployment in the first place, or they result in mistakenly positive rewards, and premature termination of episodes. FPs in contrast trigger the agent mistakenly, resulting in wasteful and unnecessary MTD deployments as episodes are either started on normal behavior or when episodes do not terminate on time. Clearly, to guarantee stable convergence of the overall learning system, the optimization of the anomaly detector, respectively the minimization of FN and FP is crucial. The next section evaluates the performance of an autoencoder pretrained for this simulation and discusses its effect on the performance of the agent as the primary maximization target.

5.2.2 Performance Evaluation

As for the previous prototype, all raw behavior data is read and split into 80% training and 20% test samples, unsuitable columns are dropped and outliers are removed from the training data. Again, Minmax-Scaling is performed based on the extreme values of normal training data only to avoid assumptions about attack behaviors. Of the normal training data 70% of samples (ca. 7000) are used for pretraining an autoencoder and 30% are set aside as sufficient for the actual RL simulation.

Anomaly Detection Performance

The autoencoder is trained on 80% of its dedicated normal data over 100 epochs with a batch size of 64 samples, a learning rate of $1e^{-4}$ and a momentum term of 0.9. The remaining 20% of the samples are used to calculate the threshold as the mean predicted MSE reconstruction loss + 2.5 standard deviations. The anomaly detection results on the test set of the accordingly trained autoencoder are depicted in Table 5.2.

Here, beurk and backdoor jakoritar stand out with a particularly bad performance, being recognized as malicious in only 12-13% of the cases, which is in line with the foregoing data exploration. Similarly, the tick attack is insufficiently detected with 32% indicating that these attacks may result in incorrect replay memory transitions due to FN, hampering the agent's learning progress. However, normal behavior as well as all other attacks are detected well with accuracies between 84% and 100%.

Table 5.2: Prototype 2: Autoencoder Performance on Raw Behavior Data

Behavior	Accuracy
normal	84.33%
ransomware_poc	100.00%
bdvl	100.00%
beurk	12.31%
the_tick	32.67%
backdoor_jakoritar	13.03%
data_leak_1	94.61%
data_leak_2	100.00%

Agent Convergence and MTD Selection Accuracy

The model of the previously trained autoencoder is stored to be employed within the simulation environment as per Figure 5.5. To ensure comparability to the first prototype, the exact same hyperparameters are selected to train the agent. However, an additional hyperparameter for this simulation is given by a probability level, to which either normal or attack behaviors are sampled at the beginning of new episodes. In reality, it is unlikely that attacks are launched constantly against the agent (as done in the first prototype). To account for this fact, the simulation samples normal behavior with a chosen probability of 80%, and else a random attack behavior before starting an episode. This ensures that the performance results consider the case of possibly many FP being introduced during training.

Another important hyperparameter is the number of samples considered to make the decision of whether a behavior is anomalous or not. In theory, it is possible to monitor an arbitrary number of samples and then make the prediction based on the majority of samples. However, at this point and for comparability only one sample will be considered here.

Figure 5.6 displays the convergence of the agent towards the maximum score as the average over the last 20 episodes (as in Figure 5.3). As already observed from the first prototype, this second agent reaches a stable level of performance after about 6000 episodes. It can be appreciated here, that the agent's learning progress is relatively constant, despite FN and FP introduced by the autoencoder.

After finishing the training process, the accuracy of the agent's greedy action choices is evaluated on the separate and unseen test set as done for the first prototype. Analogously, Table 5.3 shows the accuracies of the agent's Q^O online DQ-Network to select the correct MTD technique given attack behaviors. Note that here, no meaningful accuracy can be evaluated for normal behavior, as every MTD technique is both incorrect (as nothing needs to be deployed) and correct (as the resulting afterstate correctly corresponds to normal behavior from a learning perspective).

When comparing Table 5.2 and Table 5.3, it can be observed that the autoencoder's prediction performance approximately translates to the agent's MTD selection performance.

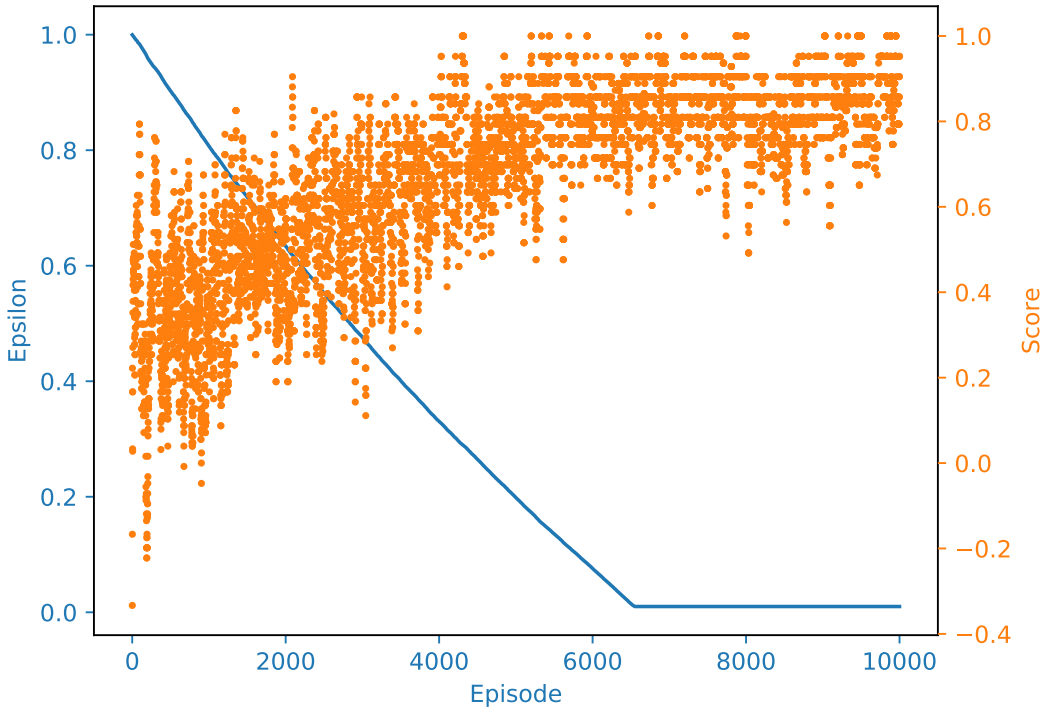

Figure 5.6: Learning process over episodes and epsilon decay: AE predictions are based on a single state sample.

The correct rootkit sanitizer technique is chosen in less than 5% of the cases when observing Beurk behavior, matching the poor autoencoder prediction for this attack with 12% accuracy. Further, the backdoor jakoritar attack is also only correctly mitigated with the IP shuffle MTD in a bit more than half of the cases. The fact that the agent's performance on the tick and the backdoor attack is not impacted as hard by their poor anomaly detection rate, may be due to the other two CnC-based data leak attacks, that are recognized with near perfect accuracy.

When comparing this agent's performance to the performance of the baseline agent from prototype one, two important things can be noticed: First, the selection accuracy decreases the most for behaviors that are hardest to distinguish from normal behavior.

Table 5.3: Prototype 2: Agent Performance on Raw Behavior Data

Behavior	Accuracy	Objective
ransomware_poc	99.14%	ransomware_mtds
bdvl	95.93%	rootkit_sanitizer
beurk	4.72%	rootkit_sanitizer
the_tick	83.28%	cnc_ip_shuffle
backdoor_jakoritar	54.69%	cnc_ip_shuffle
data_leak_1	99.47%	cnc_ip_shuffle
data_leak_2	100.00%	cnc_ip_shuffle

These behaviors have a high percentage of incorrect transitions stored in the agent's replay memory, acting as unintentional adversarial samples. Secondly, it appears that the wrongful replay buffer entries do not remarkably impact the MTD selection accuracy for other attack behaviors. In fact, ransomware, bdvl, as well as the two data leak attacks are almost perfectly recognized and considered in the agents MTD selection policy (accuracies between 95%-100%). One potential factor which contributes to this relative robustness may be due to the general variation given in DQ-Learning by the dynamic update targets and the exploration strategy. In other words, since the algorithm is designed to cope with imperfect action choices, it may be to some extent robust against incorrect rewards. However, studying this in more detail is certainly out of the scope of this work.

5.3 Summary

This chapter establishes a proof of concept that an RL agent can learn to map mitigating MTD techniques to given attack behaviors by interacting with a simulated environment that leverages ideal, raw behavior data to represent states. Two different kinds of prototypical simulations are presented where both sample behaviors from a precollected dataset in the sequential manner that corresponds to what an online agent would observe in a real environment. Both simulations sample normal behavior as next states when the agent correctly selects an MTD technique. This ensures that the assumption of perfectly working MTDs always holds during training as no unclear temporal dependencies between device states need to be considered.

The first prototype employs a supervisor to decide on the calculation of the reward and the termination of episodes. It shows what can be achieved in an optimal learning setup and therefore it establishes a reference to an upper bound for more complex simulations which more accurately model reality. The performance of an agent trained in this setup shows, that some attacks are naturally harder to map to MTD techniques than others. However, apart from two attacks that are mitigated with ≈69% accuracy, all behaviors are correctly mapped to an MTD technique with 97% accuracy and more.

The second prototype extends the previous simulation and gets rid of any supervised components in the learning process. An autoencoder is used for anomaly detection at both decision state and afterstate time. This enables to determine a start signal for episodes, to calculate rewards, and to terminate episodes in a fully unsupervised fashion. However, this comes with the drawbacks of potential misclassifications from the anomaly detector. Due to FN, episodes might be terminated prematurely, giving wrongfully positive rewards and due to FP MTD techniques are deployed despite the absence of any attack, giving mistakenly negative rewards. This leads to unintentional, adversarial samples in the agent's replay memory. The performance results show that the autoencoder's prediction performance approximately translates to the agent's performance in selecting the correct MTD techniques. The convergence of the agent is relatively stable, despite the autoencoder's imperfection and most behaviors are correctly mapped to a mitigating MTD with 95% accuracy and more. The agent only fails to find the correct MTD technique for behaviors that are very hard to distinguish from normal behavior (notably beurk and backdoor jakoritar). As the second prototype ensures that the learning process in itself

is completely unsupervised, it sets the stage for moving the RL agent towards a real, online-learning environment.

Chapter 6

Towards Full Online RL-based MTD

As stated in section 3.6.1, integrating an MTD deployment agent and its core - the Deep Q-Network - into an online environment requires additional components for monitoring, anomaly detection and orchestration of tasks. In the following, these components are presented and it is described how they can work together to achieve online learning.

Accordingly, the chapter starts off with the design of an agent controller that manages the process flow of autonomous interaction and learning in a real environment. Where this agent could, in principle, learn from scratch in an online manner, the remainder of the chapter focuses on crafting a simulation that aims to utilize the most realistic data possible to pretrain an agent, which can later be deployed in a real environment. Thus, the goal is to craft a simulation that allows the transfer of the learnt policy to the real world. As explained in Section 4.2.1, the data, respectively the states observed by an online agent are influenced by the concrete processes running on the device, including the agent's components itself. Thus, this chapter continues by presenting a dataset collected while such an agent controller has been actively running on the device in parallel. Besides detailing the steps undertaken for the data collection, the data is analyzed similarly to Chapter 4. After that, a third training simulation can be presented in Section 6.3, which extends the simulated environment from Section 5.2 with the new, more refined data. Analogously, performance results are shown and finally, the contributions of the chapter are summarized.

6.1 Online Agent Controller Design

Figure 6.1 presents an architectural view of an online MTD controller agent, including required modules for the MTD decision (WHAT & WHEN), as well as MTD specification (HOW). Each of the depicted components affects `perf` features and must be taken into account for simulations which aim to utilize realistic environment data. This comprises components for perception and interpretation of the environment, RL-based reactive MTD selection, as well as the MTD technique themselves.

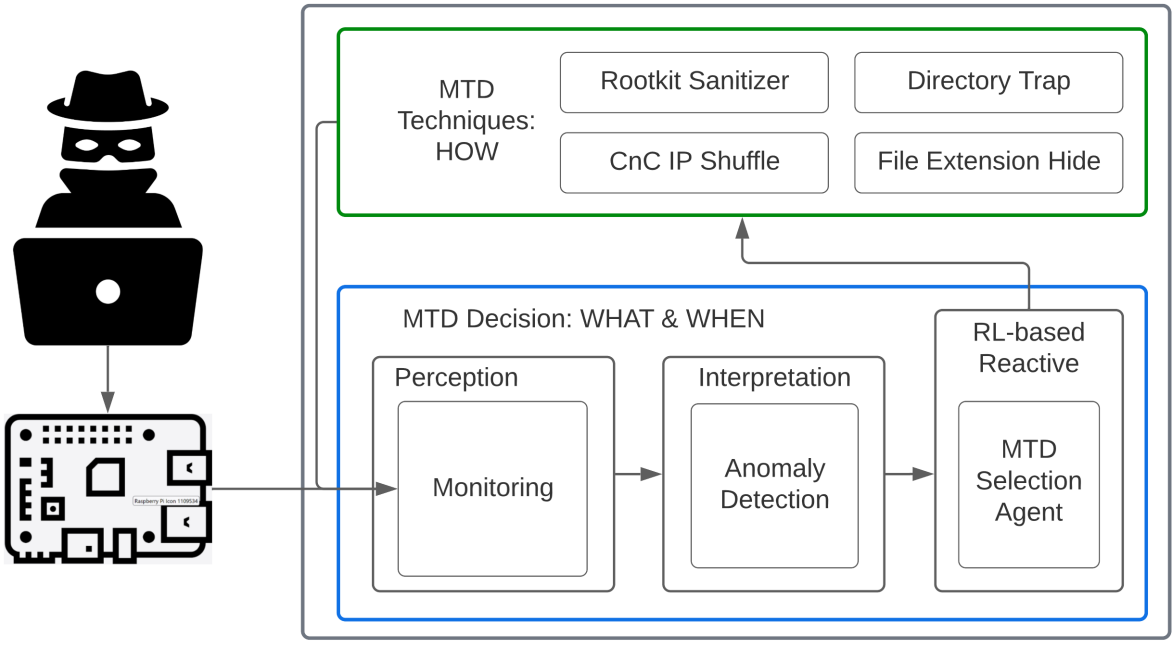

Figure 6.1: Online MTD Controller Architecture

For further clarification of how the components collaborate Figure 6.2 displays the logical view of the on-device process flow relevant for online learning. The components are listed in swimlanes, to indicate the separation of responsibilities. The controller orchestrates the timed invocation of all other components and manages the passing of information between them. Further, it is responsible for setting the points in time when a new episode should be started. As indicated in the caption of Figure 6.2, the process flow between the controller lane and the other lanes is minimized for better overview. The monitoring component is responsible for collecting relevant state information from the (real) environment. This happens via a shell script that collects all `perf` features of interest as explained in Figure 6.2 for a predefined monitoring duration. Monitoring is necessary at two different stages, at decision and afterstate time. Analogously, the anomaly detector component is invoked for two different purposes. As already discussed for the second simulation in Section 5.2, the anomaly detector is responsible for making the decision whether or not to trigger the agent at decision state time. At afterstate time, the anomaly detector estimates from the observed state whether the MTD technique was successful. In the cases where behavior is recognized as normal (TN or FN), the controller waits for a duration d until it initiates the next episode cycle. In the cases where anomalous behavior is detected (TP or FP), the agent is triggered to choose an action. The agent is the core component responsible for the policy and interactive learning updates. More concretely, it is in charge of the action selection strategy (ϵ-greedy, action prediction via DQ-Network), the storage of experienced transitions in its replay memory, as well as the whole learning update step. Finally, in order to be able to execute an MTD technique according to the agent's choice, the mechanisms must be readily available. All MTD techniques are thus summarized in a single lane for an MTD Launcher. However, the execution of a particular MTD mechanism is handled and tracked by the controller. Note that here, the process flow is considerably simpler compared to the complex sampling in Section 5.2 for prototype

2. Here, the agent's actual state observations stemming from an unknown environment replace the previous sampling of behaviors. The only defining information for taking the next step in the process flow is whether a state is considered normal or anomalous by the anomaly detector independent of whether it is a TP, FP, TN or FN. Algorithm 2 in the next chapter presents more implementation details of the online agent controller. This chapter, however abstracts away from such specific online implications, as the primary focus is on the control flow as needed for constructing a realistic simulation. Further, the complete code for this online MTD controller can be found in [56].

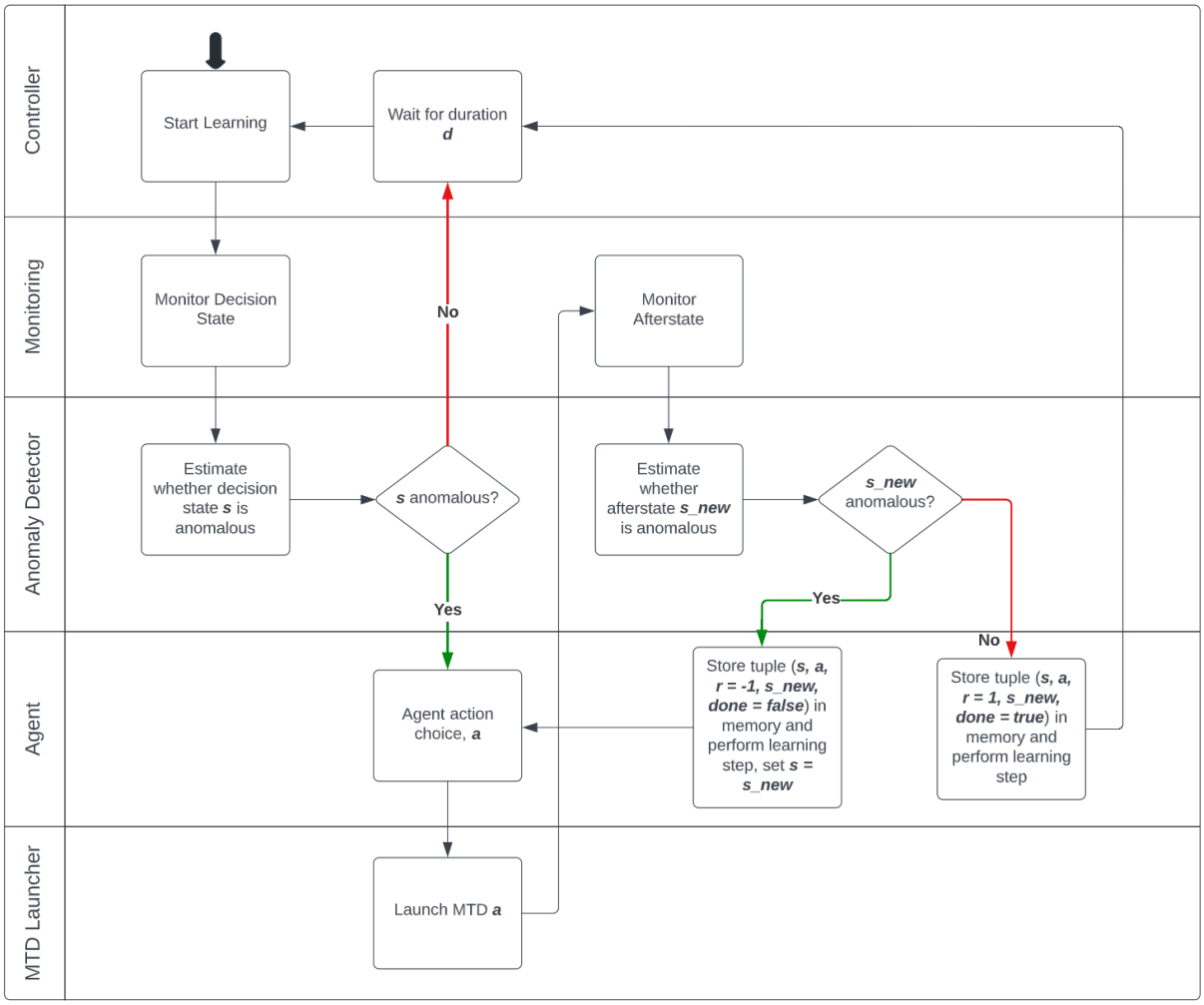

Figure 6.2: Online Agent Learning Process Flow. Technically, the controller orchestrates all of the other components. But for the sake of a better overview, the respective arrows back and forth from the controller lane are left out. Instead, information and responsibilities are directly passed between sequential components in the diagram.

All components are intended to be run on-device only. This ensures that the whole system can function in a standalone fashion, independent of any other devices. This is in line with the choice of on-device only features. Further, no communication overhead or latencies to off-device components have to be dealt with.

Another important aspect of this process design is the waiting mechanism employed by the controller to govern when the next monitoring loop should start. Theoretically, the

time to wait can be set to a fixed duration or adapted flexibly according to a separate logical component. Anyhow, setting this time duration is a generally non-trivial task as it depends on the desired security level, the resource consumption of the agent code and thus also its service quality impact. Where Chapter 7 gives more information regarding this matter, this chapter focuses more on the flow of the training process itself to construct a simulation that approximates reality.

As already mentioned above, states are monitored at two logically different places in the learning process: decision- and afterstates. Thus, a simulation which tries to learn as close as possible to this online setup must also consider both decision and afterstate data during training. The next section provides details about the collection of a corresponding dataset that enables the separate treatment of decision and afterstates within an extended simulation.

6.2 Refined Environment Data: Decision- and Afterstates

This section provides details on the collection of a dataset refined for decision and afterstates. Further, peculiarities of this dataset are analyzed to assess its suitability for simulated environments, and to get a feeling for the separability of behavior states.

6.2.1 Data Collection

For the collection of decision state and afterstate data, the exact same script is used as for the raw behaviors. However, here the monitoring is launched via the agent controller itself. Thus, as the first step, the whole online agent code is moved to a RP 3 Model B+ and all dependencies are installed. Then, the monitoring procedure slightly differs for decision- and afterstates. For decision states no MTD execution is required. The device has been infected with each of the attack behaviors already considered in previous sections and the controller script has been launched with the decision state monitoring duration set to four hours. After the monitoring has finished, the newly created datafile is just copied to a different device to perform the simulation. Monitoring real afterstates in contrast requires first to step through the whole process of monitoring decision states, detecting an anomaly, choosing an action and deploying the corresponding MTD technique. To standardize this afterstate monitoring, the monitoring duration of decision states has been set to three minutes and the anomaly detector is preconfigured to flag an anomaly. Moreover, the desired MTD technique to monitor the afterstate for is hardcoded in the controller and the subsequent afterstate monitoring duration is set to four hours. All possible combinations of available MTD techniques and device behaviors (normal and attacks) have been monitored accordingly by infecting the device and starting the preconfigured controller. [56] contains the complete code used to monitor all decision and afterstates. Further it lists the parts of the code that are left as comments due to the monitoring setup (i.e. the autoencoder prediction, or the learning update as it happens only after the afterstate monitoring). Anyhow, without loss of generality of the monitoring and training process, it is assumed

that the commented parts have negligible influence on behavior separability and thus the learning process.

Table C.1 and Table C.2 list the number of samples collected for all decision and afterstate combinations. In total, there are 77881 unfiltered samples available.

6.2.2 Data Exploration

To analyze the suitability of the collected decision- and afterstate dataset for a simulation environment, essentially the same characteristics as analyzed earlier for raw behaviors are relevant: The variation over time as well as the feature distributions. Since the results of this data exploration largely overlap with Section 4.3, only peculiarities pertaining to decision and afterstate data are presented here. This comprises comparing decision and afterstates, as well as the dimensions of different MTD techniques and behaviors. The code to generate the all plots can be found in [56].

As discussed in Section 4.3, the features should be stable over time, such that states can be randomly sampled from the precollected dataset to construct simulated episodes. Figure C.1 shows analogously to raw behaviors that the timeline of decision state behaviors occupies a dedicated range of event counts. The variation of afterstate data over time is similarly constrained. This implies that the data collected as per the procedure above is suitable for sampling realistic states in simulations.

Besides the variation, again the distributions prevailing in the decision and afterstate dataset are analyzed. Figure 6.3 compactly depicts a range of important properties that are exemplary for this dataset. Here, using only the bdvl attack and normal behavior, illustrated for the mm_page_alloc event.

First, it should be noted, that decision and afterstates may considerably differ for certain features. For instance, a normal decision state (dark green) shows a slightly different distribution than the state after launching a rootkit MTD upon normal behavior. This is in line with the foregoing analysis that all processes running on the device have an influence on the `perf` features. Further, it confirms the need to consider such a dataset when striving towards a real-world application scenario of RL-based MTD. However, decision and afterstates do not differ for all behaviors and features. For instance, bdvl (blue in Figure 6.3) decision state does not differ substantially from most of its afterstates. Concretely, the distribution of the mm_page_alloc feature is almost identical for decision state bdvl and the afterstates resulting from launching an incorrect MTD (shown as purple, red and lightblue).

In contrast, the distribution, when deploying the correct rootkit sanitizer MTD technique has a much narrower range of values. In fact, the afterstate when deploying a mitigating MTD is close to the distribution of normal behavior (the closest to the afterstate of normal behavior and the given MTD). This is necessary for the anomaly detector to estimate a positive reward signal (TN) for a mitigating MTD. But to avoid FN, inappropriate MTD techniques should yield notably differnt distributions than the correct ones. Thus, judging from exclusively looking at the kmem feature in Figure 6.3, incorrect rewards

might be predicted frequently due to the large overlap of the distributions. The fact that an afterstate combination of normal behavior and a deployed MTD m is closest to the afterstate of an attack behavior mitigated by m, further provides insights on what training data the anomaly detector component should consider. To correctly recognize mitigating MTDs, the data used for anomaly detector training should contain afterstates of deploying MTDs upon normal behavior.

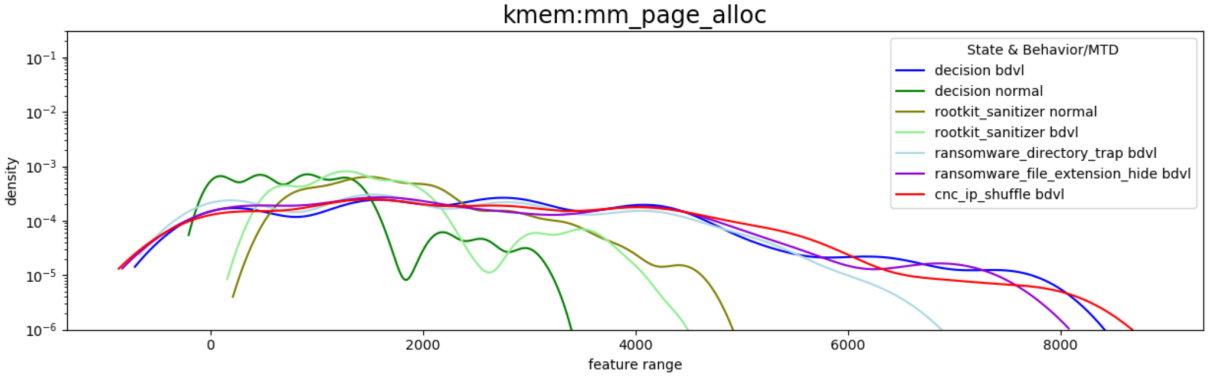

Figure 6.3: Comparison of decision states and all afterstates for bdvl as well as normal behavior for the kmem event mm_page_alloc. There are almost identical distributions for bdvl decision state as well as all afterstates with incorrect MTDs, not mitigating bdvl. However, as visible by the three green distributions, the correct rootkit sanitizer MTD leads to a distribution similar to normal behavior. Normal behavior at decision state time also follows a slightly different distribution than after having deployed the rootkit sanitizer. The afterstates for normal and bdvl behavior for the rootkit sanitizer are most similar.

When imagining all afterstate data as points in the space of $num\_features$ dimensions, ideally there are distinct clusters for each type of behavior. Under best circumstances, there is a single cluster containing normal behavior data as well as all afterstate combinations of attacks and corresponding, mitigating MTDs. Besides that, there should be separate clusters for each attack, meaning for decision states and afterstates stemming from deploying incorrect MTDs upon a given attack. Realistically, this separation boundary is blurred in most features and not recognizable in lower-dimensional visualizations. This can also be observed by the large overlap of the distributions in Figure 6.3. To learn how to map MTD techniques to states, behavior data should be separable despite the effects of any deployed MTD technique. For instance, in the worst case, an incorrect MTD technique could affect the afterstate of an attack a in a way that makes it very hard to distinguish from a certain decision or afterstate of an attack b of a different family. In order to limit this possibility of undesirable effects of MTDs, features are excluded from training if they show heavily distinct distributions between decision and afterstates. For instance, the raw_syscalls event source is extremely sensititve to such fluctuations induced by the agent itself as can be observed from Figure 6.4. For the sys_exit event, the distributions of decision state normal and normal afterstate after deploying the rootkit MTD have absolutely minimial overlap. Similarly, the distribution of all incorrect MTDs differ heavily. The complete list of features excluded via such an analysis can be found in tables A.1 and A.2.

Where Figure 6.4 compared afterstates for a single behavior, it is also interesting to visu-

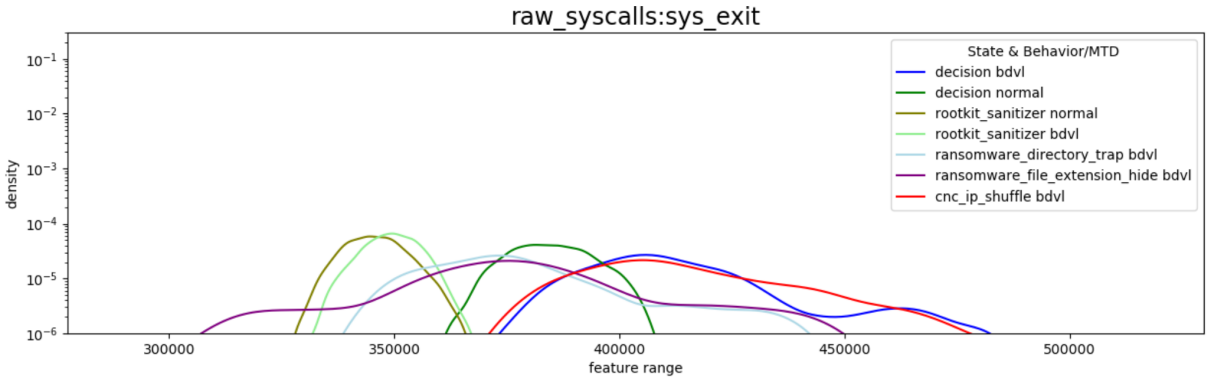

Figure 6.4: Undesired effects of MTD deployment on decision and afterstate for normal and bdvl behavior. The distributions differ significantly due to the agent, even if the behaviors to be detected are the same.

alize a the same MTD technique for the different behaviors. For the sake of completeness, and to verify that behaviors are in principle distinct, and show specific patterns despite MTD execution, an according comparison can be found in Figure C.2.

6.2.3 Refined Data Summary

The previous data exploration shows, that decision and afterstates may differ in general along the dimensions behavior, MTD techniques and features, confirming the need for an according dataset to craft realistic simulations. Further, the features remain stable over the duration of monitoring of the different states, even though the data collection is governed by the controller. This makes the collected decision- and afterstate dataset suitable for RL simulations as it is possible to randomly sample states from it to construct episodes. Further, the distributions of different behaviors show that afterstates resulting from deploying a correct MTD technique upon observing some attack closely approximate normal behavior afterstates. However, afterstates of incorrect MTD techniques follow similar distribution patterns that the malware exhibits in general. After excluding particularly sensitive features, the employment of particular MTD techniques does not seem to influence behaviors in a manner that makes them visibly less distinct.

6.3 Prototype 3: Unsupervised Simulation on Refined Data

Striving towards a more realistic online scenario, the second prototype presented in Section 5.2 is refined with close-to-online decision and afterstate data, monitored while a functionally working agent controller has been running in parallel under real conditions. In the following the implications of this new simulation are elaborated and performance results are presented.

6.3.1 Simulation Environment

As in the previous prototypes, Figure 6.5 depicts a single agent-environment interaction step of this third simulation, including all actors and components. Here, the attacker, and different combinations of malware and MTD techniques are abstracted away behind the decision- and afterstate dataset.

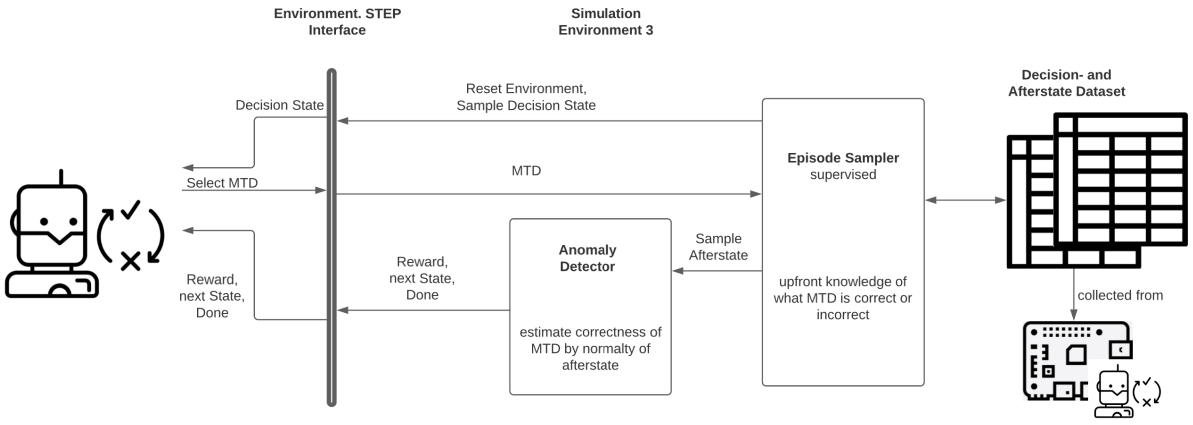

Figure 6.5: Prototype 3: Agent-Environment Interaction Step: Unsupervised on Refined Data

This refined simulation environment follows exactly the same logic as depicted in Figure 5.5, yet the complexity increases at two locations in the process: First, the sampling of states now corresponds to the given stage in the learning loop. Decision states are randomly sampled from all the behaviors monitored at decision state time, and afterstates are sampled dependent on the combination of the current behavior and the agent's MTD choice. Secondly, the anomaly detector component also needs to take into account the new data. Concretely, the anomaly detector used to decide upon rewards and whether to start or terminate an episode must consider both normal behavior at decision state and afterstate time in training. This is in line with the above data exploration.

The fact that afterstates are sampled specifically is a step towards relaxing the assumption of perfectly working MTDs (see Subsection 3.3.1). In contrast to the previous prototypes, afterstates are not just sampled as raw normal behavior if a correct MTD is chosen. Where for the first two prototypes it was not even necessary to execute any MTD technique for the data collection, now simulation afterstates correspond to the real monitoring observation after actually having deployed an MTD technique on the device. This implies that if certain MTDs alter behavior a bit, or if attacks are not mitigated fully despite the correct MTD, this is reflected in the data used in the training process. As explained in the data exploration, the effects of MTDs should be minimized, by excluding the most MTD-reactive features, but that does not change the fact that data is used which considers real MTD deployments. Essentially, this removes the assumption of perfectly working MTDs for a single agent-environment interaction step. However, across multiple interaction steps, the assumption does still hold in this simulation environment. New states are

sampled from the pool of collected decision and afterstate data irrespective of any previous malware that ran on the device. Clearly, a fast data collection procedure cannot follow the exact same steps that an online agent would perform. So this one-step relaxation of the assumption of perfectly working MTDs already aims to push the practical boundaries of how realistic simulated states can get. As already explained in detail in Chapter 3, such an assumption is fairly reasonable when considering periodic reboots, or killing of unknown processes.

6.3.2 Performance Evaluation

For this third training simulation, the whole decision- and afterstate dataset (tables C.1 and C.2) is read and split into 80% training and 20% test samples. As in the previous prototypes and motivated in the data exploration of this chapter, unsuitable columns are dropped and outliers are removed from the training data. Minmax-Scaling is performed based on the extreme values of normal decision state and normal afterstate training data, which ensures that no assumptions have to be made about any attack behavior feature values. Next, 70% of the normal decision state and normal afterstate training data are used for for pretraining an autoencoder and the remaining 30% are set aside as sufficient for the actual RL simulation. Using both decision state and afterstate normal data ensures, that normal behavior is recognized as such despite possible effects of MTD deployment on some features. Thus, it is in line with the previous data exploration.

Anomaly Detection Performance

The autoencoder is trained on 80% of its dedicated normal data over 100 epochs with a batch size of 64 samples, a learning rate of $1e^{-4}$ and a momentum term of 0.9. The remaining 20% of the samples are used to calculate the threshold as the mean predicted MSE reconstruction loss + 2.5 standard deviations. Thus, the selected hyperparameters exactly correspond to those used for the autoencoder trained for the second prototype. The results on the test set of this new autoencoder trained on refined normal data are shown in Table 6.1 for decision states and in Table 6.2.

In principle, there are many possibilities to utilize autoencoders within this simulation. One could train multiple autoencoders for each normal - MTD technique combination separately, on all such normal afterstate data combined, or on normal decision state data only. Autoencoders have been trained for all of these options, and generally they lead to comparable performance. This result can also be expected as features which are most sensitive to the effects of MTDs have been removed. However, the autoencoder trained on both normal decision- and afterstate data slightly outperforms the other variants. As it learns most accurately what exactly normal behavior is - independent of MTD deployment, it is the preferred option in our application context. The corresponding autoencoder experiments are available in [56].

Focusing first on Table 6.1 as the simpler table, it can be observed, that similar results are achieved on decision state behavior data as for the raw behaviors with the autoencoder

Table 6.1: Prototype 3: Autoencoder Performance on Behaviors at Decision State Time

Behavior	Accuracy
normal	94.99%
ransomware_poc	100.00%
bdvl	100.00%
backdoor_jakoritar	6.81%
beurk	5.09%
the_tick	6.50%
data_leak_1	100.00%
data_leak_2	100.00%

of prototype 2 (see Table 5.2). Some attacks (beurk, backdoor jakoritar and the tick) are recognized poorly with only 5%-7% accuracy due to their proximity to normal behavior. Note, that especially the tick is detected much worse here compared to the autoencoder results (83%) of the second prototype in Table 5.2. Clearly, the influence of the agent controller has a non-negligible impact on the detection rate of attack behaviors. The other attacks are detected with perfect accuracy and normal behavior is correctly recognized in 94% of the cases.

Next, Table 6.2 displays the performance on all afterstate combinations, showing the original behavior in the first column and the deployed MTD technique in the second. Despite the accuracy, this table lists the anomaly detection objective as a further column. The purpose of this column is to make clear whether the afterstate should be considered normal behavior or an anomaly. For instance, for all MTD techniques deployed upon normal behavior, the resulting afterstate should be considered normal. Similarly, all afterstates with correct MTD techniques for a given attack should be recognized as normal. For instance, the fifth and sixth row in Table 6.2 with correct MTD techniques against ransomware have a normal detection objective. Here two main results can be observed: First, the accuracies achieved on decision states more or less correspond to the accuracies achieved on afterstates. If an incorrect MTD technique is deployed, the afterstate is recognized poorly in case of the attacks beurk, backdoor jakoritar and the tick (5%-12%), and with high accuracy for the remaining attacks. The second important result is however, that in case of deploying correct MTD techniques, this is recognized for all behaviors, including beurk, the backdoor and the tick (>87% accuracy). This is a desirable result for the later RL agent training. As in the previous simulations, the agent can only deploy each MTD technique once per episode at maximum. This ensures that the behaviors beurk, backdoor jakoritar and the tick are correctly mitigated at some point, even though there might be incorrect entries in the replay memory due to many FN.

Agent Convergence and MTD Selection Accuracy

For the agent's learning process, the exact same hyperparameters are selected as for the second simulation. Again, assuming that normal behavior is running on the device most of the time, the simulation samples normal behavior before starting an episode with 80% probability level, and an attack in 20% of the cases. Further, the autoencoder is set to

Table 6.2: Prototype 3: Autoencoder Performance on Behaviors at Afterstate Time

Behavior	MTD	Accuracy	Objective
normal	ransomware_directory_trap	95.76%	normal
normal	ransomware_file_extension_hide	94.29%	normal
normal	rootkit_sanitizer	93.79%	normal
normal	cnc_ip_shuffle	94.79%	normal
ransomware_poc	ransomware_directory_trap	93.33%	normal
ransomware_poc	ransomware_file_extension_hide	94.21%	normal
ransomware_poc	cnc_ip_shuffle	100.00%	anomaly
ransomware_poc	rootkit_sanitizer	100.00%	anomaly
bdvl	ransomware_directory_trap	100.00%	anomaly
bdvl	ransomware_file_extension_hide	100.00%	anomaly
bdvl	cnc_ip_shuffle	100.00%	anomaly
bdvl	rootkit_sanitizer	88.92%	normal
backdoor_jakoritar	ransomware_directory_trap	5.05%	anomaly
backdoor_jakoritar	ransomware_file_extension_hide	12.06%	anomaly
backdoor_jakoritar	cnc_ip_shuffle	91.57%	normal
backdoor_jakoritar	rootkit_sanitizer	6.63%	anomaly
beurk	ransomware_directory_trap	5.68%	anomaly
beurk	ransomware_file_extension_hide	6.45%	anomaly
beurk	cnc_ip_shuffle	6.61%	anomaly
beurk	rootkit_sanitizer	92.91%	normal
the_tick	ransomware_directory_trap	6.06%	anomaly
the_tick	ransomware_file_extension_hide	6.72%	anomaly
the_tick	cnc_ip_shuffle	87.87%	normal
the_tick	rootkit_sanitizer	5.09%	anomaly
data_leak_1	ransomware_directory_trap	100.00%	anomaly
data_leak_1	ransomware_file_extension_hide	100.00%	anomaly
data_leak_1	cnc_ip_shuffle	88.35%	normal
data_leak_1	rootkit_sanitizer	100.00%	anomaly
data_leak_2	ransomware_directory_trap	100.00%	anomaly
data_leak_2	ransomware_file_extension_hide	100.00%	anomaly
data_leak_2	cnc_ip_shuffle	89.47%	normal
data_leak_2	rootkit_sanitizer	100.00%	anomaly

make the decision of whether the current behavior is normal or anomalous based on a single behavior sample.

Figure 6.6 shows the convergence of the agent towards the maximum episode reward as for the previous prototypes for a total of 10000 episodes.

Comparing Figure 6.6 to the convergence of the previous prototypes, it can be observed that there is much more variation in the learning progress, but the tendency to approximate an optimal policy is clearly evident. This variation may be due to the fact that the agent's learning problem has become increasingly multifaceted with the refined data. Essentially, when interacting with the environment, the agent is facing two slightly dif-

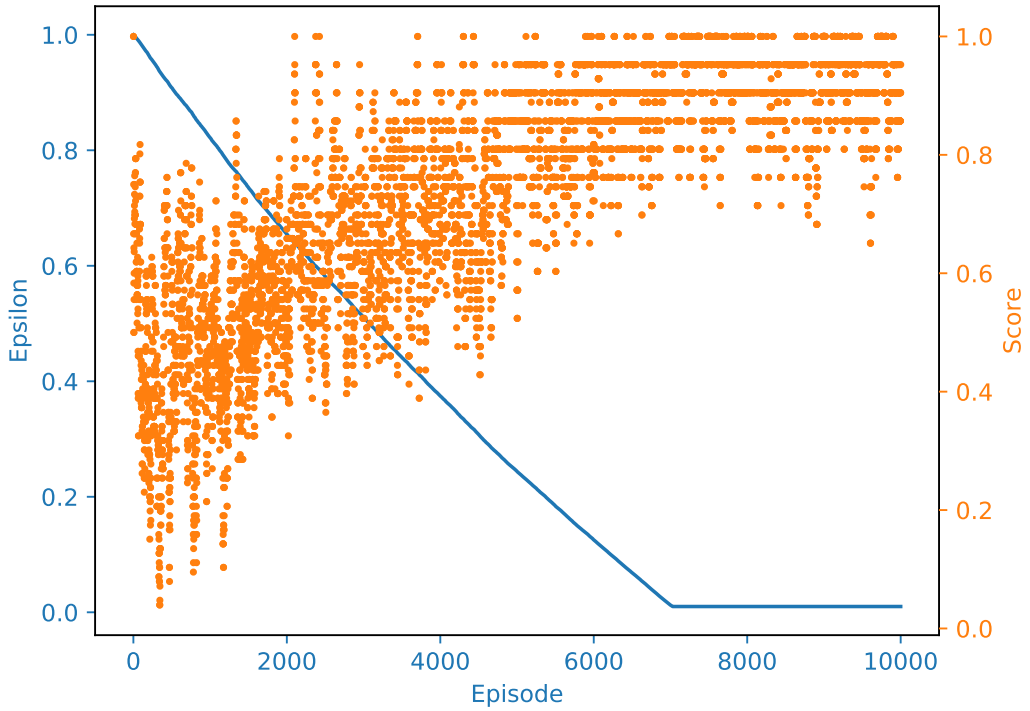

Figure 6.6: Prototype 3: Learning Progress over episodes and epsilon decay: 1 sample used for AE predictions.

ferent challenges: First, to predict a correct MTD from a decision state, and secondly, to choose the right MTD from afterstates when it fails on decision states. Analogously, the anomaly detection task has become more versatile as many combinations of MTD techniques and behaviors have to be recognized. More possibilities for states also imply that there are theoretically more points of failure and chances of encountering FN or FP. The transition from the second to the third simulation prototype reflects the growth of the general state-space which has the goal of approximating reality.

After finishing the training process, the accuracy of the agent's greedy action choices is evaluated on the separate and unseen test set as done for the previous prototypes. In contrast, here both decision state and afterstate is used for the evaluation of the agent's greedy action selection according to its Q^O online DQ-Network. Table 6.3 presents the results on decision state data, and Table 6.4 those of afterstates respectively. Note that here, no meaningful accuracy can be evaluated for all behaviors that should be considered normal (see column `Objective` in Table 6.2). Thus, all these states considered normal are not listed in the tables. After deploying the correct MTD technique, but flagging it mistakenly as a FP, every MTD technique can be executed and it is both correct (as the afterstate is normal) and incorrect (as nothing needs to be deployed).

Similarly to the MTD selection accuracies achieved in the second prototype Table 5.3, the performance on decision state data of the backdoor jakoritar (77%) and the tick attack (64%) are better than the poor autoencoder performance would suggest (5%-7%). The

Table 6.3: Prototype 3: Agent Performance on Decision State Data

Behavior	Accuracy	Objective
ransomware_poc	96.58%	ransomware_mtds
bdvl	93.15%	rootkit_sanitizer
backdoor_jakoritar	77.10%	cnc_ip_shuffle
beurk	15.31%	rootkit_sanitizer
the_tick	63.92%	cnc_ip_shuffle
data_leak_1	100.00%	cnc_ip_shuffle
data_leak_2	100.00%	cnc_ip_shuffle

Table 6.4: Prototype 3: Agent Performance on Afterstate Data

Behavior	MTD	Accuracy	Objective
ransomware_poc	cnc_ip_shuffle	100.00%	ransomware_mtds
ransomware_poc	rootkit_sanitizer	96.67%	ransomware_mtds
bdvl	ransomware_directory_trap	94.93%	rootkit_sanitizer
bdvl	ransomware_file_extension_hide	91.87%	rootkit_sanitizer
bdvl	cnc_ip_shuffle	91.47%	rootkit_sanitizer
backdoor_jakoritar	ransomware_directory_trap	70.07%	cnc_ip_shuffle
backdoor_jakoritar	ransomware_file_extension_hide	73.25%	cnc_ip_shuffle
backdoor_jakoritar	rootkit_sanitizer	60.48%	cnc_ip_shuffle
beurk	ransomware_directory_trap	16.58%	rootkit_sanitizer
beurk	ransomware_file_extension_hide	15.40%	rootkit_sanitizer
beurk	cnc_ip_shuffle	15.52%	rootkit_sanitizer
the_tick	ransomware_directory_trap	64.27%	cnc_ip_shuffle
the_tick	ransomware_file_extension_hide	63.37%	cnc_ip_shuffle
the_tick	rootkit_sanitizer	60.10%	cnc_ip_shuffle
data_leak_1	ransomware_directory_trap	100.00%	cnc_ip_shuffle
data_leak_1	ransomware_file_extension_hide	99.76%	cnc_ip_shuffle
data_leak_1	rootkit_sanitizer	100.00%	cnc_ip_shuffle
data_leak_2	ransomware_directory_trap	100.00%	cnc_ip_shuffle
data_leak_2	ransomware_file_extension_hide	100.00%	cnc_ip_shuffle
data_leak_2	rootkit_sanitizer	100.00%	cnc_ip_shuffle

agent only fails to learn to select the correct MTD for decision state beurk in the majority of cases. All other attacks are correctly mapped to the mitigating MTD technique with accuracies between 93% and 100% as can be seen in Table 6.3.

The agent's performance on afterstate is very close to the performance on the decision states as also observed from the autoencoder results. After deploying incorrect MTDs first, the correct MTD technique is selected with 60%-70% accuracy for the tick and the backdoor, and 92%-100% for all other afterstate combinations except those with beurk behavior. As beurk behavior is seldomly detected as anomalous in both decision and afterstates, incorrect replay memory entries also translate to poor MTD selection accuracy. As also shown in the second simulation, the convergence seems to be relatively robust despite the possibility of attacks (like beurk) that are not detected well and thus lead to incorrect training samples.

6.4 Summary

This chapter establishes the transition from the more theoretical offline learning scenarios presented in the previous Chapter 5, towards an increasingly realistic simulation environment that is based on real data observed from an online MTD controller agent.

First, as a base, the design of such an MTD controller is presented, including all its components, which enable to autonomously learn and interact in a real environment. Then, this controller is used to collect a second, more refined dataset that distinguishes decision- and afterstates. This dataset is analyzed and its suitability to be used for sampling states in a further, third simulation environment is demonstrated. It is shown, that features remain stable over time for given behaviors, but that there are differences for decision- and afterstates, dependent on deployed MTD techniques. Features that maximize these differences are removed, such that undesirable effects of the agent's actions on the malware detection can be reduced. In line with this, and as expected, correct MTD techniques generally lead to behavior that is close to normal, and incorrect techniques result in behavior that is close to the behavior of the device prior to MTD execution.

Coming from this analysis an autoencoder is trained on normal decision- and afterstate data and utilized for constructing the environment signals within a third simulation. With a few exceptions (beurk, backdoor jakoritar, the tick), most behavior states are recognized correctly for their normality which translates to the agent's learning capability. Thus, the optimization of the anomaly detection step is critical. Despite showing a little more variance in the learning progress due to a larger state-space, an accordingly trained agent effectively converges towards a strategy which is close to optimal. Apart from beurk, the agent learns to map mitigating MTDs to all behaviors. A beneficial factor for training the agent is also given by not allowing to deploy a technique more than once per episode. Further, when comparing the results of this chapter to the previous agents from Chapter 5, no loss in performance can be identified due to the refined data, validating the choice of the autoencoder and DQ-network models.

In summary, this chapter achieves the pretraining of an agent which effectively mitigates attacks as observed from a real environment with its available set of MTD techniques.

Within given limitations of the simulation environment, the agent thus effectively learns a policy which can be transferred to an online, real-world setting. This significantly reduces problems arising from learning online from scratch. However, to effectively work in an online scenario, further considerations need to be made. Therefore, the next chapter provides insights into concrete implications of deploying an agent online.

Chapter 7

Online Agent Implications

The previously presented simulations work offline based on precollected data. Thus, the challenge was primarily to construct environments that simulate reality as close as possible. Yet, coming from this point of view, many issues did not have to be addressed that are more urgent in the online setting. Hence, this chapter discusses the implications of utilizing an RL-based MTD agent in a real, online environment.

It heads off with implementation details of the MTD controller to lay a base for understanding the major hyperparameter options. Next, besides elaborating on options of the anomaly detection component and the state monitoring duration, the chapter discusses the case of non-mitigatable attacks. Finally, to assess the viability and feasibility of RL-based MTD, the chapter concludes with an evaluation of the agent's required resources when deployed on a RP device.

7.1 Controller

Algorithm 2 presents the pseudocode for the implementation of the online MTD controller. As already explained in Section 6.1, the controller marks the core component, which orchestrates the observation of states, anomaly detection, RL and MTD execution. In essence, Algorithm 2 displays the code corresponding to Figure 6.2, while abstracting away the details of DQ-Learning (as per Algorithm 1) behind methods of an agent object and autoencoder object.

As visible from lines one to five in Algorithm 2, several initialization steps have to be performed, before online RL-based MTD can be activated. First, an autoencoder model as pretrained from the third simulation (on normal decision- and afterstate data as per Section 6.3.2) should be loaded, with an accordingly set threshold. Analogously, a pretrained agent has to be loaded with a prefilled replay memory D. This agent should be pretrained in the most realistic simulation environment possible (as in the third prototype), to ensure that the learnt policy as given by its online DQ-Network Q^O is transferrable to the online setting. Besides Q^O, the target network Q^T also needs to be loaded to ensure stable update targets. The agent's remaining hyperparameters need to be selected specifically

for the use case at hand. The reward discount factor γ should be set close to zero to focus more on immediate rewards. Further, the batch size used for the learning update can be considered similar as in the simulation environments. However, the other hyperparameters need more careful consideration:
If the simulation environment can be trusted to model reality accurately and if the agent was trained on a large number of different attacks and selects correct MTD techniques, the exploration rate ϵ should be set to a relatively low number (i.e. performing random actions in only 2% of the cases). However, if training happens only on few attacks and the online environment differs remarkably from the simulated states, the exploration rate should be much larger. Another important aspect is given by choices regarding the replay memory. If pretraining is difficult, the replay memory does not necessarily need to be prefilled. However, the agent can only perform the learning update if at least *batch_size* samples are stored in the replay memory. Thus, in that case, sample transitions must first be stored as the agent observes them on the fly using a random policy before it starts learning.
Besides the choice of how the memory is initialized, it is crucial to decide on the size of the memory. The replay memory is set up as a ring buffer, meaning that old transitions are exchanged for new ones, based on a first-in first-out (FIFO) manner. In the previous simulation prototypes it is possible to randomly sample attack behaviors which ensures that the replay memory always contains a relatively balanced set of attacks. However, in the online case, it is realistic to assume that the device is not attacked in such a balanced manner. In fact, a certain family of malware (i.e. CnC-based) may heavily dominate other attacks. Thus, if the replay memory size is very small, almost no transitions for the other malware families are considered for learning as they are pushed out of the buffer. Of course, the problem of unbalanced attacks still persists with a very large memory size. However, with a very large memory we can guarantee that transitions of rare attacks can actually be sampled as they are at least within memory. Thus, if a lot of storage space is available the replay memory should be initialized to contain many samples.
In theory, there is again a potential problem due to non-stationarity in the environment, meaning if the environment changes over time. As already touched upon in Subsection 2.5.4, the replay memory allows to decorrelate actions that often occur in sequence. This is also the case if the device is attacked with the same malware repetedly for longer periods. So if the environment changes fast and the memory is large, there will be a lot of ”old” transitions stored in memory, not allowing the agent to timely adapt to the changed circumstances. This is also the case if the agent needs to adapt to new, unseen attacks. In the application context of ElectroSense, where the environment can be assumed stationary, the replay memory should be very large, yet limit its size according to adaptation needs. A smaller memory also comes at the expense of a slightly more unstable learning progress. The memory size should also be dependent on the number of attacks available for pretraining and certainly must be analyzed in more detail for a particular use case.

Apart from RL-related decisions, two further important parameters need to be selected related to time. First, an interval I (in seconds) must be specified for which the controller waits before it starts the next monitoring cycle. In theory, it is possible to not wait at all and continuously restart monitoring decision states, if the service quality of the device is not remarkably impacted by the MTD controller. However, in any case it would also lead to a lot of wasted resources. A waiting interval in the range of a few minutes seems to

be a good trade-off between security level and limiting resource consumption. Section 7.4 provides further details on this matter. The last and time-related parameter to select concerns the duration for which decision- and afterstates should be monitored. Certainly, this duration may neither be too long nor too short. For instance, if the duration is too short, it is possible that no state samples can be collected, or that they are removed as outliers, which results in an error. The monitoring script is set to record a vector of `perf` events every 5 seconds, besides adding some features monitored via the top command. Thus, to be safe for the given scenario, the monitoring duration should be set to at least 20-30 seconds. In case of the duration being too long, a temporary running malware might already be finished by the time the agent gets to choose an MTD technique. Aiming to exclude these two cases as far as possible, the monitoring duration should be set in the range of 50-100s. Note that in principle, for both the autoencoder prediction, as well as the agent's action choice, multiple state samples can be considered (lines 9, 12 and 16 in Algorithm 2). Moreover, based on that, it is also possible to store multiple transitions per agent-environment interaction step in the replay memory, which influences the agent's learning progress. Further considerations regarding the monitoring duration are postponed to Section 7.2.

7.2 Multisampling

As already mentioned in Section 7.1, the monitoring duration for decision- and afterstates is an important parameter influencing the security level and overall resource consumption. In addition to that, the number of behavior samples collected opens up different implementation options. This is especially relevant for the anomaly detection step (lines 9 and 16 in Algorithm 2). Instead of just using a single sample to judge a state for normality, it is also possible to feed multiple samples in the anomaly detector and make the judgement based on the average prediction. Intuitively, if there is a confidence level of above 50% for a single sample prediction, the probability for making the correct judgement increases with the number of samples if an average is taken. For instance, if there are 19 state samples available and if the autoencoder predicts with 80% accuracy on average, it is extremely likely that at least 10, meaning more than 50% of the samples are predicted correctly. Ideally, this fact can be leveraged to make a sound decision on the number of state samples to monitor and use for the anomaly detection steps. The probability for the previous example can be calculated using the upper binomial cumulative distribution function:

$$P(X \geq x) = \sum_{k=n/2}^{n} \binom{n}{k} \cdot p^k (1-p)^{n-k}$$

p denotes here the probability of success, meaning the accuracy of the anomaly detector to correctly predict a single state sample. Thus, the total probability for making a correct judgment is given as the sum of the probabilities of all cases where more than 50% of total n samples are predicted correctly.

Figure 7.1 displays the curves for accordingly calculated probabilities for different levels of anomaly detection accuracies p and number of state samples n available. As can be observed from the black curve (accuracy $p = 0.8$) it is almost certain to make the correct

Algorithm 2 Online MTD Controller Implementation Pseudocode

```
Load pretrained autoencoder model and threshold
Load pretrained agent, including online- and target networks Q^O and Q^T and prefilled replay memory D
Select learning rate, reward discount factor γ, exploration ϵ, batch size, buffer size dependent on the training status of the agent
Select time interval I in seconds to wait before starting the next episode
Select monitoring duration d for decision and afterstates
for episode 1, ∞ do
    Monitor decision state for d seconds
    Read and preprocess decision state decision_data from csv file
    isAnomaly ← autoencoder.interpret(decision_data)
    if isAnomaly then
        for t = 1, T (max timesteps within an episode = 4 MTD Techniques) do
            mtd ← agent.choose_action(decision_data)
            Execute MTD technique mtd
            Monitor afterstate for d seconds
            Read and preprocess afterstate after_data from csv file
            isAnomaly ← autoencoder.interpret(after_data)
            reward ← calculate_reward(isAnomaly)
            Store transition (decision_data, mtd, reward, after_data) in D
            agent.perform_learning_update(D)
            if not isAnomaly then
                Terminate Episode
            else
                decision_data ← after_data
            end if
        end for
    end if
    Wait for I seconds
end for
```

decision on normality with 20 samples available. Thus, it could approximate what a supervisor achieves (as in Section 5.1.2). However, if the anomaly detector performance is below 50% accuracy, the probability of making a correct decision approaches zero with n increasing.

The previous analysis shows that it can be both good and bad to make the anomaly detection decision based on multiple samples. If the autoencoder performs well with 70% accuracy and above on some behavior, the correct decision will be made with near perfect certainty with even 10 - 20 samples considered. However, the exact opposite applies for a poor detection accuracy of 30%. Thus, when such multi-sampling is applied, it can be expected that it is increasingly difficult for agent to learn to map MTD techniques to attacks which are hard to distinguish from normal behavior. Attacks distinct from normal behavior on the other hand will very rarely result in incorrect transitions for the replay memory.

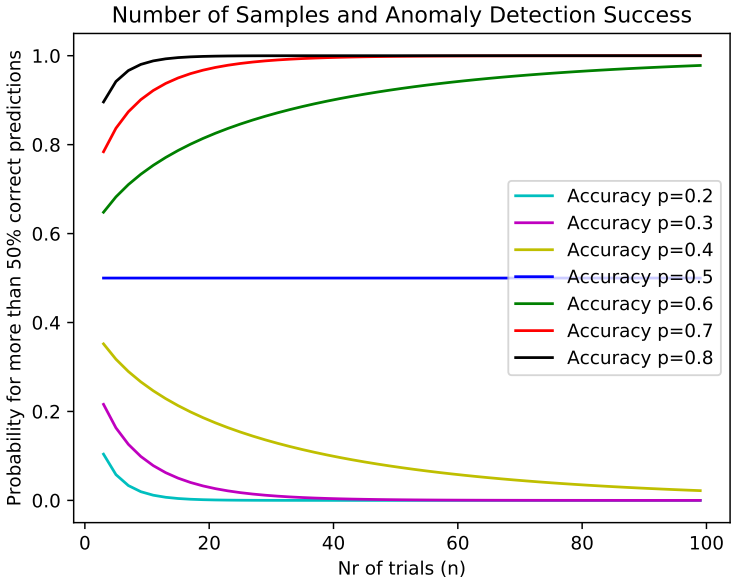

Figure 7.1: Probability for making the correct anomaly detection decision based on an average of different numbers of state samples n and anomaly detection accuracy levels p

To assess the effects of this, multisampling has been implemented for the second and third simulation and the number of samples considered can be selected as a hyperparameter [56]. Experiments show that considering multiple samples for anomaly detection generally leads to more extreme results. Some attacks (like beurk) are mitigated very rarely, others with perfect certainty (i.e. ransomware).

Thus, dependent on the security goals and type of malware considered as the most important to defend against, multiple samples should be considered or not. A security strategy that aims to reactively catch every possible type of attack should certainly go for a single sample. But dependent on the attacks which are most probable against the system, more attacks may be mitigated in total if multiple samples are leveraged. Respectively, convergence may be faster as no incorrect transitions are stored for the primary target malware. More than 15 samples (corresponding to ≈140 seconds of monitoring) should certainly be avoided such that no resources are wasted. There is no advantage in monitoring for longer durations. Further, as can be observed from Figure 7.1, the probability to detect an attack may drop to zero if the autoencoder accuracy is below 50%. Thus, in the worst case the agent is never even triggered at all and stuck in an endless loop until another attack is launched on the device that can be detected. To avoid this, a low number of samples is recommended or a combination with different security strategies, such as proactive MTD for instance. In fact, a proactive defense can be integrated naturally with the RL-based reactive MTD system. Data monitoring and anomaly detection starts at defined intervals (I in Algorithm 2). Thus, a proactive strategy could simply deploy certain MTD techniques in a random manner, despite detecting normal behavior. Another option is to launch proactive MTD techniques completely decoupled from anomaly detection, during the waiting interval I. This would certainly add to making the attack surface of the device increasingly dynamic and randomized. Further, it is symbiotic to reactive MTD as the agent does not need to have found a perfectly optimal strategy yet. However, lastly, it

should also be noted that this might impact some state features, even though they have been optimized to minimize effects of MTD techniques.

7.3 Non-Mitigatable Malware

When executing an RL-based MTD controller in an online environment, the possibility of attacks that cannot be fully defended against with the current set of MTD techniques must be taken into account. Note that this also includes the case of imperfectly working MTD which may sustainably alter device behavior. If this is not done, there is the threat of filling the replay memory with many useless samples, that possibly destroy the agent's performance.

Anomaly detection is the principal measure to decide upon the presence of malware. In theory, assuming that the anomaly detection and all MTD techniques work perfectly, each MTD technique can be executed exactly once (one episode) and then, if device behavior is still recognized as anomalous, it can be concluded that malware of an unknown, non-mitigatable type is running. In such a case, a special warning can be issued and special measures must be taken to clean the device. However, in reality, neither anomaly detection nor MTD techniques can be guaranteed to work perfectly. Thus, due to FP, one episode may not be enough to decide on whether non-mitigatable malware is present on the device. However, with multiple episodes (and deployment of all MTD techniques), the presence of non-mitigatable malware can be detected with increased confidence. Luckily, as per the autoencoder results in Table 6.1 and Table 6.2, FP are relatively rare, as normal behavior is generally detected well (normality detected in afterstates of correct MTD techniques with >87% accuracy). Thus, the probability to thwart-off mitigatable malware within the first two to three episodes already approaches near perfect certainty. In conclusion, after executing the number of available MTD techniques twice and still detecting an anomaly, the decision that non-mitigatable malware is running, can be made with very high confidence. Respectively, it is extremely likely that the original, normal device behavior has somehow been changed due to malicious activities. Thus, the device should be taken offline for a reset. In an context like ElectroSense, where it is not an issue to temporarily go offline, the decision about non-mitigatable malware may also already be made after only a single episode of MTD deployments. However, in the general case the number of episodes to consider before cleaning the device certainly depends on the impact on a system's service quality, security goals as well as the anomaly detection accuracy.

7.4 Resource Consumption Evaluation

In order to assess the feasibility of the full online agent presented in Section 6.1, it should be evaluated for its resource requirements on the platform of deployment. In the context of this work, this is done for a RP 3 Model B+ running RP OS 11 (Bullseye). Information of particular interest comprises the processing time of the agent's components, storage capacity needs, as well as CPU and RAM requirements. Such data on one hand helps to

determine the resource demands on the device itself, and on the other hand provides a reference for comparable applications in the IoT context. In order to gather such data, the complete online agent code [56], including the monitoring and anomaly detection components and MTD scripts are moved to the RP. For the anomaly detector, an autoencoder pretrained on decision- and afterstate normal behavior is used as described in Section 7.1 and Section 6.3. Similarly, the agent component itself is based on a pretrained and saved model of the agent from the third simulation.

7.4.1 Processing Time

As the proposed MTD Agent is thought to be run as a permanent service, processing times can be measured more meaningfully for each functional component than for a complete training cycle. The functional components match the different lanes presented in Figure 6.2, for monitoring, anomaly detection, MTD execution and RL. Certainly, processing times may vary dynamically, dependent on attack behaviors, MTD techniques deployed, the current status of the agent as well as external effects. Thus, it is in general most useful to estimate an upper bound for processing times.

Accordingly, Table 7.1 displays processing times monitored by setting timestamps at dedicated locations in the agent controller code for each functional component. The first column shows the time required to execute the environment monitoring script for a chosen duration of 100 seconds and to preprocess the resulting state samples (here both 11 samples for decision and afterstate). The time indicated includes outlier filtering as well as scaling with a preloaded Min-Max scaler from the third prototype (based on normal decision- and afterstate extreme values). While the monitoring duration can vary as a hyperparameter for the online agent, the preprocessing comes with negligible overhead on the RP (0.08s). The time required for the anomaly detector to evaluate the decision, respectively the afterstate (here 8 filtered samples each) is also in the range of few hundredths of a second, similar to the agent's action choice as well as the learning update step. The time for the agent's action choice is evaluated by feeding all decision state samples (8) into the Deep Q-Network and taking the most frequently predicted action. As both the anomaly detection and the action choice could also be based on a single sample, the processing times indicated mark an upper bound (lines 9, 12 and 16 in Algorithm 2). The time for the agent update comprises both storing the newly observed samples in the replay memory as well as performing the gradient descent step as shown in Algorithm 1 on a random batch of 100 stored transition samples. For this evaluation, the online agent leverages a preloaded replay memory from the agent trained in the third simulation.

Table 7.1: Processing Times

Monitoring & Data Processing decision/afterstate	**Anomaly Detector Evaluation** decision/afterstate	**Agent Action Choice** ϵ-greedy/DQN	**MTD Execution** agent action	**Agent update** DQN SGD on replay memory
100.08s/100.07s	0.025s/0.02s	0.014s	ms-mins	0.05s

Table 7.1 clearly shows that no stage of online, RL-based MTD is notably hampered by excessive processing times. This means that a pretrained agent as from the third

simulation will be an effective and efficient measure against the considered families of malware.

Besides the case of simply using a pretrained agent, it is interesting to estimate how long it would take to learn an agent from scratch in the online, real-world setting. The formula below estimates the total processing time T for the worst case scenario where the maximum of 4 available MTD techniques is deployed per episode:

$$T = num\_episodes \cdot (1 \cdot ds\_monitor\_duration + 4 \cdot sum\_processing\_time\_controller + 4 \cdot as\_monitor\_duration + 4 \cdot worst\_case\_MTD\_duration + waiting\_interval) \quad (7.1)$$

When setting *num_episodes* to 6000 in this formula, using 100s as the decision- and afterstate monitoring duration, taking 2s as an upper bound for the *sum_processing_time_controller*, 1 minute for the worst case MTD duration and choosing a waiting interval of 3 minutes between new episodes, the total processing time required to train the agent accumulates to 64.4 days. This is based on the assumption that the agent actually converges after 6000 episodes (as estimated from figures 5.6 and 6.6), and that new attacks are being launched constantly. Thus, the waiting interval is set as relatively short with only three minutes from a realistic point of view. The result of requiring more than sixty days to train an online agent once more clearly demonstrates the need for pretraining in simulations.

However, an agent does not need to have fully converged to achieve significantly better than a random MTD deployment strategy. Thus, by assuming that there is an agent pretrained on some attack behaviors and moved online, we can simply take a lower number of episodes in the above formula to estimate the time required to a sufficient level convergence. For instance by setting $num\_episodes = 3000$ the required time is reduced to 32.2 days with the same assumptions as above. Despite that this is still a very long duration, it might not be as dramatic. One reason for this is that each MTD is deployed at most once per episode, so any attack behavior that is detected as an anomaly is mitigated at some point. Only attacks too close to normal, or attacks that only run temporarily might slip this defense. However, if an attack is too close to normal, not even a fully trained agent may correctly react to it. Further, short and temporary attacks are hard to reactively mitigate in general without continuous monitoring, independent of the chosen defense approach.

Additionally, due to the possibility of zero-day attacks, an agent can always be seen as only partially converged. This is the advantage of RL in the first place: To dynamically figure out new mitigation strategies without making any assumptions about current device behavior or attacks in general. Thus, partially trained or not, the agent's MTD selection policy is still significantly better than random action selection or proactive MTD only. As already mentioned at the end of Section 7.2, by additionally employing proactive MTD deployments aside from the agent's reactive deployment strategy, a fairly good level of security can be achieved, less dependent on the current learning progress.

7.4.2 Disk, CPU and RAM Requirements

To measure disk requirements imposed by the agent, the `du` command is used. The complete directory of the online controller, including all MTD techniques, agent code and models requires less than a single MB of storage (964KB). The pretrained agent takes the majority of space as it stores all hyperparameters, online and target networks, as well as a prefilled replay memory of 500 transitions. The model of the pretrained autoencoder takes 8KB of disk space and the csv files resulting from monitoring decision and afterstates for 100 seconds (11 samples) take 8KB each as well.

Certainly required storage depends a lot on all the different hyperparameters selected, as well as the concrete implementation and functionality. However, the overall disk space occupied is absolutely minimal. Thus, storage is certainly not a limiting factor for the application of RL-based MTD in the ElectroSense context.

In order to assess CPU and memory requirements of the agent, we are again interested in an upper bound that an IoT device must be able to handle. Ransomware certainly belongs to the most CPU and memory impacting malwares, due to the resource-intensive encryption process. This has been shown in [12], and is also visible from the data analysis in this work. Correspondingly, the MTD technique that requires most resources is the MTD that aims to trap the encryptor process with an expanding and collapsing directory structure.

Thus, CPU and RAM usage is recorded along the execution of ransomware-poc and the directory trap MTD launched from within a single agent-environment interaction step. The exact monitoring procedure performed on the RP is as follows: First, Nigel's performance monitor (nmon) is launched in recording mode on the RP using the command `nmon -f -s1 -c400`. This ensures that 400 samples on memory and cpu statistics are taken at an interval of one second and stored to a file. Next, after 10s, ransomware-poc is used to attack the device by encrypting a nested test directory structure containing 1MB of random files. Finally, the online agent is started with a monitoring duration set to 100s, and the directory trap technique hardcoded after the online-network action choice. After finishing a single agent-environment interaction step, the recorded vectors for CPU and RAM usage are plotted against the timeline. Figure 7.2 displays the according resource usage.

Focusing first on the figure showing the free memory over time, it is possible to see the different stages of the agent execution. First, about 70% of the memory are free as neither malware, nor agent code is running. Then, there is a sudden drop in free memory to about 30% when the ransomware is launched against the RP. The minimum is reached at $\approx 20\%$ when the agent controller is actively running. At about 120 seconds in the timeline, the controller has finished monitoring and launches the directory trap MTD. After mitigation at about 200 seconds the level of free memory stabilizes at $\approx 45\%$. The percentage of CPU dedicated to user-level operations reaches its maximum at approximately 55%, after 140 seconds which is in line with the MTD execution. The maximum of CPU system usage is reached during the same time window with $\approx 25\%$. Together, the minimum level of idle CPU is reached at about 25%. Essentially, this window corresponds to the time when the agent has launched the directory trap MTD, and it is trying to capture the

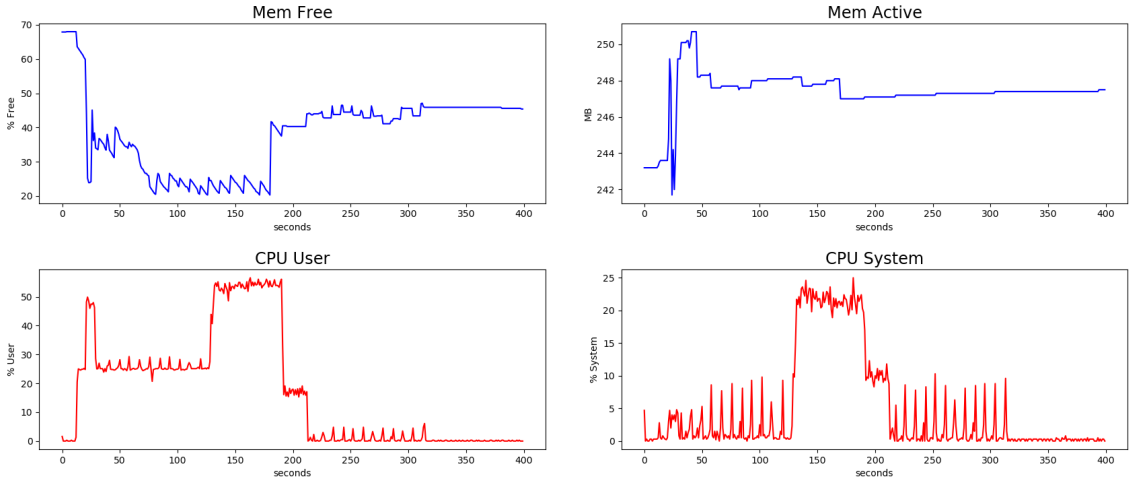

Figure 7.2: CPU and RAM Usage of the MTD Controller Agent for Ransomware and Mitigation with the Directory Trap MTD Technique

ransomware's encryptor process. Once the encryptor is killed, a lot of memory can be freed, as visible by an increase of $\approx 20\%$ at 190 seconds. Similarly, a drop to close to zero till 7% CPU usage can be observed after a bit more than 200 seconds. The percentage of the active memory also stabilizes after this timewindow on a about a fourth of the available memory (250MB). The timewindow between ca 210 and 320 seconds is used for monitoring the afterstate and performing the learning update and storing of transitions in the replay memory.

Even though this is the most resource-consuming scenario that the agent may encounter with the given attacks and MTD techniques, the RP 3 does not seem to have any troubles executing all the agent's processes in parallel. Thus, from a resource requirements perspective, the agent should be fully functional without delays or other issues along its execution.

7.5 Summary

This chapter presents the major implications of setting up an MTD controller agent in a real, online environment. First, the learning loop as orchestrated by the controller is analyzed with a view towards selecting the most important hyperparameters. An autoencoder pretrained on normal decision- and afterstate data is necessarily required for the learning loop to work and its model needs to be preloaded. The agent itself does not necessarily need to be pretrained in a simulation, but it is highly desirable to ensure an effective MTD policy.

Important hyperparameters regarding the agent concern the initialization and the size of the replay memory, as well as the waiting interval (I) between episodes, and the state monitoring duration. If sufficient storage capacity is available and attacks can be expected to be launched against the device in a relatively balanced manner, the replay

memory should contain as many samples as possible. However, if attacks occur relatively unbalanced, a medium size replay memory size is recommended to not limit the agent's adaptation capability, but still ensure decorrelation of transitions. If possible, the replay memory should be initialized with monitored samples, such that learning can occur right at time of online deployment.
For the ElectroSense use case the waiting interval between episodes should be set to a few minutes as a tradeoff between security level and resource consumption. This also takes into account the option for possible proactive MTD deployments during the waiting interval.
The duration to monitor states must ensure that there is at least one sample available. Taking outlier filtering and processing delays into account, the duration should not be set below 30 seconds for the given use case. Further, the monitoring duration should not exceed approximately 140 seconds (corresponding to around 15 samples) to avoid wasting resources and precious mitigation time.
The number of state samples collected can further be exploited for improved anomaly detection. When the accuracy of the autoencoder is well above 50%, confidence in deciding whether the device behavior is abnormal can be greatly improved by considering an average of the predictions of multiple state samples. However, behaviors that are detected poorly (<50% accuracy) are detected worse when such multisampling is applied. Thus, dependent on the security strategy a single to few samples should be used for anomaly detection. Ideally, a multisampling strategy of the reactive MTD controller could be combined with a proactive strategy that randomly deploys techniques dependent on how long they have not been utilized.

Further, in the online scenario, the case of non-mitigatable malware cannot be neglected. If an attack cannot be fully defended against with the current set of available MTD techniques, the learning loop must be interrupted to avoid filling the replay memory with useless transitions. If correct MTD techniques are chosen that thwart off a given attack, the normality in the afterstate is recognized with high accuracy (>87%). Thus, a warning about the presence of non-mitigatable malware can be issued with high confidence after very few episodes of complete MTD technique deployments.

Finally, this chapter evaluates the resource requirements of the implemented MTD controller agent along the dimensions processing time, as well as disk, CPU and RAM usage. Processing times needed for each functional component show that there is no notable time-related issue that could prevent the proposed system from effectively mitigating attacks. However, the time required for training from scratch poses a severe limitation. Accounting for 64 days in a selected worst case scenario once again clearly shows the need for realistic pretraining in a simulation. But even with pretraining, the adaption to new attacks may take a considerable amount of time, such that further strategies are needed. The complete MTD controller, including all models and MTD techniques takes less than a MB of disk space. The CPU and RAM usage was evaluated for the most resource-consuming scenario as given by the ransomware attack mitigated by the directory trapping MTD technique. The results show that the amount of free memory never drops below 20% (leaving approximately 800MB free for the RP 3 with 1 GB RAM) and the total CPU load never exceeds 70%-80%. Thus, the feasibility and viability of the proposed RL-based MTD system is not impeded by any hardware constraints.

Chapter 8

Conclusions, Limitations and Future Work

8.1 Conclusions

This work explores the use case of RL for MTD technique selection to thwart off a range of malware families (ransomware, rootkits and CnC-based) against an exemplary, real-world crowdsensing platform. In particular, this work analyzes the problem of integrating reactive MTD into the RL framework and distills the functional components required to enable the interaction of an agent with an online environment.

Further, this work presents three prototypical simulations for training MTD selection agents. Initially starting from rather theoretical simulated environments to explore the application domain, the complexity is increased at different levels to account better for a real online scenario.

The first simulation prototype marks a baseline and shows what RL can achieve under ideal conditions. It samples states from a precollected dataset of ideal and raw attack behaviors upon which the RL agent has to learn what MTD technique to choose. The feedback on the agent's action is given in a supervised manner based on prior knowledge of what MTD technique mitigates which attack. This ensures always providing correct rewards and episode states to the agent, but obviously it cannot transfer to an online scenario. All malware families considered (ransomware, rootkits and CnC-based) are correctly mapped to mitigating MTD techniques with slight differences among specific attacks (≈70% minimum accuracy for beurk, and backdoor jakoritar, 98% and above for all others).

The second prototype removes the previous need for a supervisor by adding an anomaly detection component which is used to interprete states after MTD execution to estimate environment signals. This is based on the assumption that correct MTD techniques result in normal behavior, and should be rewarded, whereas incorrect MTDs do not change the attack behavior and should be penalized. This prototype theoretically enables the transfer to fully unsupervised and autonomous online learning, but it comes with the

drawback of imprecisions given by anomaly detection. Thus, while still sampling from ideal raw behavior data, this simulation measures how much impact anomaly detection has on the system's learning capabilities. Poor anomaly detection accuracy on attacks results in the storage of incorrect transitions in the agents replay memory, which effects the agent's MTD selection performance. When comparing the accordingly trained agent to the supervised prototype, the selection accuracy drops considerably for attacks which are hard to distinguish from normal behavior (notably beurk and the backdoor jakoritar attack). The MTD selection accuracy on other attacks is, however, not affected.

The third prototype works exactly as the second, yet increases the complexity with respect to the data used for state sampling. It leverages a second precollected environment dataset that accounts for disturbances induced by an MTD agent controller itself, besides the raw attack behavior. This dataset includes afterstates monitored after launching all combinations of MTD techniques and attack behaviors on an ElectroSense sensor. As this simulation considers fully realistic data, that also an online agent would observe, it is not only useful for analyzing the learning process, but also for pretraining agents that can be used in a real, online environment. Hence, it allows the transfer of the policy learnt in the simulation, to the real world, which ensures that there is no need for an extremely resource-consuming process of learning online from scratch. The evaluation of such a pretrained agent shows, that the MTD selection performance does not decrease compared to the previous prototypes, but that there may be a slightly larger variation in the learning progress. One influential factor for this is certainly an increased state-space as the simulated environment approximates reality. The agent's learning is impacted by anomaly detection inaccuracies similarly to the second prototype, highlighting again the importance of improving anomaly detection performance for the given approach.

Furthermore, aside from training in simulations, this work presents the design and implementation of a fully functional, online MTD controller agent. This controller can autonomously perceive states in a real sensor environment, detect anomalies, select and execute MTD techniques reactively and is capable of interpreting resulting afterstates. Hence, it theoretically allows to learn from scratch without any prior knowledge. Yet within specified limits, by making use of the autoencoder and action-value networks as pretrained in the third simulation, it represents a fully operational online learning agent right after deployment.

Finally, this work presents an evaluation of the resource consumption induced by the online agent on a RP 3 device with 1GB RAM as it is used for ElectroSense. The full code takes less than a single MB of disk space of the RP and the usage of CPU and RAM never exceeds 80%. Hence, the leveraged hardware clearly does not pose a limiting factor for the feasibility or viability of the proposed MTD agent in the crowdsensing use case. However, executing 6000 episodes is estimated to take as long as $\approx$64days in the worst case. Thus, dependent on the pretraining options and possibility of policy transfer to the online setup, it could take very long for an agent to become both effective and efficient. For a more holistic defense, and to reduce the issues of online-learning, the combination of the current reactive approach with proactive MTD is recommended.

8.2 Limitations

As given by the assumptions discussed throughout this work, RL-based MTD selection comes with some limitations. The most important limitation is the time and data required for RL. RL is a very resource- and time-consuming process that requires a lot of interaction data. Online trial and error learning from scratch is generally impractical or unfeasible such that a realistic simulation is needed for pretraining an agent before deploying it online. But even in case of employing a pretrained agent, the agent might take very long to learn to mitigate new attacks that have not been observed in the simulation. Thus, a simulation environment must consider a wide range of different attacks as they might occur for real and leverage a precollected dataset of real state samples. If this cannot be guaranteed, the transferrability of the policy learnt in the simulation to a real environment is severely limited.

A further potential limitation to the learning capability of an MTD agent is the case of imperfectly working MTD techniques. If for instance, some part of a malware continues to run after all MTD techniques are executed but it does not have a malicious effect, the RL-based MTD system is trapped in a bad state of changed normal behavior and requires a reset (as per Section 7.3).

Finally, an important limitation with respect to extensibility of the proposed RL-based approach is given by the action value networks utilized for DQ-Learning. The size of the output layer of the Q-networks must be set according to the number of actions, respectively MTD techniques available. Thus, if new MTD technique must be integrated, the output layer of the action-value networks need to be removed, its size must be increased by the number of new techniques and retraining is required. Thus, to avoid expensive learning operations, the set of MTD techniques utilized for RL should not be changed frequently.

8.3 Future Work

Besides addressing the previously described limitations, there are multiple promising options for future research going in a similar direction as the approach proposed in this work. First, future work could consider incorporating a larger amount of attacks and MTD techniques in the current RL-based MTD system and assess different learning scenarios. In particular, it would be beneficial to analyze how long it takes a pretrained agent to learn new, unseen attacks for different ways of pretraining and behaviors available. Next, a more elaborate feature engineering approach could be pursued to select the features considered for state samples. Thereby, a special focus should pe put on the improvement of anomaly detection. Further, in order to fully exploit MTD as a paradigm and apply it more holistically, it could be a great improvement to the current reactive RL-based system, to explore the integration with proactive defense and their joint optimization.

A further highly promising direction of future work encompasses the integration of more fine-grained MTD decisions in the current RL-based MTD selection framework. As in this work the primary goal is to decide upon WHAT to move by selecting among multiple MTD

techniques, the RL-based defense could be extended to also include decisions regarding WHEN and HOW to move. At the core this involves creating a more complex reward function that takes not only the fact of correct mitigation of attacks into account, but also HOW efficient the defense system operates. For instance, further MTD techniques could be implemented and be made parameterizable, and the agent's goal is extended to select an optimal combination of parameters that thwarts off an attack in the most economic manner possible. Essentially, this includes analyzing not only decision- and afterstates, but also the moving phase during the execution of MTD techniques. Finally, one possible approach for learning decisions on WHAT, WHEN and HOW to move could be to leverage hierarchical RL for multiple stages of learning granularity.

Bibliography

[sti] STIG security technical implementation guide. `https://www.stigviewer.com/stigs`. Accessed: 2022-06-20.

[2] Al-rimy, B. A. S., Maarof, M. A., and Shaid, S. Z. M. (2018). Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions. *Computers & Security*, 74:144–166.

[3] Al-Shaer, E., Duan, Q., and Jafarian, J. H. (2012). Random host mutation for moving target defense. In *International Conference on Security and Privacy in Communication Systems*, pages 310–327. Springer.

[4] Antonatos, S., Akritidis, P., Markatos, E. P., and Anagnostakis, K. G. (2007). Defending against hitlist worms using network address space randomization. *Computer Networks*, 51(12):3471–3490.

[5] Azab, M., Hassan, R., and Eltoweissy, M. (2011). Chameleonsoft: a moving target defense system. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 241–250. IEEE.

[6] Ben-Asher, N., Morris-King, J., Thompson, B., and Glodek, W. J. (2016). Attacker skill defender strategies and the effectiveness of migration-based moving target defense in cyber systems. In *Proc. 11th Int. Conf. Cyber Warfare Security (ICCWS)*, page 21.

[7] Brown, N., Sandholm, T., and Machine, S. (2017). Libratus: The superhuman ai for no-limit poker. In *IJCAI*, pages 5226–5228.

[8] Bunten, A. (2004). Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*.

[9] Cai, G.-l., Wang, B.-s., Hu, W., and Wang, T.-z. (2016). Moving target defense: state of the art and characteristics. *Frontiers of Information Technology & Electronic Engineering*, 17(11):1122–1153.

[10] Calvo, R. (2020). es-sensor. `https://github.com/electrosense/es-sensor`. [Online; accessed 19-July-2022].

[11] Carroll, T. E., Crouse, M., Fulp, E. W., and Berenhaut, K. S. (2014). Analysis of network address shuffling as a moving target defense. In *2014 IEEE international conference on communications (ICC)*, pages 701–706. IEEE.

[12] Cedeño, J. (2022). Mitigating cyberattacks affecting resource-constrained devices through moving target defense (mtd) mechanisms.

[13] Chai, X., Wang, Y., Yan, C., Zhao, Y., Chen, W., and Wang, X. (2020). Dq-motag: deep reinforcement learning-based moving target defense against ddos attacks. In *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*, pages 375–379. IEEE.

[14] Cho, J.-H., Sharma, D. P., Alavizadeh, H., Yoon, S., Ben-Asher, N., Moore, T. J., Kim, D. S., Lim, H., and Nelson, F. F. (2020). Toward proactive, adaptive defense: A survey on moving target defense. *IEEE Communications Surveys & Tutorials*, 22(1):709–745.

[15] Colbaugh, R. and Glass, K. (2012). Predictability-oriented defense against adaptive adversaries. In *2012 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2721–2727. IEEE.

[16] CortexVacua (2022). MTDFramework. `https://github.com/CortexVacua/MTDFramework`. [Online; accessed 11-Sept-2022].

[17] Dass, S. and Siami Namin, A. (2021). Reinforcement learning for generating secure configurations. *Electronics*, 10(19):2392.

[18] de Melo, A. C. (2009). Performance counters on linux. In *Linux Plumbers Conference*, volume 118.

[19] Ding, J., Nemati, M., Ranaweera, C., and Choi, J. (2020). Iot connectivity technologies and applications: A survey. *arXiv preprint arXiv:2002.12646*.

[20] Eghtesad, T., Vorobeychik, Y., and Laszka, A. (2020). Adversarial deep reinforcement learning based adaptive moving target defense. In *International Conference on Decision and Game Theory for Security*, pages 58–79. Springer.

[21] Error996 (2017). bdvl. `https://github.com/Error996/bdvl`. [Online; accessed 11-Sept-2022].

[22] Fatayer, T. S., Khattab, S., and Omara, F. A. (2011). Overcovert: Using stack-overflow software vulnerability to create a covert channel. In *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, pages 1–5. IEEE.

[23] Gardiner, J., Cova, M., and Nagaraja, S. (2014). Command & control: Understanding, denying and detecting-a review of malware c2 techniques, detection and defences. *arXiv preprint arXiv:1408.1136*.

[24] Gonzalez, D. and Hayajneh, T. (2017). Detection and prevention of crypto-ransomware. In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 472–478. IEEE.

[25] Gregg, B. (2016). perf Examples. `https://www.brendangregg.com/perf.html`. [Online; accessed 10-July-2021].

[26] Hong, J. B. and Kim, D. S. (2015). Assessing the effectiveness of moving target defenses using security models. *IEEE Transactions on Dependable and Secure Computing*, 13(2):163–177.

[27] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., and Franz, M. (2011). Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer.

[28] Jafarian, J. H., Al-Shaer, E., and Duan, Q. (2012). Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132.

[29] jakoritarleite (2020). backdoor jakoritar. `https://github.com/jakoritarleite/backdoor`. [Online; accessed 25-Sept-2022].

[30] jimmy ly00 (2021). Ransomware-PoC. `https://github.com/jimmy-ly00/Ransomware-PoC/tree/a2949e775d3fbd765b5c62eb30cdc66aeb775863`. [Online; accessed 11-Sept-2022].

[31] Jordan, J. (2018). Introduction to autoencoders. `https://www.jeremyjordan.me/autoencoders/`. [Online; accessed 10-July-2021].

[32] Jovanović, ■. D. and Vuletić, P. V. (2019). Analysis and characterization of iot malware command and control communication. In *2019 27th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE.

[33] Larsen, P., Brunthaler, S., and Franz, M. (2015). Automatic software diversity. *IEEE Security & Privacy*, 13(2):30–37.

[34] LEHMANN, J. (2018). Z-score Basics, Math Support Center blog, University of Baltimore. `https://ubalt.pressbooks.pub/mathstatsguides/chapter/z-score-basics/`. [Online; accessed 10-July-2021].

[35] Lei, C., Zhang, H.-Q., Tan, J.-L., Zhang, Y.-C., and Liu, X.-H. (2018). Moving target defense techniques: A survey. *Security and Communication Networks*, 2018.

[36] Li, Y., Dai, R., and Zhang, J. (2014). Morphing communications of cyber-physical systems towards moving-target defense. In *2014 IEEE International Conference on Communications (ICC)*, pages 592–598. IEEE.

[37] Liu, J., Shen, H., Narman, H. S., Chung, W., and Lin, Z. (2018). A survey of mobile crowdsensing techniques: A critical component for the internet of things. *ACM Transactions on Cyber-Physical Systems*, 2(3):1–26.

[38] Luo, Y.-B., Wang, B.-S., and Cai, G.-L. (2014). Effectiveness of port hopping as a moving target defense. In *2014 7th International Conference on Security Technology*, pages 7–10. IEEE.

[39] Manual, L. (2021). ld.so - dynamic linker/loader. `https://man7.org/linux/man-pages/man8/ld.so.8.html`. [Online; accessed 15-June-2022].

[40] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[41] Navas, R. E., Cuppens, F., Cuppens, N. B., Toutain, L., and Papadopoulos, G. Z. (2020). Mtd, where art thou? a systematic review of moving target defense techniques for iot. *IEEE internet of things journal*, 8(10):7818–7832.

[42] ncc group (2020). the tick. `https://github.com/nccgroup/thetick`. [Online; accessed 25-Sept-2022].

[43] Network, T. E. (2022). ElectroSense: Collaborative Spectrum Monitoring. `https://electrosense.org/#!/`. [Online; accessed 11-Sept-2022].

[44] Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX*, pages 363–372. Springer.

[45] NITRD (2009). Nitrd csia iwg cybersecurity game-change research and development recommendations.

[46] Owoh, N. P. and Singh, M. M. (2020). Security analysis of mobile crowd sensing applications. *Applied Computing and Informatics*.

[47] Peng, W., Li, F., Huang, C.-T., and Zou, X. (2014). A moving-target defense strategy for cloud-based services with heterogeneous and dynamic attack surfaces. In *2014 IEEE International Conference on Communications (ICC)*, pages 804–809. IEEE.

[48] perf wiki (2020). perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`. [Online; accessed 10-July-2021].

[49] Pilloni, V. (2018). How data will transform industrial processes: Crowdsensing, crowdsourcing and big data as pillars of industry 4.0. *Future Internet*, 10(3):24.

[50] Prakash, A. and Wellman, M. P. (2015). Empirical game-theoretic analysis for moving target defense. In *Proceedings of the second ACM workshop on moving target defense*, pages 57–65.

[51] Rajendran, S., Calvo-Palomino, R., Fuchs, M., Van den Bergh, B., Cordobés, H., Giustiniano, D., Pollin, S., and Lenders, V. (2017). Electrosense: Open and big spectrum data. *IEEE Communications Magazine*, 56(1):210–217.

[52] Rajput, P. H. N., Sarkar, E., Tychalas, D., and Maniatakos, M. (2021). Remote non-intrusive malware detection for plcs based on chain of trust rooted in hardware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 369–384. IEEE.

[53] Riad, K., Huang, T., and Ke, L. (2020). A dynamic and hierarchical access control for iot in multi-authority cloud storage. *Journal of Network and Computer Applications*, 160:102633.

[54] Rowe, J., Levitt, K. N., Demir, T., and Erbacher, R. (2012). Artificial diversity as maneuvers in a control theoretic moving target defense. In *National Symposium on Moving Target Research*.

[55] Sánchez, P. M. S., Celdrán, A. H., Schenk, T., Iten, A. L. B., Bovet, G., Pérez, G. M., and Stiller, B. (2022). Studying the robustness of anti-adversarial federated learning models detecting cyberattacks in iot spectrum sensors. *arXiv preprint arXiv:2202.00137*.

[56] Schenk, T. (2022). Reinforcement Learning based Moving Target Defense. `https://github.com/Leitou/rl-based-mtd`. [Online; accessed 3-Oct-2022].

[57] Sengupta, S. and Kambhampati, S. (2020). Multi-agent reinforcement learning in bayesian stackelberg markov games for adaptive moving target defense. *arXiv preprint arXiv:2007.10457*.

[58] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

[59] Soussi, W., Christopoulou, M., Xilouris, G., and Gür, G. (2021). Moving target defense as a proactive defense element for beyond 5g. *IEEE Communications Standards Magazine*, 5(3):72–79.

[60] Sovarel, A. N., Evans, D., and Paul, N. (2005). Where's the feeb? the effectiveness of instruction set randomization. In *USENIX Security Symposium*, volume 10.

[61] Stellios, I., Kotzanikolaou, P., Psarakis, M., Alcaraz, C., and Lopez, J. (2018). A survey of iot-enabled cyberattacks: Assessing attack paths to critical infrastructures and services. *IEEE Communications Surveys & Tutorials*, 20(4):3453–3495.

[62] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[63] Taguinod, M., Doupé, A., Zhao, Z., and Ahn, G.-J. (2015). Toward a moving target defense for web applications. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 510–517. IEEE.

[64] Thompson, M., Evans, N., and Kisekka, V. (2014). Multiple os rotational environment an implemented moving target defense. In *2014 7th International Symposium on Resilient Control Systems (ISRCS)*, pages 1–6. IEEE.

[65] Trendmicro (2022). Umbreon Linux Rootkit Hits x86, ARM Systems. `https://www.trendmicro.com/zh_hk/research/16/i/pokemon-themed-umbreon-linux-rootkit-hits-x86-arm-systems.html`. [Online; accessed 15-June-2022].

[66] Vijayan, V. (2020). Deep Reinforcement Learning. `https://medium.com/@vishnuvijayanpv/deep-reinforcement-learning-value-functions-dqn-actor-\critic-method-backpropagation-through-83a277d8c38d`. [Online; accessed 10-July-2022].

[67] Vormayr, G., Zseby, T., and Fabini, J. (2017). Botnet communication patterns. *IEEE Communications Surveys & Tutorials*, 19(4):2768–2796.

[68] Ward, B. C., Gomez, S. R., Skowyra, R., Bigelow, D., Martin, J., Landry, J., and Okhravi, H. (2018). Survey of cyber moving targets second edition. Technical report, MIT Lincoln Laboratory Lexington United States.

[69] Yoon, S., Cho, J.-H., Kim, D. S., Moore, T. J., Free-Nelson, F., and Lim, H. (2021). Desolater: Deep reinforcement learning-based resource allocation and moving target defense deployment framework. *IEEE Access*, 9:70700–70714.

[70] Zahra, A. and Shah, M. A. (2017). Iot based ransomware growth rate evaluation and detection using command and control blacklisting. In *2017 23rd international conference on automation and computing (icac)*, pages 1–6. IEEE.

[71] Zapata, M. (2022). beurk. `https://github.com/unix-thrust/beurk`. [Online; accessed 11-Sept-2022].

[72] Zhuang, R., Zhang, S., DeLoach, S. A., Ou, X., Singhal, A., et al. (2012). Simulation-based approaches to studying effectiveness of moving-target network defense. In *National symposium on moving target research*, volume 246.

Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CnC	Command and Control
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CSV	Comma-Separated Values
DL	Deep Learning
FN	False Negatives
FP	False Positives
FSM	Finite State Machine
GB	Gigabyte
GELU	Gaussion Error Linear Units
HPC	Hardware Performance Counters
IID	Independent and Identically Distributed
IoT	Internet of Things
LAN	Local Area Network
MB	Megabyte
ML	Machine Learning
MSE	Mean Squared Error
MTD	Moving Target Defense
RAM	Random Access Memory
ReLU	Rectified Linear Units
GeLu	Gaussian Error Linear Units
RF	Radio Frequency
SGD	Stochastic Gradient Descent
SSH	Secure Shell
TN	True Negatives
TNR	True Negative Rate
TP	True Positives
TPR	True Positive Rate

List of Figures

List of Tables

Appendix A

Environment Data

A.1 perf

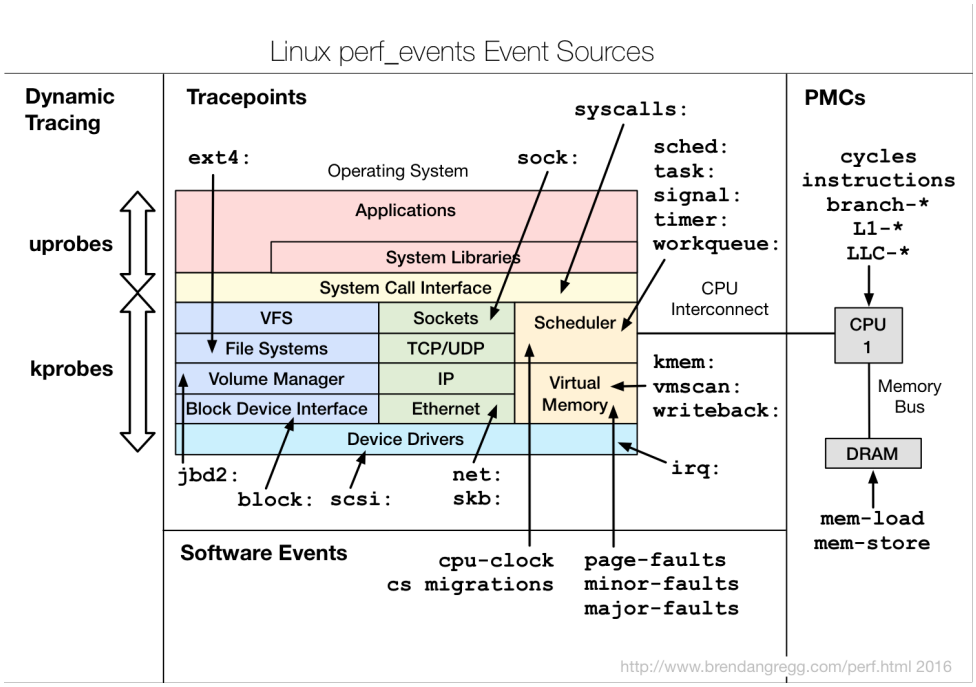

Figure A.1: Linux perf Event Sources [25]

A.2 Environment Features

A.2.1 Perf Events

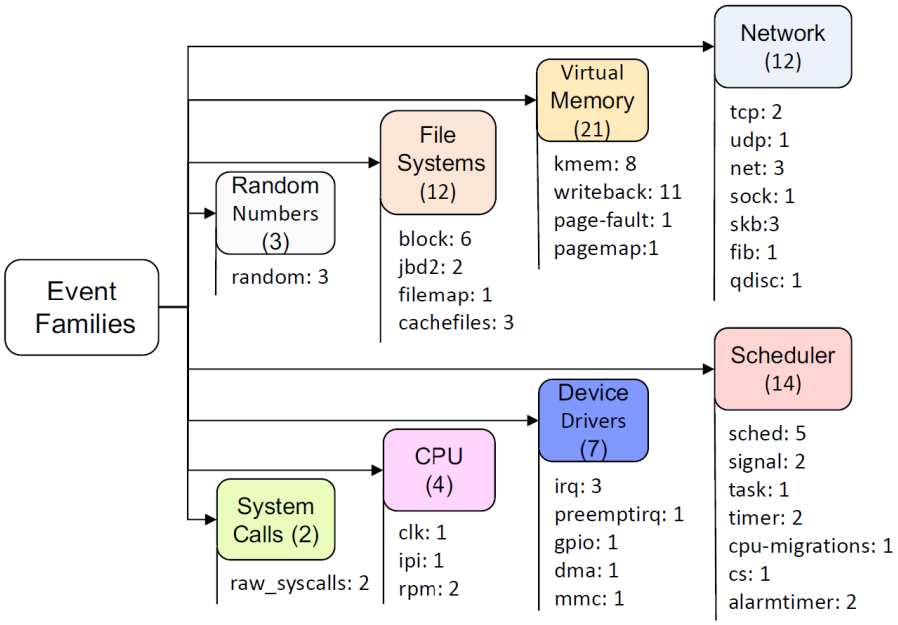

Figure A.2: Considered Event families [55]

A.2.2 List of Features

CSV column names which are too long are abbreviated and .. is appended. In this case, the full column name is composed as the concatenation of the Event Source, a colon and the Event.

Table A.1: Monitored Features and Basic Properties: constant, time-status related, excluded or used as feature for the models.

CSV Column	Event Source	Event	Constant	Status	Excluded	Feature
time				x		
timestamp				x		
seconds				x		
connectivity				x		
cpuUser		cpuUser				x
cpuSystem		cpuSystem			x	
cpuNice		cpuNice	x			
cpuIdle		cpuIdle			x	
cpuIowait		cpuIowait			x	
cpuHardIrq		cpuHardIrq	x			
cpuSoftIrq		cpuSoftIrq			x	
tasks		tasks			x	
tasksRunning		tasksRunning			x	
tasksSleeping		tasksSleeping			x	
tasksStopped		tasksStopped	x			
tasksZombie		tasksZombie			x	
ramFree		ramFree			x	
ramUsed		ramUsed			x	
ramCache		ramCache			x	
memAvail		memAvail			x	
iface0RX		iface0RX			x	
iface0TX		iface0TX			x	
iface1RX		iface1RX			x	
iface1TX		iface1TX			x	
numEncrypted		numEncrypted			x	
alarmtimer:al..	alarmtimer	alarmtimer_fired	x			
alarmtimer:al..	alarmtimer	alarmtimer_start	x			
block:bl..	block	block_bio_backmerge			x	
block:bl..	block	block_bio_remap				x
block:bl..	block	block_dirty_buffer			x	
block:bl..	block	block_getrq				x
block:bl..	block	block_touch_buffer			x	
block:bl..	block	block_unplug				x
cachefiles:ca..	cachefiles	cachefiles_create	x			
cachefiles:ca..	cachefiles	cachefiles_lookup	x			
cachefiles:ca..	cachefiles	cachefiles_mark_active	x			
clk:cl..	clk	clk_set_rate			x	
cpu-migrations		cpu-migrations			x	
cs		cs			x	
dma_fence:dm..	dma_fence	dma_fence_init	x			
fib:fi..	fib	fib_table_lookup				x
filemap:mm..	filemap	mm_filemap_add_to_page_cache				x
gpio:gp..	gpio	gpio_value				x
ipi:ip..	ipi	ipi_raise				x
irq:ir..	irq	irq_handler_entry			x	

Table A.2: Monitored Features and Basic Properties: continued

CSV Column	Event Source	Event	Constant	Status	Excluded	Feature
irq:so..	irq	softirq_entry			x	
jbd2:jb..	jbd2	jbd2_handle_start				x
jbd2:jb..	jbd2	jbd2_start_commit			x	
kmem:kf..	kmem	kfree				x
kmem:km..	kmem	kmalloc				x
kmem:km..	kmem	kmem_cache_alloc			x	
kmem:km..	kmem	kmem_cache_free			x	
kmem:mm..	kmem	mm_page_alloc			x	
kmem:mm..	kmem	mm_page_alloc_zone_locked				x
kmem:mm..	kmem	mm_page_free			x	
kmem:mm..	kmem	mm_page_pcpu_drain				x
mmc:mm..	mmc	mmc_request_start				x
net:ne..	net	net_dev_queue				x
net:ne..	net	net_dev_xmit				x
net:ne..	net	netif_rx				x
page-faults		page-faults				x
pagemap:mm..	pagemap	mm_lru_insertion				x
preemptirq:ir..	preemptirq	irq_enable			x	
qdisc:qd..	qdisc	qdisc_dequeue				x
random:ge..	random	get_random_bytes				x
random:mi..	random	mix_pool_bytes_nolock				x
random:ur..	random	urandom_read			x	
raw_syscalls:sy..	raw_syscalls	sys_enter			x	
raw_syscalls:sy..	raw_syscalls	sys_exit			x	
rpm:rp..	rpm	rpm_resume				x
rpm:rp..	rpm	rpm_suspend				x
sched:sc..	sched	sched_process_exec				x
sched:sc..	sched	sched_process_free				x
sched:sc..	sched	sched_process_wait				x
sched:sc..	sched	sched_switch			x	
sched:sc..	sched	sched_wakeup			x	
signal:si..	signal	signal_deliver				x
signal:si..	signal	signal_generate				x
skb:co..	skb	consume_skb			x	
skb:kf..	skb	kfree_skb				x
skb:sk..	skb	skb_copy_datagram_iovec				x
sock:in..	sock	inet_sock_set_state			x	
task:ta..	task	task_newtask				x
tcp:tc..	tcp	tcp_destroy_sock				x
tcp:tc..	tcp	tcp_probe				x
timer:hr..	timer	hrtimer_start			x	
timer:ti..	timer	timer_start				x
udp:ud..	udp	udp_fail_queue_rcv_skb	x			
workqueue:wo..	workqueue	workqueue_activate_work				x
writeback:gl..	writeback	global_dirty_state			x	
writeback:sb..	writeback	sb_clear_inode_writeback				x
writeback:wb..	writeback	wbc_writepage				x
writeback:wr..	writeback	writeback_dirty_inode				x
writeback:wr..	writeback	writeback_dirty_inode_enqueue				x
writeback:wr..	writeback	writeback_dirty_page				x
writeback:wr..	writeback	writeback_mark_inode_dirty				x
writeback:wr..	writeback	writeback_pages_written				x
writeback:wr..	writeback	writeback_single_inode				x
writeback:wr..	writeback	writeback_write_inode				x
writeback:wr..	writeback	writeback_written				x

A.3 Raw Behavior Dataset Size

Table A.3: Number of Raw Samples per Behavior for Raspberry Pi 3 Model B+, 1GB RAM, Total size of 59004 samples

Behavior	Count
normal	14702
bdvl	5698
beurk	7358
backdoor_jakoritar	4312
the_tick	7704
data_leak_1	5687
data_leak_2	4162
ransomware_poc	9381

Table A.4: Number of Raw Samples per Behavior for Raspberry Pi 4 Model B, 2GB RAM, Total size of 9414 samples.

Behavior	Count
normal	5092
bdvl	853
beurk	636
backdoor_jakoritar	1002
the_tick	739
ransomware_poc	1092

A.4 Data Exploration

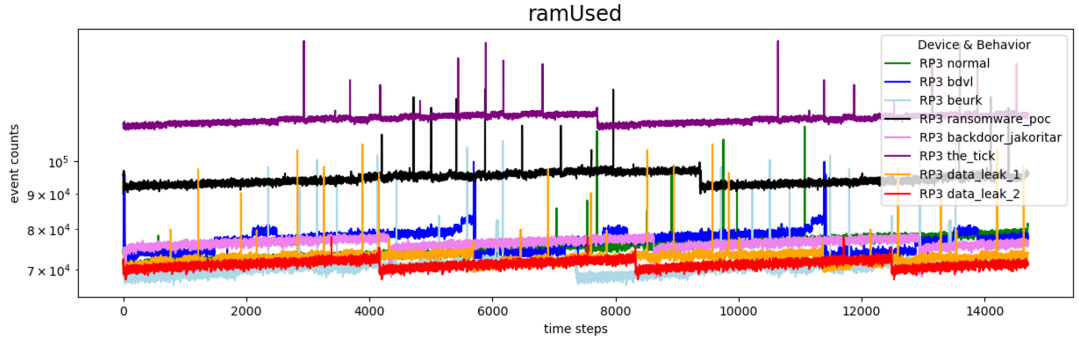

Figure A.3: Timeline for different malware families for the ramUsed feature monitored via the `top` command. Excluded in training due to cyclic patterns, intransparent temporal dependencies.

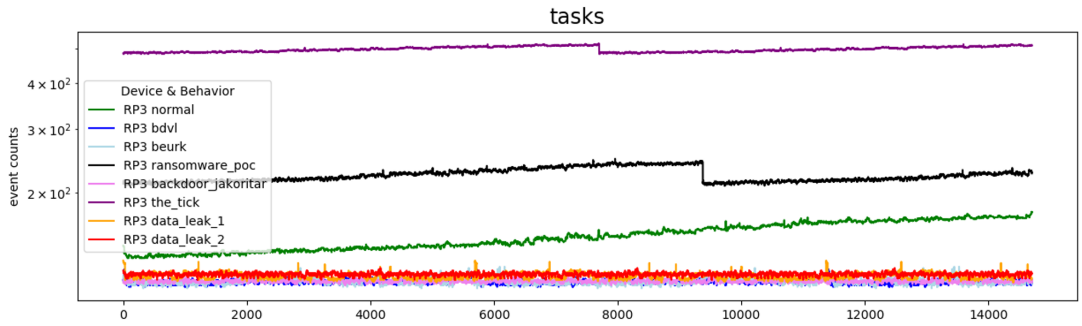

Figure A.4: Timeline for different malware families for the tasks feature monitored via the `top` command. Excluded in training due to cyclic patterns, intransparent temporal dependencies.

Appendix B

Anomaly Detection

B.1 Prototype 2 and 3: Unsupervised State Interpretation

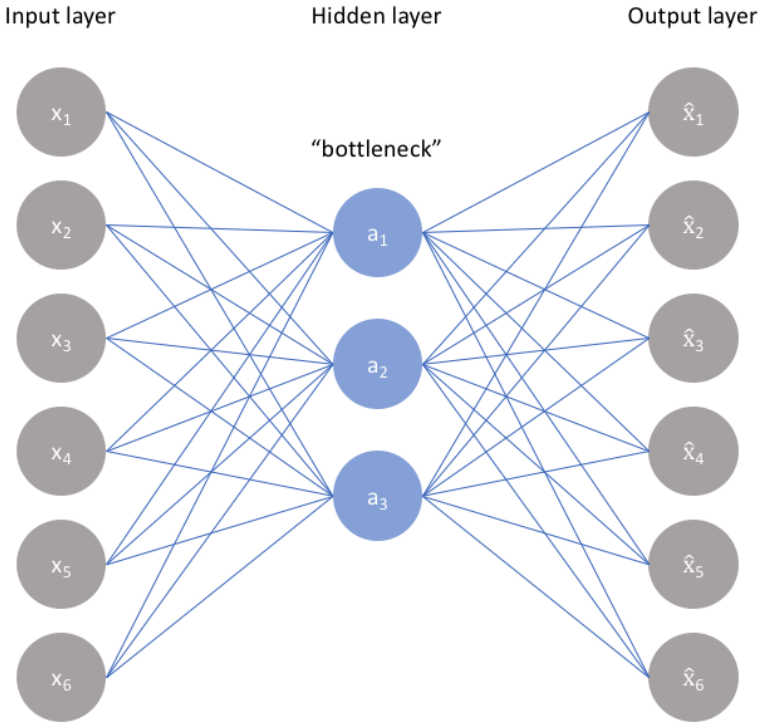

Figure B.1: Autoencoder Architecture [31]

Appendix C

Refined Simulation Data

C.1 Decision- and Afterstate Dataset Size

Table C.1: Number of Decision State Samples per Behavior for RP 3 Model B+ 1GB RAM, Total size of 17332 samples.

Behavior	State	Count
normal	decision	4178
bdvl	decision	1658
beurk	decision	2012
backdoor_jakoritar	decision	2018
the_tick	decision	1507
data_leak_1	decision	2080
data_leak_2	decision	2075
ransomware_poc	decision	1804

Table C.2: Number of Afterstate Samples per Behavior-MTD technique combination for RP 3 Model B+ 1GB RAM, Total size of 60549 samples.

Behavior	State	Count
normal	after cnc_ip_shuffle	2031
normal	after ransomware_directory_trap	2084
normal	after ransomware_file_extension_hide	1971
normal	after rootkit_sanitizer	1971
bdvl	after cnc_ip_shuffle	657
bdvl	after ransomware_directory_trap	1392
bdvl	after ransomware_file_extension_hide	624
bdvl	after rootkit_sanitizer	1995
beurk	after cnc_ip_shuffle	1975
beurk	after ransomware_directory_trap	1969
beurk	after ransomware_file_extension_hide	1990
beurk	after rootkit_sanitizer	2081
backdoor_jakoritar	after cnc_ip_shuffle	2024
backdoor_jakoritar	after ransomware_directory_trap	2017
backdoor_jakoritar	after ransomware_file_extension_hide	2013
backdoor_jakoritar	after rootkit_sanitizer	2085
the_tick	after cnc_ip_shuffle	2086
the_tick	after ransomware_directory_trap	2095
the_tick	after ransomware_file_extension_hide	2087
the_tick	after rootkit_sanitizer	2093
data_leak_1	after cnc_ip_shuffle	2079
data_leak_1	after ransomware_directory_trap	2091
data_leak_1	after ransomware_file_extension_hide	2085
data_leak_1	after rootkit_sanitizer	2081
data_leak_2	after cnc_ip_shuffle	2089
data_leak_2	after ransomware_directory_trap	2072
data_leak_2	after ransomware_file_extension_hide	2095
data_leak_2	after rootkit_sanitizer	2082
ransomware_poc	after cnc_ip_shuffle	636
ransomware_poc	after ransomware_directory_trap	2095
ransomware_poc	after ransomware_file_extension_hide	2092
ransomware_poc	after rootkit_sanitizer	1812

C.2 Data Exploration

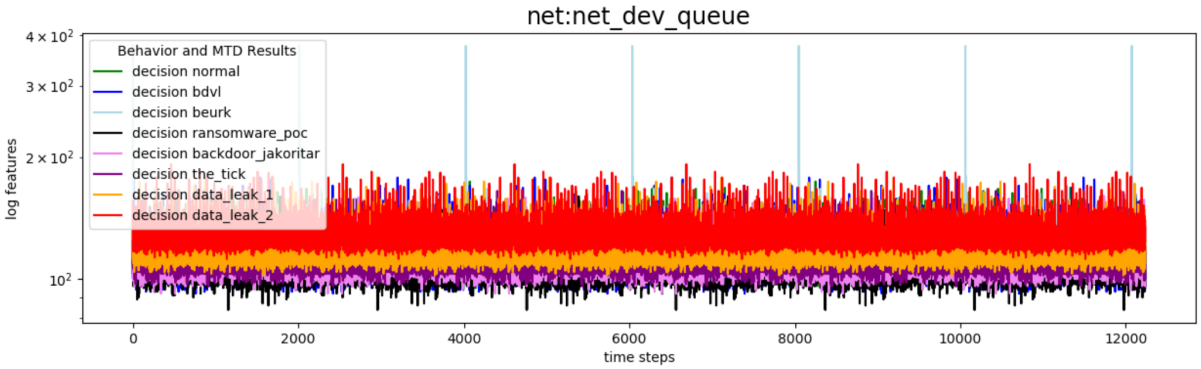

Figure C.1: Comparison of different behaviors at decision state for the net_dev_queue event. The of event counts vary across behaviors but remain stable within a certain range, as also shown in Section 4.3 for raw behaviors

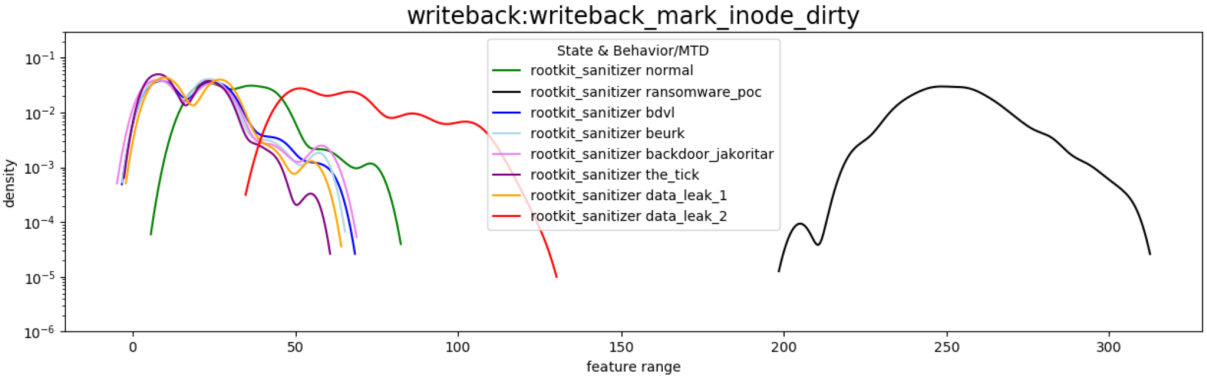

Figure C.2: Comparison of different behaviors after deploying the rootkit sanitizer MTD for the writeback_mark_inode_dirty event. Ransomware and the data leak 2 attack are most distinct, independent of any MTD deployment.