BACHELORARBEIT

Self-Propagating Malware Containment via Reinforcement Learning

ausgeführt am

Institut für
Analysis und Scientific Computing
TU Wien

unter der Anleitung von

Assoc. Prof. Dipl.-Ing. Dr.techn. Clemens Heitzinger

durch

Sebastian Eresheim
Matrikelnummer: 1608049

Wien, am 27. Mai 2021

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 27. Mai 2021

Sebastian Eresheim

Contents

1 Introduction

In 2017 two computer viruses – *WannaCry*[1] and *NotPetya*[2] – emerged and together are responsible for an estimated damage of about 14 billion dollars worldwide. These viruses are typical examples of ransomware, where a virus encrypts valuable information of the victim and demands a ransom in exchange for the decryption key. One reason why these two viruses were particularly dangerous was their method of dissemination. Besides leveraging stolen credentials from the victim host in order to spread via file sharing, both viruses also exploited a vulnerability in an older Windows implementation of the file sharing protocol SMB. This way an infected computer could write and execute new files on any computer on the local network, as long as the new victim was accepting the vulnerable file sharing protocol version. Unfortunately, back in 2017 these were surprisingly many, given that the vulnerability was already publicly known.

Before the malware presents the user its ransom note, it already tried to infect other computers on the local network. If additional infections on the network are successful, then each new infection itself tries to infect other hosts on the network, before the ransom note is shown. Thus by the time the first ransom note appears on any host in the network, many more could actually be affected at that time. Therefore the sooner an infected host can be detected and disconnected from the rest of the network, the more damage could be avoided. A counteracting entity therefore needs to act fast to prevent future spreads, but also needs to look ahead if an additional infection has already happened. For human security experts such scenarios are a tough challenge and often times can only be resolved by inflicting collateral damage to the rest of the operative network.

In recent years the field of reinforcement learning (RL) has gained momentum. The combination of traditional reinforcement learning methods and deep learning techniques, managed to outperform even the best humans in certain areas. The first milestone was reached in 2013 when a single RL agent managed to play a large collection of Atari 2600 games, most of them on a human level, some even on a super-human level [MKS+13]. Only a few years later, in 2016, a RL agent – called AlphaGo – managed to beat Lee Sedol in the game of Go [SHM+16], who was one of the best Go players at that time. The game of Go was believed to be a long lasting challenge for an artificial intelligence, because of its large state space. Therefore it was even more surprising, when only shortly afterwards AlphaZero managed to not only learn Go, but also the games of Chess and Shogi and even defeated AlphaGo by a large margin [SAH+20]. After Go, one of the most difficult board games to master, was "solved", the community turned towards complex

[1] https://www.heise.de/newsticker/meldung/WannaCry-Was-wir-bisher-ueber-die-Ransomware-Attacke-wissen-3713502.html

[2] https://www.heise.de/security/meldung/Alles-was-wir-bisher-ueber-den-Petya-NotPetya-Ausbruch-wissen-3757607.html

computer games [VEB+17]. In 2019 a team from OpenAI created an agent based on deep RL for the computer game Dota 2 [BBC+19] which was capable of defeating one of the world's best human teams. Meanwhile a team from Google DeepMind trained a deep RL agent in the game of Starcraft 2 [VBC+19] which also won against high rank human players. These agents also showed that reinforcement learning agents are capable of finding (nearly) optimal policies in environments which are

- high-dimensional in states,
- high-dimensional in actions,
- (quasi) time-continuous,
- scarce of reward and
- only partially observable.

1.1 Research Question

Due to the ability of RL agents making quick decisions in difficult situations which may have long-term impacts, RL is a potential technology to prevent self-propagating malware from spreading in local networks. Compared to supervised learning, a reinforcement learning approach could have three key advantages:

1. RL is able to counteract not just in a timely manner, but also individually in different situations. Supervised learning approaches usually detect the occurrence of malware and then notify a higher instance. Additionally a supervised-learning-based system can react in a predefined manner (e.g. by putting questionable programs in a quarantine), but this heavily relies on domain knowledge from an expert and usually suffers from a small degree of individualization. RL on the other hand can react quickly and is able to find optimal reactions to different situations.

2. Given a rich action space, RL is be able to decrease uncertainty by interacting with the object in question before further actions are taken. An analogy for a supervised learning system would be an observer that is restricted to images of an object in order to determine what it is and how to proceed with it. However, RL enables the observer to interaction with the object and observe reactions. In the previous analogy, the observer could for example twist and turn the object to get different points of view, before making a final decision on the future interaction. In the specific context of self-propagating malware, an RL agent could interact with a potentially infected host and look out for anomalies in the encountered reaction of the host.

3. Another downside of supervised learning is the reliance on sufficient labeled data. Labeling a sufficient amount of data can cost experts a lot of time and is therefore expensive. RL on the other hand relies on an environment with which the agent can interact. Once the environment is built, the amount of data generated by agent interactions can be arbitrarily large. In the scenario of self-propagating malware

labeling all data packets that indicate an infection of a host is a very time consuming task. Having an environment that marks the time of a released infection and an agent that learns to distinguish malware-based data packets based on an according reward function is a more time efficient approach.

The research question that this thesis tries to answer is therefore: "Is it possible to train a computer agent via reinforcement learning to prevent self-propagating malware from spreading in a local network?". The resulting RL agent should be seen as a proof of concept prototype, which might not include all previously discussed advantages, and not as a fully functioning optimal solution. Yet it can be seen as a baseline for further improvements that include more advanced methods.

The rest of this thesis is structured as follows: chapter 2 introduces the methods which are used and referenced throughout this thesis in more detail, chapter 3 describes the setup for the experiments, chapter 4 shows the results of the experiments, chapter 5 discusses these results and chapter 6 concludes this thesis.

2 Preliminary

2.1 Reinforcement Learning

Reinforcement Learning [SB18] is a sub-field of *machine learning*. In contrast to the better known sub-fields *supervised learning* and *unsupervised learning* RL does not focus on data itself. Instead an agent – an autonomously acting system – and its environment are the focus of this field. These two entities stand in a cyclic interaction relationship to each other. Given discrete time-steps $t = 0, 1, 2, 3, \ldots$, the agent receives a state S_t from the environment and responds with an action A_t. The environment then answers with a numerical reward R_{t+1}, which determines how good the agent's chosen action was and a new state S_{t+1}, which marks the beginning of a new cycle. Multiple of such interactions lead to a trajectory $S_0, A_0, R_1, S_1, A_1, R_1, S_2, A_2, R_3, \ldots$ (by convention the transition from one time-step to the next one is at the transition from action to reward). The agent's goal is to choose actions such that the received total reward is maximized.

2.1.1 Markov Decision Process

In this context a Markov Decision Process (MDP) models the fundamental setting.

Definition 2.1.1 (Markov Decision Process)**.** A tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \chi, \gamma)$ is called a *Markov Decision Process*, if

- $\mathcal{S}$ is a state space,
- $\mathcal{A}$ is an action space,
- $\mathcal{R} \subset \mathbb{R}$ is a set of rewards,
- p is an inner dynamics function as

$$p\colon \quad \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0,1], \quad (s', r, s, a) \mapsto \mathbb{P}(S'_t = s', R_t = r | S_t = s, A_t = a),$$

- $\chi : \mathcal{S} \to [0,1]$ is the initial distribution of the states,
- $\gamma \in (0,1]$ is a discount factor and
- the process fulfills the *Markov property*

$$\begin{aligned} & \mathbb{P}(S'_t = s', R_t = r | S_t = s_l, A_t = a_l, \ldots, S_0 = s_0, A_0 = a_0) \\ = \quad & \mathbb{P}(S'_t = s', R_t = r | S_t = s_t, A_t = a_t). \end{aligned}$$

The MDP is called *deterministic* if $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to \{0,1\}$ and otherwise *stochastic*.

Throughout this thesis the uppercase variables S_t, A_t and R_t denote random variables for the discrete time-step t, that map into their respective spaces $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$. Lowercase variables on the other hand, like s, s', s_0, s_l, a or r are specific elements of the state- or action space or the reward set.

2.1.2 Value Methods

RL algorithms can be separated in two distinct groups – value-based methods and direct policy methods. This thesis only focuses on value-based methods. First, so called *tabular methods* are considered, where state- and action spaces are finite and reasonably small. Therefore it is possible to store a table that contains a value for either each state or each combination of state and action. These tables are referred to as *value functions*. The approaches introduced in this section for tabular methods are later extended to infinite state- or action spaces via *function approximation*, which is described in section 2.2.

Definition 2.1.2 (Policy)**.** Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \chi, \gamma)$ be an MDP and $\Delta(\mathcal{A})$ be the set of probability distributions over $\mathcal{A}$.
Then a function

$$\pi\colon \quad \mathcal{S} \to \Delta(\mathcal{A})$$

that assigns a probability distribution over the action space to a state is called a *policy*.

The purpose of the agent is to find an optimal policy π^* that maximizes the reward. In contrast to direct policy methods the agent does so by leveraging value functions, that determine how "good" a state/state-action-pair is.

Definition 2.1.3 (Value Functions)**.** Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \chi, \gamma)$ be an MDP, π be a policy, $G_t := \sum_{i=0}^{T} \gamma^i R_{t+i}$ the future discounted return at time-step t and $T \in \mathbb{N} \cup \{\infty\}$ be the terminal time-step. Then the function

$$v_\pi\colon \quad \mathcal{S} \to \mathbb{R}, \quad s \mapsto \mathbb{E}_\pi[G_t | S_t = s]$$

is called *state value function* and the function

$$q_\pi\colon \quad \mathcal{S} \times \mathcal{A} \to \mathbb{R}, \quad (s, a) \mapsto \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

is called *action value function*. The measure of goodness for a state/state-action-pair, is the expected cumulative discounted reward, given a state/state-action-pair for the current time-step and a policy to follow for future action selection. The value functions regarding the optimal policy π^* are called *optimal value functions* and denoted by $v^*(s)$ and $q^*(s, a)$.

Definition 2.1.4 ($(\epsilon$-)Greedy Policy)**.** Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \chi, \gamma)$ be an MDP, $q(s, a)$ be an action value function and $\epsilon > 0$. A policy of the structure

$$\pi(s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a q(s, a), \\ 0 & \text{else} \end{cases}$$

is called the *greedy policy with respect to q*. A policy of the structure

$$\pi(s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \text{argmax}_a\, q(s,a), \\ \frac{\epsilon}{|\mathcal{A}|} & \text{else} \end{cases}$$

is called a *ϵ-greedy policy with respect to q*.

If q is clear from the context, then these policies are simply called *(ϵ-)greedy policy*.

Generalized Policy Iteration

The process of *Policy Iteration* consists of two tasks. First, during *policy evaluation*, a policy π is used to interact with the environment in order to create an estimation $\hat{q} \approx q_\pi$ for the actual value function q_π. At this point, different algorithms can be used to calculate $\hat{q}$. Monte Carlo and temporal-difference methods are two examples of such algorithms. After policy evaluation is finished, *policy improvement* leverages the created value function $\hat{q}$ to form a new policy π', which is the greedy policy with respect to $\hat{q}$. This new policy π' is then the starting point for a new cycle of policy iteration. The longer this process of alternating policy evaluation and policy improvement continues, the closer $\hat{q}$ and π' get to q^* and π^*.

Generalized Policy Iteration includes variants of policy iteration, where policy evaluation and policy improvement are stopped prematurely. This is possible because it is not necessary for $\hat{q}$ to actually get close to q_π in one policy evaluation step, as long as it gets closer to q^*.

2.1.3 Monte Carlo Methods

Because the return of the current time-step $G_t := \sum_{i=t}^{T} \gamma^i R_i$ depends on future rewards, it is a natural approach to hold out until it is known. Monte Carlo methods therefore wait until an episode finished. Then it iterates an episode from back to start and calculates the return for every time-step. The returns are then used to update the estimates of the state- or action-value function of the encountered states during the episode. The approach builds on the law of large numbers, which ensures that the sample mean of the returns converges to the expected value of the return for every state and action. The Monte Carlo approach in more detail can be seen in algorithm 1.

2.1.4 Temporal-Difference Learning

Temporal-difference learning takes a different approach to learning the value function compared to Monte Carlo methods. Its basic principle is to use (already learned) estimators in the process of updating other estimators. This enables them to learn already within an episode instead of waiting until all rewards of the episode have manifested. This idea is based on $[R_{t+1} + \gamma v(S_{t+1})] \approx G_t$. An example for this approach can be seen in Equation 2.1, where $v(S_{t+1})$ – itself an estimator – is being used to update the estimator $v(S_t)$

$$v(S_t) \leftarrow v(S_t) + \alpha[R_{t+1} + \gamma v(S_{t+1}) - v(S_t)]. \tag{2.1}$$

Algorithm 1: Monte Carlo

Initialize:
$\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$ arbitrarily
$q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily
$Returns(s, a)$ empty list for all $s \in \mathcal{S}, a \in \mathcal{A}$

while *True* **do**
  Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(s)$ randomly such that all pairs have probability > 0
  Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, S_1, A_1, ..., A_{T-1}, R_T$
  $G \leftarrow 0$
  foreach *step of episode,* $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_t$
    Append G to $Returns(S_t, A_t)$
    $q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
    $\pi(S_t) \leftarrow \text{argmax}_a\, q(S_t, a)$

This is especially beneficial in environments with a long episode duration and in continuous RL, where the agents lifetime is not structured in episodes, but instead continuous indefinitely.

SARSA

SARSA combines the two approaches of general policy iteration and temporal-difference learning. It leverages an action value function Q, which is updated via

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

during policy evaluation. For the t-th time-step this update is performed at time-step $t+1$, meaning that the following state S_{t+1} and the following action A_{t+1} already need to be known when the update is calculated. The algorithm's name comes from the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. SARSA is a so called *on-policy* algorithm, where the policy by which the agent interacts with the environment is the same policy as which is updated. In other words during the policy evaluation step, Q_π is the action value function of the policy that is to gather the experience samples for updating. A more detailed description of SARSA can be seen in algorithm 2.

Q-Learning

A related algorithm to SARSA is Q-Learning [WD92]. Formally, Q-Learning looks very similar to SARSA, only a minor change in the update formula distinguishes them:

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

Unlike SARSA, Q-Learning is an *off-policy* algorithm. Here, the policy the agent follows through the environment is different to the policy that is being updated. This is the case,

Algorithm 2: SARSA

Parameters: step size $\alpha \in (0, 1]$, $\epsilon > 0$
Initialize $q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily except $q(\text{terminal}, \cdot) = 0$

foreach *episode* **do**
  Initialize S
  Choose A from S using policy derived from Q
  foreach *step of episode* **do**
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q
    $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  until S is terminal

because the $\max_a$ operation leads to updating Q directly in the direction of Q^*. This is due to $(R_{t+1} + \gamma \max_a q(S_{t+1}, a)) \approx G_t$ being an estimator of the return, if the optimal policy was followed. In SARSA the policy evaluation updates Q in the direction of Q_π, which is then used to move π in the direction of π^*. With Q-Learning, however, the agent tries to estimate the optimal action values from the beginning, regardless of the policy that is being followed. A description of the Q-Learning algorithm, is shown in algorithm 3.

Algorithm 3: Q-Learning

Parameters: step size $\alpha \in (0, 1]$, $\epsilon > 0$
Initialize $q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily except $q(\text{terminal}, \cdot) = 0$

foreach *episode* **do**
  Initialize S
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q
    Take action A, observe R, S'
    $q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)]$
    $S \leftarrow S'$
  until S is terminal

2.2 Function Approximation

Tabular methods, as described in the previous section, have one crucial disadvantage: the state and action spaces need to be reasonably small, such that a table for all states/state-action-pairs is computable in a reasonable time. For most real-world applications this requirement is not tractable. Function approximation (FA) is a common method to over-

come this limitation.

The most common form of function approximation is parameterized FA. Here a vector $\mathbf{w} \in \mathbb{R}^d$ is used to parameterize the value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$. It is important to note, that the dimension of the vector $\mathbf{w}$ is greatly smaller than the size of the state space ($d \ll |\mathcal{S}|$). For parameterized FA, many supervised learning algorithms can be applied. For example $\hat{v}(s, \mathbf{w})$ can be a decision tree, where each entry of the vector $\mathbf{w}$ is a decision boundary in the tree. $\hat{v}(s, \mathbf{w})$ can also be a neural network, where all the neuron weights and biases are stored in the vector $\mathbf{w}$. Since the dimension of the parameter vector $\mathbf{w}$ is much smaller than the state space not all states can be equally accurate estimated. Because the agent should at least be accurate in the training data, these FA methods need to adapt $\mathbf{w}$ in a way that more frequently appearing states are more accurately approximated than less frequent states. Semi-gradient TD is one example of a learning algorithm leveraging parameterized FA.

Another form of function approximation is memory-based FA. Unlike their parameterized counterpart, where experience is only used once during the training part of the supervised learning algorithm, memory-based function approximation keeps the encountered experience in memory. When an estimated value for a query state is needed, the information in memory is retrieved and used as basis for its value estimate. For example when an agent encounters the exact state frequently, then the estimate could be the average of the rewards it received previously, just like with tabular methods. Such approaches are often called *lazy learning*, because in contrast to *eager learning*, most of the computational calculation is done at query time and not at training time. This thesis focuses on *local-learning* algorithms, where not all previously encountered experience is used for the value estimation of a query state, but instead only experience that relies in some sort of neighborhood.

2.2.1 k-Nearest Neighbors

One such local-learning method is the *k-nearest neighbors* algorithm. This algorithm requires a metric space (M, d), where $M \subset \mathbb{R}^l$ is an l-dimensional space and $d : M \times M \to \mathbb{R}$ is a metric that determines the distance between two points of the space M. Then the k-nearest neighbors algorithm assigns the (weighted) average of the k closest points in memory to a query state. The parameter k is a hyper-parameter which needs to be determined upfront.

2.3 Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) was initially introduced to boost the speed in approximate nearest neighbor search [IM98]. The general idea is that for two input vectors x and y the probability of creating a hash collision increases as a similarity measure for the two input vectors increases. Thus the more similar x and y are, the more likely it is that $hash(x) = hash(y)$. In this thesis a hash collision of high probability should appear when $\|x - y\|_p$ is low and vice versa.

The LSH algorithm used in this thesis [DIIM04] is based on *p-stable distributions*.

Definition 2.3.1 (p-stable Distribution)**.** A distribution $\mathcal{D}$ over $\mathbb{R}$ is called *p-stable* if there

exists a $p \geq 0$ such that for andy i.i.d random variables $X_1, \ldots, X_n, X$ with distribution $\mathcal{D}$ and any numbers $v_1, \ldots, v_n$, the random variable $\sum_i v_i X_i$ has the same distribution as the random variable $(\sum_i |v_i|^p)^{1/p} X$.

An example for a p-stable distribution is a Gaussian distribution, which is 2-stable.

Let α be a random vector of a d dimensional space (for our purposes, d is high), where each entry is drawn independently from a p-stable distribution. Then, according to the definition of a p-stable distribution and given a d dimensional vector v, the dot product αv is distributed as $(\sum_i |v_i|^p)^{1/p} X$, where X is a random variable with p-stable distribution. For two d dimensional vectors v_1 and v_2 it follows that $\alpha v_1 - \alpha v_2 = \alpha(v_1 - v_2)$ is distributed as $\|v_1 - v_2\|_p X$, where again X is a random variable with p-stable distribution. Because creating the dot product with α projects each vector v to the real line, partitioning it into different buckets and assigning each vector v the value of that bucket after projection, results in a locality preserving hashing scheme. To be precise, the hash function is defined as

$$h_{\alpha,b}(\mathbf{v}): \quad \mathbb{R}^d \to \mathbb{N}, \quad \mathbf{v} \mapsto \left\lfloor \frac{\alpha \mathbf{v} + b}{r} \right\rfloor, \tag{2.4}$$

where α is, as before, a d dimensional vector where each component is independently drawn from a p-stable distribution, b is a real number chosen uniformly from the range $[0, r]$ and r is a predefined hyper parameter. Finally the probability of a collision for $c := \|v_1 - v_2\|_p$ can be calculated via

$$p(c) = Pr_{\alpha,b}[h_{\alpha,b}(\mathbf{v_1}) = h_{\alpha,b}(\mathbf{v_2})] = \int_0^r \frac{1}{c} f_p\left(\frac{t}{c}\right)\left(1 - \frac{t}{r}\right) dt, \tag{2.5}$$

where f_p denotes the probability density function of the absolute value of the p-stable distribution.

3 Methods

In order to let an RL agent learn to quarantine (potentially) infected hosts within a computer network, we decided to use an episodic approach. During one episode a host gets infected at a random point in time and the agent needs to learn to intervene only after an infection. Each episode is divided into equidistant time-steps where the agent can choose an action.

3.1 Environment

The theoretical foundation of the environment is a MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \chi, \gamma)$. The state space $\mathcal{S}$, the action space $\mathcal{A}$ and the reward function are explained in more detail in the following sub-sections. The inner dynamics function p is unknown to the agent. In fact it tries to learn this function indirectly via the reward signal. The initial distribution χ is deterministic, because the agent starts each episode from the same initial state. The discount factor is $\gamma = 0.99$, because the agent should maximize its long term goal.

3.1.1 State Space

The formal definition of an MDP requires the state to contain all the information the agent can base its decision on. In our setup, the information the agent needs is information about the network messages passing the interface the agent is listening on. Each network message that passes the network interface is encoded in a feature vector $\phi \in \mathbb{N}^d$ where $d-1$ dimensions are 0 and only one dimension contains a 1. This 1 stands for one data packet and the dimension this 1 is placed contains further information according to the feature selection. The state of the environment is then defined as the sum

$$s := \left(\sum_{i=0}^{N_t} \phi_i, \sum_{j=t-10}^{t} \sum_{i=0}^{N_j} \phi_i \right) \tag{3.1}$$

of all feature vectors, where N_j is the number of network packets in time-step j and t is the current time-step. The left component of the state is a short term depiction of the current situation, summing all feature vectors of the previous time-step. The right component adds more context as it additionally sums up all feature vectors of the previous 10 time-steps. Having the agent only look at the network traffic of the previous time-step enables it to react to short term events, but makes it oblivious to longer lasting trends. Therefore the right component was added. This definition results in $\mathcal{S} = \mathbb{N}^d \times \mathbb{N}^d$.

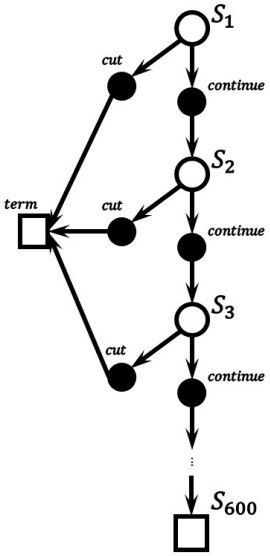


Figure 3.1: A backup diagram of the environment. White circles represent states, black circles represent actions and white squares represent terminal states.

3.1.2 Action Space

The action space only consists of two actions $\mathcal{A} = \{\text{continue}, \text{cut}\}$ in order to record and replay single episodes. The action *continue* lets the environment continue without interruption and the action *cut* disconnects the network connection to the vulnerable VM and ends the episode. Figure 3.1 depicts the backup diagram of the environment, which displays the connections between states and actions.

3.1.3 Reward Function

The purpose of the reward function is to teach the agent the desired behavior. The desired behavior in this thesis is on one hand to cut the network connection as soon as an infection has happened, and on the other hand to not intervene otherwise. In order to accomplish these goals, the reward function is defined as: let $I \in \mathbb{N}$ be the time-step an infection happens on the vulnerable VM and let $T \in \mathbb{N}$ be the final time-step. Also let $I \leq T$ that during an episode a host is infected and $T < I$ that there is no infection during an episode. Finally let $t \in \mathbb{N}$ be the current time-step. Then the reward function $R : \mathbb{N} \times \mathbb{N} \times \mathcal{A} \to \mathbb{R}$ is defined by

$$R(I, t, \text{continue}) := \begin{cases} 0 & \text{if } \ t < T, \\ 1 & \text{if } \ t = T < I, \\ -1 & \text{if } \ I < t = T, \end{cases}$$

$$R(I, t, \text{cut}) := \begin{cases} -1 & \text{if } \ t \leq I, \\ \frac{1}{t-I} & \text{if } \ I < t. \end{cases}$$

.

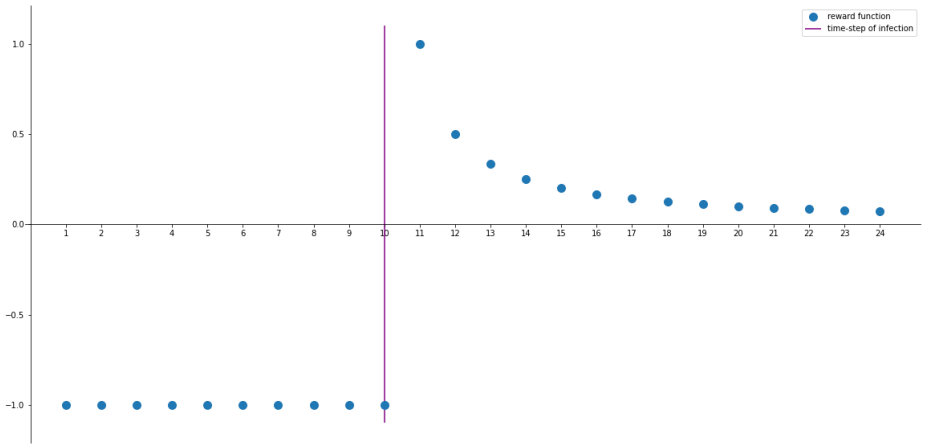


Figure 3.2: An example reward function for the action *cut*. In this scenario the infection of the host happens in time-step 10. If the action *cut* is applied before or at the same time-step than the infection, then a reward of -1 is given. If the action is applied after the infection, then the reward is $\frac{1}{t-10}$, where t is the current time-step.

The rapid decrease in reward the longer the infection dates back ensures that the agent prefers to cut the connection earlier than later. On the other hand the negative reward punishes the agent if it acts too early or not at all when it should have. Figure 3.2 depicts the second part of the reward function.

3.2 Feature Selection

This thesis introduces two separate feature selection methods. First, a rather simple feature selection method is applied, in order to verify that RL is actually applicable. This method only consist of a one-dimensional vector, returning $(1) \in \mathbb{R}^1$ if a data packet uses the *arp* protocol and (0) otherwise. This particular feature encoding was chosen, because the malware is known to use the *arp* protocol for host discovery. Therefore experiments with this feature encoding show whether learning generally works.

Second, a more sophisticated feature selection method is applied. Similar to the first method, only network meta data is considered for the feature encoding. In this more sophisticated method, data packets are distinguished by their type and by their destination. Because all of the relevant data is categorical the approach is to use a single dimension in the feature vector for each combination of categorical values. This results in a very high, yet mutual exclusive, number of dimensions.

The packet destination (within the LAN) is stored in its IP address. This address is split in a network and a host part, where in this specific setup 3 networks are distinguished and each network can contain 256 separate hosts. Thus there are 768 different addresses in this setup. Encoding all addresses, even though only 6 of them are used, is necessary, because the malware actively enumerates all IP addresses in the network and listens for

replies. Not encoding all possible addresses would lead to either not being able to model such packets at all or to not being able to distinguish between these addresses. Also all IPv6 packets are grouped together, regardless of their target, because although it is desired to distinguish between IPv6 from IPv4 traffic, IPv6 is not the main focus of this work (due to its enormous address space).

The type of data packet is defined by the network protocols *arp*, *icmp*, *tcp*, and *udp*. Additionally for *tcp* and *udp* the destination port further differentiates the type as long as it is within the threshold of the well-known ports at 1024. Above this threshold all ports are grouped in one dimension for all addresses and relevant protocols for convenience reasons. Also additional dimensions for a coarser distinction were added, counting all *ip*, *arp*, *tcp* and *udp* traffic.

Finally there is one dimension in the feature vector of the more sophisticated method for each combination of address and type previously mentioned, resulting in 3,154,440 mutual exclusive dimensions.

3.3 Agent

The agent uses generalized policy iteration as a general method to find an optimal policy π^*, as described in section 2.1. During interaction the agent uses an ϵ-greedy policy, where different values for ϵ are considered. For the simple feature extraction the agent directly computes all action values in a tabular approach, whereas in the sophisticated feature extraction case memory-based function approximation is applied. To be specific we apply a k-nearest neighbors function approximation in combination with locality sensitive hashing, as described in section 2.3, in order to decrease the query time of the k-NN. This means whenever the value of a state is queried, the average value of the k-nearest neighbors is calculated and used as approximation. The buffer size for the k-NN algorithm, comprised 100,000 states. In the simple feature extraction case, Monte Carlo methods, SARSA and Q-Learning are compared for learning algorithms. In the more sophisticated feature extraction case only SARSA and Q-Learning are applied.

4 Experiments

In our experiments, we apply an episodic RL approach, because a network where every host is infected marks a clear terminal state in the environment. In order to return from this terminal state to the initial state, where no infection has happened, we deployed *virtual machines* (VMs) connected via a virtual network. Also considering VMs makes it possible to use real software and real malware. Using real software and real malware is in some form a unique selling point, due to many environments in the RL field being abstract games [LLL+19, BNVB13], or simulations of real world scenarios [TET12, DRC+17, AR19]. Furthermore, VMs are used in corporations as well as physical computers. This is why they are more a subset of a real world scenario than a simulation of it. As already mentioned, VMs allow to create so called snapshots – an image of the virtual machine's state at a specific point in time – which the VMs can be reverted to and thus make it possible to fast and reliably undo the damage a malware infection causes.

The virtual network that connects the VMs, is shaped in a star topology, meaning there is a central VM, that all others are connected to. In our setup there are 4 VMs, the central one, called *agent VM*, and three VMs that can be infected, called *vulnerable VMs*. Figure 4.1 shows a schematic representation of this setup. The software component that represents the agent is located on the *agent VM*, which contains also some components of the environment, like feature extraction. Network traffic, that originates from vulnerable VMs is routed via the agent VM, where each connection to a vulnerable VM can be independently blocked via firewall settings.

Each episode is limited to 10 minutes of real time, which is divided into equal time-steps of 1 second. During an episode malware is released on one machine at a random time-step, which then tries to spread to all machines on the local network. At each time-step the agent can interact with the network. This results in a maximum of 600 actions per episode. In order to prevent the agent from interfering with benign hosts, baseline episodes are introduced, where no malware is released. In these episodes the agent should learn to not hinder the network. The ratio of episodes where no malware is released is $1 : 5$, meaning for each baseline episode there are 5 malware episodes.

In order to overcome a time consuming experiment setup, the environment is adapted to record episodes and replay them faster than real time during the learning process. The downside of this approach is a more restricted action space, since not every consequence of every action at each time-step can be recorded. Therefore the agent's observations are limited to a single network interface instead of all three interfaces. Additionally its action space is reduced to two distinct actions, one no-operation action called *continue* and one for blocking all network traffic called *cut*. By blocking all network traffic, the host is effectively put in a quarantine, where it resides until a human expert has had a further inspection. With these adaptations it is possible to record and replay episodes. In the case of the first action, the episode continues normally, without any agent interaction and in the case of

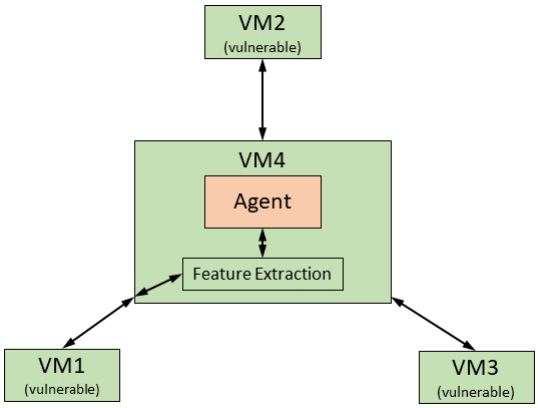


Figure 4.1: A schematic representation of the environment. The green elements depict the environment, the light brown element resembles the agent. VMs 1-3 (the *vulnerable* VMs) are only connected to VM4 (the *agent* VM), which acts as a router and forwards network data packets. Besides forwarding the agent VM also captures the data packets and passes them on to a feature extraction unit. The output of this process is then buffered and handed to the agent in the environment's state representation. Based on this information the agent then chooses an action.

the second action, all traffic coming from the host is blocked. Since this block cannot be lifted again during an episode, the host stays in the same state for the rest of the episode, where it is incapable of sending network messages to other hosts. Also the agent cannot cut a hosts network connection a second time, leaving it left with one action only. Since there is no semantically benefit in waiting until the episode ends, while applying the only action left (continue), the cut action effectively ends the episode as soon as it is applied. For a recorded episode this means that in the case of continue, all recorded messages are replayed in the order as they appeared. In the case of cut, all further recorded messages are suppressed (as the network connection is cut) and the episode ends immediately.

Capturing the network traffic is done via *tshark*[1], a well known tool for live capturing network data. For the record/replay functionality the recorded network packets are stored in JSON format. An example data structure can be seen in listing 1. During replay, only certain features are extracted from the JSON data. These features are gathered in a feature vector, which serves as the input to the state representation of the environment. Although only certain features are used, the data is still stored in JSON format in order to enable future comparisons of feature selection methods. More details on the feature selection are provided in section 3.2.

In a real world scenario the initial infection on the network often requires user interaction. A malicious link on a website or a malicious attachment in an e-mail are common initial incident vectors. In this environment the initial user behavior is simulated. The file containing the actual malware is already in place on every vulnerable VM, but is not exe-

[1] https://www.wireshark.org/docs/man-pages/tshark.html

```
{
    "_source":{
        "layers":{
            "arp":{
                "arp.proto.type":"0x00000800",
                "arp.src.hw_mac":"**:**:**:**:**:**",
                "arp.hw.type":"1",
                "arp.src.proto_ipv4":"192.168.1.1",
                "arp.dst.proto_ipv4":"192.168.1.254",
                "arp.dst.hw_mac":"00:00:00:00:00:00",
                "arp.hw.size":"6","arp.proto.size":"4",
                "arp.opcode":"1"
            },
            "frame":{
                "frame.time_relative":"4.599587614",
                "frame.interface_id_tree":{
                    "frame.interface_name":"enp0s8"
                },
                "frame.offset_shift":"0.000000000",
                "frame.cap_len":"60",
                "frame.ignored":"0",
                "frame.protocols":"eth:ethertype:arp",
                "frame.time_delta":"0.004918768",
                "frame.encap_type":"1",
                "frame.len":"60",
                "frame.time_delta_displayed":"0.004918768",
                "frame.number":"33",
                "frame.marked":"0",
                "frame.interface_id":"0"
            },
            "eth":{
                "eth.dst_tree":{
                    "eth.addr":"ff:ff:ff:ff:ff:ff",
                    "eth.dst_resolved":"Broadcast",
                    "eth.ig":"1",
                    "eth.addr_resolved":"Broadcast",
                    "eth.lg":"1"
                },
                "eth.src_tree":{
                    "eth.src_resolved":"PcsCompu_00:a3:3d",
                    "eth.addr":"**:**:**:**:**:**",
                    "eth.ig":"0",
                    "eth.addr_resolved":"PcsCompu_00:a3:3d",
                    "eth.lg":"0"
                },
                "eth.dst":"ff:ff:ff:ff:ff:ff",
                "eth.src""**:**:**:**:**:**",
                "eth.type":"0x00000806",
            }
        }
    }
}
```


Listing 1: Example JSON data of an *arp* request

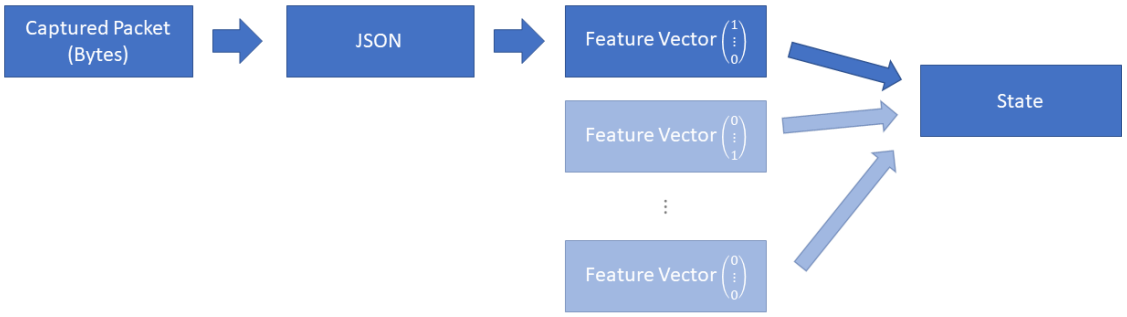


Figure 4.2: Schmeatic representation of a network packet. The raw data packet is captured from the network interface. For storage as well as for additional processing the packet is transformed into JSON format. The feature selection (described in section 3.2) continues from that and returns a feature vector $x \in \mathbb{N}^d$. The representation is then the sum of all these feature vectors originating from data pacets of the same time-step.

cuted by default. A small script, that automatically runs at startup, listens on port 4449 for an instruction to infect the machine. This port is not included in the data collection of the agent VM, as it would enable the agent to learn to look out for this very instruction. As soon as such an instruction comes from the agent VM the script executes the stored malicious file and releases the malware on the host and in the network. The agent VM is the only VM that sends such infection instructions to the vulnerable VMs.

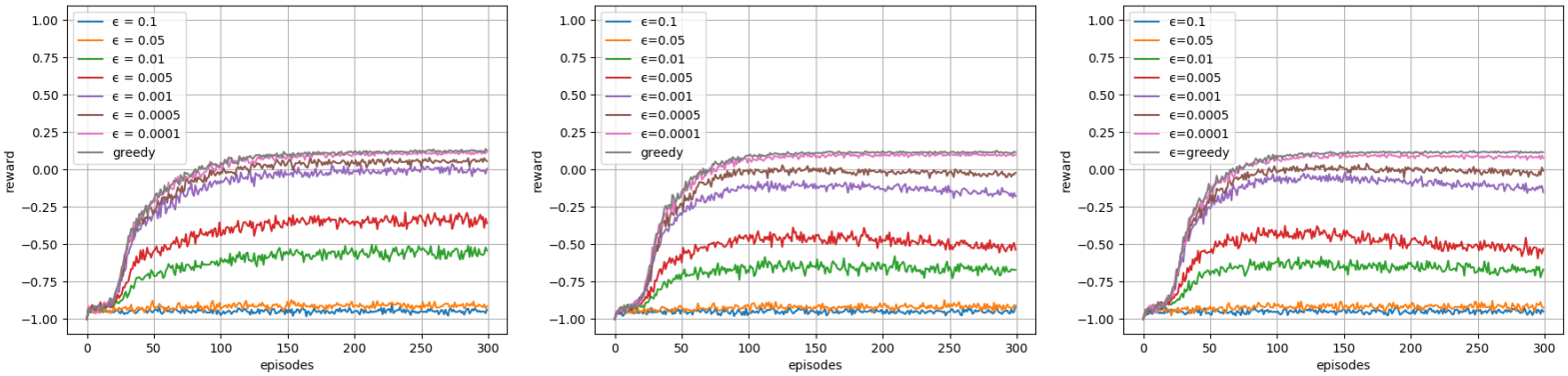
For the experiments we created a data set of 20 recorded episodes, 5 in which no infection happened and 15 with a random infection. This data set is then extended by two of similar size in order to see how representative the generated data is and how the transfer of learned knowledge between the data sets performs.

4.1 Simple Feature Encoding

Since the simple feature encoding method results in a one dimensional state of integers, function approximation was omitted and tabular reinforcement learning was applied. A comparison between the learning algorithms Monte Carlo method, SARSA and Q-Learning, and also the differences by varying ϵ can be seen in Figure 4.3. Figure 4.4 shows a direct comparison of the 3 best ϵ values of the different learning algorithms.

4.2 Sophisticated Feature Encoding

Figure 4.5 and Figure 4.6 show the results of a hyper-parameter search in the sophisticated feature extraction setting, for the step size α, the greediness ϵ and the amount of nearest neighbors k. These are searched independently for SARSA and Q-Learning. All further



(a) Monte Carlo (b) SARSA (c) Q-Learning

Figure 4.3: A comparison of 3 different learning algorithms (Monte Carlo, SARSA and *Q*-Learning) using the simple feature extraction (counting *arp* packets within a time-step) and different parameters for exploration (ϵ). Each line in all 3 plots is averaged over 500 runs.

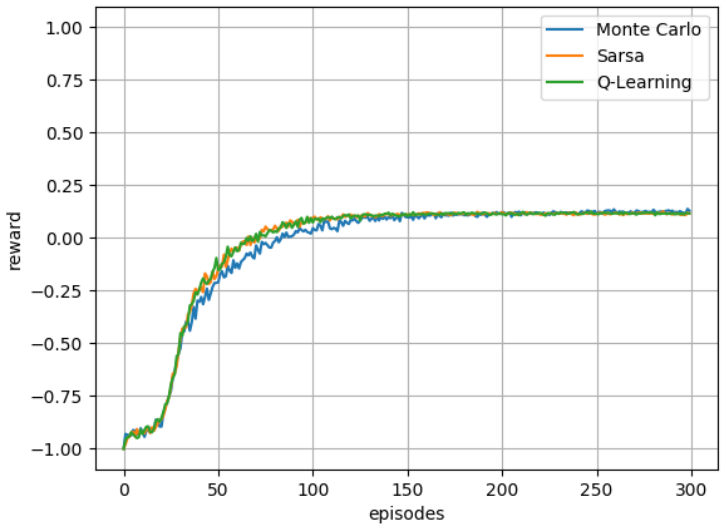


Figure 4.4: A direct comparison of the 3 best performing agents per learning algorithm of Figure 4.3.

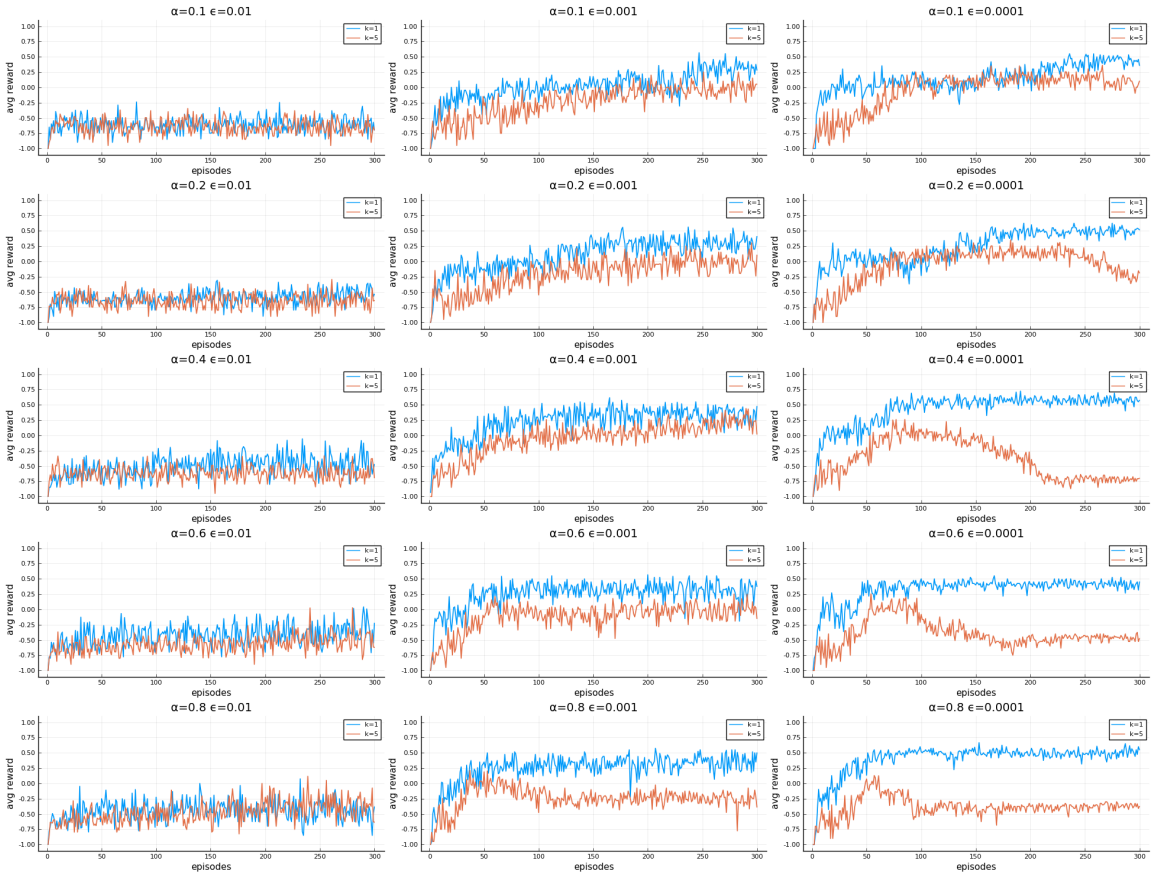


Figure 4.5: A hyper-parameter search for the SARSA algorithm using the sophisticated feature extraction. The hyper-parameters are step size α, exploration ϵ and number k of nearest neighbors. α changes over rows, ϵ changes over columns. Each line in all subplots is averaged over 20 runs.

results are generated with SARSA as a learning algorithm with $\alpha = 0.4$, $\epsilon = 0.0001$ and $k = 1$, because it turned out to have the best results during the hyper-parameter search.

Besides the initial data set (data set 1) – where the previous results are based on – two additional data sets of similar size were recorded (data set 2 and data set 3) to further evaluate the setup. Figure 4.7a shows a comparison of all 3 data sets, when the agent is only trained on a single data set. The other sub-figures (Figure 4.7b, Figure 4.7c and Figure 4.7d) show different comparisons between the data sets. In each of these sub-figures in the first 50 episodes all 3 agents were trained on one of the 3 different data sets, each agent on a separate one. After 50 episodes, all agents were switched to the same data set. Thus these 3 sub-figures display how beneficial it is to transfer knowledge based on one particular data set to an environment based on another data set. If the data distributions of the data sets are the same, then an agent with transferred knowledge should perform as good as an agent that was trained only on the switched to data set.

Finally, Figure 4.8, Figure 4.9 and Figure 4.10 show the results when these three data

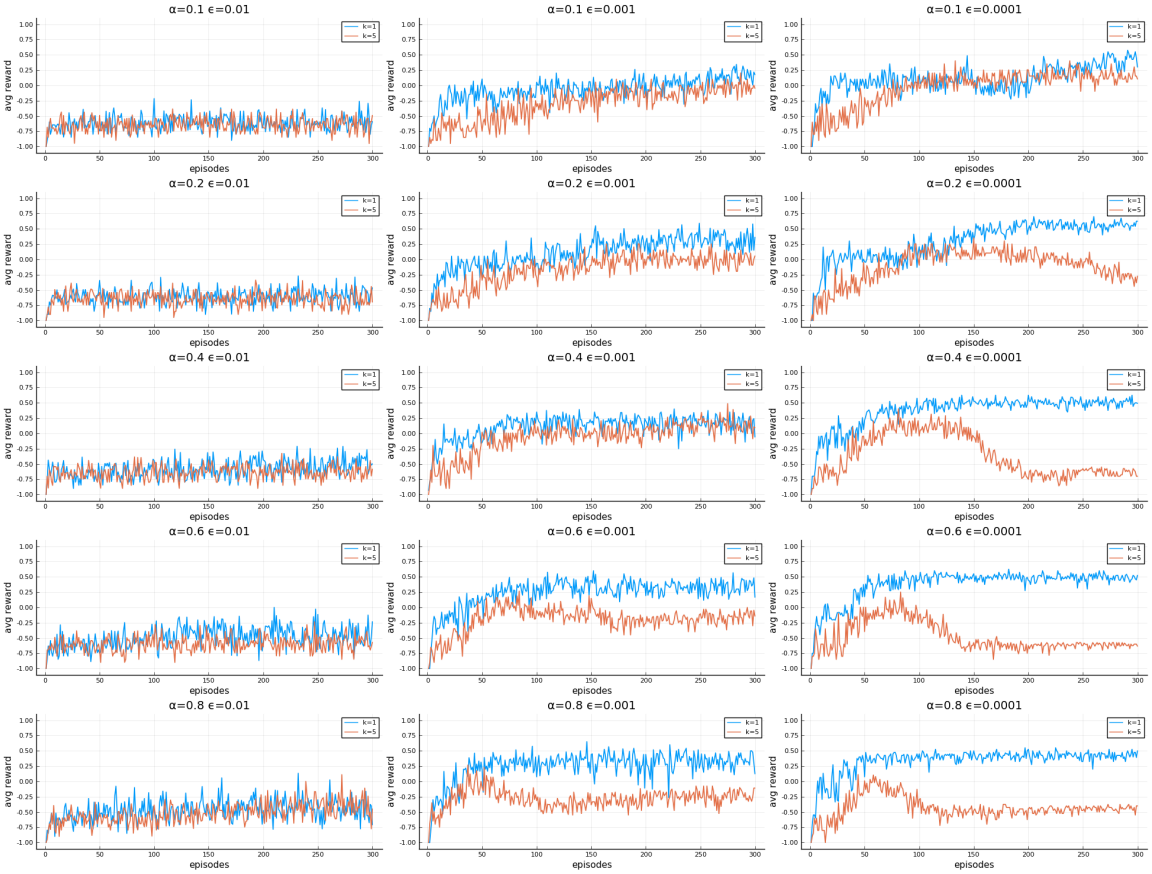
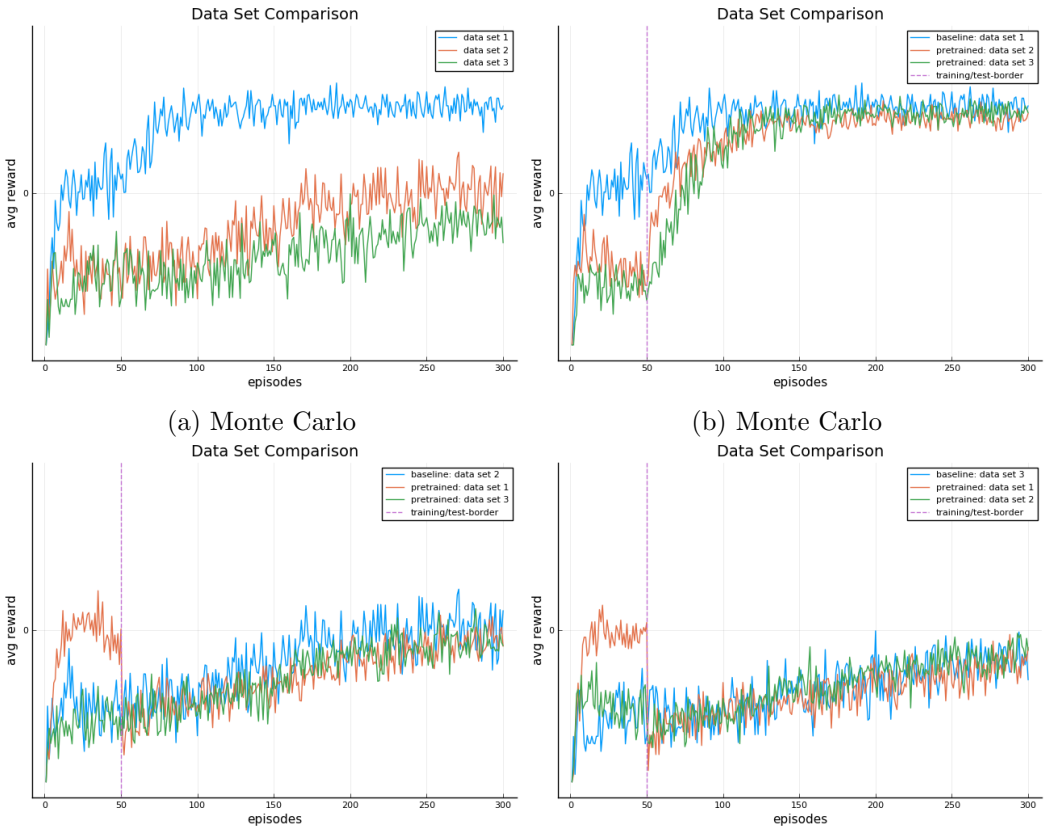


Figure 4.6: A hyper-parameter search for the Q-Learning algorithm using the sophisticated feature extraction. The hyper-parameters are step size α, exploration ϵ and number k of nearest neighbors. α changes over rows, ϵ changes over columns. Each line in all subplots is averaged over 20 runs.



(a) Monte Carlo (b) Monte Carlo

(c) Monte Carlo (d) Monte Carlo

Figure 4.7: Besides the initial data set (data set 1), two additional data sets were recorded. Figure (a) shows a comparison of the 3 data sets with agents performing on the sets separately. Figures (b) – (d) show a comparison of transferred knowledge. For the first 50 episodes, each agent trains on its separate data set. After 50 episodes all agents are transferred to the same baseline data set. This shows how good a trained knowledge base performs on a different data set. Each line in all 4 subplots is averaged over 50 runs.

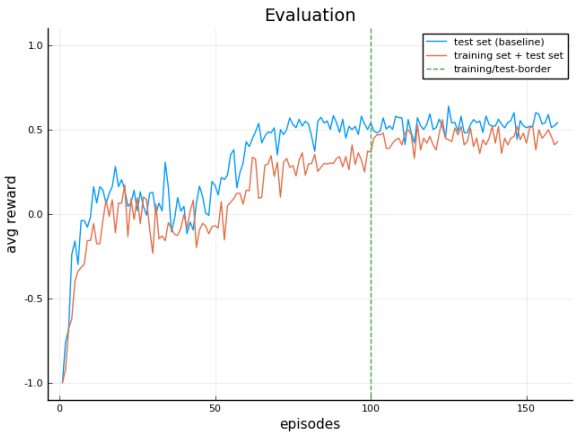


Figure 4.8: A train-test comparison on data set 1. Data set 1 is split into a training- and test set (3 : 1). An agent is then trained on the training set and evaluated on the test set. Another agent solely operating on the test set acts as upper bound. Each line in the graph is averaged over 50 runs.

sets are each split into a training and test set. This split is done in a ratio of 3 : 1. Each figure uses an agent that only learns from the test set as an upper bound. Such an agent is clearly overfit to the test set, but also shows a good estimate of the maximum possible average reward. The training phase includes 100 episodes from the training set and the following test phase includes 50 episodes from data of the test set.

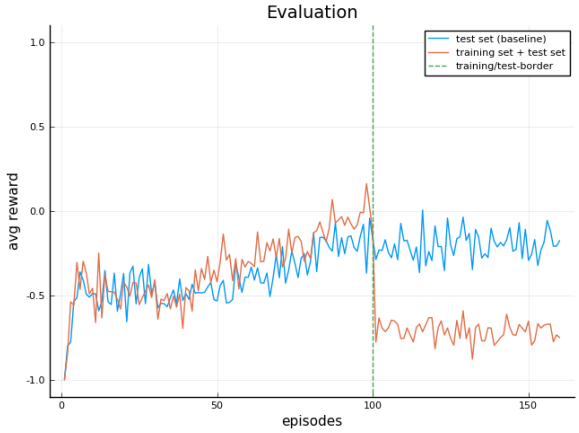


Figure 4.9: A train-test comparison on data set 2. Data set 2 is split into a training- and test set (3 : 1). An agent is then trained on the training set and evaluated on the test set. Another agent solely operating on the test set acts as upper bound. Each line in the graph is averaged over 50 runs.

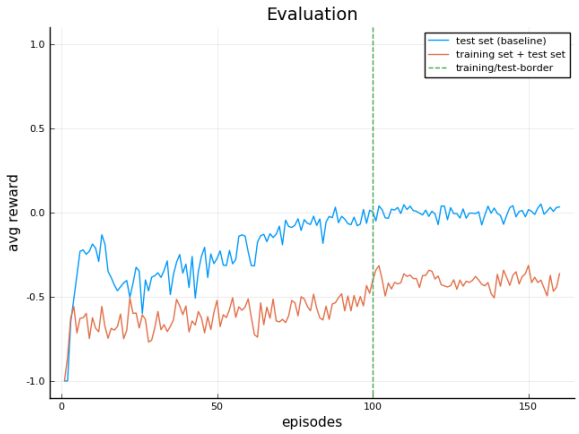


Figure 4.10: A train-test comparison on data set 3. Data set 3 is split into a training- and test set (3 : 1). An agent is then trained on the training set and evaluated on the test set. Another agent solely operating on the test set acts as upper bound. The base line is averaged over 50 runs, the train-test split is averaged over 100 runs.

5 Discussion

Our experiments using the simple feature encoding (Figure 4.3) show that, regardless of the learning algorithm, a greedy action selection algorithm performs best. This result is not too surprising as there is not much noise integrated in the environment, which would let the agent be uncertain in its value function. The simple feature encoding only counts the amount of *arp* packages within a time-step. Non-infected hosts only send out *arp* packages once in a while and then cache the result. Infected hosts on the other hand enumerate all possible hosts within a network and try to contact each of them using the *arp* protocol, resulting in 3-4 data packets per time-step after the infection. This situation does not appear in non-infected hosts in our lab environment and therefore is a clear distinguishing factor and resulting in greedy action selection performing best.

A more surprising observation is the performance drop of agents using SARSA and Q-Learning with ϵ between 0.01 and 0.0005 after an initial peak around episode 100.

Figure 4.4 clearly shows the faster learning capabilities of SARSA and Q-Learning, compared to Monte Carlo methods, although this learning advantage was expected to be larger. One possible reason behind this is again a lack of random noise in the environment and because of a general simple environment structure.

The hyper-parameter searches for the more sophisticated feature extraction (Figure 4.5 and Figure 4.6) again show, that an ϵ value of 0.01 is still too large and no learning takes place. An interesting observation in these plots is a drastic decrease in performance for $k = 5$ in combination with $\epsilon = 0.0001$ and $\alpha \geq 0.2$ or $\epsilon = 0.001$ and $\alpha \geq 0.6$. Again we suspect this appears because of a lack of random noise in the environment. Therefore once the buffer fills with more data, the $k = 5$ parameter leads to mix good estimator points with bad ones, resulting in a worse estimation than using only a single estimator point. In general, the results for SARSA and Q-Learning appear very similar.

The results of Figure 4.7a suggest that unfortunately an agent with the same hyper-parameters is not able to achieve the same performance on all three data sets. We suspect the reason behind this is a discrepancy between the time-step the environment records the release of the malware and the earliest time-step the agent is able to detect an infection. The reward calculation heavily depends on the assumption that this time difference is negligible. The theoretical maximum of the reward function (1) is only met if the agent cuts the network connection to the infected host exactly one time-step after the infection has happened. If the malware does not interact with the network for several time-steps, the agent is not able to detect the infection that early. Instead it can only detect the infection when the malware first sends data packets across the network, resulting in a lower actual reward maximum. We suspect the graphs for data set 2 and data set 3 in Figure 4.7a perform poorly, because of this effect.

Switching from data set 1 to one of the other two results in a performance drop, as depicted in Figure 4.7c and Figure 4.7d. This is expected, assuming the time delay as-

sumption, introduced in the previous paragraph, holds. Also the performance after the switch from data set 1 to 2 or 3 does not behave significantly worse than the respective baseline. Switching from data set 2 or 3 to data set 1 (Figure 4.7b) results in a steep increase in average reward already in the first few episodes after the change. This reinforces the time delay assumption, because an increase based on new experience would require more episodes in the new environment. On the other hand the average reward does not increase immediately to the level of the baseline, therefore also suggesting a difference in the data distribution.

Figure 4.8 shows no performance drop after the train-test border. The results nearly match the ones of the baseline, suggesting that the data of train and test set are of similar distribution and the agent is capable of learning the relevant information from the training set. Figure 4.9 on the other hand, shows a significant drop in average reward right after the train-test border and a lower performance than the baseline agent afterwards. A likely explanation for this outcome is the agent lacking necessary information it requires in the test set. Since the model (the k-NN algorithm) basically uses raw data, we assume that the missing information is generally missing in the training data set and not excluded in the model building process. Figure 4.10 displays a similar image to Figure 4.8, albeit in a lower range of average reward and also with a larger gap to the baseline. A closer inspection of the raw data of individual runs indicated, that the learning success depends on the replay order of the recorded episodes, further indicating that some time-steps are better predictors in the k-NN algorithm than others. We suspect this phenomenon shows, due to a lack of intelligence in the data point decision mechanism of the k-NN algorithm. Currently all data is added to the comparison buffer of the k-NN algorithm in a first in first out order, regardless of points already in the buffer. Adding more intelligence to this selection process could be a good starting point for future work.

6 Conclusion

In this thesis we showed that it is possible to create a self-propagating malware containment system via trial- and error learning. For that, we applied reinforcement learning algorithms to an environment of virtual machines containing real world software and also real world malware. In particular, we compared SARSA and Q-Learning by also leveraging a k-nearest neighbors based function approximation approach. This approach compares states via the amount of data packets sent within a certain time-span, grouped by destination and packet type. The agents action spaces only stretches from no action to cutting network connections, with no action in between. Our empirical results show that a trained agent is capable of distinguishing when to cut a network connection to an infected host and when to not intervene with the system. Additionally, the RL based approach allows our agent to learn the difference between benign and malicious data packets without the need of labeling each packet.

Bibliography

[AR19] Aqeel Anwar and Arijit Raychowdhury. Autonomous Navigation via Deep Reinforcement Learning for Resource Constraint Edge Nodes using Transfer Learning. *arXiv e-prints*, page arXiv:1910.05547, Oct 2019.

[BBC+19] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680*, 2019.

[BNVB13] Marc G. Bellemare, Yafar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: an Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: an Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[LLL+19] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A Framework for Reinforcement Learning in Games. *arXiv preprint arXiv:1908.09453*, 2019.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[SAH+20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis,

Thore Graepel, et al. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839):604–609, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. MIT press, 2018.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *nature*, 529(7587):484–489, 2016.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A Physics Engine for Model-Based Control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster Level in StarCraft II using Multi-Agent Reinforcement Learning. *Nature*, 575(7782):350–354, 2019.

[VEB+17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft II: A new Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782*, 2017.

[WD92] Christopher J.C.H. Watkins and Peter Dayan. Q-Learning. *Machine learning*, 8(3-4):279–292, 1992.