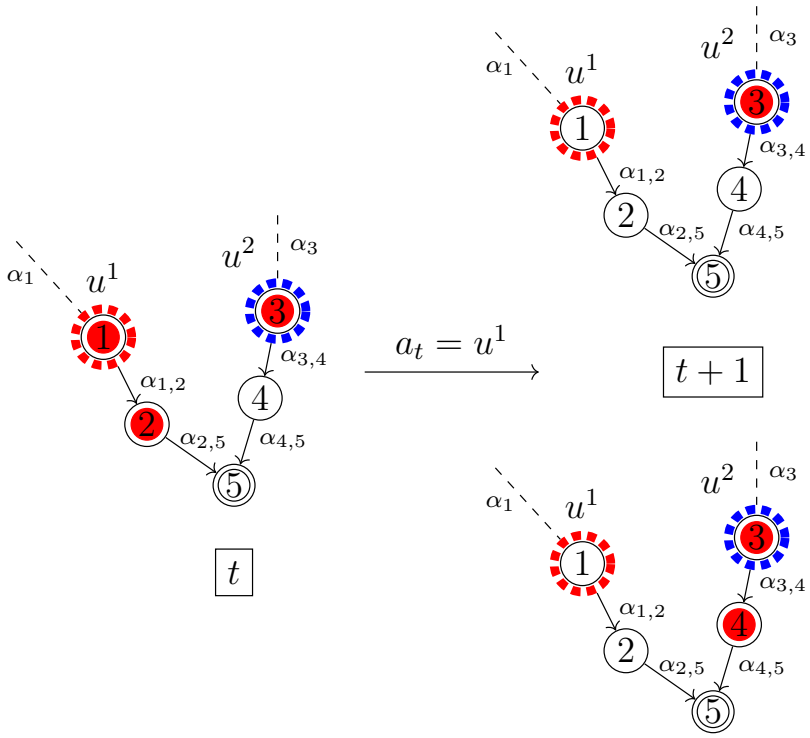
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Efficient Solving Methods for POMDP-based Threat Defense Environments on Bayesian Attack Graphs

Master's thesis in Computer Science – Algorithms, Languages and Logic

JOHAN BACKMAN
HAMPUS RAMSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Efficient Solving Methods for POMDP-based Threat Defense Environments on Bayesian Attack Graphs

JOHAN BACKMAN
HAMPUS RAMSTRÖM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Efficient Solving Methods for POMDP-based Threat Defense Environments on Bayesian Attack Graphs

Johan Backman
Hampus Ramström

Supervisor: Christos Dimitrakakis, Department of Computer Science and Engineering
Examiner: Devdatt Dubhashi, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example computer system modeled as a Bayesian Attack Graph and formulated as a POMDP defense problem. First in time-step t, then in two different potential states at $t + 1$, after performing a countermeasure action u^1 in t. Full details in Section 3, Figure 3.1.

Typeset in LaTeX
Gothenburg, Sweden 2018

Efficient Solving Methods for POMDP-based Threat Defense Environments on Bayesian Attack Graphs

JOHAN BACKMAN
HAMPUS RAMSTRÖM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this work, we show how to formulate a threat defense environment as a Partially Observable Markov Decision Process (POMDP) that allows for fast approximate defense algorithms against multiple attackers. It is done through an action extension, coined the Inspect action, which allows the agent to reveal the true state of the environment, thereby reducing the problem into a traditional Markov Decision Process (MDP) for the current time-step. The work is an extension of previous definitions of the same problem. Furthermore, based on the new definition we define and show the optimal policy, as well as two new solving algorithms, n-Myopic and n-Lookahead. To evaluate their performance, we show and compare the results of these new algorithms to more standard solving algorithms, such as Q-learning and Policy Gradients.

The experimental results show that the new algorithms perform better than previous attempts and allows for larger scale threat environments thanks to the approximate MDP reduction. Additionally, to facilitate future research, two OpenAI Gym environments were developed and are publicly available for new research to build upon. We encourage new research with similar problem description to use this software library, opening up to standardized performance results.

Keywords: Reinforcement Learning, POMDP, Bayesian Attack Graphs, Security, Defense Policies, OpenAI Gym, Threat Defense

Acknowledgements

First and foremost we would like to thank our supervisor, Christos Dimitrakakis, for his invaluable knowledge, insights and ideas throughout the whole project. Another thanks goes out to Christos together with the whole CSE Institution at Chalmers for being so flexible, allowing us to work on the project from remote locations.

We also would like to thank Devdatt Dubhashi for making the time and effort acting as examiner for this thesis project.

Special thanks to Johan Ek, for the rewarding cooperation and peer-reviewing, as well as for being the appointed opponent for this thesis as a whole.

Sincere thanks to our friends and family providing feedback on the content, formatting and language. Special mention goes out to: Sune Ramström, Trina Vana and Alexander Wong. You made this report a hundred times better!

Thank You,

Johan Backman & Hampus Ramström
Gothenburg, Aug 2018

Contents

List of Figures

List of Tables

1

Introduction

As more and more devices connect to the internet and utilize its services, the needs for robust security and auditing solutions are growing. There is a wide range of attacks targeting computer systems, such as malware, zero-day exploits, and network worms. In 2016, the Internet Complaint Centre at the Federal Bureau of Investigation (FBI) received almost 300 000 reports of cyber-crime, which totalled $1.3 billion in losses in the United States alone (Federal Bureau of Investigation, 2016).

The traditional way of performing security revolves around locking down the network and the operating system. When any kind of event happens that falls outside normal behavior, a response team of operators investigate the incident (Krutz & Vines, 2010). But as software is growing in size and complexity every year, the number of potential security incidents is growing with it. In most cases it is impossible for a security team to audit every single event, validating that the traditional way is insufficient, and that new automated methods needs to be explored. By having a security system that monitors an environment and suggests or handles incidents automatically, the human effort can be alleviated.

The introduction is divided into five sections: aim, problem, related work, the contribution to the field, and a general overview of the whole thesis. This pattern is applied throughout the thesis where an organizational introduction is presented in each chapter followed by more detailed sections.

1.1 Aim

The aim of the project is to formulate a threat defense environment as a POMDP-model, that allows for fast approximate defense algorithms to detect security vulnerabilities in a computer system. The model has to scale well for a system of increasing size. Previous attempts have shown the complexity of solving large POMDP-based models. Therefore, finding trade-offs and optimizations in the model to make it solvable within a usable time frame is of high relevance (Shmaryahu, Shani, Hoffmann, & Steinmetz, 2017).

The system has to be fully automated. Many attacks are carried out at such a fast pace that a human operator would not be able to interpret the data and act fast enough. Additionally, the scale and cost of operating human inspection on security events is not feasible due to the scale and complexity of modern systems.

By modeling the threat defense environment and developing fast approximate defense algorithms, the thesis tries to answer the following question:

> "Can an automated system be modeled as a POMDP in such a way that it works in a real-time context to find and efficiently respond to simultaneous attacks against a computer system?"

There is previous research showing how to model this problem. Early work done by Liu and Man (2005) and Poolsappasit, Dewri, and Ray (2012), show the viability of attack graphs. They are a special case of scenario graphs, where each path in an attack graph is a series of exploits that leads to an undesirable state (Sheyner, 2004). In recent years, advancements in the field were made by Miehling et al. (2015), by deploying defense policies in attack graphs. The limitation of previous work is the constraint to small attack graphs, i.e. to solely model small systems, and the focus on restrictive defensive actions, temporally shutting down parts of the system, instead of information gathering defensive actions. This thesis addresses the research question through several criteria:

1. Model multiple attackers in a system, and implement reinforcement learning algorithms such that a defender chooses defensive actions to counter the attacks. The actions will be based on imperfect information yielded from the environment, regarding the state of the network, i.e. if an attack is taking place or not.

2. Extend the set of defensive actions from solely availability restricting to incorporate an information gathering action. It will consist of an expert inspecting the state of the network.

3. Model and self-develop new reinforcement learning algorithms utilizing the new defensive action.

4. Implement the models to prove their viability.

5. Evaluate the implementations.

Since finding optimal policies for POMDP is *PSPACE-complete* (Papadimitriou & Tsitsiklis, 1987), we need approximate strategies to make it feasible to deploy the model in any system, regardless the size of the system. This can be approached from several different angles, e.g. by finding approximations through Policy Gradient methods (Aberdeen & Baxter, 2002) or work on the attack graph generation of the system, such that more efficient models are created (Ou, Boyer, & McQueen, 2006).

In comparison to Miehling et al. (2015), we interpret how the defensive countermeasures affect the environment, and the definition of the optimal policy differently. Thus, a comparison between our results would not be justifiable. Instead, we will compare the developed, and applied algorithms to each other, as well as to the self-defined optimal policy.

1.2 Problem

An operator wants to defend its environment of a computer system from external attacks. The preventive actions have to happen in real-time to make sure no sensitive data leaves the system. The attacks are assumed to happen faster than the operator manually can respond to. With this in mind the problem involves modeling a system as a POMDP, that can be used to analyze the vulnerabilities and exploits posed to the operator's environment. Using this analysis of imperfect information of the attacker(s), the system chooses appropriate countermeasures. Either by, if necessary, disabling services that the vulnerabilities and exploits depend upon, or by investigating the state of the network by applying expertise. The defense system should also lean upon the more secure option, i.e. take no unnecessary risks, where a partial shutdown is favored to a scenario in which the attacker has partial control of the system. However, the cost of countermeasure deployment has to be considered, as we want to maintain accessibility to the system without having a force of experts continuously monitoring it, which for large systems would be unfeasible.

The problem is divided into two parts; modeling the attackers capabilities and spread throughout the environment as a *Bayesian Attack Graph*, and determining the optimal defense strategy as a POMDP for the defender observing said attacks, and take appropriate action(s). The formulation of the problem, and the notations and definitions utilized, generally follows the work done by Miehling et al. (2015).

1.3 Related work

The idea of attempting to detect attacks and vulnerabilities automatically is not new within the security research community. The general term used to describe the problem is *Intrusion Detection*. Researched solutions range from using fixed signatures of malicious code, to anomaly detection models, utilizing advanced machine learning techniques. The different approaches all have trade-offs and challenges. Signature-based solutions are efficient and easy to implement, but fail to detect unknown attacks. Meanwhile, anomaly-based solutions offer a way to detect unknown attacks, but adds computational complexity, false positives, and in turn makes it harder to trigger events in real-time due to the delay in processing (Liao, Lin, Lin, & Tung, 2013).

A related research topic is the detection of security vulnerabilities in a system by exposing it to friendly attacks, so called *penetration testing*, where the objective is to obviate potential paths of intrusion. Network security has successfully been targeted for this type of evaluation, by generating friendly attacks and then automatically running them against an environment (Sarraute, Buffet, & Hoffmann, 2012). One of the more advanced platforms utilize *Partially Observable Markov Decision Processes* (POMDP) to model the problem. This area faces the same kind of challenges as intrusion detection, it should be done with minimal intervention from human operators, and needs to accurately find vulnerabilities (Hoffmann, 2015). The problem with using POMDP as a modeling strategy is the computational complexity when scaling to larger graphs (Shmaryahu et al., 2017).

Finding optimal policies for POMDP is *PSPACE-complete* (Papadimitriou & Tsitsiklis, 1987), but recent research results on the issue of scalability show great promise. Silver and Veness (2010) introduced *POMCP*, a Monte-Carlo algorithm for online planning that has successfully scaled up to very large POMDPs. Following, Somani, Ye, Hsu, and Lee (2013) introduced *R-DESPOT* that avoids POMCP's poor worst-case behaviour by evaluating policies on a small number of sampled *scenarios* instead. They further show through theoretical analysis and experiments that their new approach outperforms two of the fastest online POMDP planning algorithms to that date, including *POMCP*. In addition to showing strong results, both projects are open source.

Much less research focuses on the actual defense when an attack or vulnerability has been found. Some early research by Wells, Pazandak, Nodine, and Cassandra (2004), show promise using a POMDP as basis for an automatic intervention system. They implement a full software solution that proves they can assess and intervene in a multi-agent environment. This furthermore highlight the demand for trade-offs between accuracy and evaluation time. Some more recent research by Miehling et al. (2015) has shown promise in using a POMDP-based model to model attacks and their corresponding defense strategies. Using previously mentioned penetration testing ideas, they build a model with a planner that looks at potential threats, with the addition of a defender that can take actions, and alter the possible attack vectors.

In the research done by Miehling et al. (2015), solely defensive actions of restricting the availability by disabling services that vulnerabilities and exploits depend upon are considered. Thus, we chose to extend the set of actions by adding an information gathering defensive action, where an expert investigates the state of the computer system. It is not the first time that an action of this kind is implemented (Hansen & Feng, 2000), but to the best of our knowledge, it is the first time that it is implemented in these specific settings.

When performing a study of the current literature within the subject, we found that it was hard to reproduce, and compare results due to little standardization on the metrics used by researchers. Leading to the development of two OpenAI gym environments, to facilitate better and easier peer reviews within this specific topic (Brockman et al., 2016).

1.4 Contributions

There are three major contributions that will be presented in this thesis, each enumerated below. The code for each contribution is publicly available, see *Appendix A* for details.

1. An attack graph model that improves on previous work, by introducing an additional information gathering action. Additionally, two model specific algorithms, based on the new action, to solve the environment.

2. Definition of the optimal policy strategy, and experimental results with comparisons.

3. Standardization of the evaluation of two attack graph-based reinforcement learning models through two OpenAI gym environments.

1.5 Overview

The thesis is organized into *five* main chapters. The first chapter details fundamental theory that the rest of the thesis builds upon. A reader familiar with the reinforcement learning field should focus on the part denoting theory related to the threat environment definition, but can skip the section on reinforcement learning solving algorithms.

The problem has two parts: modeling the environment and defining solving algorithms for said environment. Therefore, the presentation of results is organized into a chapter about the environments, and a separate chapter detailing the solving algorithms, Chapters 3 and 4 respectively. The specific parameters of each environment and algorithm, with experimental results are presented in Chapter 5. Finally, the last chapter 6 Conclusion, contains discussion of the results, future work and ethical considerations.

2

Theory

This chapter covers the fundamental theory required to define the two attack graph-based models, also incorporating used definitions from previous work. Furthermore, this section details the theory of the standard solving algorithms in Chapter 4, used in the experimental results in Chapter 5.

2.1 POMDP

As a POMDP is a generalization of a MDP, we first introduce the definition of a MDP in Definition 2.1.1, followed by the definition of a POMDP in Definition 2.1.2.

Definition 2.1.1 (Markov Decision Process)**.** A Markov Decision Process (MDP) is defined by a 5-tuple $\langle \mathcal{S}, \mathcal{A}, T_a, R_a, \gamma \rangle$, where

- $\mathcal{S}$ is the set of states.
- $\mathcal{A}$ is the set of actions.
- $T_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$, is the transition probability that action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ will lead to $s' \in \mathcal{S}$.
- $R_a(s)$ is the scalar reward received by taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.
- $\gamma \in [0, 1]$ is the discount factor, denoting the difference in how much immediate and distant reward should affect the agent.

Definition 2.1.2 (Partially Observable Markov Decision Process)**.** A Partially Observable Markov Decision Process (POMDP) is a generalization of a MDP. In this generalization the 5-tuple is extended to a 7-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, T_a, O_a, R_a, \gamma \rangle$, where in addition to the previous MDP definition in 2.1.1

- $\mathcal{O}$ is the set of observations.

- $O_a(s') = P(o_{t+1} = o | s_{t+1} = s', a_t = a)$ is the observation probability that action $a \in \mathcal{A}$ resulting in state $s' \in \mathcal{S}$ will lead to the observation $o \in \mathcal{O}$.

At each time-step, the process is in a state $s \in \mathcal{S}$. The environment updates its state when an agent performs an action $a \in \mathcal{A}$. By taking the action the environment transitions from state s to state s' based on the transition probability $T_a(s, s')$. When performing action a in state s, the agent receives a reward $r = R_a(s)$. In the case of a POMDP, the true state s' is unknown to the agent, instead the agent receives an observation $o \in \mathcal{O}$ based on the observation probability $O_a(s')$, which can be action dependent (Smallwood & Sondik, 1973).

In both cases of a MDP and a POMDP, the goal of an agent is to find an optimal policy π^*. A policy π is the agent's behaviour function. It is a mapping from state to action, which can both be deterministic and stochastic, dictating what action $a \in \mathcal{A}$ to take given a particular state $s \in \mathcal{S}$ (Silver, 2015a). It can be defined as

$$\begin{aligned} &\pi(s) = a && \text{if deterministic,} \\ &\pi(s|a) = P(a_t = a | s_t = s) && \text{if stochastic.} \end{aligned}$$

In the optimal case, π^* should maximize the agent's *expected future discounted reward*, which can be defined as

$$V^* = V^{\pi^*}(b_0) = \underset{\pi}{\text{maximize}} \quad \mathbb{E}^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t | b_0\right],$$

where b_0 is the initial probability distribution over states (Dimitrakakis & Ortner, 2018).

2.2 Conflict environment model

The formulation of the problem, notations, and definitions utilized in this thesis generally follows the work done by Miehling et al. (2015). The problem is divided into two parts; modelling the attackers capabilities and spread throughout the environment as a *Bayesian Attack Graph* and determining the optimal defense policy as a POMDP for the defender observing said attacks and based on the policy take appropriate action(s).

2.2.1 Bayesian Attack Graphs

In a *Bayesian Attack Graph* the nodes are denoted *attributes*, and the edges are denoted *exploits*. The attributes are *attacker capabilities*, or vulnerability probabilities from the defender's viewpoint. Some example attributes could be the level of permission the attacker has on a given machine, information leakage, or a specific vulnerability in a service or a system. Exploits on the other hand are events that allow the attacker to use their current set of capabilities (attributes) to obtain further capabilities. The formal definition of a Bayesian Attack Graph is defined in Definition 2.2.1 (Miehling et al., 2015).

Definition 2.2.1 (Bayesian Attack Graph)**.** A Bayesian Attack Graph, $\mathcal{G}$, is defined as the tuple $\mathcal{G} = \langle \mathcal{N}, \theta, \mathcal{E}, \mathcal{P} \rangle$, where

- $\mathcal{N} = \{1, \ldots, N\}$ is the set of nodes, termed *attributes*.
- θ is the set of node types. Each non-leaf attribute (a node that has at least one predecessor) is assumed to be one of two types, $\theta = \{\wedge \ (\textbf{AND}), \vee \ (\textbf{OR})\}$. The respective sets of nodes are denoted by $\mathcal{N}_\wedge$ and $\mathcal{N}_\vee$.
- $\mathcal{E}$ is the set of directed edges, termed *exploits*.
- $\mathcal{P}$ is the set of *exploit probabilities* associated with edges. Each exploit (directed edge), $e = (i, j) \in \mathcal{E}$, has an associated probability $\mathcal{P}(e) = \alpha_{ij} \in [0, 1]$.

An assumption made is that each attribute $i \in \mathcal{N}$ can either be enabled, i.e. the attacker hold attribute (capability) i, or disabled, i.e. attribute i is not held by the attacker. Thus, the network state is defined, at time t, as a binary vector $s_t = (s_t^1, \ldots, s_t^N) \in \mathbf{S} := \{0, 1\}^N$, where s_t^i is the state of attribute i at time t and denoted as

$$s_t^i \in \{0 \ (\textit{disabled}), 1 \ (\textit{enabled})\}.$$

The *Bayesian Attack Graph* $\mathcal{G}$ contain both leaf nodes $\mathcal{N}_\mathrm{L} \subseteq \mathcal{N}$, defined as nodes with no predecessors, and root nodes $\mathcal{N}_R \subseteq \mathcal{N}$, which are nodes with no successors. The leaf nodes, $\mathcal{N}_\mathrm{L}$, can be viewed as the bridges connecting to the external world. These nodes are assumed to be the entry-points into the attack graph for the attacker(s). Conversely, root nodes, $\mathcal{N}_\mathrm{R}$, are the attributes found at the deepest exploit level. Among these, there is a subset of root nodes that are viewed by the defender as the most crucial and important attributes to protect, $\mathcal{N}_\mathrm{C} \subseteq \mathcal{N}_\mathrm{R}$. Following the notation defined in Definition 2.2.1, the state of a subset of attributes, e.g. $\mathcal{N}_\mathrm{C}$, from a state s_t, at time t, is denoted as

$$s_t^{\mathcal{N}_C} \in \{0\ (disabled), 1\ (enabled)\}.$$

The node types θ, are assigned to non-leaves $i \in \mathcal{N} \setminus \mathcal{N}_L$, that are either of type **AND**, $i \in \mathcal{N}_\wedge$, or of type **OR**, $i \in \mathcal{N}_\vee$ which sets the condition that the attribute's direct predecessors need to satisfy if the attribute is to become enabled. If $i \in \mathcal{N}_\wedge$, then there exist a non-zero probability of attribute i being enabled at $t+1$ *iff all* direct predecessors, denoted by $\bar{\mathcal{D}}_i = \{j \in \mathcal{N} | (j,i) \in \mathcal{E}\}$ (with $\bar{\mathcal{D}}_i = \varnothing$ for $i \in \mathcal{N}_L$), are enabled at time t. If $|\bar{\mathcal{D}}_i| = 1$, i.e. there exist solely one direct predecessor to i, then i is classified as an **AND** attribute as well. If $i \in \mathcal{N}_\vee$, then if *any* attribute $\bar{\mathcal{D}}_i$ is enabled at time t there exist a non-zero probability that attribute i will be enabled at time $t+1$. An example of a network containing both **AND** and **OR** attributes can be viewed in Figure 2.1.

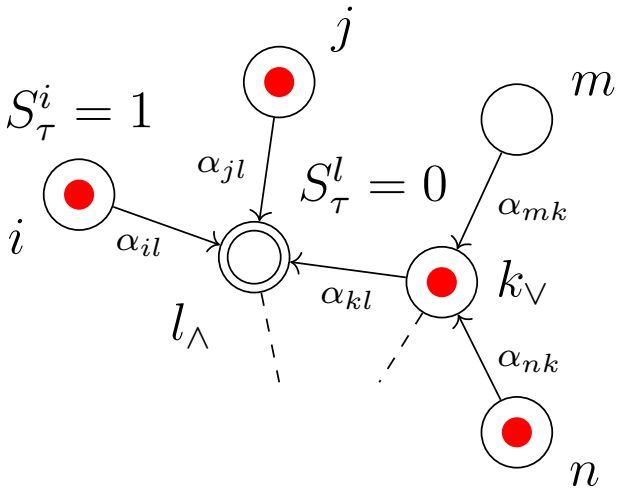


Figure 2.1: An example network at time $t = \tau$ where node $l \in \mathcal{N}_\wedge \cap \mathcal{N}_C$ is an **AND** attribute and a critical attribute, $k \in \mathcal{N}_\vee$ is an **OR** attribute and $i, j, k, n \in \{1\}$, i.e. the nodes with an inner red circle, are enabled. Given this it can be observed that k is enabled without the enabling of m, which is possible since k is an **OR** attribute. The dotted lines indicate that what we observe is solely a part of the graph as a whole.

All exploits that can be found in a system may not be straightforward to utilize. They may be difficult to carry out, or even bypassed, such that the capabilities to perform an exploit is possessed by an attacker, but not carried out. These uncertainties are captured by the *exploit probabilities*, where a probability is assigned to each exploit representing the probability of success.

2.2.2 Attackers

As previously described in Section 2.2.1, the gateway for an attacker into the network is through the leaf nodes, $\mathcal{N}_L$, thus it is assumed in accordance of probabilistic

dynamics that the contagion (attributes enabled by the attacker, i.e. the spread of its capabilities) has its first step at $\mathcal{N}_{\mathrm{L}}$. The probability $\alpha_i \in [0, 1]$ that attribute $i \in \mathcal{N}_{\mathrm{L}}$ is enabled at time-step t is then defined as

$$\alpha_i := P(s_{t+1}^i = 1 | s_t^i = 0),\ i \in \mathcal{N}_{\mathrm{L}}$$

and denoted as the *probability of attack*. The steps of contagion spread that continues are described by the so called *predecessor rules*, coined by Miehling et al. (2015), that describe how the process spreads to an attribute as a function of three factors: i) the attribute's type, ii) the states of the attribute's direct predecessors, and iii) the exploit probabilities. It is denoted as the *probability of spread*, and for the **AND** attributes, $i \in \mathcal{N}_{\wedge}$, we can mathematically define the probability of i transitioning from the disabled state, 0, at time t to the enabled state, 1, at $t + 1$ as

$$P(s_{t+1}^i = 1 | s_t^i = 0, s_t) = \begin{cases} \prod_{j \in \bar{\mathcal{D}}_i} \alpha_{ji} & \text{if } s_t^j = 1\ \forall\ j \in \bar{\mathcal{D}}_i \\ 0 & \text{otherwise} \end{cases}$$

and for the **OR** attributes, $i \in \mathcal{N}_{\vee}$, as

$$P(s_{t+1}^i = 1 | s_t^i = 0, s_t) = \begin{cases} 1 - \prod_{j \in \bar{\mathcal{D}}_i} (1 - \alpha_{ji}) & \text{if } \exists\ j \in \bar{\mathcal{D}}_i \text{ s.t. } s_t^j = 1 \\ 0 & \text{otherwise} \end{cases}$$

where the probabilities are fixed during the whole time span of the decision problem. Worth mentioning, is that these probabilities are unknown to the defender. This follows from the problem description, where the goal is to model a real-world situation as closely as possible. In other words the specific probabilities to enable an attribute is unknown to the security operators.

2.2.3 Defender

As the attacker(s) propagate through the system, the defender will receive observations at each time step t in the form of a vector $o_t \in \mathcal{O} = \mathbf{S}$. The defender will not have perfect information, but instead limited surveillance of the network and thus the defender is not aware of the capabilities that the attacker(s) possess in any given time-step. This is modelled as a *probability of detection*, where at each time step t the defender will observe attribute i as enabled with the probability

$$\beta_i := P(o_t^i = 1 | s_t^i = 1).$$

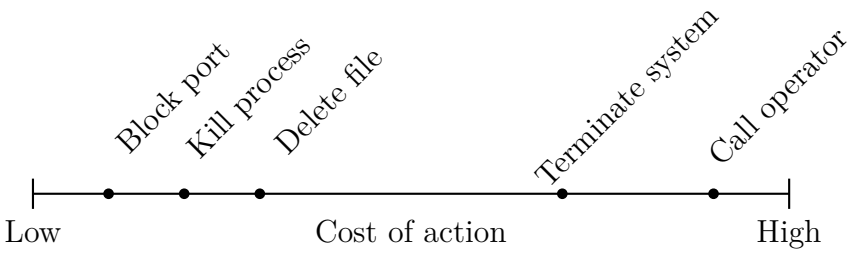


Figure 2.2: Example illustrating the difference in cost for some potential countermeasures that the defender can choose to perform.

The defender will thereby be forced to form a belief of the current state of the network. As with the exploiting probabilities α, the probability of detection is unknown to the defender following the same reasoning.

In order to keep the network secure, the defender will counter the attacker(s) attempts and remove its capabilities by utilizing countermeasures. The defender will have access to M binary actions that forms the countermeasures, i.e. all possible subsets (the power set $\mathcal{P}$) of binary actions constitutes the space of countermeasure actions $\mathcal{U} := \mathcal{P}(\{u^1, \ldots, u^M\})$. Each binary action u^m in countermeasure $u \in \mathcal{U}$ will disable a set of attributes, denoted by $W_{u^m} \subseteq \mathcal{N}$, and each countermeasure will have a cost $C(s, u)$ for state $s \in \mathbf{S}$. The cost will try to mirror the negative and positive effect that the action have on the system, i.e. how effective it is at keeping the critical attributes out of the attacker's control and how it affects authorized users ability to utilize the system. Intuitively, we can define accessibility as if all the system parts are available, i.e. if no attributes are disabled by a countermeasure for a specific amount of time. Our interpretation in this thesis is that it is for one time-step. However, it is far from being a definitive interpretation and is discussed later in Section 3.4.

Some examples of countermeasures and their relative cost can be viewed in Figure 2.2. A defender is not required to perform a countermeasure, where choosing the empty set $u = \varnothing$ implies that the defender is allowing the system to operate uninterrupted. In other words do nothing to gather more observations that can yield a better understanding of the environment.

The *history* of all the countermeasures performed by the defender up to and including time $t-1$, as well as the observations up to and including time t, can be formulated as

$$H_t = (b_0, u_0, o_0, u_1, o_1, \ldots, u_{t-1}, o_t),$$

where $b_t \in S$ is the defender's current belief of the state at time t, where it exist a function $\mathcal{T}$ that updates the defender's current belief b_t with the new (realized) information obtained, i.e. $\{o_{t+1}, u_t\}$, between time-steps t and $t+1$, such that

$$b_{t+1} = \mathcal{T}(b_t, o_{t+1}, u_t).$$

A summary of the history can then be formulated as an *information state*, I_t, which satisfies the conditions that it can be evaluated from H_t, and that there exist a function, $\mathcal{F}_t$, such that $I_{t+1} = \mathcal{F}_t(I_t, o_{t+1}, u_t)$ where o_{t+1} and u_t composes the new information received between time t and $t+1$. It is not certain however that these two conditions constitutes enough information to make an optimal decision. One alternative is to simply use the belief itself, i.e. the probability mass function

$$b_t^s = P(s_t = s|H_t)$$

of being in a certain state, where $s \in \mathbf{S}$, and with the drawback that H_t is *unbounded* in time.

2.2.4 POMDP-formulated defense problem

Based on the problem definition and the attack graph environment in Sections 1.2 and 2.2.1, the problem for the defender becomes to determine an appropriate (optimal) defense policy, $\pi : \mathcal{S} \rightarrow \mathcal{U}$, that map the current belief of the defender b_t at each time-step t to a countermeasure action $u_t \in \mathcal{U}$. With the constraints that the countermeasure should make critical attributes out of control of the attacker(s), while concurrently preserving availability of the system. A realization from this fact is that the defender's policy should not solely rely on the current belief of the network, but also take into account on how the system will change due to action(s) of attackers and the effects of deployed countermeasures. The defender attempts to formulate an optimal defense policy π^* that minimizes the infinite-horizon discounted expected cost V^*, defined as

$$\begin{aligned} V^* = V^{\pi^*}(b_0) = \underset{\pi}{\text{minimize}} \quad & \mathbb{E}^\pi\left[\sum_{t=0}^{\infty} \gamma^t C(b_t, u_t)|b_0\right] \\ \text{subject to} \quad & u_t = \pi(b_t) \\ & b_{t+1} = \mathcal{T}(b_t, o_{t+1}, u_t), \end{aligned}$$

where $\mathbb{E}^\pi[\cdot]$ denotes the expectation with respect to the probability measure induced by policy π. The *discount factor* $\gamma \in [0, 1]$ models the fact that an immediate cost is more certain than a cost in the future, and the expected instantaneous cost is defined as

$$C(b_t, u_t) = \sum_{s \in \mathbf{S}} b_t^s C(s, u_t),$$

where $C(s, u)$ is the instantaneous cost of performing counter measure $u \in \mathcal{U}$ in state $s \in \mathbf{S}$. Dynamic programming is used to obtain a solution to the infinite-horizon discounted expected cost problem that $V^*(b)$ constitutes. As it satisfies the dynamic programming (Bellman) optimality equation,

$$V^*(b) = \min_{u \in \mathcal{U}} Q^*(b, u),$$

where $Q^*(b, u)$ is defined as

$$Q^*(b, u) := C(b, u) + \gamma \sum_{o \in \mathcal{O}} P_o^{b,u} V^*(\mathcal{T}(b, o, u)),$$

with $P_o^{b,u} = P(o_{t+1} = o | b_t = b, u_t = u) = \sum_{s \in \mathbf{S}} b^s P(o_{t+1} = o | s_{t+1} = s, u_t = u)$ (where $P(o_{t+1} = o | s_{t+1} = s, u_t = u)$ is the observation probability), we can define an optimal defense policy by taking

$$\pi^*(b) = \operatorname*{argmin}_{u \in \mathcal{U}} Q^*(b, u).$$

2.3 Reinforcement learning algorithms

Two well-known solving algorithms for reinforcement learning problem are presented in this section. These are later used in the experimental results and comparisons detailed in Chapters 4 and 5 respectively. The definitions follow descriptions from the book *"Decision Making Under Uncertainty and Reinforcement Learning"* written by Dimitrakakis and Ortner (2018) if not stated otherwise.

2.3.1 Q-learning

One of the simplest, but also well-known, reinforcement algorithms is Q-learning. The algorithm is an *off-policy* reinforcement learning algorithm, where the goal is to learn the optimal behaviour by approximating a Q-function based on the immediate reward given by taking an action.

The algorithm (Algorithm 1 where μ is the environment, $\epsilon_{t_{tot}}$ is the exploration parameter, $\alpha_{t_{tot}}$ is the learning rate and $\boldsymbol{v}_{t_{tot}}$ is the value function at time $t_{tot} = (i-1) \cdot t_{max} + t$, i.e. the total time since the simulation began) easily be implemented using a table to store the Q-function. However, despite its simplicity, it appears to show good performance in practice (Dimitrakakis & Ortner, 2018).

Algorithm 1 Q-learning

Input: $\mu, \mathcal{S}, \mathcal{A}, \epsilon_{t_{tot}}, \alpha_{t_{tot}}, \gamma_{t_{tot}}$

Initialize $Q_0 \in \mathcal{V}$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
    Initialize $s_t \in \mathcal{S}$
    for $t = 1, 2, \ldots, t_{max}$ **do**
        $a_t \sim \pi^*_{\epsilon_{t_{tot}}}(a|s_t, Q_{t_{tot}})$
        $s_{t+1} \sim T_a(s_t, s')$
        $Q_{t_{tot}+1}(s_t, a_t) = (1 - \alpha_{t_{tot}})Q_{t_{tot}}(s_t, a_t) + \alpha_{t_{tot}}[R_{a_t}(s_t) + \gamma_{t_{tot}}\boldsymbol{v}_{t_{tot}}(s_{t+1})]$,
        where $\boldsymbol{v}_t(s) = \max_{a \in \mathcal{A}} Q_t(s, a)$
    end for
end for

2.3.2 Policy Gradient

In contrast to Q-learning, where the policy function is inferred from a learned Q-function, the policy function is inferred directly in Policy Gradient methods. Generally, policy-based methods find the optimal policy by optimizing for long term reward directly, employing gradient ascent on the expected utility to find a locally maximizing policy (Dimitrakakis & Ortner, 2018).

In practice there are several different approaches to policy gradient methods. A widely used one is Monte Carlo Policy Gradient, also called REINFORCE, which is defined in Algorithm 2. This method is *unbiased* but suffers from high variance (Silver, 2015b).

The policy gradient is defined as

$$\bigtriangledown_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\bigtriangledown_\theta \log \pi_\theta(s_t, a_t) Q^{\pi_\theta}(s_t, a_t)],$$

where the long-term value of $Q^{\pi_\theta}(s_t, a_t)$ replaces the instantaneous reward $r = R_a(s)$. The goal is to find the optimal θ for a stochastic policy π given each state $s \in S$ and action $a \in A$ pair. In the REINFORCE algorithm in Algorithm 2, v_t is used as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$ and α_t denotes the learning rate. The gradient for this particular algorithm is then defined as

$$\triangle\theta_t = \alpha_t \bigtriangledown_\theta \log \pi_\theta(s_t, a_t) v_t,$$

Algorithm 2 REINFORCE

Input: $\mu, \mathcal{S}, \mathcal{A}, \alpha$

Initialize $s_t \in \mathcal{S}$, set θ arbitrarily
for each episode $\langle s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T \rangle \sim \pi_\theta$ **do**
    for $t = 1, 2, \ldots, T$ **do**
        $\theta \leftarrow \theta + \alpha_t \bigtriangledown_\theta \log \pi_\theta(a_t, s_t) v_t$
    end for
end for

3

Environments

This chapter details the implementation of each environment. In addition, the assumptions and limitations are stated which applies to both of the environments. Furthermore, both environments are based on the theoretical definitions in Chapter 2.

The chapter is organized into four sections, where first the assumptions are stated and justified. Following, each environment is detailed in a separate section respectively, where the Inspect environment in Section 3.3, is based on the minimal environment in Section 3.2. Lastly, an analysis of the reward (cost) function is detailed in Section 3.4.

3.1 Assumptions

Several assumptions are imposed on the attacker to simplify the problem. All assumptions can be removed but requires changes to each solving algorithm and increases complexity. Each assumption and justification is detailed below where some of them are inherited from work by Miehling et al. (2015).

1. An attacker is assumed to not follow any specific intelligent behaviour, but instead propagate randomly throughout the network attempting to reach the critical attribute(s). It can be viewed as an adversary applying trial and error to succeed, where a more sophisticated strategy would add unwanted complexity to the model. Thus, with this assumption, we may model it as a probabilistic spreading process.

2. No false positives can occur, i.e. excluding the possibility that the defender observes that an attribute is enabled ($o_t^i = 1$) when actually disabled ($s_t^i = 0$). As the defender already acts under imperfect information, adding additional uncertainty is undesirable. Mathematically, this can be defined as $P(o_t^i = 1 | s_t^i = 0) = 0$.

3. All leaf attributes, $\mathcal{N}_{\mathrm{L}}$, are covered by a binary action. That is, for each attribute $i \in \mathcal{N}_{\mathrm{L}}$ there exists a countermeasure $u \in \mathcal{U}$ containing action u^m that disables i, i.e. $i \in \mathcal{W}_{u^m}$. This follows from the extreme case of using the countermeasure of shutting down the system completely, blocking all entry points to the system, i.e. the leaf nodes $\mathcal{N}_{\mathrm{L}}$.

4. If all binary actions $u = \{u^1, \ldots, u^M\}$ are deployed as a countermeasure the simulation ends. As we have assumed that all leaf attributes, $\mathcal{N}_{\mathrm{L}}$, are covered by a binary action deploying all countermeasures will result in a full system shut down. Thus, ending the simulation with the instantaneous cost of $C(u, s) = C(\{u^1, \ldots, u^M\}, s)$ where $s \in \mathbf{S}$.

5. The attack graph only contains **AND** attributes, i.e. $N_{\wedge} = \mathcal{N} \setminus \mathcal{N}_{\mathrm{L}}$ and $N_{\vee} = \varnothing$. This assumption is justified in the remaining assumptions.

6. The only feasible states are *monotone*, denoted by the set $\mathcal{S}$. A state $s = \{s^1, \ldots, s^N\} \in \mathcal{S} \subseteq \mathbf{S}$ is monotone, if for every attribute i that is enabled in an attack graph that solely is built up by **AND** attributes, all of i's predecessors $j \in \mathcal{D}_i$ are enabled, i.e. $s^i = s^j = 1$. In other words we make the assumption that attacker's behave with *monotonicity* and thereby eliminating cycles present in the attack graph. Conceptually speaking we assume that attackers will never give up previous capabilities gained while increasing its control over the network. The justification for this follows the arguments made by Poolsappasit et al. (2012), and by the fact that informally it makes logical sense from the goals of an attacker that they would not willingly give up anything gained. With this limitation, we restrict ourselves to the $|\mathcal{S}| = K \leq |\mathbf{S}| = 2^N$ *feasible* states instead of all 2^N states in space $\mathcal{S}$. Thus, the probability mass function need solely to concern K possible states and we can redefine the belief as

$$\Pi_t^s = P(s_t = s | H_t, s \in \mathcal{S}).$$

7. An adversary can enable attributes while countermeasure $u \in \mathcal{U}$ is applied, if the attributes and their predecessors are unaffected by u, i.e not in the *reach* of u. It can in turn be defined as the set of nodes that are disabled as a result of deploying a countermeasure action $u \in \mathcal{U}$

$$\mathcal{R}_u := \{i \in \mathcal{N} | i \in \mathcal{S}_j, j \in \mathcal{W}_u\},$$

where $\mathcal{S}_j$ is the set of successors of j. It follows from intuition that if an adversary is about to enable an attribute at time-step t, and that a countermeasure u is simultaneously deployed that affects another part of the system, the adversary should be able to continue and enable the attribute as it is unaffected.

An example is presented in Figure 3.1, to highlight this behaviour. Worth noting and illustrated in Figure 3.1, is that an adversary can exclusively enable attributes that are direct successors to itself and that has all of it predecessors enabled in a single time-step, following the *monotonicity* assumption.

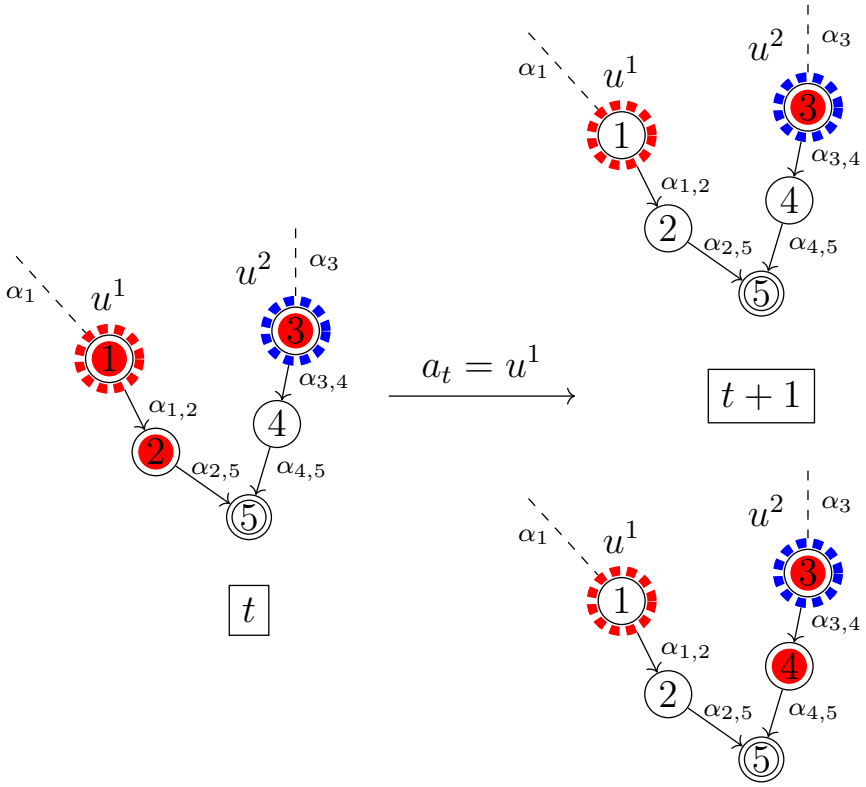


Figure 3.1: An example network, first at time-step t and then with the two potential states at time-step $t+1$, after performing the countermeasure action $a_t = u^1 \in \mathcal{U}$ at time-step t. All nodes are **AND** attributes, following our assumption, with the critical attribute $\mathcal{N}_{\mathrm{C}} = \{5\}$, the leaf attributes $\mathcal{N}_{\mathrm{L}} = \{1, 3\}$ and that there exists two binary actions $u^1, u^2 \in \mathcal{U}$. As can be observed, critical attributes are indicate by its double outer circles, leaf attributes by the dotted line with the attack probability as they are the entry-points for the attacker(s) into the network, and the colored outer strips the set of attributes W_{u^m} disabled by the binary action u^m where $m \in \{1, 2\}$. At time-step t, attribute $1, 2, 3 \in \{1\}$, i.e. the nodes with an inner red circle, are enabled. At time-step $t+1$, after deploying countermeasure u^1, attribute $1, 2 \in \mathcal{R}_{u^1}$ are disabled, i.e. unavailable for both authorized users and adversaries, as attribute 2 is the successor of the disabled attribute 1. However, attribute $3 \in \{1\}$ is still enabled as it is unaffected by u_1, and $4 \in \{1\}$ is enabled with probability $\alpha_{3,4}$.

3.2 Minimal environment

To understand bottlenecks and problem areas, as well as to have a model and experiments to use for comparisons to our later self-developed new model, a baseline had to be created. As the work done on defense policies for partially observable spreading processes on Bayesian attack graphs by Miehling et al. (2015) is the most relevant related to this thesis, we chose to use it as the foundation.

In their work, a toy numerical example is implemented which can be viewed in Figure 3.2. Being of an ideal size, we made the decision to use the network as the main framework in the implementations to come.

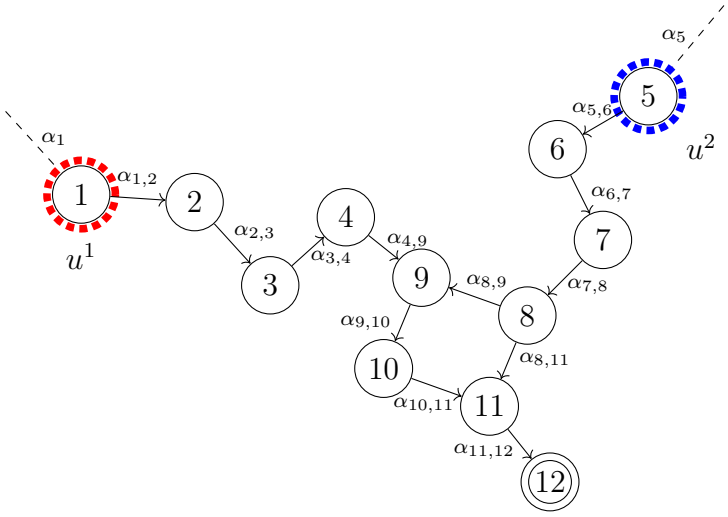


Figure 3.2: The sample network given by Miehling et al. (2015). All nodes are considered to be **AND** attributes and α to be the exploit probabilities. There exists two leaf attributes $\mathcal{N}_{\mathrm{L}} = \{1, 5\}$, one critical attribute $\mathcal{N}_{\mathrm{C}} = \{12\}$ and two binary actions $u^1, u^2 \in \mathcal{U}$. Thus the set of countermeasure actions are $\mathcal{U} = \{\varnothing, \{u^1\}, \{u^2\}, \{u^1, u^2\}\}$ where $\varnothing$ is the countermeasure action of doing nothing, i.e. applying no countermeasure.

The following sections define the different components of the 29-state/observation, 4-action POMDP defense problem for the sample network in Figure 3.2.

3.2.1 Attributes

The sample network is constituted of 12 attributes, where two leaf attributes exist $\mathcal{N}_{\mathrm{L}} = \{1, 5\}$ and one critical attribute $\mathcal{N}_{\mathrm{C}} = \{12\}$. To give the network a more realistic appearance, Miehling et al. interpret the attributes, i.e. the vulnerabilities of a computer system, as follows:

1. Vulnerability in WebDAV on machine 1

2. User access on machine 1
3. Heap corruption via SSH on machine 1
4. Root access on machine 1
5. Buffer overflow on machine 2
6. Root access on machine 2
7. Squid port scan on machine 2
8. Network topology leakage from machine 2
9. Buffer overflow on machine 3
10. Root access on machine 3
11. Buffer overflow on machine 4
12. Root access on machine 4

3.2.2 Actions

The defender is assumed to have access to the *two* following actions:

u^1: Block WebDAV service
u^2: Disconnect machine 2

With these two binary actions, we have a set of $2^2 = 4$ countermeasures, i.e. $\mathcal{U} = \{\varnothing, \{u^1\}, \{u^2\}, \{u^1, u^2\}\}$ where $\varnothing$ is the countermeasure action of doing nothing, i.e. applying no countermeasure.

3.2.3 Cost function

In the minimal example, the cost function is assumed to be additively separable, i.e $C(s, u) = C(s) + D(u)$ for all $s \in \mathcal{S}, u \in \mathcal{U}$ where $C(s)$ is the state cost and $D(u)$ the availability cost of a countermeasure. The state cost $C(s)$ is defined as

$$C(s) = \begin{cases} 1 & \text{if } s^{12} = 1 \text{ where } 12 \in \mathcal{N}_{\mathrm{C}} \\ 0 & \text{otherwise.} \end{cases}$$

The availability cost of a countermeasure $D(u)$ is in the sample network defined to be proportional to the number of attributes that the countermeasure action disables. In other terms, it can be defined as the countermeasure with largest reach $\mathcal{R}_u$ with $u \in \mathcal{U}$. Since $|\mathcal{R}_{u^1}| = |\mathcal{R}_{u^2}|$, $D(u^1)$ and $D(u^2)$ are set to be equal, i.e. $D(u^1) = D(u^2) = 1$. $D(\{u^1, u^2\}) = 5$, i.e. a higher cost since this reduces availability to zero by resetting all attributes in the graph.

3.2.4 State space

Without any assumptions on attribute types and attacker behaviour, the state space would be $|\mathcal{S}| = 2^{12} = 4096$. However, with the restriction of **AND** attributes and monotone states, the state space is greatly reduced to $|\mathcal{S}| = 29$, which is a reduction of over 140 times.

3.2.5 Observation space

The same reduction can be applied to the observation space as a result of the assumptions made. Not that the dimensionality of the observations can be reduced, as we still will observe any of the $|\mathcal{O}| = 2^{12}$ possible combinations of enabled attributes in the observation set $\mathcal{O}$, but rather how to interpret the observations. Since we assumed that there are no possibility of false positives, we know that a given observation is always a subset of the true underlying state $s \in \mathcal{S}$. Thus, the observation space can be reduced to $\hat{\mathcal{O}} \subseteq \mathcal{O}$ observations that exclusively give useful information by mapping a given observation $o \in \mathcal{O}$ to $\hat{o} \in \hat{\mathcal{O}}$.

3.2.6 Parameters

For the network sample problem, Miehling et al. set the parameters to the values defined in Table 3.1.

Network sample problem parameters	
Probabilities of detection	$\boldsymbol{\beta} = (0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.7, 0.6, 0.7, 0.85, 0.95)$
Attack probabilities	$\alpha_1 = 0.5$ and $\alpha_5 = 0.5$
Spread probabilities	$\alpha_{1,2} = 0.8$, $\alpha_{2,3} = 0.8$, $\alpha_{3,4} = 0.9$, $\alpha_{4,9} = 0.8$, $\alpha_{5,6} = 0.8$, $\alpha_{6,7} = 0.9$, $\alpha_{7,8} = 0.8$, $\alpha_{8,9} = 0.8$, $\alpha_{8,11} = 0.8$, $\alpha_{9,10} = 0.9$, $\alpha_{10,11} = 0.8$, $\alpha_{11,12} = 0.9$

Table 3.1: The parameters for the toy numerical example given by Miehling et al. (2015).

3.2.7 Time maximum

Also necessary, is to define the maximum time for an episode t_{max}, as it was not done by Miehling et al. (2015). We have already made the assumption that a simulation will end if all binary actions $\{u^1, u^2\}$ are deployed, but in order to compare results we need to define a maximum time for a simulation. A prerequisite to take into consideration is that the maximum time must be of a size such that all attributes $\mathcal{N}$ can be reached in the specified time frame. By intuition, there must be a possibility to visit these as the defender has to take the whole system into consideration when building its defensive strategies.

When running the Minimal environment with the parameters defined, we observe that the average time it takes for an adversary to enable all attributes is approximately $t_{avg} \sim 12$. Thus, a sufficient amount of time is added to define $t_{max} = 100$. The reason is that we want the choice of a bad defensive strategy to respectively show bad performance in the results and not be hidden due to a short ending of the simulation. If we then choose to solely perform the countermeasure of doing nothing at each time-step, i.e. $a_t = \varnothing$ for $t = 1, \ldots, t_{max}$, an average total cost of $100 \cdot C(s) - t_{avg} = 100 \cdot 1 - 12 = 88$ will be received for the whole episode where $s^{12} \in \{1\}$.

3.3 Inspect Environment

The *Inspect Environment* was created by taking the minimal environment and extending it with an additional action $u^3 \in \mathcal{U}$, *Inspect*. The new action allows the agent to inspect and retrieve the true state $s_t \in \mathcal{S}$ in parallel to the observation $o_t \in \hat{\mathcal{O}}$ at time step t. Performing this action has a small penalty $D(u^3)$. Basically, reducing the POMDP into a MDP for a single time-step. It can be viewed as the action of letting an expert examine a computer system, in a fashion where it is assumed that the expert possesses such knowledge that he or she always will discover potential intrusions into the system. Enabling the possibility to approximate the detection and the attack and spread probabilities, $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$, and use these to perform better informed actions.

Another alternative implementation of the Inspect action would be to solely retrieve the true state of attribute $i \in \mathcal{N}$ at time step t, i.e. $s_t^i \in \mathcal{S}_i$. However, as we were interested in implementing and evaluating algorithms that would empirically approximate the spread and attack probabilities α and use these, we made the decision to implement the Inspect action such that the true state of all attributes are retrieved. In addition, algorithms relying on retrieving s_t^i can be implemented in an environment where the inspection action corresponds to retrieving s_t.

The new action space with the addition of Inspect:

u^1: Block WebDAV service
u^2: Disconnect machine 2
u^3: Inspect

With the addition of the new action the set of countermeasure actions is now $\mathcal{U} = \{\varnothing, \{u^1\}, \{u^2\}, \{u^3\}, \{u^1, u^2\}\}$, where the Inspect action, u^3, cannot be combined with the other two actions u^1 and u^2. The reasoning behind this decision is that the inspect action distinguishes itself from the others as an information gathering action in contrast to an availability restricting action. The size of the action space is now $|\mathcal{U}| = 1 + 2^2 = 5$.

3.4 Cost function analysis

When the defense agent performs a countermeasure there is an associated cost attached to it, in the context of reinforcement learning this is what defines the cost function. Since the application of the research is security, and that the goal was to have an agent that performs actions in an environment that attempts to approximately mimic a real setup, an analysis on the actual real-world cost of scenarios was used to define the cost function for the new Inspect action u^3. The method of finding the cost was based on previous research and statistics, in order to find the real world money value of breaches, human intervention and system down time.

Gathering data from research papers on the cost of a multitude of attacks and availability issues in computer systems, we ended up with the data in Table 3.2. The table denote the cost in millions of dollar for data breaches and the response to them in the United States in 2017. The data verified that the relative cost of actions were realistically set by Miehling et al. (2015). Therefore, the same cost values were used with the exception of the Inspect action, which was inferred by picking a value that had a reasonable relative cost to the initial actions based on the security data.

Average Real-world cost of security issues	
Data breach	$3.62
Breach recovery	$1.07
Response team labor	$0.905

Table 3.2: Average cost of a data breach in millions of dollars for mid- to large-size companies in the United States in 2017 (Ponemon Institute LLC, 2017).

The cost of each countermeasure in both environments is defined in Table 3.3. As can be observed, performing nothing obviously does not have a cost associated with it. But we need to keep in mind that there is a penalty for letting an attacker take

over critical nodes in the graph. Meanwhile, the special Inspect action has a minimal cost associated with it, due to the time aspect of this countermeasure. The Inspect action will gather information about the true state of the environment, which in a real system would account to time spent querying for this information. Basically, there has to be a cost associated with it even if the countermeasure itself does not directly affect the attacker and its capabilities. At the same time, it is resources used and time for an attacker to unopposed progress within the system. Thus, in relation to doing nothing, it is more expensive as resources connected to the defence of the system is used.

The last countermeasures u^1 and u^2 denote actions directly impacting an attacker. But they also impact the availability of the system by forcing resources to be unavailable. Countermeasure u^2 might seem like an countermeasure that has severe impact, but in the model, they both impact the usage of the system equally, i.e. a user of the system loses as much connectivity.

Action	Cost
$\varnothing$	0
u^3: Inspect	0.2
u^2: Disconnect Machine	1
u^1: Block service	1
$\{u^1, u^2\}$	5

Table 3.3: The cost of each countermeasure in both environments.

In a scenario where the availability is modelled more precisely, the cost for each countermeasure that changes the availability of the system is dependent to the time of lowered accessibility. Currently, countermeasures affecting the availability of the system, e.g. u^1, only affects the availability for one time-step and the cost received is not time-dependent. A more realistic scenario would be that the simulation will continue after countermeasure is deployed, as implemented now, and that the cost then should accrue over the time of lowered availability. For example, the countermeasures u^1 and u^2 in this setting would have the costs

$$\begin{cases} D(u^1) = t_{u^1} \cdot k_{u^1} + 1 \\ D(u^2) = t_{u^2} \cdot k_{u^2} + 1 \end{cases}$$

where t_{u^n} is the time $\mathcal{R}_{u^n}$ is unavailable to authorized users due to the deployment of $u^n \in \mathcal{U}$, and the weight k_{u^n} acts as the availability cost over time for the specific countermeasure, i.e. also dependent on the reach $\mathcal{R}_{u^n}$. Likewise, the cost of high impact countermeasures such as the union of u^1 and u^2 could be defined with a weight that penalizes taking multiple binary actions extra

$$D(\{u^1, u^2\}) = k_{u^1,u^2}(D(u^1) + D(u^2)),$$

as well as giving a built in negative cost over time due to the previous definition where the countermeasure is penalized over time. Formally, we can generalize this into

$$D(\{u^1, u^2 \ldots, u^n\}) = k_{u^1,u^2\ldots,u^n} \sum_{i=1}^{n} D(u^i)$$

As one can see, there is some significant improvements that can be made to the cost function to make it more closely match something in an actual real system. A realization of adding a time cost for unavailability, is that there should probably be a way for the agent to reverse an countermeasure. In other words each countermeasure can be reset, so in the case of shutting down a machine, the agent can bring it back up and stop the cost adding up over each time-step. However, these suggested improvements are not implemented, but rather highlighted and specified as areas of future work.

3.4.1 Time of cost

In the implementation of the environments described in this thesis, the cost is given at each time-step of the simulation. In other words an action gives instantaneous cost to the agent. This allows algorithms such as Q-learning and to some degree Policy Gradient Methods to learn directly from the cost, which is required in the case of Q-learning. Policy Gradient methods could probably work with a non-instantaneous cost, but would exhibit a long training time.

Modelling it in such a way where the cost is given after a full simulation, allows for more accurate representation. But it complicates and increases the convergence time of most algorithms. This comes from the fact that the agent now has to correlate a series of observations with a single action. It is nonetheless an area of high interest, as it would take steps towards a more realistic model. Therefore, labeled as future work where a deeper analysis is conducted in Section 6.2.1.

3.5 OpenAI Gym

When reviewing previous research within the field and with our specific application in mind we found that no standardized comparison benchmark existed. To perform fair and hopefully allow for future performance comparisons within the research

community we decided to develop two OpenAI Gym environments, one for the Minimal environment and one for the Inspect environment. These were based on the environment definitions and used for all experimental results.

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms (Brockman et al., 2016). It contains a vast and growing collection of environments with a shared interface, allowing users to write general algorithms and run them across several different domains. More details on the specific implementation of the software library in Appendix A.

4

Algorithms

Solving POMDP:s is a computationally heavy task. Some efficient approximate solving algorithms exist but none has proven to scale to thousands of nodes. By using the addition of Inspect in the environment as explained in Section 3.3, we can use simpler algorithms that can perform well. This chapter details the implementation of each of the developed algorithms as well as implementation details for some standard algorithms used in the experimental comparisons in Chapter 5.

The first section details the actual optimal policy for each environment. The implementation for standard algorithms are then detailed and in the last two sections, Section 4.4 and 4.5, *two* new algorithms are introduced, *n-Myopic* and *n-Lookahead.*

4.1 Optimal policy

In the work done by Miehling et al. (2015) they define the optimal policy as:

> "The optimal policy is intuitive. It can be seen ... that optimal countermeasure is the one that disables the attributes that have a sufficiently high probability of being enabled..." (Miehling et al., 2015)

They do not declare the specific probability required, but if we reflect over the parameters defined for the Minimal environment, the probability to enable one of the leaf nodes $s^1, s^5 \in \mathcal{N}_{\mathrm{L}}$ are

$$P(s^1_{t+1} = 1 | s^1_t = 0) = P(s^5_{t+1} = 1 | s^5_t = 0) = \alpha_1 = \alpha_5 = 0.5.$$

After two time-steps the probability for them being enabled respectively is

$$P(s^1_{t+2} = 1 | s^1_t = 0) = P(s^5_{t+2} = 1 | s^5_t = 0) = 1 - \alpha_1 \cdot \alpha_1 = 1 - \alpha_5 \cdot \alpha_5 = 0.75$$

which already is a high probability. The result would be that the optimal policy would deploy binary actions heavily and in an early stage leading to a high cost. At the same time, the defender is not punished if an adversary enables non-critical attributes $i \in \mathcal{N} \setminus \mathcal{N}_{\mathrm{C}}$, or oppositely, penalized if enabled non-critical attributes are disabled. Hence, we chose to define optimality differently and put emphasis on protecting the critical attributes $\mathcal{N}_{\mathrm{C}}$ and not all attributes $\mathcal{N}$. If the case however is to protect all attributes, it can simply be done by redefining critical attributes $\mathcal{N}_{\mathrm{C}} \nsubseteq \mathcal{N}_{\mathrm{R}}$ and making all attributes critical, i.e. $\mathcal{N}_{\mathrm{C}} = \mathcal{N}$. It can be discussed if this is the correct stance to it, as an owner of a system does not want any adversary to have unauthorized access to any part of the system, and should be further looked into in future work.

The desired property of a policy is to maintain availability while preventing any adversary from enabling any critical attribute $i \in \mathcal{N}_{\mathrm{C}}$, and simultaneously minimizing the cost of doing so. Thus, all of these three properties need to be fulfilled for a policy to be optimal. To exemplify, in the Minimal environment defined in Section 3.2, the optimal policy is not to directly apply both countermeasures and thereby ending the episode. While ensuring that no critical attribute is enabled and minimizing the cost to 5 for an entire episode, it fails to maintain the availability of the system as it is online for 1 time-step of $t_{max} = 100$ possible time-steps. This reveals a contradiction, as the goal defined in Section 2.2.4 is to minimize the infinite-horizon discounted expected cost V^*, not to maintain the availability of the system. However, both can be accomplished by appropriate selection of the discount factor γ, as it determines how much immediate and distant costs affect the agent. I.e., if the discount factor is set to $\gamma = 1$ it will take all future costs into consideration, and thus terminate the simulation directly by applying both countermeasures as it will minimize the expected cost. On the contrary, if gamma is set to $\gamma = 0$, it will consider the action that minimizes the expected immediate cost, which in general will be to do nothing. Thus, by tuning we can chose an appropriate γ that will accomplish both minimization of the expected cost and availability of the system. It is nonetheless further discussed in Section 6.2, as tuning the discount factor is not the singular path to achieve the defined optimality.

A general optimal policy can instead be described as the one seen in Algorithm 3, where we provide the environment μ, the set of states $\mathcal{S}$ and the set of countermeasures $\mathcal{U}$ as input. If all the direct predecessors $\bar{\mathcal{D}}_c$ to a subset of the critical attributes $c \subseteq \mathcal{N}_{\mathrm{C}}$ is enabled in the current state $s_t \in \mathcal{S}$, we deploy the countermeasure action $u \in \mathcal{U}$. It should disable at least one of the direct predecessors to each critical attribute in c and maximize

$$\frac{\bar{\tau}_u^c}{D(u)},$$

where $\bar{\tau}_u^c$ is the average time it takes from being in a state where all the direct predecessors $\bar{\mathcal{D}}_c$ to $c \subseteq \mathcal{N}_{\mathrm{C}}$ are enabled until being in a state where all the direct

predecessors $\bar{\mathcal{D}}_{c'}$ to $c' \subseteq \mathcal{N}_{\mathrm{C}}$ are enabled, by deploying the countermeasure $u \in \mathcal{U}$. Thus, we want to maximize the ratio of the average time $\bar{\tau}_u^c$ to the cost of deploying u. Otherwise, we deploy the countermeasure $\varnothing$ of doing nothing.

In Figure 4.1, we can observe an example on how the general optimal policy in Algorithm 3 is applied on a network. In the network, the critical attributes are $\mathcal{N}_{\mathrm{C}} = \{5, 6\}$, the leaf attributes are $\mathcal{N}_{\mathrm{L}} = \{1, 2\}$, the nodes with an inner red circle are enabled and there exist two binary actions $u^1, u^2 \in \mathcal{U}$. As can be observed, in time-step t, the network is in state $s_1 \in \mathcal{S}$, where the direct predecessors $\bar{\mathcal{D}}_5 = \{1, 3\}$ are enabled to the subset of critical attributes $\{5\} \subseteq \mathcal{N}_{\mathrm{C}}$. Thus, the defender should deploy a countermeasure action $u \in \mathcal{U}$, that should disable at least on of the attributes in $\bar{\mathcal{D}}_5 = \{1, 3\}$, and maximize $\bar{\tau}_u^5/D(u)$. If we further assume that the cost of u^1 and u^2 are equal, i.e. $D(u^1) = D(u^2)$, it all depends on the average time $\bar{\tau}_u^5$ to reach a new state where all the direct predecessors $\bar{\mathcal{D}}_{c'}$ to $c' \subseteq \mathcal{N}_{\mathrm{C}}$ are enabled. For this, there exists three different scenarios, i.e. three subsets c'. The direct predecessors $\bar{\mathcal{D}}_{c'}$ can respectively be found enabled in s_2, s_3 and s_4 in Figure 4.1. Thus, a countermeasure u should be chosen such that one of these states are reached as late as possible, i.e. maximizing $\bar{\tau}_u^5$. Noteworthy, is that s_2 and s_4 can be reached by deploying either u^1 or u^2, but s_3 only through u^1. Furthermore, it may seem natural to deploy u^2 as it disables more attributes than u^1, i.e. have a larger reach $\mathcal{R}_{u^1} < \mathcal{R}_{u^2}$, but that may not be the optimal choice. If $\alpha_1 \ll \alpha_2, \alpha_{2,3}$, then u^1 probably is the better choice as the expected time to enable attribute 1 will be larger than to enable attributes 2 and 3.

The optimal policy requires to have full knowledge of the state to be aware if the current state has all the direct predecessors to a subset of critical attributes enabled. Thereby the policy is not applicable to our POMDP problem at hand, but to a MDP version of it. However, its performance is optimal.

Algorithm 3 General optimal policy for MDP environments

Input: $\mu, \mathcal{S}, \mathcal{U}$

for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $s_t = \mathbf{0}$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    if $\exists\, c \subseteq \mathcal{N}_{\mathrm{C}}$ s.t. $\forall\, j \in c:\ s_t^{\bar{\mathcal{D}}_j} \in \{1\}$ **then**
      $a_t = \underset{u \in \mathcal{U}}{\operatorname{argmax}} \frac{\bar{\tau}_u^c}{D(u)}$ s.t. $\forall\, j \in c:\ s_t^{\bar{\mathcal{D}}_j} \cap W_u \neq \emptyset$
    else
      $a_t = \varnothing$
    end if
    $s_{t+1} \sim T_{a_t}(s_t, s_{t+1})$
  end for
end for

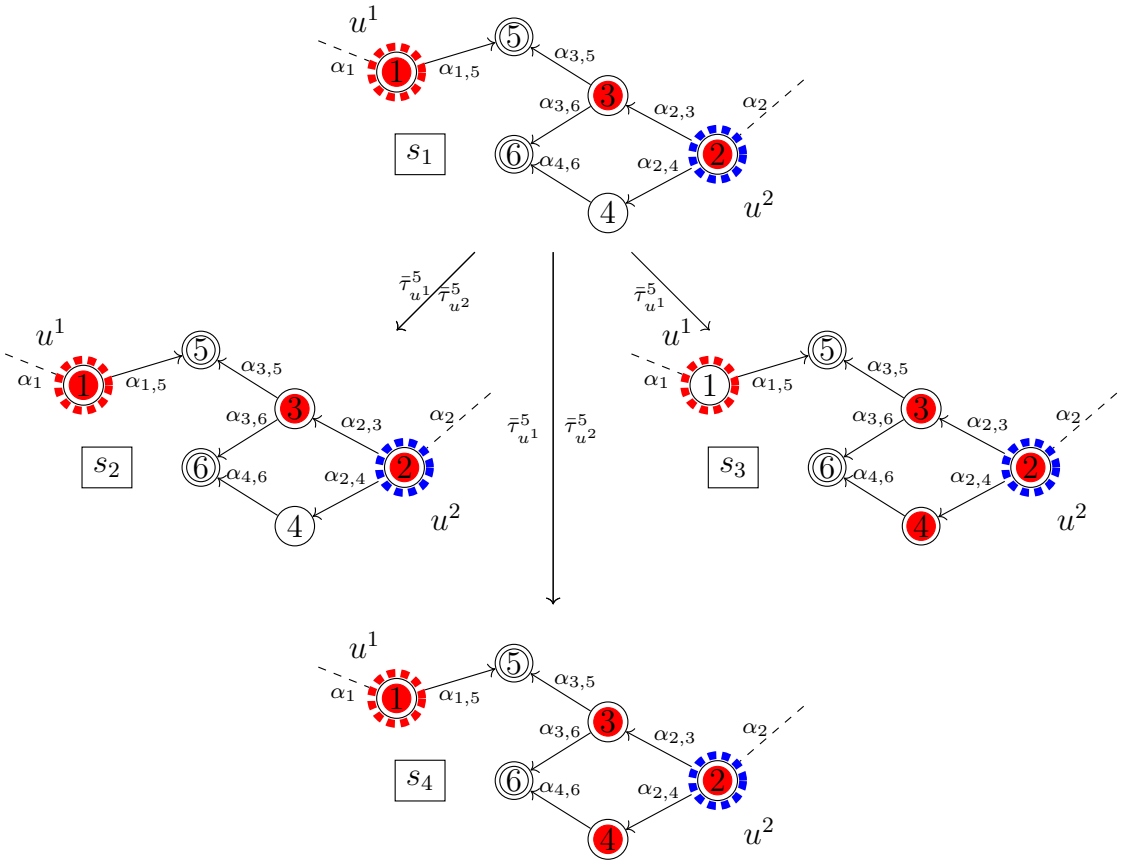


Figure 4.1: An example network on which the general optimal policy in Algorithm 3 is applied upon, where the critical attributes are $\mathcal{N}_\mathrm{C} = \{5, 6\}$, the leaf attributes are $\mathcal{N}_\mathrm{L} = \{1, 2\}$, the nodes with an inner red circle are enabled and there exist two binary actions $u^1, u^2 \in \mathcal{U}$. We can observe the three possible states $s_2, s_3, s_4 \in \mathcal{S}$ that on average is reached after $\bar{\tau}^5_u$ time-steps, depending on the countermeasure u^1 or u^2 deployed when all the direct predecessors $\bar{\mathcal{D}}_5 = \{1, 3\}$ are enabled to the subset of critical attributes $\{5\} \subseteq \mathcal{N}_\mathrm{C}$ in state $s_1 \in \mathcal{S}$. The optimal countermeasure $u \in \mathcal{U}$ thus depends on which one that maximizes $\bar{\tau}^5_u / D(u)$.

4.1.1 Minimal and Inspect environment

As the Minimal and Inspect environment are equivalent with the exception that the Inspect environment has an additional countermeasure, i.e. the Inspect action, they share the same optimal policy for the MDP version of the problem. It can be viewed in Algorithm 4, where we provide the environment μ, the set of states $\mathcal{S}$, and the set of countermeasures $\mathcal{U}$ as input. It solely performs an action, as described earlier in the general optimal policy in Algorithm 3, when all the direct predecessors, $\bar{\mathcal{D}}_{12} = \{11\}$, to a critical attribute, $12 \in \mathcal{N}_\mathrm{C}$, are enabled. The ratio $\bar{\tau}^{12}_u / D(u)$, is equivalent for u_1 and u_2. It follows from the fact that the size of the reach for the two available actions u_1 and u_2 are equivalent, $|\mathcal{R}_{u^1}| = |\mathcal{R}_{u^2}|$, that the joint probability for the two reaches are equal, $\alpha_1 \cdot \alpha_{1,2} \cdot \ldots \cdot \alpha_{11,12} = \alpha_5 \cdot \alpha_{5,6} \cdot \ldots \cdot \alpha_{11,12}$, and that they share the same cost $D(u^1) = D(u^2) = 1$, as well as they both disable

the enabled direct predecessor 11 for the critical attribute 12. Thus, we randomly chose one of the actions with probability $P(a_t = u_1) = P(a_t = u_2) = 0.5$. As the policy maintains availability, by not closing down the system by applying the countermeasure $\{u^1, u^2\}$, while preventing any adversary from enabling the critical attribute 12 and simultaneously minimizing the cost, it is optimal.

Algorithm 4 Optimal policy for the MDP Minimal and Inspect environment

Input: $\mu, \mathcal{S}, \mathcal{U}$

for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $s_t \in \mathbf{0}$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    if $s_t^{11} = 1$ **then**
      $a_t \sim \begin{cases} u^1 & \text{w.p. } 0.5 \\ u^2 & \text{w.p. } 0.5 \end{cases}$
    else
      $a_t = \varnothing$
    end if
    $s_{t+1} \sim T_{a_t}(s_t, s_{t+1})$
  end for
end for

4.2 Q-learning

The implementation of Q-learning on the Minimal environment can be seen in Algorithm 5, where we need to provide the environment μ, the set of states $\mathcal{S}$, the set of countermeasures $\mathcal{U}$, the learning rate α, the discount factor γ and the exploration factor δ as input. Noteworthy, is that $U(0, 1)$ defines a random value from the standard uniform distribution with minimum 0 and maximum 1.

Algorithm 5 Defender Q-learning

Input: $\mu, \mathcal{S}, \mathcal{U}, \alpha, \gamma, \delta$

Initialize $Q_{t_{tot}} = \mathbf{0}$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $o_t = \mathbf{0}$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    $a_t \sim \text{ChooseAction}(o_t, Q_{t_{tot}}, i, \delta)$
    $o_{t+1} \sim O_{a_t}(s_{t+1})$
    $Q_{t_{tot}+1}(o_t, a_t) = (1-\alpha)Q_{t_{tot}}(o_t, a_t) + \alpha[R_{a_t}(s_t) + \gamma \boldsymbol{v}_{t_{tot}}(o_{t+1})]$,
    where $\boldsymbol{v}_t(o) = \max_{u \in \mathcal{U}} Q_t(o, u)$
  end for
end for

function ChooseAction$(o_t, Q_{t_{tot}}, i, \delta)$
  $\epsilon = 1 - \delta \cdot i$
  if $U(0,1) < \epsilon$ **then**
    return A random countermeasure $u \in \mathcal{U}$
  else
    return $\operatorname{argmax}_{u \in \mathcal{U}} Q_{t_{tot}}(o_t, u)$
  end if
end function

4.2.1 Windowed Q-learning

To improve the results of Q-learning, an additional version was implemented on the Minimal environment, named *windowed* Q-learning. The workings of it is to keep a *window* denoted w, a buffer, of the last n_{window} observations, actions and costs. Every time an action is executed the buffer is filled up, and then the buffer is used to do multiple updates of the Q-function in each time-step. Details of this algorithm can be seen in Algorithm 6, where we provide the environment μ, the set of states $\mathcal{S}$, the set of countermeasures $\mathcal{U}$, the learning rate α, the discount factor γ, the exploration factor δ and the size of the window n_{window} as input. The work showed little impact on the result compared to the simpler Q-learning implementation. Thus, it was omitted in the experimental results.

Algorithm 6 Defender Windowed Q-learning

Input: $\mu, \mathcal{S}, \mathcal{U}, \alpha, \gamma, \delta, n_{window}$

Initialize $Q_{t_{tot}} = \mathbf{0}$, $w = empty\ queue$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $o_t = \mathbf{0}$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    $a_t \sim$ ChooseAction$(o_t, Q_{t_{tot}}, i, \delta)$
    $o_{t+1} \sim O_{a_t}(s_{t+1})$
    if $|w| \geq n_{window}$ **then**
      $w.pop()$
    end if
    $w.push(\{o_{t+1}, R_{a_t}(s_{t+1}), a_t\})$
    for $o', R_{a'}(s'), a' \in w$ **do**
      $Q_{t_{tot}+1}(o', a') = (1-\alpha)Q_{t_{tot}}(o', a') + \alpha[R_{a'}(s') + \gamma \boldsymbol{v}_{t_{tot}}(o')]$,
      where $\boldsymbol{v}_t(o) = \max_{u \in \mathcal{U}} Q_t(o, u)$
    end for
  end for
end for

---

function ChooseAction$(o_t, Q_{t_{tot}}, i, \delta)$
  $\epsilon = 1 - \delta \cdot i$
  if $U(0,1) < \epsilon$ **then**
    return A random countermeasure $u \in \mathcal{U}$
  else
    return $\operatorname{argmax}_{u \in \mathcal{U}} Q_{t_{tot}}(o_t, u)$
  end if
end function

4.3 Policy Gradient

In the Minimal environment we use an adaptation of *Monte Carlo Policy Gradient* using a single hidden layer neural network and *Softmax*

$$softmax(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)},$$

as policy function, where z is some vector for $i = 1 \ldots j$. The chosen policy function is well suited for a discrete action space, as in the environments we are trying to solve. An overview of the network configuration can be viewed in Figure 4.2. For each batch b the network is sequentially fed all observations $o \in \hat{\mathcal{O}}$ in that batch, and the weights are updated using the corresponding countermeasure actions $u \in \mathcal{U}$ together with the costs $C(s, u)$, where $s \in \mathcal{S}$.

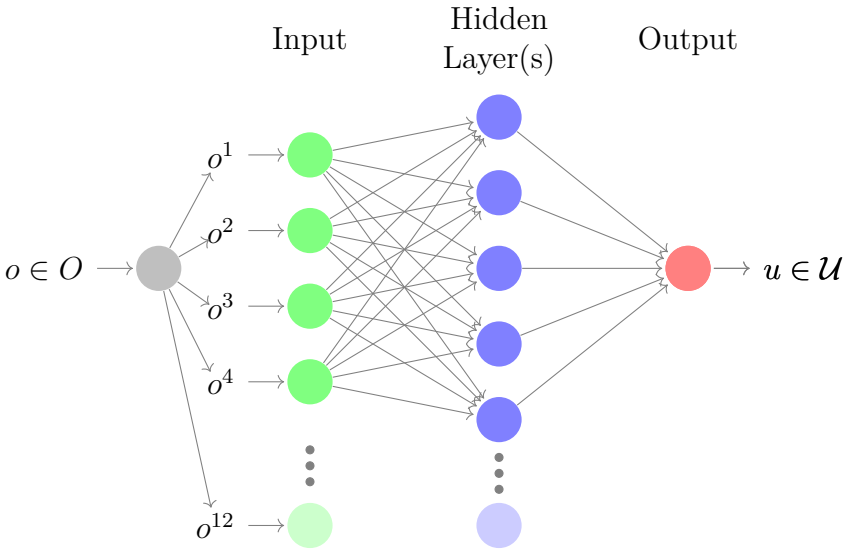


Figure 4.2: The fully connected neural network configuration of the adapted Policy Gradient method. The input layer takes a observation $o \in \hat{\mathcal{O}}$ and feeds that through, where the output layer outputs a corresponding countermeasure $u \in \mathcal{U}$, which are sampled using a multinomial distribution. The updates are done through gradient descent, with a softmax policy function.

At the core of the algorithm we perform batch updates of the network in size of $n_{batches}$. Each of these batch updates a full simulation is run and all observations, actions and costs are collected. The costs are normalized, and then all three of these collected sets are used to run and update the neural network. For each simulation step, the current best action sampled from the network is performed, and then we gradually improve. Given this fact, the current method of evaluating could perform decent on a configuration of the environment where costs are given at the end of a full simulation rather than at each action. With this current configuration the algorithm should intuitively converge much faster on an optimal policy.

The algorithm, as a whole, can be examined in Algorithm 7, where we provide the environment μ, the set of states $\mathcal{S}$, the set of countermeasures $\mathcal{U}$, a small constant ϵ, and the batch size $n_{batches}$ as input. In the algorithm the neural network is denoted by ξ.

Algorithm 7 Defender Policy Gradient

Input: $\mu, \mathcal{S}, \mathcal{U}, \epsilon, n_{batches}, \xi$

Initialize $s_t \in \mathcal{S}$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
  $\nu = \emptyset, O = \emptyset, A = \emptyset$
  for $b = 1, 2, \ldots, n_{batches}$ **do**
    for $t = 1, 2, \ldots, t_{max}$ **do**
      $O \cup o_t$
      $a_t \sim$ ChooseAction(o_t, ξ)
      $A \cup a_t$
      $\nu \cup r_t$
    end for
  end for
  $Z \sim \sum \frac{\nu - \frac{1}{|\nu|} \sum \nu}{\sigma_\nu + \epsilon}$
  Update(Z, A, O, ξ)
end for

---

function ChooseAction(o_t, ξ)
  return $u \in \mathcal{U} \sim$ feeding o_t into ξ
end function

---

function Update(Z, A, O, ξ)
  Update ξ using Z, A and O given our loss function
end function

4.4 *n*-Myopic

The n-Myopic solving algorithm works on the Inspect environment by creating an empirical matrix, E, through the usage of the inspect countermeasure, i.e. u_3, to create an informed view of the *real* state of the environment. By inspection, the agent will retrieve the true state $s_t \in \mathcal{S}$ as well as the observation $o_t \in \hat{\mathcal{O}}$ at time-step t. Thus, we can create the so called empirical matrix E, that contains an approximation of the probability of being in state $s \in \mathcal{S}$ when receiving observation $o \in \hat{\mathcal{O}}$, $P(s_t = s | o_t = o)$.

The empirical matrix is built by defining the number of episodes ϵ to utilize for tuning the matrix. Then during each episode, we perform the countermeasure inspection action u_3 for each time-step t, and update the empirical matrix according to the received state and observation. However, if all the direct predecessors $\bar{\mathcal{D}}_c$ to a subset of the critical attributes $c \subseteq \mathcal{N}_\mathrm{C}$ are enabled, we perform a random countermeasure $u \in \mathcal{U}$, excluding the countermeasures of doing nothing, $\varnothing$, inspecting, u_3, or shutting down the system entirely, $\{u_1, u_2\}$. We then measure the time τ_u^c it takes after performing the randomly chosen action u when the direct predecessors

$\bar{\mathcal{D}}_c$ to a subset $c \subseteq \mathcal{N}_\mathrm{C}$ are enabled, for an adversary to yet again enable the direct predecessors $\bar{\mathcal{D}}_{c'}$ to a subset $c' \subseteq \mathcal{N}_\mathrm{C}$ of the critical attributes, and store it in the time matrix, L. Thus, after performing ϵ episodes, we can determine an optimal countermeasure to deploy for each possible subset of $c \subseteq \mathcal{N}_\mathrm{C}$, by comparing the ratio of the average time $\bar{\tau}_u^c$ to reach a new state where all the direct predecessors $\bar{\mathcal{D}}_{c'}$ to $c' \subseteq \mathcal{N}_\mathrm{C}$ are enabled after performing the countermeasure u to the cost of deploying u. Hence, the length t_{max} of an episode affects the approximation quality of the tuning, as larger t_{max} implies that more information is gathered by the Inspect action in contrast of a smaller t_{max}.

In the episodes following after ϵ, we check for each observation received what the probability is for the adversary to be in one of the n-predecessors n_{pred} from the direct predecessors $\bar{\mathcal{D}}_c$ to a subset $c \subseteq \mathcal{N}_\mathrm{C}$ of the critical attributes defined as $Y^c_{n_{pred}}$. If the maximum probability is to be in one of these n-predecessors, we perform the optimal countermeasure found during the tuning of the empirical matrix for the specific subset $c \subseteq \mathcal{N}_\mathrm{C}$ of the critical attributes. It can be viewed as we make the assumption that all attack and spread probabilities α_i and α_{jk}, where $i \in \mathcal{N}_\mathrm{L}$, $j \in \mathcal{N}$ and $k \in \mathcal{N} \setminus \mathcal{N}_\mathrm{L}$, are equal to 1. I.e., that adversaries will enable all available attributes at each time-step with probability 1. Thus, by setting n_{pred} we determine the time buffer that we want to use, as if the adversaries would have absolute knowledge of the system. Additionally, if an observation $o_t \in \hat{\mathcal{O}}$ is received at time-step t containing enabled critical attributes $\exists o_t^c \in \{1\}$ we deploy the optimal countermeasure for the specific subset $c \subseteq \mathcal{N}_\mathrm{C}$ found during the tuning. It follows from our assumption made previously; that no false positives can occur, thus we can be certain that the critical attributes in c are enabled. The state that we now are in is however different from the one used in the tuning, as we deployed countermeasures when all the direct predecessors to a subset of critical attributes were enabled, but as it were for the direct predecessors, the optimal countermeasure learned from tuning should still be the optimal with a high certainty.

In Algorithm 8, we can see the Myopic solving algorithm, where the input environment μ, the set of states $\mathcal{S}$, the set of countermeasures $\mathcal{U}$, the tuning parameter ϵ and the number of predecessors n_{pred} is provided as input. For a more high level description, n-Myopic can be found represented as a flowchart in Figure 4.3.

Algorithm 8 n-Myopic

Input: $\mu, \mathcal{S}, \mathcal{U}, \epsilon, n_{pred}$

Initialize $E_t = \mathbf{0}$, $A_{optimal} = \mathbf{0}$, $L_t = \emptyset$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $s_t = \mathbf{0}$, $o_t = \mathbf{0}$, $\tau_u^c = 0$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    $a_t \sim$ ChooseAction$(s_t, o_t, A_{optimal}, E_t, i, \epsilon, n_{pred})$
    $o_{t+1} \sim O_{a_t}(s_{t+1})$
    if $a_t = u_3$ **then**
      $s_{t+1} \sim T_{a_t}(s_t, s_{t+1})$
      $E_{t+1}(o_{t+1}, s_{t+1}) = E_t(o_{t+1}, s_{t+1}) + 1$
      $\tau_u^c = \tau_u^c + 1$
    else if $a_t \in \mathcal{U} \backslash \{\varnothing, u_3\}$ **and** $i < \epsilon$ **then**
      $L_{t+1}(a_u^c, c) = L_t(a_u^c, c) \cup \tau_u^c$
      $\tau_u^c = 0$
    else if $i = \epsilon$ **and** $t = 1$ **then**
      $A_{optimal}(c) = \underset{u \in \mathcal{U}}{\operatorname{argmax}} \frac{\frac{1}{|L_t(u,c)|} \sum_{x \in L_t(u,c)} x}{D(u)}, \quad \forall c \subseteq \mathcal{N}_{\mathrm{C}}$
    end if
  end for
end for

---

function ChooseAction$(s_t, o_t, A_{optimal}, E_t, i, \epsilon, n_{pred})$
  if $i < \epsilon$ **then**
    if $\exists\, c$ s.t. $s_t^{\bar{\mathcal{D}}_c} \in \{1\}$ **then**
      return A random countermeasure $a \in \mathcal{U} \backslash \{\varnothing, u_3, \{u_1, u_2\}\}$
    else
      return u_3
    end if
  else
    if $\exists\, c$ s.t. $\left[\underset{s \in \mathcal{S}}{\operatorname{argmax}}\, E(o_t, s)\right]^{Y_{n_{pred}}^c} \notin \{0\}$ **or** $\exists\, c$ s.t. $o_t^c \in \{1\}$ **then**
      return $A_{optimal}(c)$
    else
      return $\varnothing$
    end if
  end if
end function

4.5 *n*-Lookahead

The n-Lookahead algorithm works on the Inspect environment in a similar fashion to the n-Myopic algorithm in Section 4.4. In addition to the previously defined

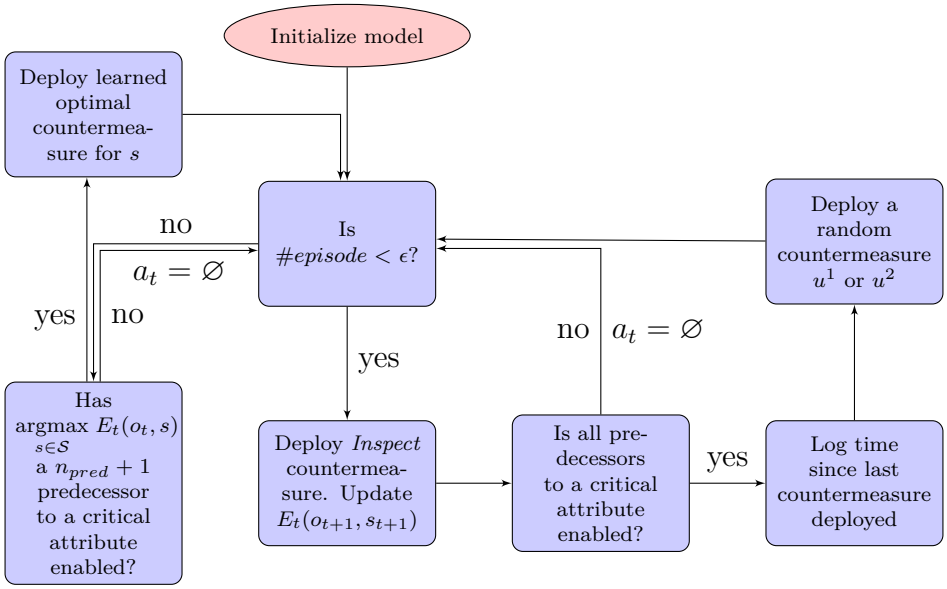


Figure 4.3: An overview of the n-Myopic algorithm, which can be found with all its details in Algorithm 8. In the flowchart $a \in \mathcal{A}$, $s \in \mathcal{S}$, $o \in \hat{\mathcal{O}}$ where E is the empirical matrix, ϵ is the number of episodes used for tuning, $\#episode$ is the current episode and n_{pred} is the number of predecessors from a direct predecessor to a critical attribute that we monitor to see if any has been enabled.

empirical matrix, E, and the time matrix, L, a transition matrix Υ is created, denoting the transition probabilities $P(s_{t+1} = s' | s_t = s, a_t = a)$. As before, these are built by defining the number of episodes ϵ to utilize for tuning by deploying the Inspect action u_3. The transition matrix is built by deploying the inspect action in two consequent time-steps, i.e. t and $t+1$, and observing the states s_t and s_{t+1} to approximate the transition probabilities. Using the learned state of the environment the probability of the next state $s_{t+1} \in \mathcal{S}$ is a state $s' \in \mathcal{S}$ where all the direct predecessors $\bar{\mathcal{D}}_c$ to a subset of the critical attributes $c \subseteq \mathcal{N}_\mathrm{C}$, given the current observation $o_t \in \hat{\mathcal{O}}$, is calculated by using the empirical and the transition matrix, i.e.

$$P(s_{t+1} = s' | o_t = o) \sim \sum_{s \in \mathcal{S}} E(o, s) \Upsilon(s, s').$$

With the calculated transition matrix, we can chose freely in how many time-steps n_{future} into the future that we want compute the probability to be in a state $s' \in \mathcal{S}$ with all direct predecessors to critical attributes enabled, i.e.

$$P(s_{t+n_{future}} = s' | o_t = o) \sim \sum_{s_0 \in \mathcal{S}} \sum_{s_1 \in \mathcal{S}} \cdots \sum_{s_{n_{future}-1} \in \mathcal{S}} E(o, s_0) \Upsilon(s_0, s_1) \Upsilon(s_1, s_2) \ldots \Upsilon(s_{n_{future}-1}, s').$$

As we already have the possibility to compute the probability that the current state

is $s' \in \mathcal{S}$, through the usage of the empirical matrix

$$P(s_t = s' | o_t = o) \sim E(o, s'),$$

we may calculate the probability that we have visited state s' in n_{future} time-steps as

$$\begin{aligned} P(s' \in \{s_t, \ldots, s_{t+n_{future}}\} | o_t = o) &= 1 - P(s' \notin \{s_t, \ldots, s_{t+n_{future}}\} | o_t = o) \\ &= 1 - \prod_{n=0}^{n_{future}} P(s_{t+n} \neq s' | o_t = o) \\ &= 1 - \prod_{n=0}^{n_{future}} (1 - P(s_{t+n} = s' | o_t = o)) \end{aligned}$$

The optimal countermeasure for each subset of critical attributes are calculated in similar fashion as in Section 4.4. When ϵ episodes of tuning has passed the received observations are used to approximate the probability of an adversary to be in a state where all predecessors to a subset of critical attributes are enabled in n_{future} time-steps. If the probability is over a specified threshold λ to be in one of these states, the optimal countermeasure is performed, found during the tuning. Following the no false positives assumption discussed in the previous n-Myopic Section 4.4, if an observation $o_t \in \hat{\mathcal{O}}$ is received at time-step t, containing enabled critical attributes $\exists\, o_t^c \in \{1\}$, the optimal countermeasure is deployed for the specific subset $c \subseteq \mathcal{N}_{\mathrm{C}}$ found during the tuning as well.

The n-Lookahead algorithm is denoted in Algorithm 9, where the environment μ, the set of states $\mathcal{S}$, the set of countermeasures $\mathcal{U}$, the tuning parameter ϵ and the number of time-steps n_{future} are provided as input. To obtain a more comprehensive view, a flowchart representation of n-Lookahead is provided in figure 4.4.

Algorithm 9 n-Lookahead

Input: $\mu, \mathcal{S}, \mathcal{U}, \epsilon, n_{future}, \lambda$

Initialize $E_t = \mathbf{0}$, $A_{optimal} = \mathbf{0}$, $\Upsilon_t = \mathbf{0}$, $L_t = \emptyset$
for $i = 1, 2, \ldots, n_{episodes}$ **do**
  Initialize $s_t = \mathbf{0}$, $o_t = \mathbf{0}$, $\tau_u^c = 0$
  for $t = 1, 2, \ldots, t_{max}$ **do**
    $a_t \sim$ ChooseAction$(s_t, o_t, A_{optimal}, \Upsilon_t, i, \epsilon, n_{future}, \lambda)$
    $o_{t+1} \sim O_{a_t}(s_{t+1})$
    if $a_t = u_3$ **then**
      $s_{t+1} \sim T_{a_t}(s_t, s_{t+1})$
      $E_{t+1}(o_{t+1}, s_{t+1}) = E_t(o_{t+1}, s_{t+1}) + 1$
      $\Upsilon_{t+1}(s_t, s_{t+1}) = \Upsilon_t(s_t, s_{t+1}) + 1$
      $\tau_u^c = \tau_u^c + 1$
    else if $a_t \in \mathcal{U} \backslash \{\varnothing, u_3\}$ **and** $i < \epsilon$ **then**
      $L_{t+1}(a_u^c, c) = L_t(a_u^c, c) \cup \tau_u^c$
      $\tau_u^c = 0$
    else if $i = \epsilon$ **and** $t = 1$ **then**
      $A_{optimal}(c) = \underset{u \in \mathcal{U}}{\operatorname{argmax}} \frac{\frac{1}{|L_t(u,c)|} \sum_{x \in L_t(u,c)} x}{D(u)}, \quad \forall c \subseteq \mathcal{N}_{\mathrm{C}}$
    end if
  end for
end for

---

function ChooseAction$(s_t, o_t, A_{optimal}, E_t, i, \epsilon, n_{future}, \lambda)$
  if $i < \epsilon$ **then**
    if $\exists\, c$ s.t. $s_t^{\bar{\mathcal{D}}_c} \in \{1\}$ **then**
      return A random countermeasure $a \in \mathcal{U} \backslash \{\varnothing, u_3, \{u_1, u_2\}\}$
    else
      return u_3
    end if
  else
    if $\exists\, c, s'$ s.t. $P(s' \in \{s_t, \ldots, s_{t+n_{future}}\} | o_t = o) > \lambda$ **and** $s'^{\bar{\mathcal{D}}_c} \in \{1\}$
    or $\exists\, c$ s.t. $o_t^c \in \{1\}$ **then**
      return $A_{optimal}(c)$
    else
      return $\varnothing$
    end if
  end if
end function

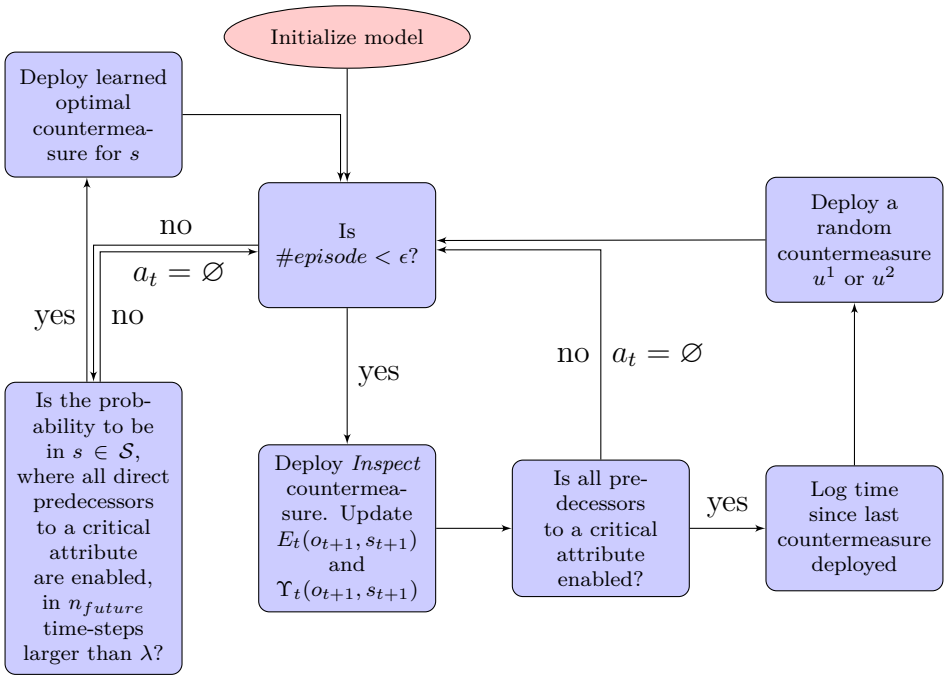


Figure 4.4: An overview of the n-Lookahead algorithm, which can be found with all its details in Algorithm 9. In the flowchart $a \in \mathcal{A}$, $s \in \mathcal{S}$, $o \in \hat{\mathcal{O}}$ where E is the empirical matrix, Υ is the transition matrix, ϵ is the number of episodes used for tuning, $\#episode$ is the current episode and λ is the threshold used to determine if the probability for an adversary to be in a state s, where all direct predecessors to a critical attribute are enabled, in n_{future} time-steps is large enough.

5

Results

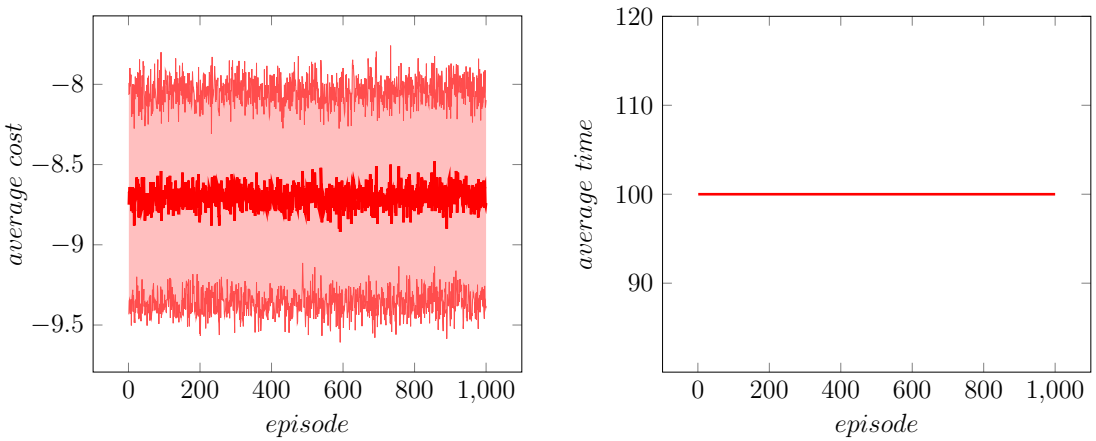
In this chapter each of the experimental simulation results of each algorithm, detailed in previous chapters, are presented. Each algorithm is dedicated its own section and then lastly a section that compares the results between each algorithm is presented. A deeper discussion of the results can be viewed in Section 6.2.

5.1 Performance evaluation

To evaluate and compare the performance of the applied and developed algorithms, we followed a consistent path. First, if necessary, by performing hyper-parameter optimization through *grid search*, also know as *parameter sweep*. I.e. a simple exhaustive search on a manually specified subset of the hyper-parameter space of the algorithm. It is followed by running the algorithm for a predefined number of 1000 episodes, where an average cost is computed over 100 simulations for each episode. An interesting benchmark is obviously the average episodic cost. It is however not the sole interesting metric, as the average episodic length has key importance and the convergence and stability of the algorithm have high relevance. Noteworthy, is that for both environments the cost is instantaneous which is further discussed in Section 6.2.

5.2 Optimal policy

In Figure 5.1, the average episodic cost and time over 100 simulations, for 1000 episodes with the Optimal Policy implementation seen in Algorithm 4, on the MDP version of the Minimal environment is illustrated. The averages are denoted by the the thick lines, while the filled area indicate the standard deviation. The standard deviation in Figure 5.1b can however not be observed due to that it is 0, i.e. the episodic time is always $t_{max} = 100$, which is expected as it is the optimal policy.



(a) The average episodic cost.

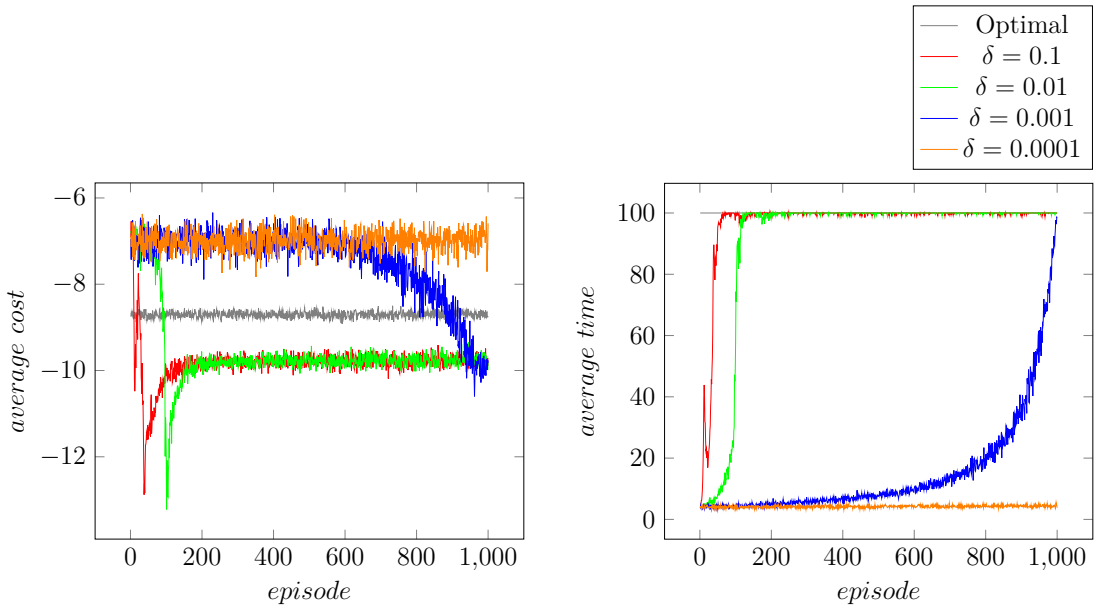
(b) The average episodic time.

Figure 5.1: The average episodic cost and time over 100 simulations, for 1000 episodes with the Optimal Policy implementation seen in Algorithm 4, on the MDP version of the Minimal environment. The filled area indicates the standard deviation, meanwhile the thick line denotes the actual average cost and time.

5.3 Q-learning

In Figure 5.2 the average episodic cost and time over 100 simulations, for 1000 episodes, with the learning rate $\alpha = 0.1$, the discount factor $\gamma = 0.7$ and the exploration factors $\delta = 0.1, 0.01, 0.001, 0.0001$ in the Q-learning implementation seen in Algorithm 5 on the Minimal environment can be observed. The amount of exploration seems to have little effect on the average episodic cost, where they all converge towards the same point with the exception of $\delta = 0.0001$ as it continues exploration and thereby acting randomly until the end. It may seem odd that before converging, the result revolves around a better value, i.e. -7. The reason however is as it chooses countermeasures randomly, it also chooses the counter measure $\{u_1, u_2\}$ that effectively terminates the simulation which is undesirable. Similarly, it may seem strange that the algorithm narrows in on approximately -13 before converging. The reason follows from how the randomness is applied, i.e. gradually decreased through $\epsilon = 1 - \delta \cdot i$ where δ is the exploration factor and i the current episode. Thus, near convergence the agent is closing in on behaving non-randomly and behaves quasi-random, thereby yielding even worse result in contrast to behaving completely random or non-random.

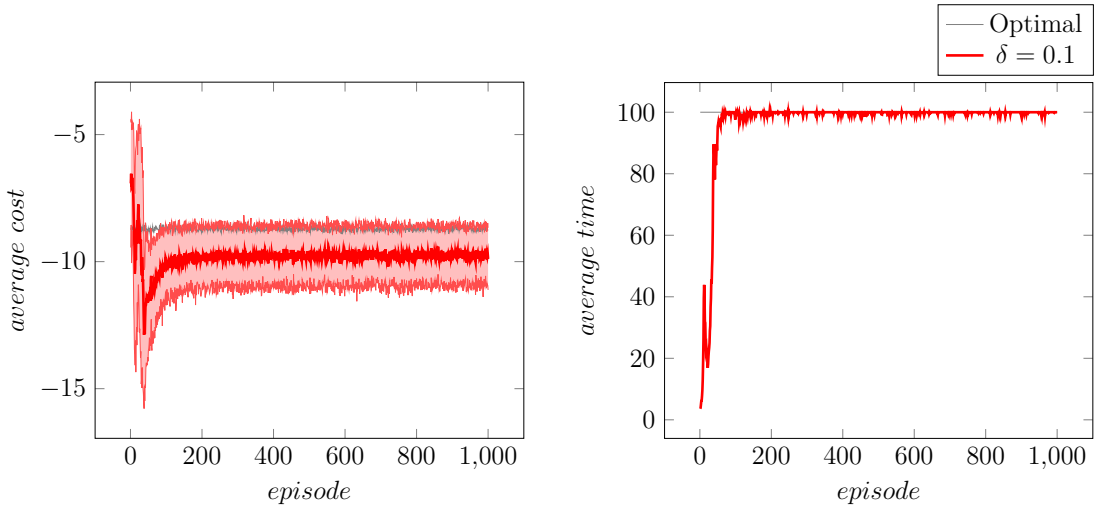
In a similar manner the average episodic time converges towards the same value, i.e. the optimal one t_{max} except $\delta = 0.0001$ due to its continuous randomness. Therefore, $\delta = 0.1$ is considered to be the optimal exploration factor, as it converges to the optimal average episodic time $t_{max} = 100$ and to the same average episodic cost as the others but faster. Noteworthy, is that the variance seen regarding the average episodic time is low after the exploitation phase, where we only observe



(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.2: The average episodic cost and time over 100 simulations, for 1000 episodes, the discount factor $\gamma = 0.7$ and the exploration factors $\delta = 0.1, 0.01, 0.001, 0.0001$ with the Q-learning implementation seen in Algorithm 5 on the Minimal environment.

small outliers.



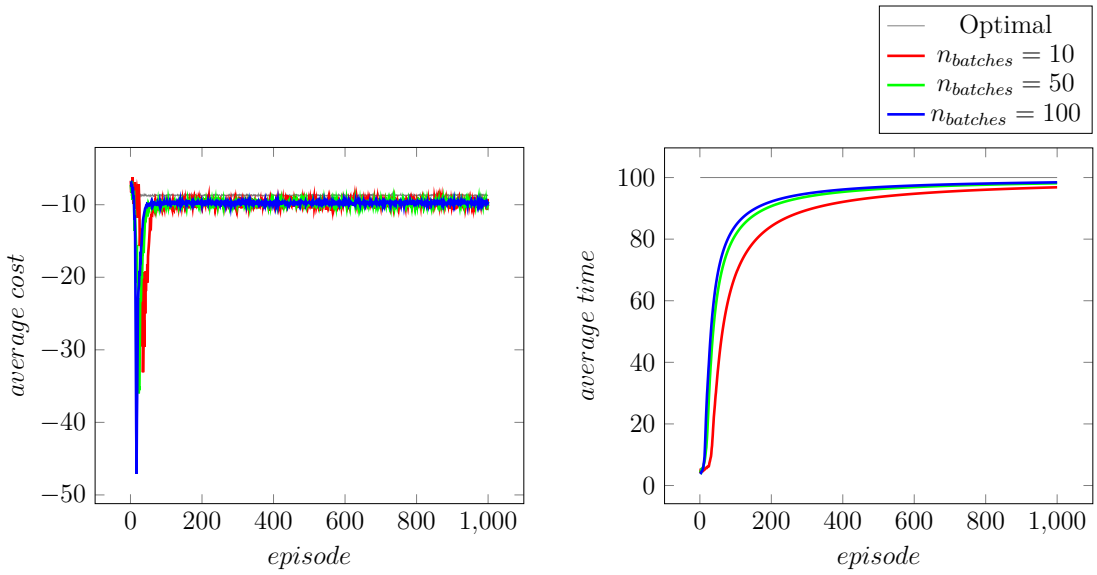
(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.3: The average episodic cost and time over 100 simulations, for 1000 episodes, the discount factor $\gamma = 0.7$ and the exploration factor $\delta = 0.1$ with the Q-learning implementation seen in Algorithm 5 on the Minimal environment. The filled area indicates the standard deviation, meanwhile the thick line denotes the actual average cost and time.

5.4 Policy Gradient

The Policy Gradient implementation is evaluated on the Minimal environment, where the results show a near optimal cost. For our application the network was configured with 30 hidden layer nodes, and the output layer consisted of a vector of size $1 \times |\mathcal{U}|$. Additionally, for each experiment the learning rate was set to 0.01. The graph in Figure 5.4, shows that the algorithm tries to optimize based on the countermeasures first and goes for a low cost which would be to do both action u^1 and u^2. Thereby, further into the simulation the agent starts optimizing for the availability of the system and achieves a longer simulation time, and the cost closes in on the optimal value.

The difference in convergence comes from the varying batch size being used. Having a larger batch size lowers the variance and improves convergence slightly. Though, the training time increases exponentially with increases in batch size. Clearly, we can make the conclusion that a larger batch size increases the performance on this problem.

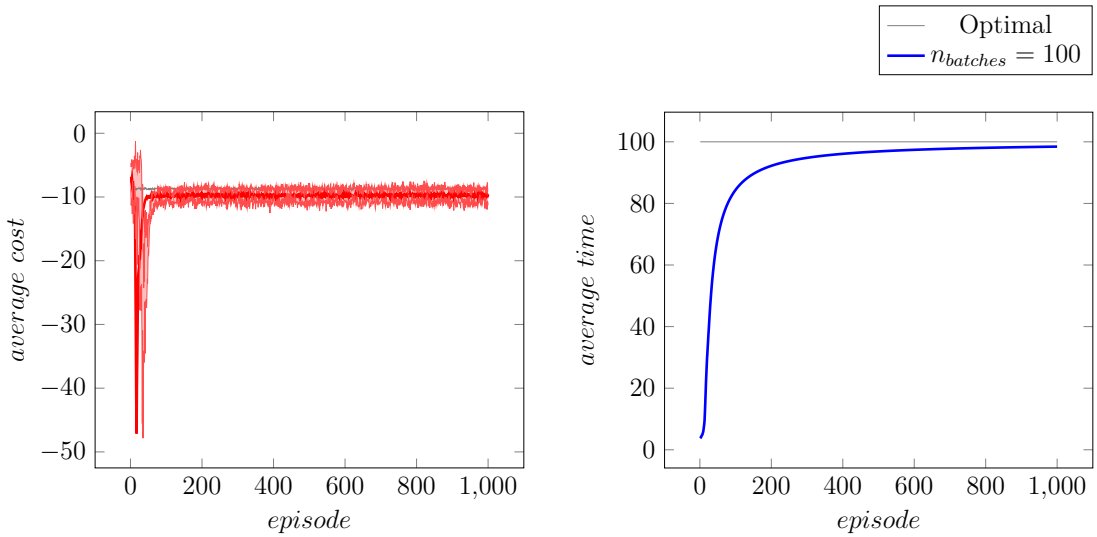


(a) The average episodic cost.

(b) The average episodic time.

Figure 5.4: The average episodic cost and time over 100 simulations, for 1000 episodes, with a batch size of $n_{batches} = 10, 50, 100$ for the Policy Gradient implementation seen in Algorithm 7.

Looking at the cost and time for batch size $n_{batches} = 100$, in Figure 5.5, where the thick lines display the averages.



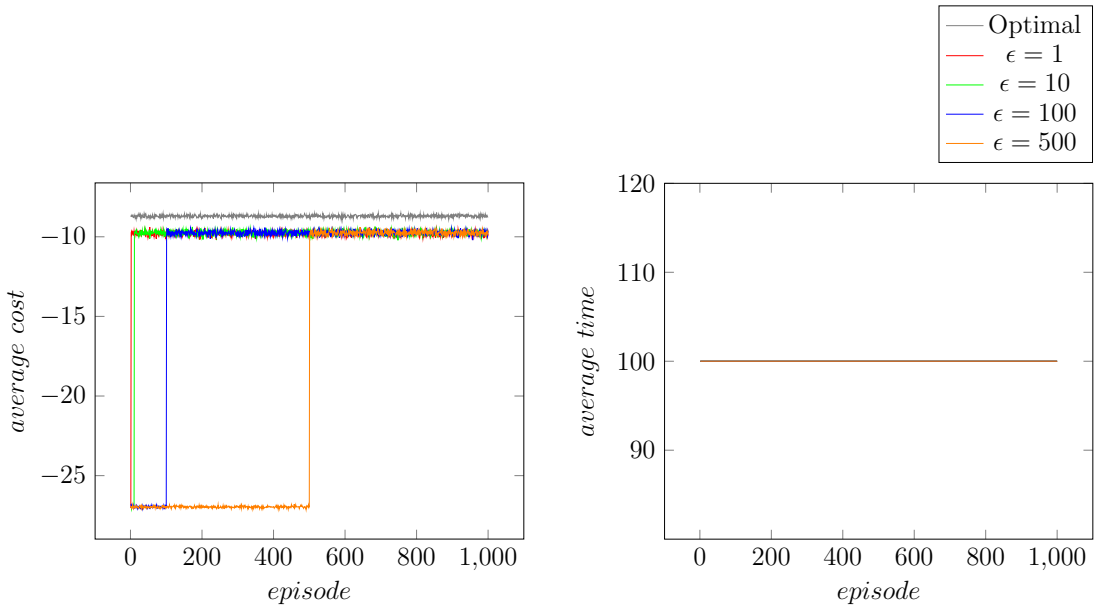
(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.5: The average episodic cost and time over 100 simulations, for 1000 episodes, with a batch size of $n_{batches} = 100$ for the Policy Gradient implementation seen in Algorithm 7. The filled area indicates the standard deviation, meanwhile the thick line denotes the actual average cost and time.

5.5 *n*-Myopic

In Figure 5.6 we can observe the average episodic cost and time over 100 simulations, for 1000 episodes, the number of predecessors to $n_{pred} = 0$ and the tuning parameters $\epsilon = 1, 10, 100, 500$ with the n-Myopic implementation seen in Algorithm 8 on the Inspect environment. The amount of tuning seems to have little effect on the average episodic cost, where they all converge towards the same result which is near optimal. Thus, the empirical matrix E seems to contain sufficient information after one episode of information gathering. Nonetheless, not lowering the tuning parameter any further to $\epsilon = 0$ is beneficial as E then would not contain any information at all and the algorithm would thereby act randomly. However, with larger tuning parameters, we receive more episodes with the episodic tuning cost, which lowers the average episodic result to approximately -27. The reason follows from the cost of performing the Inspect action, which lies at 0.2 and is performed nearly at each time-step in the episode.

Noteworthy, is that the average episodic time for the different tuning parameters are equivalent, fixed at t_{max} for all episodes. The reason follows from the design choice of n-Myopic, as it is constructed such that no countermeasures are deployed that terminates the simulation. Thus, $\epsilon = 1$ is considered to be the optimal tuning parameter as it has the fastest convergence rate to the optimal average episodic time $t_{max} = 100$ and to the same average episodic cost as the others.



(a) The average episodic cost.

(b) The average episodic time.

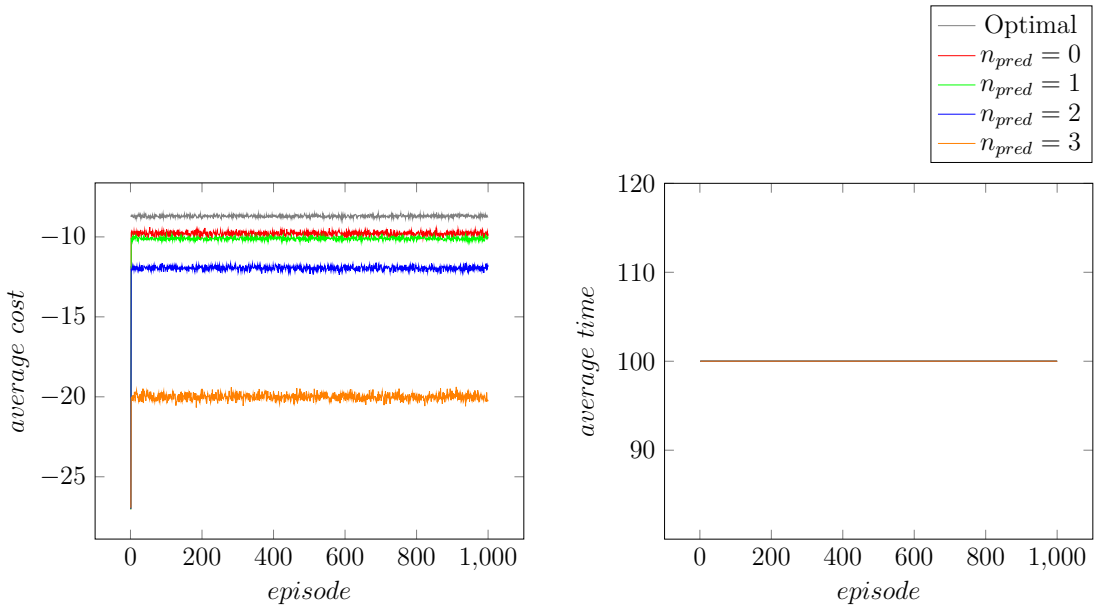
Figure 5.6: The average episodic cost and time over 100 simulations, for 1000 episodes, the number of predecessors to $n_{pred} = 0$ and the tuning parameters $\epsilon = 1, 10, 100, 500$ with the n-Myopic implementation seen in Algorithm 8 on the Inspect environment.

Following, we investigate the effect of the number of predecessors n_{pred} taken into account, for $\epsilon = 1$. In Figure 5.7, we can respectively observe the average episodic cost and time over 100 simulations, for 1000 episodes, with the number of predecessors set to $n_{pred} = 0, 1, 2, 3$. The result is degraded when taking more predecessors into account, thus we consider $n_{pred} = 0$ to be the optimal number of predecessors.

In Figure 5.8, the average episodic cost and time over 100 simulations for 1000 episodes with the number of predecessors to set $n_{pred} = 0$ and the tuning parameters $\epsilon = 1$ can be viewed. This is the set of parameters that were identified as optimal.

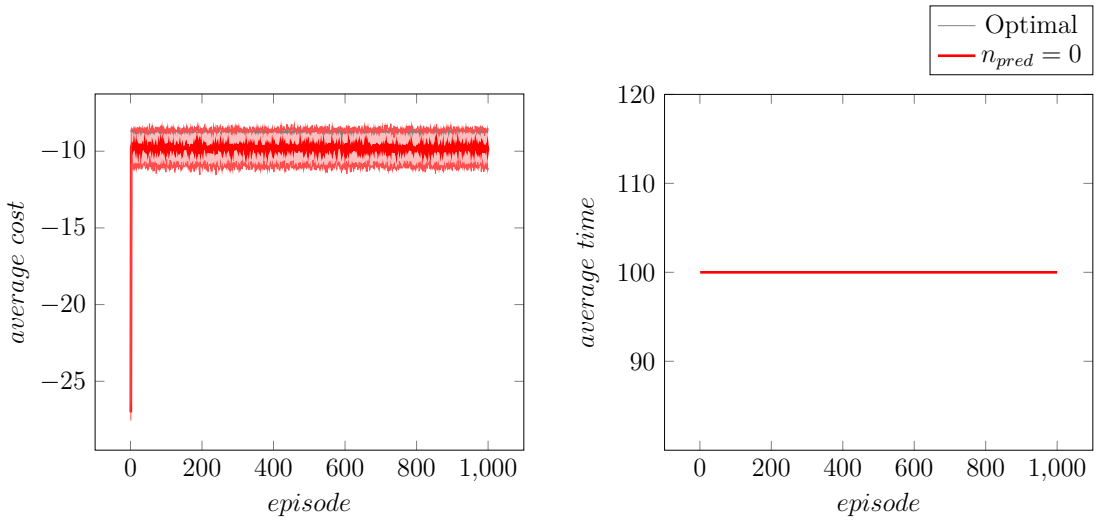
5.6 *n*-Lookahead

In Figure 5.9, we can respectively observe the average episodic cost and time over 100 simulations, for 1000 episodes, the threshold set to $\lambda = 0.4$, the number of time-steps to $n_{future} = 0$ and the tuning parameters $\epsilon = 1, 10, 100, 500$ with the n-Lookahead implementation seen in Algorithm 8 on the Inspect environment. Yet again as with the n-Myopic implementation, we can observe that the amount of tuning seems to have little effect on the average episodic cost, where they all converge towards the same result. Also, the result is affected similarly of the episodic tuning cost brought by a larger tuning parameter, where we witness the same average episodic result of



(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.7: The average episodic cost and time over 100 simulations, for 1000 episodes, the number of predecessors to $n_{pred} = 0, 1, 2, 3$ and the tuning parameter to $\epsilon = 1$ with the n-Myopic implementation seen in Algorithm 8 on the Inspect environment.

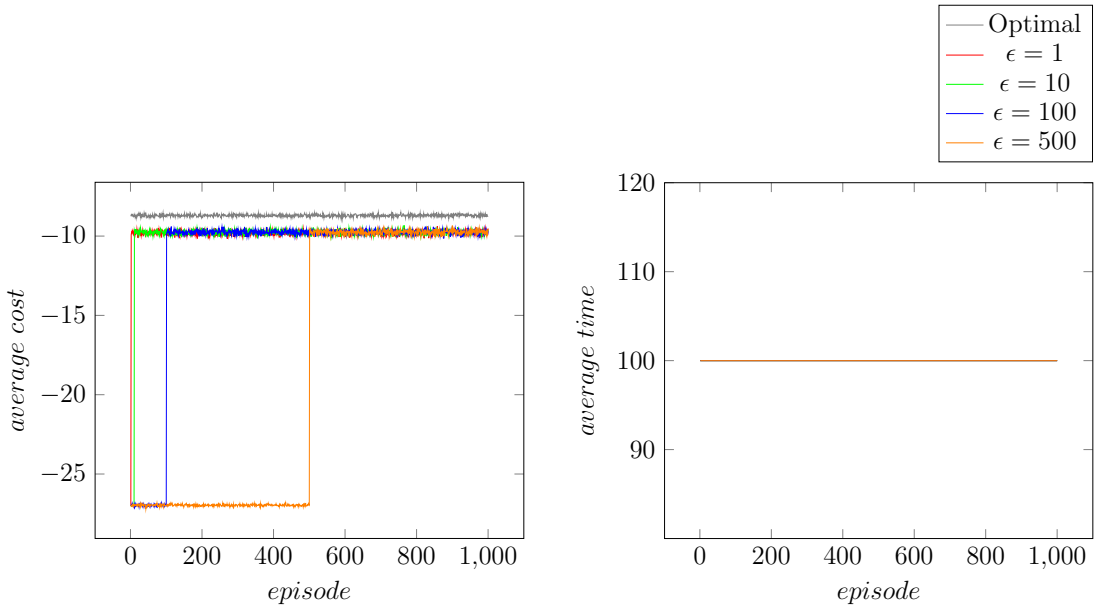


(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.8: The average episodic cost and time over 100 simulations, for 1000 episodes, the number of predecessors to $n_{pred} = 0$ and the tuning parameter to $\epsilon = 1$ with the n-Myopic implementation seen in Algorithm 8 on the Inspect environment. The thick lines denotes the average cost and time.

approximately -27 for episodes used for tuning.

Thus we consider $\epsilon = 1$ again to be the optimal exploration factor, as it converges to the optimal average episodic time $t_{max} = 100$ and to the same average episodic cost as the others, but as before, faster. We also note, as earlier, that the average episodic times are equivalent for the different tuning parameters, where the reason is the same as before, i.e. through the design of the algorithm.



(a) The average episodic cost. **(b)** The average episodic time.

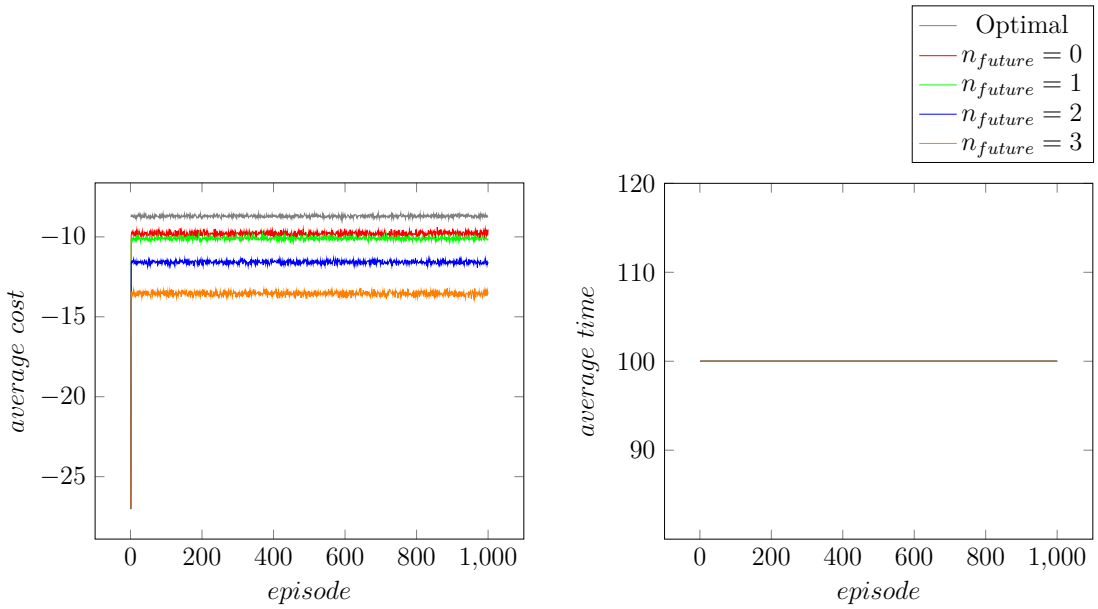
Figure 5.9: The average episodic cost and time over 100 simulations, for 1000 episodes, the threshold set to $\lambda = 0.4$, the number of predecessors to $n_{future} = 0$ and the tuning parameters $\epsilon = 1, 10, 100, 500$ with the n-Lookahead implementation seen in Algorithm 9 on the Inspect environment.

Continuing, in Figure 5.10, we can respectively observe the average episodic cost and time over 100 simulations, for 1000 episodes, where we investigate the effect of the different number of time-steps $n_{future} = 0, 1, 2, 3$. As with n-Myopic, we can observe that the result worsen with the increase of time-steps.

Thus, we can conclude that the optimal parameters are with the tuning parameter set to $\epsilon = 1$ and the number of time-steps to $n_{future} = 0$, where we in Figure 5.11 can respectively observe the average episodic cost and time over 100 simulations, for 1000 episodes.

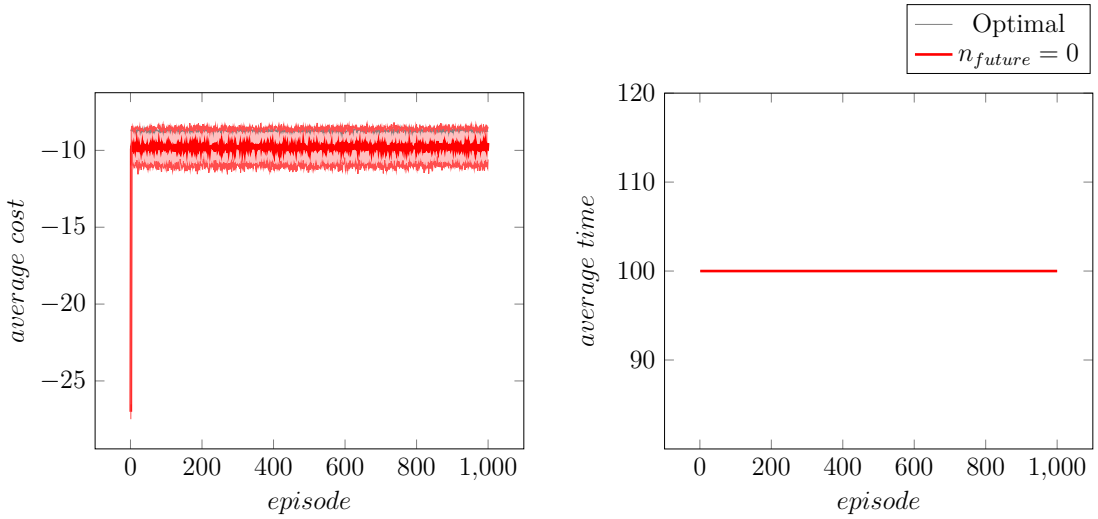
5.7 Comparison

In Figure 5.12, a comparison of all the algorithms is presented. Each algorithm entry is based on the best performing parameters from previous sections. Based on these plots, we can see that all of the algorithms perform close to the optimal



(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.10: The average episodic cost and time over 100 simulations, for 1000 episodes, the threshold set to $\lambda = 0.4$, the number of time-steps to $n_{future} = 0, 1, 2, 3$ and the tuning parameter to $\epsilon = 1$ with the n-Lookahead implementation seen in Algorithm 9 on the Inspect environment.

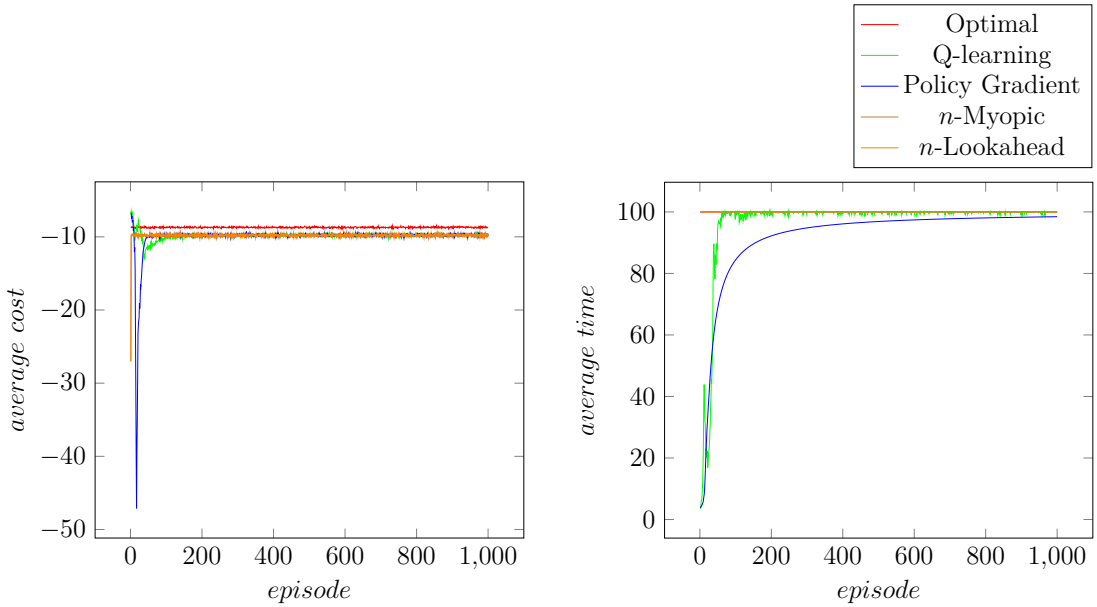


(a) The average episodic cost. **(b)** The average episodic time.

Figure 5.11: The average episodic cost and time over 100 simulations, for 1000 episodes, the threshold set to $\lambda = 0.4$, the number of time-steps to $n_{future} = 0$ and the tuning parameter to $\epsilon = 1$ with the n-Lookahead implementation seen in Algorithm 9 on the Inspect environment. The filled area indicates the standard deviation, meanwhile the thick line denotes the actual average cost and time.

solution. A difference however lies within the convergence of the result. For n-Myopic and n-Lookahead, we solely require one episode of tuning, but which brings the high episodic result of approximately -27. Simultaneously, for Policy Gradient and Q-learning, we respectively approximately require 50 and 100 episodes before converging, where Policy Gradient have its trough with average episodic results as low as approximately -47. Thus a consideration is needed in order to decide if a fast convergence with a low average episodic cost is more desirable than a slower convergence with a more modest average episodic cost.

An even larger difference however comes in the availability metric, where Policy Gradient and Q-learning, exhibits high variance and significantly more episodes until time convergence, compared to n-Myopic and n-Lookahead. In fact, the Policy Gradient implementation never reaches an average maximum time of $t_{max} = 100$ during the trial runs of 1000 episodes. For the two remaining algorithms, n-Myopic and n-Lookahead, the average episodic time is equivalent, which has its cause in their design. Thus, these can be viewed as near equal, in terms of optimality, as they both share similar average episodic costs and times.



(a) The average episodic cost.

(b) The average episodic time.

Figure 5.12: The average episodic cost and time over 100 simulations, for 1000 episodes, for all the algorithms with the best performing parameters. I.e., Q-learning, Policy Gradient, n-Myopic and n-Lookahead.

6

Conclusion

This chapter discusses the results in comparison to each of the algorithms developed and presented in this thesis, but also from the perspective of previous definitions of this problem. Additionally, a section about future research and improvements as well as ethical considerations is presented.

To help the reader follow along in the discussion, the first section, Section 6.1, is a short summary of the paper.

6.1 Summary

The goal of this thesis was to develop a model of an attack environment for a computer system. Additionally, based on this environment model, develop defense strategies to protect against attackers. We have presented *two* formulations of threat environments based on a POMDP. Where the first one follows previous research by Miehling et al. (2015), and the second environment improves on it by introducing a new action, that allows an agent to approximately reduce the problem into a MDP. The reduction allows for fast and simple solving algorithms. Given this feature, we have presented several solving algorithms, based on traditional approaches, as well as introducing *two* new algorithms. The traditional approaches of using Q-learning and Policy Gradient showed good results, but the new introductions, n-Myopic and n-Lookahead, outperformed both.

6.2 Discussion

The thesis tried to answer if an automated system could be designed to respond to computer threats. We have shown that the problem can be formulated and solved as a POMDP, using different solving algorithms. The work improves on previous definitions such as the one made by Miehling et al. (2015), by adding a new formulation of the environment. The new formulation allows for fast solving algorithms

by approximately reducing the problem into a MDP for a set of time steps. Based on both the previous definition and the new formulation, two OpenAI Gym environments were developed to help future research develop new solving agents with the same environments. This contribution allows for equal comparisons between this, and future research. We have shown two new solving algorithms proving that fast and simple algorithms can be formulated to solve the problem. Furthermore, by defining the optimal solution, and comparing to some traditional reinforcement learning algorithms, we have shown a baseline for future research.

On the flip side, the research still has a long way to go, and we do not think this is applicable to a production system yet. There is several improvements that needs to made, and how much it can be scaled has to be formally proven. More details on future directions in Section 6.2.1.

There are aspects with the model and environment that are inherited from the work of Miehling et al. (2015) which could be improved to approach real-word scenarios further. One of these are how we embody the desire of keeping the system available as long as possible. Currently we use the discount factor γ to map this behaviour on the agent, as explained in Section 4.1. However, an alternative method which could prove to be more fruitful is by implying a reward for each time-step that the system is kept online, i.e. changing the cost of doing nothing as a countermeasure to a positive value $C(\varnothing) > 0$. That said, we also create an additional tuning factor, potentially rewarding up-time too much could lead to never taking any action and not shutting attackers out, since it is more rewarding to just keep the system up. Nonetheless, it is an area which should be explored and address as future work.

An intended design choice to be mentioned and discussed is that a simulation solely ends when all available binary actions are deployed as a countermeasure. I.e., if solely a part of the system is temporally shut down, the simulation will continue. One may argue that if an adversary has enabled a number of attributes, i.e. found and exploited a number of vulnerabilities in a computer system, that the adversary would learn how to use these vulnerabilities. Thus if a countermeasure was applied that disabled the enabled attributes, the adversary would enable them again at a faster pace. Our reasoning however, is that the defender, i.e. the operators of the computer system, would resolve the weaknesses while the specific part of the system is disabled. The eventuality of these weaknesses happening should not be removed from the model, as the adversary could learn to bypass the defensive changes made or that another superior adversary with a different skill set could use a similar method to make breaches. How the probabilities should change for a set of attributes after defensive changes have been made while being disabled is another area for future work.

What also need to be considered, is the limitations of the environment defined by Miehling et al. (2015). As could be observed in Figure 5.12, the result of all algorithms with the best performing parameters, was close to the optimal solution once converged. A reason for this could be that the environment can be viewed as

quite simple. It is constituted of solely 12 attributes, where two are leaf attributes and one is a critical attribute, with two binary actions. Further, the detection probabilities $\boldsymbol{\beta} \geq 0.5$, are high, where they are even higher for the most crucial attributes, i.e. the critical attribute and its direct predecessor, $\beta_i \geq 0.85$ where $i \in \{11, 12\}$. Thus one can more or less apply the optimal policy for the Minimal and Inspect environment in Algorithm 4 directly on the POMDP environment, as the probability to witness the correct state at the direct predecessor to the critical attribute $\bar{\mathcal{D}}_{12} \in \{11\}$ is close to 1. Therefore, all the algorithms applied to the environment achieves good result, as the problem is close to a MDP when in close distance to the critical attribute in the Minimal and Inspect environment.

An example where we can observe the result of this, is the choosing of parameters yielding best result for n-Myopic and n-Lookahead. As the best number of predecessors for n-Myopic, and the best number of time-steps for n-Lookahead, are both zero, i.e. $n_{pred} = n_{future} = 0$, the algorithms are equivalent with the difference that n-Myopic observes if the maximum probability when receiving an observation is to be in the state where $s^{11} \in \{1\}$, where n-Lookahead instead examines if the probability to be in the same state is over the specified threshold, i.e. $\lambda = 0.4$. I.e., both algorithms solely uses the built empirical matrix, to see if they in the current time-step is at the direct predecessor to the critical attribute, as the success probability of doing such is high.

Also to highlight, is if the cost was not instantaneous, and instead yielded at the end of the simulation, Q-learning and Policy Gradient would not achieve the same result, as they are dependent on receiving direct feedback for deploying a specific action for a specific observation. At the same time, the result of n-Myopic and n-Lookahead would be unchanged, as they solely rely on the empirical and transition matrix built.

To remember is that the self-developed algorithms n-Myopic and n-Lookahead are model-specific to the contrary of Q-learning and Policy Gradient, i.e. that n-Myopic and n-Lookahead can not be applied to any general problem environment. However, as the developed algorithms solely were intended for the area of computer security, and that we have a method to create a model of a computer system, it is not viewed as an issue.

6.2.1 Future work

The environments presented in this thesis are static, manually constructed, and would be considered small formulations of the problem. A first step of improvement would be to automatically develop environments constituting of real-world sized systems, that in turn would require large Bayesian attack graphs. Thus, future research would focus on having a dynamic data-stream that defines the environment, actions, and the corresponding cost function. The challenge constitutes to automat-

ically identify vulnerabilities in a system and create a Bayesian attack graph with these, as well as accurately set the spread and attack probabilities before and after defensive changes in system have been performed. Additionally, find a way to translate data from for example a vulnerability database into an accurate probabilistic representation of how hard it is to carry out and its likelihood.

The simulation currently stops if the action of shutting down, u^1, as well as doing the block action, u^2, are performed. It would be valuable to explore having a cost for the amount of time a node is shut down. Potentially allowing the agent to bring it back up again when the vulnerability(s) is resolved. Some analysis and formulation of a cost function for this type of problem is presented in Section 3.4, but deeper analysis and experimental results are missing. Furthermore, we have discussed the difference in having instantaneous cost versus giving the cost at the end of a simulation. The algorithms deployed in this paper abuses the instantaneous cost to achieve good, and comparative results. But, forcing a cost at the end of simulation, would achieve a more realistic formulation of the problem, and would therefore be of interest to investigate further. Additionally, adding a explicit positive reinforcement for system availability, in other word a reward for every time-step the system is alive, could more closely match the real world.

To receive a more truthful result, a more complex environment should be developed with more attributes, containing more critical and leaf attributes, and thus more binary actions. The detection probabilities $\boldsymbol{\beta}$ should be minimized, and a larger variety of attack and spread probabilities should be used. Applying the algorithms at hand on this new and more complex environment, would give more varying results, where it should be clearer to identify the optimal algorithm. And if the cost was reconfigured from being instantaneous, one would see even higher variance between the results of the different algorithms.

As discussed in Section 4.1, it should further be looked into if the definition of the problem is correct and represents the real world. Currently, a cost is not received if an adversary enable attributes that are not considered critical, i.e. $i \in \mathcal{N} \setminus \mathcal{N}_{\mathrm{C}}$. It it is doubtful however, if an owner of a system would be indifferent to any kind of unauthorized access to any part of the system, but would rather see it as a security failure.

Naturally, it would be interesting to lift the assumptions made in Section 3.1, in order to approach real-word scenarios even further. As an example, we currently look at the problem without a notion of false positives. Future work would include defining it and coming up with a way to measure it.

Our work has focused on using simple solving algorithms for this complex environment definition. It would be of high value looking into combining more advanced POMDP solving algorithms with the addition of the Inspect action. A limitation we found during the literature study was that these algorithms scaled poorly to larger formulations of environments. But, with the reduction to an approximate

MDP that Inspect gives us, we have fast and stable solving methods that could be deployed in conjunction with advanced POMDP algorithms to achieve larger scale. Examples of such potential algorithms would be *POMCP* (Silver & Veness, 2010) and *R-DESPOT* (Somani et al., 2013) that both have showed strong performance on large-scale POMDP problems as well as being open source projects. Additionally, we provide no convergence guarantees or analysis of the algorithms, which would help in more theoretical comparison.

The way the relative cost for each action is defined based on real-world data is done in a somewhat hand-wavy fashion. There could be interesting results in doing a deeper analysis and developing a framework for a general environment.

Currently, the hyper-parameter optimization is quite crudely performed through a grid search with a sufficient distance between each tested hyper-parameter tested, limited by our time and computing power. As an example, the learning rate α and the discount factor γ combinations for Q-learning is evaluated with a distance of 0.1, i.e. $\alpha, \gamma \in \{0, 0.1, 0.2, \ldots, 1\}$. This can obviously be improved by evaluating with a shorter distance of 0.1, requiring more time and computing power. Another option would be to consider other hyper-parameter optimization methods, e.g. *random search* (Bergstra & Bengio, 2012), *Bayesian optimization* (Snoek, Larochelle, & Adams, 2012), *gradient based optimization* (Bengio, 2000) or *evolutionary optimization* (Friedrichs & Igel, 2005).

It could also be further investigated on how to deduce if the probability is sufficient for being at a specific state, to make the decision to deploy an appropriate countermeasure or not. At the time being, e.g. for n-Myopic, we investigate if the maximum probability is to be in one of the n-predecessors defined by $Y^c_{n_{pred}}$, and if so deploy the optimal countermeasure. Following, for n-Lookahead, we do not rely on the maximum probability but instead of a threshold λ, found through grid search, where we deploy an optimal countermeasure if the probability to be in a specific state bypasses the threshold. Thus it could be further looked into what strategy yields best result, and if using the threshold, consider different hyper-parameter optimization methods as discussed earlier.

6.2.2 Ethical considerations

In any type of automation there is going to potentially be people losing a job that existed before the automation. New technology usually introduces a new set of professions to support the technology, but the number of positions and the educational level changes. Eventually, finding oneself in a situation where knowledge about the fundamental issues have dissipated. Furthermore, only a few individual know how to operate and maintain the system.

Having an automated AI that potentially takes action in an real-world environment there is a concern around who is responsible if something goes wrong. Imagine this system being deployed in an environment serving a health care related application that patients rely on for medication or management of a condition. In this scenario if the system decides to shut down servers in a false positive scenario, and patients loses access to the service. It might be simple to put responsibility on the entity running the infrastructure and the defense system, which is probably correct. However, it results in a situation where the administrative entity can argue no blame, because having a breach would be of a worse severity, and therefore would have taken the best action to their knowledge.

Having knowledge about the workings of any security system can allow you to abuse it, and thus become undetectable. Additionally, knowing how the system deploys actions could lead to a new set of denial-of-service attacks, where an attacker purposely reveals its attacks, so that the system shut down resources that are essential to the overall service.

Trusting an automated system for security can lead to decisions that are hard to explain and motivate. In general machine learning/AI systems tend to lead to situation where a result is hard to backtrack. In other words the system performs actions or gives results we have a hard time explaining. An interesting direction to expand the research would be to create a co-existing policy AI, that connects all the information of the decision-making AI, and tries to explain it as a human would interpret it.

One way of alleviate the concerns that are associated with the automated AI response to breaches is to transform the system into a recommendation engine. The difference would be that no countermeasures are deployed directly by the AI. Instead, the AI solely performs the calculations and thereafter presents its choice of viable countermeasures to the operators, which will perform the countermeasures if deemed fit. In other words a human operator is now the final decision-maker, but with access to fast and reliable information from the security system.

References

Aberdeen, D., & Baxter, J. (2002). Scaling internal-state policy-gradient methods for pomdps. In *Proc. icml-02* (pp. 3–10).

Bengio, Y. (2000). Gradient-based optimization of hyperparameters. *Neural computation*, *12*(8), 1889–1900.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*(Feb), 281–305.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *Openai gym.*

Dimitrakakis, C., & Ortner, R. (2018). *Decision making under uncertainty and reinforcement learning.* Retrieved from `http://www.cse.chalmers.se/~chrdimi/downloads/book.pdf`

Federal Bureau of Investigation. (2016). *2016 internet crime report.* Retrieved from `https://pdf.ic3.gov/2016_IC3Report.pdf`

Friedrichs, F., & Igel, C. (2005). Evolutionary tuning of multiple svm parameters. *Neurocomputing*, *64*, 107–117.

Hansen, E. A., & Feng, Z. (2000). Dynamic programming for pomdps using a factored state representation. In *Aips* (pp. 130–139).

Hoffmann, J. (2015). Simulated penetration testing: From" dijkstra" to" turing test++". In *Icaps* (pp. 364–372).

Krutz, R. L., & Vines, R. D. (2010). *Cloud security: A comprehensive guide to secure cloud computing.* Wiley Publishing.

Liao, H.-J., Lin, C.-H. R., Lin, Y.-C., & Tung, K.-Y. (2013). Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, *36*(1), 16 - 24. Retrieved from `http://www.sciencedirect.com/science/article/pii/S1084804512001944` doi: https://doi.org/10.1016/j.jnca.2012.09.004

Liu, Y., & Man, H. (2005). Network vulnerability assessment using bayesian networks. In *Data mining, intrusion detection, information assurance, and data networks security 2005* (Vol. 5812, pp. 61–72).

Miehling, E., Rasouli, M., & Teneketzis, D. (2015). Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the second acm workshop on moving target defense* (pp. 67–76). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2808475.2808482` doi: 10.1145/2808475.2808482

Ou, X., Boyer, W. F., & McQueen, M. A. (2006). A scalable approach to attack

graph generation. In *Proceedings of the 13th acm conference on computer and communications security* (pp. 336–345). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1180405.1180446 doi: 10.1145/1180405.1180446

Papadimitriou, C., & Tsitsiklis, J. N. (1987, August). The complexity of markov decision processes. *Math. Oper. Res.*, *12*(3), 441–450. Retrieved from http://dx.doi.org/10.1287/moor.12.3.441 doi: 10.1287/moor.12.3.441

Ponemon Institute LLC. (2017, Sep). *2017 cost of cyber crime study.* Accenture. Retrieved from https://www.accenture.com/t20170926T072837Z__w__/us-en/_acnmedia/PDF-61/Accenture-2017-CostCyberCrimeStudy.pdf

Poolsappasit, N., Dewri, R., & Ray, I. (2012). Dynamic security risk management using bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*, *9*(1), 61–74.

Sarraute, C., Buffet, O., & Hoffmann, J. (2012). Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Aaai.*

Sheyner, O. M. (2004). *Scenario graphs and attack graphs* (Unpublished doctoral dissertation). US Air Force Research Laboratory.

Shmaryahu, D., Shani, G., Hoffmann, J., & Steinmetz, M. (2017). Partially observable contingent planning for penetration testing. In *Iwaise: First international workshop on artificial intelligence in security* (p. 33).

Silver, D. (2015a). *Lecture 1: Introduction to reinforcement learning.* UCL. Retrieved from http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html

Silver, D. (2015b). *Lecture 7: Policy gradient methods.* UCL. Retrieved from http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html

Silver, D., & Veness, J. (2010). Monte-carlo planning in large pomdps. In *Advances in neural information processing systems* (pp. 2164–2172).

Smallwood, R. D., & Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, *21*(5), 1071-1088. Retrieved from https://doi.org/10.1287/opre.21.5.1071 doi: 10.1287/opre.21.5.1071

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems* (pp. 2951–2959).

Somani, A., Ye, N., Hsu, D., & Lee, W. S. (2013). Despot: Online pomdp planning with regularization. In *Advances in neural information processing systems* (pp. 1772–1780).

Wells, D., Pazandak, P., Nodine, M., & Cassandra, A. (2004, aug). Adaptive defense coordinator for multi-agent systems. In *Proceedings of the first ieee symposium on multi-agent security and scalability.*

A

OpenAI Gym Environments

A.1 Requirements

- Python 2.7
- Git
- OpenAI Gym
- Numpy

A.2 Installing

Github repository: `https://github.com/hampusramstrom/gym-threat-defense`

Fetch the public source code from Github:

```
$ git clone git@github.com:hampusramstrom/gym-threat-defense.git
$ cd gym-threat-defense/
```

Then, to globally install the library run:

```
$ pip install -e .
```

The library can now be imported into any Python 2.7 project. The next sections detail how to use the library in a project. Furthermore, there is a fully functioning example agent based on Q-learning in Section A.4.

A.3 Usage

The goal of the Gym project is to make it easy to run and create agents for several reinforcement learning domains by having a consistent interface. The three required functions are:

- *reset()*
- *step()*
- *render()*

The *reset()* function resets the environment to the starting state, and the *render()* function makes it possible to render a graphical representation of observations. The interesting detail here is the *step()* function that progresses the environment based on an action being passed as an argument to the function. For each step t in the Minimal environment the agent receives a reward r_{t+1}, and an observation o_{t+1} based on the action a_t that was taken. The important difference in the second implementation of the environment is that when a step is taken the agent receives a tuple containing the observed state, and the true state (o_{t+1}, s_{t+1}), but only if the action taken was Inspect. In any other cases, the environment behaves just as the Minimal implementation, where the result of a step is an observation.

A.4 Example

The example below shows how a simple Q-learning agent could be implemented and run with the Gym environment.

```
"""
A simple example on how to use the Threat Defense environment,
applying Q-learning where a table is used to store the data.

Authors:
Johan Backman - johback@student.chalmers.se
Hampus Ramstrom - hampusr@student.chalmers.se
"""

import numpy as np
import random
import gym
import gym_threat_defense  # noqa


def choose_action(env, observation, q, i):  # noqa
    """
    Chooses a new action, either randomly or the one with
```

```
19      max value in the Q table, depending on the amount of randomness.
20
21      Arguments:
22      env -- the Threat Defense gym environment.
23      observation -- an observation as its numeric index in the states
        matrix,
24          containing all states.
25      q -- the Q table, containing the data.
26      i -- the current episode in the simulation.
27
28      Returns:
29      An action containing a numeric value [0, 3].
30      """
31      dec = 0.01
32      eps = 1 - i * dec
33
34      if random.uniform(0, 1) < eps:
35          return env.action_space.sample()
36      else:
37          return np.argmax(q[observation])
38
39
40  def get_index_in_matrix(env, observation):
41      """
42      Retrieves the index of an observation in the STATES matrix,
43      containing all states.
44
45      Arguments:
46      env -- the Threat Defense gym environment.
47      observation -- an observation as a binary vector of length 12.
48
49      Returns:
50      A numeric index.
51      """
52      for i in range(env.all_states.shape[0]):
53          if np.array_equal(observation, env.all_states[i]):
54              return i
55
56
57  def q_learning(env):
58      """
59      Runs Q-learning with a simple table for storing the data and prints
        the
60  mean reward for the last 100 episodes as well as printing the Q-table
        at the
61      end of the simulation.
62
63      Arguments:
64      env -- the Threat Defense gym environment.
65      """
66      q = np.zeros([env.observation_space.n, env.action_space.n])
67      alpha = 0.1
68      gamma = 0.7
69      num_episodes = 2000
70
71      rewards = []
```

```
72
73      for i in range(num_episodes):
74          o_list = env.reset()
75          o = get_index_in_matrix(env, o_list)
76          done = False
77          r_all = 0
78
79          while True:
80              a = choose_action(env, o, q, i)
81
82              on_list, r, done, _ = env.step(a)
83              on = get_index_in_matrix(env, on_list)
84              q[o, a] = q[o, a] + alpha * (r + gamma * np.max(q[on]) - q[
    o, a])
85              o = on
86              r_all += r
87
88              if done:
89                  break
90
91          rewards.append(r_all)
92          if i % 100 == 0 and i > 0:
93              print 'Episode: %s' % i
94              print "Score over the last 100 episodes: " + \
95                  str(sum(rewards[(i - 100):i]) / 100)
96
97      print "Score over time: " + str(sum(rewards) / num_episodes)
98      print "Final Q-Table values"
99      print q
100
101
102 env = gym.make('threat-defense-v0')
103 q_learning(env)
```