KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Technology

Second Cycle, 30 credits

MetaNet: A Meta Learning Model for Automated Penetration Testing of Networked Systems

Application of Meta Learning Ideas on Penetration Testing Problems

Chang Fu

Stockholm, Sweden, 2024

Authors

Chang Fu <changf@kth.se>
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden

Examiner

György Dán
KTH Royal Institute of Technology

Supervisor

Yeongwoo Kim
KTH Royal Institute of Technology

Abstract

With the development of networked systems, vulnerabilities underlying a network have kept increasing in recent years, and cyber security has become an essential part when building such networks. One of the most popular methods of evaluating the security of a network is penetration testing. However, we have seen a shortage of experts in the penetration testing field due to its complexity and the training cost. One way to alleviate this problem is automated penetration testing, which automates the penetration test process using algorithms, including the Attack Graph model, Partially Observable Markov Decision Process (POMDP) method.

In this thesis, we demonstrates the application of reinforcement learning algorithms on penetration testing problems and shows the potential application of meta-learning methods on such problems to boost the generalization ability of reinforcement learning algorithms. We first test the performance of Advantage Actor Critic (A2C) and Double Deep Q-Network (DDQN) on different static networks and compare their convergence speed, stability and total rewards achieved. Then we incorporate meta-learning ideas into reinforcement learning algorithms and propose a new model named MetaNet. Our results show that reinforcement learning algorithms are capable of solving penetration testing problems with little prior knowledge, and by using meta-learning methods, MetaNet shows a great improvement in generalization ability.

To conduct our experiments, we first create a test environment, which is a structured network mimicking actual communication networks in real-world. Each network is composed of several hosts, and each host contains several services that can be compromised. Then we apply A2C and DDQN on these networks. The algorithms start from a certain host and try to compromise the target host. Our results show that both A2C and DDQN are capable to compromise the target host and achieve positive rewards under most circumstances. To increase the generalization ability of these algorithms, we propose MetaNet, where we add additional inputs to the model, wrap the model with Long Short-Term Memory (LSTM) and train the model on different networks at once. Our results show that MetaNet not only keeps high winning ratios on networks that it is trained on but also performs better than the vanilla algorithms on other unseen networks.

Keywords

Penetration Testing, Reinforcement Learning, A2C, Meta Learning

Abstract

I och med utvecklingen av nätverkssystem har sårbarheterna i ett nätverk ökat under de senaste åren, och cybersäkerhet har blivit en viktig del när man bygger sådana nätverk. En av de mest populära metoderna för att utvärdera säkerheten i ett nätverk är penetrationstestning. Vi har dock sett en brist på experter inom penetrationstestområdet på grund av dess komplexitet och utbildningskostnaden. Ett sätt att lindra detta problem är automatiserad penetrationstestning, som automatiserar penetrationstestprocessen med hjälp av algoritmer, inklusive Attack Graph-modellen, POMDP-metoden.

Denna avhandling demonstrerar tillämpningen av förstärkningsinlärningsalgoritmer på problem med penetrationstestning och visar den potentiella tillämpningen av meta-inlärningsmetoder på sådana problem för att öka generaliseringsförmågan hos förstärkningsinlärningsalgoritmer. Vi testar först prestandan hos A2C och DDQN på olika statiska nätverk och jämför deras konvergenshastighet, stabilitet och totala uppnådda belöningar. Sedan införlivar vi meta-lärande idéer i förstärkningsinlärningsalgoritmer och föreslår en ny modell som heter MetaNet. Våra resultat visar att förstärkningsinlärningsalgoritmer kan lösa penetrationstestningsproblem med få förkunskaper, och genom att använda meta-inlärningsmetoder visar MetaNet en stor förbättring av generaliseringsförmågan.

För att genomföra våra experiment skapar vi först en testmiljö, som är ett strukturerat nätverk som efterliknar faktiska kommunikationsnätverk i verkligheten. Varje nätverk består av flera värdar, och varje värd innehåller flera tjänster som kan äventyras. Sedan tillämpar vi A2C och DDQN på dessa nätverk. Algoritmerna utgår från en viss värd och försöker äventyra målvärden. Våra resultat visar att både A2C och DDQN är kapabla att äventyra målvärden och uppnå positiva belöningar under de flesta omständigheter. För att öka generaliseringsförmågan hos dessa algoritmer föreslår vi MetaNet, där vi lägger till ytterligare input till modellen, slår in modellen med LSTM och tränar modellen på olika nätverk samtidigt. Våra resultat visar att MetaNet inte bara håller höga vinstkvoter på nätverk som det är tränade på utan också presterar bättre än vaniljalgoritmerna på andra osynliga nätverk.

Nyckelord

Penetrationstestning, Förstärkningsinlärning, A2C, Meta Learning

Acknowledgements

From January 2021 to December 2022, it has been two years since I started this thesis, and I can't think of any reason for this apart from myself. I did a similar project when I was doing my bachelor's thesis, where I tried to solve Autonomous Vehicle Routing Problems using reinforcement learning algorithms, and it's hard to say that I actually achieved anything after I completed it. Thus I choose this thesis topic, hoping I can start from where I have left and go further. Although there are difficulties, I have received numerous help from different people.

First, I'd like to thank my examiner, György Dán, for giving me this opportunity to carry on this thesis. Reinforcement learning is actually a rarely used method in the cyber security area compared with other methods, but he grants me a chance to choose the method I'd like to use, which eventually results in this thesis.

Then, I want to give a huge thanks to my supervisor, Yeongwoo Kim, who has been supporting me to finish this thesis from the very start till the very end. A lot of ideas, whether they are implemented in this thesis or not, are proposed by him. I can say for sure that I can't reach this far without his help along the way. Thank you again for your ideas and reviews.

Finally, I want to thank my parents for motivating me to move forward on this thesis. I always tend to put this thesis as a low priority, but they emphasize the importance of this thesis and push me forward all the time.

Looking back, I have spent quite an amount of time on this thesis. Still, it would be hard for me to reach this far without the help of people around me. Thank you all again, for your instruction, your accompany and your support.

Acronyms

A2C Advantage Actor Critic. iii, iv, 2, 3, 5, 12–14, 17, 26, 27, 29–33, 36, 37

A3C Asynchronous A2C. 14

AI Artificial Intelligence. 2

ANN Artificial Neural Network. 7, 8

DAG Directed Acyclic graph. 21

DDoS Distributed Denial of Service. 1

DDQN Double Deep Q-Network. iii, 3, 11–13, 17, 26, 27, 29–32, 36, 37

DQN Deep Q-Network. 2, 5, 10, 11, 17

KEV Known Exploited Vulnerabilities. 2

LSTM Long Short-Term Memory. iii, 8, 9, 14, 27, 28, 33, 34, 36

MAML Model-Agnostic Meta-Learning. 5, 14–16

MC Monte Carlo. 13, 26

MDP Markov Decision Process. 2, 7, 9, 10

NLP Natural Language Processing. 15

PDDL Plan Domain Definition Language. 17

POMDP Partially Observable Markov Decision Process. iii, 2, 7, 16, 17

PPO Proximal Policy Optimization. 10

ReLU Rectified Linear Unit. 11, 13

RNN Recurrent Neural Network. 8, 14

SQL Structured Query Language. 5

SSH Secure Shell. 5

TD Temporal Difference. 13, 26

Contents

Chapter 1

Introduction

1.1 Background

With the development of communication networks and the increasing complexity of software applications, vulnerabilities underlying networks keep increasing in recent years [35], [39]. These vulnerabilities can be exploited by attackers to perform forbidden actions such as accessing sensitive data and interfering functionality of networks. What accompanies the increasing number of vulnerabilities is the increasing difficulty of preventing potential attacks. More and more intricate networks and numerous services running on them make it more important to ensure the security of networks, and cyber security gains the interest of many researchers in the computer science field.

Cyber security is the practice of ensuring the security of hosts, services and data in a specific network as well as maintaining the integrity of the network. According to the Cyber Security Breaches Survey conducted by the UK government in 2020, the extent of cyber security threats had not diminished, and cyber attacks had evolved and become more frequent [35]. According to the survey, almost half of businesses (46%) and a quarter of charities (26%) reported having cyber security breaches or attacks in the last 12 months.

Common cyber security threats are malware attacks, social engineering attacks, Distributed Denial of Service (DDoS) attacks and so on. Malware attacks include Trojan viruses, worms and other malicious software that intends to gain unauthorized access to data and establish backdoors. Social engineering attacks include phishing, malvertising and other human-targeted attacks that urge the employee to perform unintended actions. DDoS attacks attempt to disrupt the normal traffic and to overwhelm the resources of a network to make it unable to function normally. These attacks are often used in combination with other cyber attacks. Among all these security threats, Aslan et al. survey the top threats for networks and cyber security, where remote procedure calls and SQL injections are the leading threats [46].

To protect networks from potential attacks and infiltration, thorough checks should be performed regularly. These checks include security audits, risk assessments, red team assessments, etc. Also, researchers have proposed different vulnerability assessment tools to help improve cyber security [17]. Among these checks, the most straightforward way is to check all known vulner-

abilities in the network and identify possible attack paths that attackers may follow.

Penetration testing (PT) is a security assessment of networks that conducts controlled attacks on a specific network and tries to identify vulnerabilities. This is one of the most common and effective methods to evaluate the security of a network since it simulates what may happen in the real-world. However, performing penetration tests requires both trained experts and time, and we have already seen a shortage in this field [25]. To alleviate this problem, tools like Zmap [10], Metasploit [14] and libraries of known exploits such as the Known Exploited Vulnerabilities (KEV) catalog have been developed to assist penetration testers to automate parts of their work, such as port scanning, payload configuration and vulnerability identification.

After identifying vulnerabilities, the next step is to compromise hosts where these vulnerabilities are detected. In the early days, people would formulate this problem using the attack tree model [3] and the attack graph model [4], which are representations of all possible paths of attack against a cyber security network. Recently, automated penetration testing, that is to use algorithms to automate the penetration testing process, has been a popular research topic and has gained much attention in Artificial Intelligence (AI) community [44]. By using various algorithms to automate the whole procedure, companies can not only save money on hiring cyber security professionals but also keep their systems secure all the time.

However, simulating cyber attacks is not an easy task. From attackers' perspective, the network is neither fully observable nor deterministic. The information attackers know about the network is partial, and attackers cannot be sure whether a specific attack will succeed. Classical planning approaches, where attackers know the whole structure of the network, and all actions that the attacker performs are deterministic can not model cyber attacks perfectly. POMDP models the problem better, which involves prior probabilities of network configurations and stochastic observations. However, the weakness of this approach is that it can not scale to the required network size due to the curse of dimensionality and the policy space complexity.

Reinforcement learning (RL), as a flexible and powerful approach for solving Markov Decision Process (MDP) problems, becomes a natural candidate for addressing penetration testing problems. Reinforcement learning is an area of machine learning concerning how intelligent an agent could be in a certain environment after enough long time of interacting with the environment. The environment is typically stated in the form of a POMDP, which switches between different states and outputs a reward given a specific action from the agent. Due to the generality, reinforcement learning can be applied to many problems including penetration testing problems, and some researchers have tried different reinforcement algorithms in this field such as Deep Q-Network (DQN) and A2C [43].

1.2 Problem

1.2.1 Problem Definition

Penetration testing requires finding possible attack paths in a network given the information of the network such as hosts, services, topology, etc. The attack paths are sequences of actions that attackers may perform on the network, and algorithms are designed as attackers to find the optimal path efficiently. When designing such algorithms, two aspects need to be considered. The first is the scalability of the algorithm, which means to which extent the algorithm can be

applied on large networks. The second is whether the algorithm requires prior knowledge about the network, which may be the topology or information about services. In a network where a defender exists, the winning ratio of the attacker against the defender is also an important metric of the algorithm.

The objective of this thesis is to test the performance of reinforcement learning algorithms on penetration testing problems and try to improve their achieved rewards on different networks. In cases where a defender exists, the objective is to maximize the winning ratio of the attacker against the defender. Another objective of this thesis is to improve the generalization ability of reinforcement learning algorithms using meta-learning ideas.

1.2.2 Research Questions

We aim to answer the following questions in this thesis:

1. Can reinforcement learning algorithms such as A2C and DDQN gain positive rewards when they are applied to penetration testing problems?
2. Can we find an appropriate formalism to model penetration testing problems and potentially improve the convergence speed and performance of reinforcement learning algorithms by incorporating meta-learning ideas?

1.2.3 Research Methodology

In order to test the performance of reinforcement learning algorithms on penetration testing problems, we simulate our own networks according to the structure of real-world communication networks, and test A2C and DDQN algorithms on these simulated networks separately. Here *test* means that we train these algorithms on a target network, wait for their convergence and find their average rewards after convergence. Using other criteria such as the number of compromised hosts is also applicable, but the average reward is the most commonly used criterion in reinforcement learning papers, and is also much more intuitive.

To improve the performance of reinforcement learning algorithms, we incorporate meta-learning ideas and build our model called MetaNet. Then, we train this model on simulated networks, and compare its performance and convergence speed with other reinforcement learning algorithms that we have tested before. We also test its generalization ability by pre-training the model on multiple networks, and then test the speed of convergence on target networks.

Related works are found mainly on the Internet such as topics regarding red teaming using reinforcement learning on Google Scholar, technical journals such as IEEE, online databases such as PNNL as well as references of these papers. There is some literature on this research topic, but few have similar network structures as in this thesis. Since we can not find a solid baseline, we focus on comparing the stability and convergence speed of A2C and DDQN to get a review of their performance.

1.2.4 Contribution

In this thesis, reinforcement learning algorithms are adopted for automated penetration testing, where algorithms such as A2C and DDQN are used as our test algorithms to find out the poten-

tial application of reinforcement learning algorithms on penetration testing problems. We also try to incorporate meta-learning ideas into these algorithms to boost the generalization ability and the training speed. The contributions of this thesis are:

1. Demonstrate the application of reinforcement learning algorithms on penetration testing problems, showing that reinforcement learning algorithms are capable of finding the optimal attack path in a network.
2. Add a defender which uses *defend minimally* strategy to be the opponent of these algorithms, showing that reinforcement learning algorithms can achieve a satisfying winning ratio against the defender after enough time of training.
3. Incorporate the idea of meta-learning, propose MetaNet and show the potential application of meta-learning on penetration testing problems.

1.3 Ethics and Sustainability

All experiments and corresponding results in this thesis are for research only, and should not be used to exploit communication network systems in real-world. Our main aim is to understand the behavior of attackers, and this will contribute to the development of defensive strategies in the future. Also, in order to prevent potential malicious usage of our code, we do not publish the code in this thesis.

Networks used in this thesis are simulated networks, and no actual network system is involved during the experiments. Thus, using our approach requires prior knowledge about the target network. This implies that attackers who can not gain additional knowledge about the target network by scanning will have difficulty in using our model, while defenders will be able to model and study the vulnerabilities of their networks.

This thesis also contributes to social justice and sustainability. The studies about penetration testing would help companies learn methods and algorithms that attackers may use, thus preventing attackers from achieving their final goal, i.e., compromising critical services. This prevents financial losses for companies, and improves social justice by decreasing data leaks from public services.

1.4 Thesis Organization

This thesis is structured as follows. Chapter 2 explains the background of penetration testing and reinforcement learning. Chapter 3 formally defines the problems that we aim to solve in this thesis. Chapter 4 explains the methodology that we use in this thesis, including the structure of MetaNet. Chapter 5 presents the results of this thesis. Chapter 6 summarizes this thesis with conclusions.

Chapter 2

Background

This chapter explains the background knowledge needed for this thesis, which mainly consists of two parts: penetration testing and machine learning. In the first part, we provide some basic knowledge of penetration testing and introduce some tools as well as traditional algorithms for penetration testing problems. In the second part, we first explain some principles of neural networks, and then introduce some reinforcement learning algorithms such as DQN and A2C and meta-learning algorithms such as Model-Agnostic Meta-Learning (MAML).

2.1 Penetration Testing

2.1.1 Introduction

Penetration testing has been a critical step in testing network systems to ensure the system's robustness and improve information security. It performs controlled attacks toward the target network aiming to gain access to sensitive information on specific hosts. There may exist various methods for penetration tests, but generally they consist of four steps: information gathering, exploitation, privilege escalation and cleaning up [30].

Information gathering A penetration test usually starts with information gathering, where attackers try to collect relevant information about the network in hope of finding some vulnerabilities. Scanning tools such as Nmap are frequently used to monitor the traffic, scan ports and detect operating systems [32]. During information gathering, attackers may find the best action to get control of a specific host, which can be a service or an application having some vulnerabilities.

Exploitation Malicious actions against the service or the application are called an *exploit*. The aim of an exploit is to gain control of a specific host or to cause abnormal behaviors in the network, and common exploits include Structured Query Language (SQL) injection [7], Secure Shell (SSH) attacks [11], etc.

Privilege escalation Once attackers have gained control of a specific host, the next step is to perform privilege escalation, where attackers use various local exploits to try to gain administrative privileges and to expand their action space. The penetration then continues from the

compromised host and goes deeper into the rest of the network until attackers reach the final target host.

Cleaning up The final step is to clean up all evidence of penetration in order to avoid legal affairs and to perform the same penetration next time. One of the most common actions of cleaning up is to delete security log files. Once a successful attack is found during the penetration test, the whole attacking process and related vulnerabilities will be reported to system administrators. After the penetration test, necessary actions will be taken to prevent such attacks and fix the vulnerabilities.

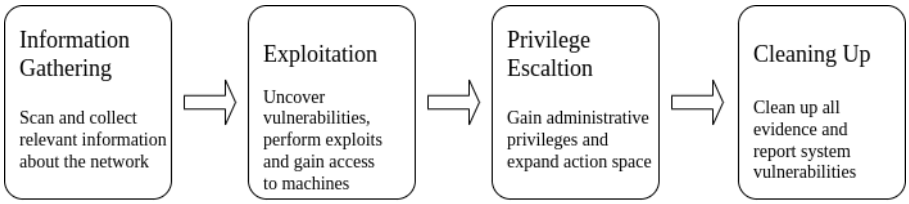


Figure 2.1.1: General procedures for a penetration test. Depending on different scenarios there might be additional stages such as gathering intelligence to better understand how the network works and its potential vulnerabilities.

Due to the difference of target networks, there may be additional steps when performing penetration tests. For example, at the beginning of a penetration test, penetration testers may need to define the scope and goals of a test. On large networks, they may need a high-speed port scanner to find out all services [33].

There are different penetration testing methods available, and each of them has their own traits. Depending on the position where attackers start their exploits, penetration tests can be divided into external testing and internal testing [30]. The external testing is conducted outside the network, while the internal testing is conducted from the internal of the network. The internal attack does not necessarily need to be a malicious employee, and it can be that an attacker uses a credential stolen from an employee by phishing attack.

2.1.2 Traditional Algorithms

Attack Graph Model

Automated attack planning is an essential part of penetration testing, which aims to automatically find all possible attack paths to sensitive hosts on a specific network. Schneier first proposed an attack tree model to represent the security of a system, where the root node is the goal and the paths from leaf nodes to the root node are different ways of achieving the goal [3]. However, this model only supports single-objective scenarios, which does not accommodate multi-objective scenarios. Here, single-objective means that the target of the attacker is a single host, while multi-objective means the target may be multiple hosts lying in different places in the network.

Sheyner et al. proposed the attack graph model that represents all possible attack paths on a network [4]. The formal definition is given as follows:

An attack graph is a tuple $G = (S, T, S_0, S_s)$, where S is a set of states, $T \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, and $S_s \subseteq S$ is a set of success states.

Sheyner et al. also proposed an algorithm for constructing the attack graph. The first step is to determine the set of states S_r which are reachable states from initial states S_0. Then, the algorithm computes the set of states S_{unsafe} which have a path to an unsafe state. By restricting the transition relation R to states S_{unsafe}, the algorithm finally returns an attack graph. The paper also gave an intrusion detection example and the corresponding attack graph generated by this algorithm.

The attack graph model has been a key model in the penetration testing field. However, both attack tree and attack graph model can be enormously large even for modestly-sized systems [4], and they require full knowledge about the system. Also, the vanilla attack graph model can not address dynamic networks where a defender is involved. Kordy et al. proposed attack-defense trees (ADT), an extension to attack trees, to analyze complex security and privacy problems [8].

POMDP Model

To have a more general formalism of penetration testing problems, researchers turn to the MDP model to formulate penetration testing problems. A MDP is a tuple (S, A, tr, R), where S is a set of states, A is a set of actions, $tr(s, a, s')$ is the probability of transitioning from a state s to a state s' using action a and $R(s, a, s')$ is the reward for executing action a in state s arriving at state s'. A MDP is a mathematical model of sequential decision, which is an extension of Markov Chains.

POMDP is a variation of MDP, where the agent can not directly observe the underlying state of the system. A POMDP is a tuple $(S, A, \Omega, tr, O, R, b_0)$, where S, A, tr, R are the same as MDP, Ω is a set of possible observations, $O(a, s', o)$ is the probability of observing $o \in \Omega$ after executing action a, arriving at state s', and b_0 is the initial belief, that is a probability distribution over the possible initial states.

The state is the set of compromised security conditions where each condition represents the configuration of the network, the state of its hosts and the state of services on its hosts. The configuration includes the operating system, open ports, running services and other information for each host. The initial belief is hence a probability distribution over possible network states.

Although the POMDP-based approach has achieved great success, it still suffers a lot from large state space and action space, which make POMDP difficult to solve. Researchers have proposed different methods to reduce its complexity while acquiring relatively precise results such as approximate POMDP solvers [5].

2.2 Machine Learning

2.2.1 Artificial Neural Networks

Artificial Neural Network (ANN)s are structured computing nodes designed by computer scientists that are meant to simulate human brains. An ANN can be regarded as a black box, which produces desired outputs given certain inputs. Backpropagation is an algorithm instructing how an ANN should be trained under different circumstances. Nowadays, ANNs have been

applied to a wide range of industries to boost efficiency, and researchers continue to explore their potential to their best.

ANNs are composed of computing nodes called *artificial neurons*. The basic structure of a neuron is shown in Figure 2.2.1.

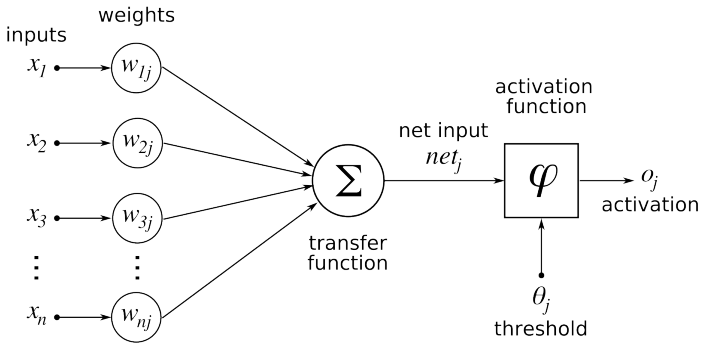


Figure 2.2.1: Structure of an artificial neuron with activation function. The structure is designed to mimic neurons in human brains but a significant performance gap exists. A single artificial neuron cannot learn XOR function.

There are different variants of neural networks. ANNs are the simplest version of neural networks, whose neurons pass information in one direction until the output layer. Recurrent Neural Network (RNN)s are a variant of neural networks, where the output from one node can be passed as the input to the same node, creating a cycle within the model. An RNN has an internal state to process sequences of inputs, which makes it useful in time series prediction problems.

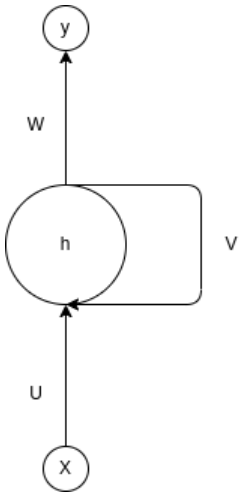


Figure 2.2.2: Structure of an RNN, where X is the input, y is the output and U, W are weight matrices. Each cell has a hidden state h which is involved in calculating the next hidden state.

Figure 2.2.2 shows the basic structure of an RNN network. Each cell has a hidden state, which is calculated by the combination of the current input and the previous hidden state. This mechanism grants RNN the ability to convey information between inputs, which helps deal with sequential problems that ANNs can not solve.

LSTM networks are a variant of RNN, which are explicitly designed to learn long-term dependencies [2]. In standard RNNs, each node has a very simple structure, while in LSTM, there

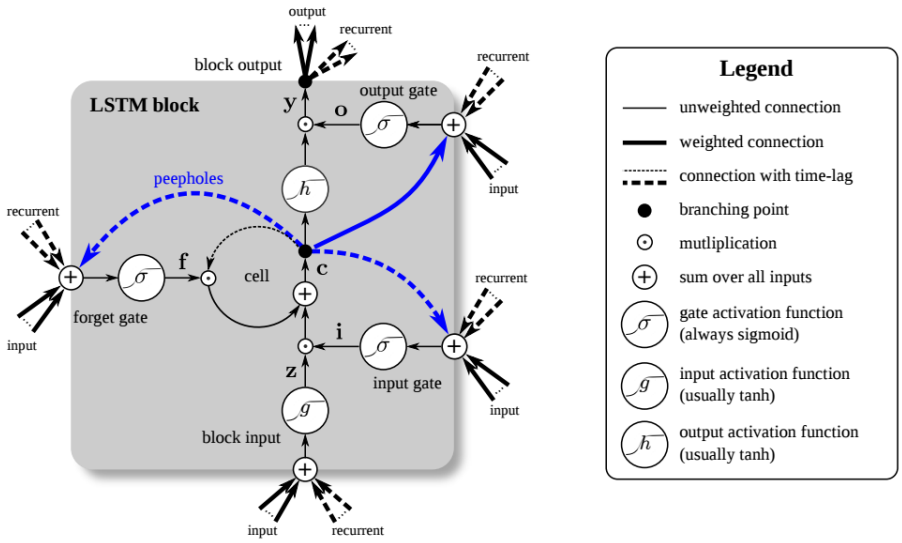


Figure 2.2.3: An LSTM unit. The unit has four input weights (from the data to the input and three gates) and four recurrent weights (from the output to the input and the three gates). Image source [23].

are three *gates* in the recurrent structure that grant LSTM extreme ability to deal with inputs with long-term dependencies.

Three gates in LSTM are *forget gate*, *input gate* and *output gate*, and each gate has its own function. The forget gate decides whether LSTM should remember or forget the information. It outputs an indicator which is multiplied by the cell state. If the indicator is 0, the output of this gate is also 0, and the information is forgotten in this cell. The input gate decides the importance of the new information carried by the input. It outputs a value between -1 and 1, and the new cell state is updated by the combination of this value, the current cell state and the output from the forget gate. Finally, the output gate decides the value of the next hidden state, which contains information from previous inputs.

2.2.2 Reinforcement Learning Algorithms

Reinforcement learning is one of the most popular frameworks that solves problems which can be expressed as MDPs. A reinforcement learning problem involves an *agent* acting in an unknown environment to achieve a goal. The agent has no prior knowledge about the environment, and interacts with the environment by trial-and-error. At each step, the agent takes a certain action depending on the *state* of the environment, and the environment gives a feedback to the agent called *reward*, and switches to another state.

The main elements of a reinforcement learning problem are:

1. an agent interacting with the environment,
2. an environment that the agent interacts with,
3. a policy that the agent follows to take actions at each step,
4. and a reward system that gives feedback to the agent after taking actions.

Figure 2.2.4 demonstrates the framework of a reinforcement learning process. Formally, re-

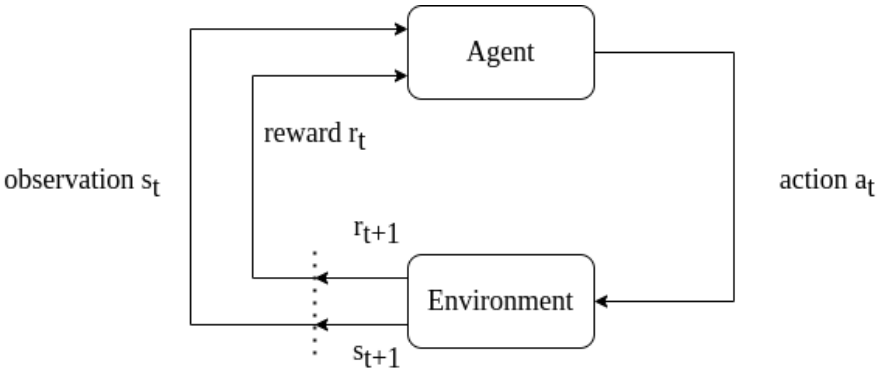


Figure 2.2.4: Reinforcement learning process. The agent takes an action according to the state of the environment, and the environment gives feedback to the agent and changes to a new state.

inforcement learning problems can be represented as a tuple (S, A, R, P, γ), where S is a set of states, A is a set of actions, R is the reward function, P is the state transition function and $\gamma \in (0, 1)$ is a discount factor. The objective of learning is to find a policy $\pi^*(s, a)$ such that the expected cumulative reward is maximized,

$$\pi^* = \arg\max_{\pi} \mathrm{E}[\sum_{t=0}^{T} \gamma^t r_{t+1}]. \tag{2.1}$$

The Bellman equation relates the optimal policy π^* to the states and the actions of the MDP:

$$V^*(s) = \max_{a} \mathrm{E}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a], \tag{2.2}$$

and

$$Q^*(s, a) = \mathrm{E}[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a], \tag{2.3}$$

where $V^*(s)$ is the maximum expected reward in state s, and $Q^*(s, a)$ is the maximum expected reward in state s when taking action a. Thus, we have

$$\pi^*(s) = \arg\max_{a} Q^*(s, a). \tag{2.4}$$

A reinforcement learning algorithm is a procedure to compute or approximate π^*. There are two classes of such algorithms: model-free algorithms and model-based algorithms. Within model-free algorithms, there are two types of algorithms: value-iteration algorithms such as DQN and policy-iteration algorithms such as REINFORCE and Proximal Policy Optimization (PPO). For model-based algorithms, there are algorithms such as Dyna-Q [1].

Deep Q Networks

Q-learning is an off-policy temporal-difference algorithm for learning state-action values. Q-learning can be implemented either in a tabular way or as a function approximation. Tabular methods store the state-action values in a table and update values in each iteration, while function approximation methods use a function to generate state-action values, and the algorithm

updates parameters of the function iteratively. This applies when the state space is large, and since a neural network is used to approximate the function, this method is also called DQN. According to the Bellman equation, the loss function for DQN is

$$L(\theta) = \mathrm{E}[(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s, a))^2], \tag{2.5}$$

where γ is the discount factor which balances immediate and future rewards.

Q-learning is an off-policy algorithm, which means it uses a different policy from the target policy for exploring actions called the behavior policy. The behavior policy is the policy used to explore the environment and collect data, and the target policy is the policy that the agent tries to learn and improve. Having two policies is beneficial because the target policy can learn from the behavior policy and get a more balanced view of the environment. But, this also makes the evaluation more challenging when the past experiences act very differently from the newer ones.

One of the problems of DQN is that it tends to overestimate the true rewards because the Q values think the agent will obtain a higher return than what the agent actually gets. To address this problem, Hasselt et al. suggest using a simple trick: decoupling the action selection from the action evaluation, which results in DDQN [18]. The Bellman equation for DDQN becomes

$$Q(s, a) = r + \gamma Q(s', \arg\max_{a'} Q(s', a')). \tag{2.6}$$

This simple trick has shown to reduce overestimations, resulting in a better policy than vanilla DQN. There is another variant of DQN called Dueling DQN, which splits the Q values into two parts and leads to a better policy evaluation [21].

Algorithm 1: DDQN

Initialize action network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, T;
for *each iteration* **do**
  for *each environment step* **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$;
    Execute a_t and observe the next state s_{t+1} and reward $r_t = R(s_t, a_t)$;
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$;
  end
  for *each update step* **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$;
    Compute target Q value $Q^*(s_t, a_t) \approx r_t + \gamma Q_\theta(s_{t+1}, \arg\max_{a'} Q_{\theta'}(s_{t+1}, a'))$;
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$;
    Update target network π parameters $\theta' \leftarrow T * \theta + (1 - T) * \theta'$;
  end
end

DDQN models have two networks. One is the *online network* and another is the *target network*. Two networks share the same structure, which is a linear Rectified Linear Unit (ReLU) stack. The only difference is that parameters in the target network are untrainable and are updated by copying parameters from the online network.

To train DDQN, the trainer keeps track of the current state, the next state, the action, the reward and a terminal indicator from the environment. Also, the DDQN trainer memorizes all history interactions and saves them into a separate memory space. Each time the trainer performs parameter updates, it samples a batch of history interactions from the memory and uses them as training data.

The DDQN training process includes three parts. First, it estimates a Q value by feeding the current state to the online network, which is called the *estimated Q value*. Then, it generates a predicted best action by feeding the next state to the online network. This best action is fed into the target network, and the output is combined with the reward to estimate another Q value called the *target Q value*. Finally, these two Q values are put into the loss function to update the parameters of the online network.

The training actually starts after certain steps when the memory space has enough records for sampling. Also, a hyperparameter called *sync_step* is used in the training process, which controls how often we synchronize parameters between the online network and the target network.

A2C

A2C is a reinforcement learning algorithm which is both value-based and policy-based. In A2C, g_t is replaced with an evaluator $A^{\pi_\theta}(s, a)$, which is called the advantage function that gives an estimate of how advantageous action a is compared to the average action.

$$\nabla_\theta J(\theta) = \mathrm{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s, a)]. \tag{2.7}$$

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s). \tag{2.8}$$

Mnih et al. tried several variants of A^{π_θ} and adopted the following form in the original paper [16]:

$$A^{\pi_\theta}(s, a) = r_t + V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t), \tag{2.9}$$

which is also the advantage function used in the project.

Algorithm 2: A2C

Initialize parameter θ and θ_v randomly;
while *not converge* **do**
  Use policy π_θ to collect a trajectory $T \leftarrow \{s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_H, a_H, r_H\}$;
  for *each episode in* T **do**
    Calculate expected rewards $V^{\pi_\theta}(s_{t+1})$ and $V^{\pi_\theta}(s_t)$ respectively using the critic network;
    Calculate advantage $A^{\pi_\theta}(s, a) \leftarrow Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$;
    Use the trajectory to estimate the gradient of the actor network $\nabla_\theta J(\theta) \leftarrow \mathrm{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s, a)]$;
    Update θ of the policy $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$;
    Update $\theta_v \leftarrow \theta_v + \beta \nabla_{\theta_v} A^{\pi_\theta}(s, a)$;
  end
end

There are two different methods that can be used to estimate $V(s)$ in A2C: the Monte Carlo (MC) method and the Temporal Difference (TD) method. The MC method learns from complete episodes and is suitable for problems with finite number of steps. The value function $V(s)$ is updated as:

$$V(S_t) = V(S_t) + \alpha[G_t - V(S_t)], \tag{2.10}$$

where $V(S_t)$ is the state value that we are going to estimate. G_t is calculated as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T, \tag{2.11}$$

where T is the termination time and γ is a discount factor.

The TD method approximates $V(s)$ by comparing estimates at two steps in time. This solves the problem that MC needs to wait until the end of the episode, and makes TD applicable to problems with long episodes. The value function $V(s)$ is updated as follows:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{2.12}$$

where $R_{t+1} + \gamma V(S_{t+1})$ is called *TD target value*.

A2C models are composed of two parts: the Actor network and the Critic network. The Actor network is trained for decision, that is to decide the next step given the input. The Critic network is trained for judging the performance of the Actor network, whose feedback will be used for training both networks.

The process that the agent interacts with the environment is the same as the DDQN algorithm. Before each episode starts, the network is reset to its initial state. At each step, both the Actor network and the Critic network take in the current state and give corresponding outputs. The output of the Actor network is a probability distribution, indicating the possibility of taking each possible action. Then a hyper parameter decides the Actor network to either explore or exploit. If the algorithm chooses to explore, then the Actor will take a random action to explore the network. If the algorithm chooses to exploit, it will sample an action according to the probability distribution and try to exploit a certain host. The output of the Critic network is a single value, which will be used to calculate the advantage later.

In this thesis, the structure of the Actor network is a linear ReLU stack, which is composed of several linear layers connected by ReLU activation functions. The default number of layers is three, and the default layer size is set as 24 for the current testing. At the end of the Actor network, another Softmax layer is added to normalize the output result. The structure of the Critic network is similar to the Actor network, which is also a linear layers stack connected by ReLU, except that the output of the last layer is a single value.

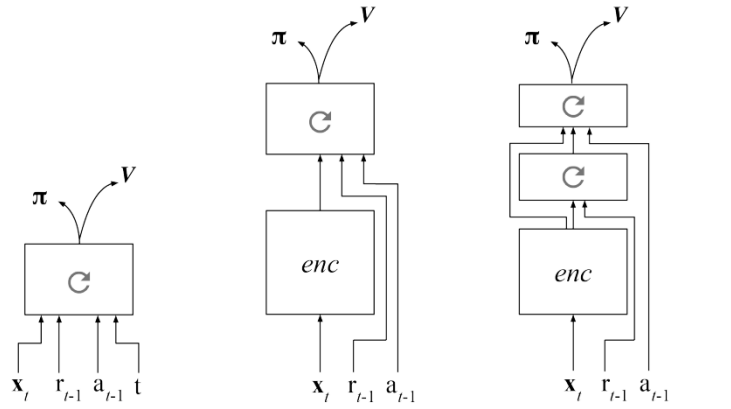
2.3 Meta Learning

Meta-learning has been another important research branch besides reinforcement learning and aims to overcome some shortages of reinforcement learning. Common reinforcement learning algorithms rely on a great amount of training as well as an accurate reward system to be able to perform well, but these conditions are not always available. Some problems, such as training a

robot to perform a specific movement, do not have a very accurate reward system, while some other problems have a very large action space, which is difficult to deal with using existing algorithms. Such problems require quick learning and learning from past experiences, which is the key problem that meta-learning tries to solve. Meta-learning learns how to learn [27], and is an essential part of achieving general artificial intelligence. During recent years, researchers have proposed various ideas to approach meta-learning. One straightforward method is to add memory to neural networks. Santoro et al. add an external memory to their model, which stores the mapping of training data and labels [42]. This structure enables later inputs to be compared with external memory to achieve better prediction. To improve the speed of training, Andrychowicz et al. propose a method that can predict the gradient descent, which results in a better optimizer than Adam [15]. The attention mechanism is also used in meta-learning [19].

Meta-learning can also be applied to reinforcement learning algorithms. By adding additional inputs such as action and reward from the previous step into the next step to force the algorithm to learn some connections between steps [20]. The overall configuration of this method is very similar to vanilla reinforcement learning training procedures, except that two additional values, the last action a_{t-1} and the last reward r_{t-1} are also fed to the reinforcement learning algorithms as inputs. In reinforcement learning, algorithms take actions according to the state s_t, while in meta-learning, algorithms take actions according to a combination of the state, the last action and the last reward (s_t, a_{t-1}, r_{t-1}). The idea of this design is to force algorithms to adjust their strategies according to dynamics between states and learn some task-level knowledge. This is achieved by adding a RNN structure to the network, and the overall structure of the model is shown in Figure 2.3.1.

LSTM serves as a memory for tracking dynamics between different states, and A2C or Asynchronous A2C (A3C) are the actual reinforcement learning algorithms for learning.



(a) LSTM A2C (b) LSTM A3C (c) Stacked-LSTM A3C

Figure 2.3.1: Add LSTM structure to A2C and A3C and feed past actions and rewards into the model. Image source [20].

Another method to apply meta-learning to reinforcement learning algorithms aims to learn a base model on multiple tasks to find a model with good initial parameters. Finn et al. propose an algorithm named MAML to find initial parameters for a model which makes it adapt to new tasks [22]. Figure 2.3.2 shows how a model is trained using MAML.

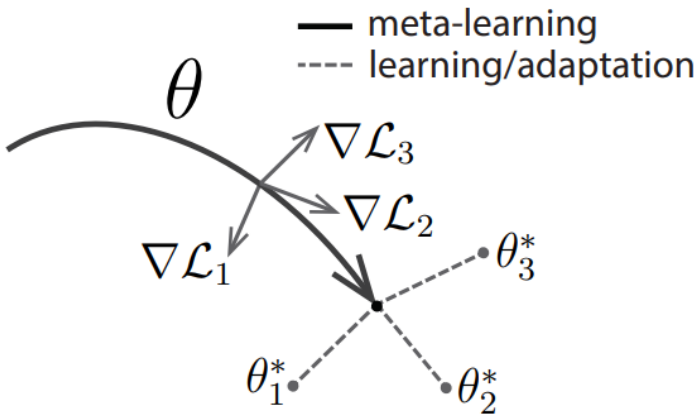


Figure 2.3.2: Diagram of MAML which tries to find initial parameters θ that enable the model to adapt to new tasks within little additional training. Image source [22].

Algorithm 3: MAML

Require: $p(\mathcal{T})$: distribution over tasks;
Require: α, β: step size hyperparameters;
Randomly initialize θ;
while *not done* **do**
  Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$;
  for *all* $\mathcal{T}_i$ **do**
    Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples;
    Compute adapted parameters with gradient descent $\theta_i' \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$;
  end
  Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$;
end

The training of MAML is based on tasks. A task is a fixed number of data sampled from the training set. We start training MAML by initializing the training parameters of the model randomly. We then sample a batch of tasks, and for each task, we copy the original model and update the parameter of the copied model using the task. This update can be done multiple times. After this step, each copied model is tested using a test set, and we get an overall loss function which is the sum of loss functions of each copied model. We update the original model using the overall loss function, and repeat the whole procedure from sampling a new batch of tasks. The trained model is then fine-tuned using a different task and should perform well after only several steps of fine-tuning.

2.4 Transfer Learning

Transfer learning is a machine learning method where existing pre-trained models are used for new tasks. This method is usually used to reduce the training time of the model, especially when the model is large. For example, BERT [28] is a language model designed by Google for Natural Language Processing (NLP) problems. To use this model on a new problem which requires classifying financial articles, a common method is to add a new layer at the end of

BERT and fine-tune the new model using financial articles. By using transfer learning, models usually get a higher start and a more smooth training procedure.

Meta-learning shares a lot of similarity with transfer learning (TL) [9]. Basically, they both aim to improve the generalization ability of models and provide a good start for training on other problems, but they take different routes. Transfer learning tries to find optimal parameters on one task, and then fine-tunes these parameters directly to different tasks, while meta-learning tries to find optimal initial parameters, and these initial parameters will be first applied to a model and then be trained to optimal to address different tasks.

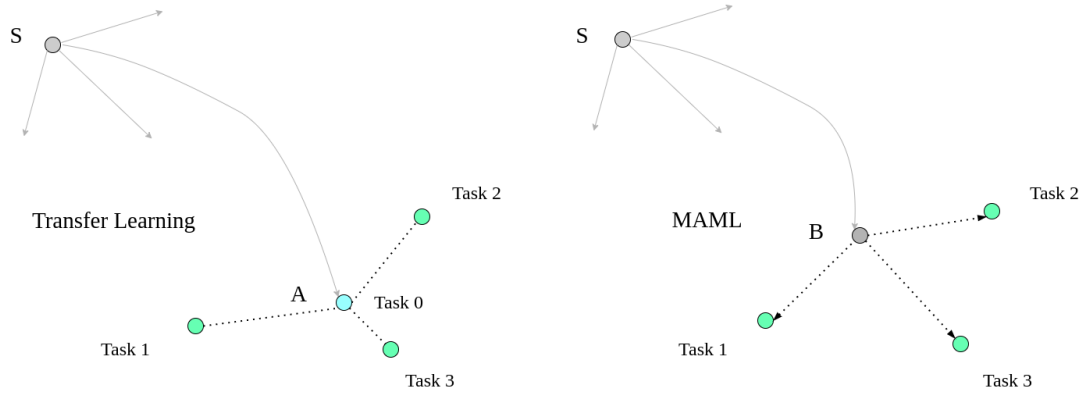


Figure 2.4.1: Difference between transfer learning and MAML. Transfer learning does not take the following adaptation into consideration, only to find an optimal point A on all tasks. MAML aims to find optimal initial parameters for the model that ensure optimal performance after adaptations on each task.

Figure 2.4.1 shows the difference between meta-learning and transfer learning. Transfer learning does not consider adaptation and tries to find a point A which has the shortest distance to all three tasks. This point has optimal performance on these three tasks overall, but does not guarantee to have optimal performance after adaptation on each task. Meta-learning, however, tries to find a point B that makes the model have optimal performance on each task after adaptation since it considers the second derivative and updates gradient by gradient.

2.5 Related Work

Different solutions for automated penetration testing have been proposed over time. Schneier et al. first propose an attack tree model [3] to find possible attack paths of communication networks. The shortcoming of this model is that the network can only have a single target. To model the scenario where a network has multiple targets, Sheyner et al. propose the well-known graph model [4], where they provide a new algorithm to automatically generate and analyze the attack graph of a network. The attack graph model has been a key model in the penetration testing field, and it has been improved and refined to meet various requirements. However, this attack planning method needs full knowledge about the network, that is attackers need to know the configuration and topology of the network before exploitation.

To take uncertainty into account, POMDP was incorporated to solve penetration testing problems. Sarraute et al. model such problems using POMDP, which grounds penetration testing problems a well-researched formalism [13]. Also, Hauer et al. use POMDP to verify the security of industrial control systems [24]. Despite its ability of modeling problems better, POMDP

requires much computation to solve, and solving POMDP for large networks is not feasible. Researchers have proposed different ideas to take advantage of its ability while reducing computation complexity. One method is to introduce intermediate models such as partially observable contingent planning [26]. Another method is to divide large networks into smaller ones and solve each network using POMDP [6]. Apart from the methods mentioned above, Obes et al. described penetration testing problems using two files in Plan Domain Definition Language (PDDL) and adopted a classical planning algorithm to find attack paths [12], which scales up to medium-sized networks with hundreds of hosts. Kim et al. propose an active learning approach to prioritize alerts generated by intrusion detection systems so as to maximize real-time situational awareness, which is tested to be robust to both false alerts and investigation errors [45].

In recent years, researchers have turned to machine learning algorithms to solve penetration testing problems. Schwartz et al. implement ϵ-greedy DQN and try to find out if reinforcement learning algorithms are suitable for penetration testing problems [31]. Zhou et al. propose a network information gain-based automated attack planning (NIG-AP) algorithm, where they define network information gain and use it as the reward [34]. Hammar et al. try different reinforcement learning algorithms around the use case of intrusion prevention and propose an opponent pool to increase the diversity of policies [40]. Panfili et al. also try multi-agent scenarios and propose a game-theoretical approach for cyber security insurance [29]. In the first part of this thesis, we formulate a network and compare the performance of A2C and DDQN algorithms.

Meta-learning has also been widely used in cyber security area [47]. Olowookere et al. propose a framework that combines the potentials of meta-learning ensemble techniques and cost-sensitive learning paradigm for fraud detection [37]. Huang et al. propose a meta-learning-based computation offloading algorithm for dynamic computation tasks in mobile edge computing networks [41]. He et al. address the problem of predicting the network traffic by proposing a meta-learning scheme that consists of a set of predictors. However, most use cases of meta-learning are classification problems, and little research has been done in penetration testing area. In the second part of this thesis, we further our work and incorporate meta-learning ideas to increase the generalization ability of our models, and demonstrate how meta-learning can be applied to penetration testing problems.

Chapter 3

Problem Definition

In this chapter, we define the problem that we aim to address in this thesis, including the environments and their assumptions, prerequisites, rewards and objectives that we need to achieve.

3.1 Environments

3.1.1 Networks

Networks Networks used in this thesis have a hierarchical structure, and each component is composed of smaller components. From top to bottom, these components are networks, hosts and services. A network is a comprehensive system that provides many services for public usage. In reinforcement learning problems, it serves as the environment that the attacker interacts with.

Hosts Hosts, i.e. physical machines comprise the network. Hosts are the minimal units of the network that can be any device such as a laptop, a smartphone, etc. To compromise a host, the attacker usually exploits services on a host and places a backdoor for future access and steals information.

For simplicity, in this thesis, hosts are identified by their unique IP addresses, and we assume that there are no private IP addresses. The attacker distinguishes different hosts by their IP addresses, by which he determines whether a host is the target host. However, the attacker initially does not have access to all hosts since some hosts are unreachable. Notice that the premise of compromising a host is to have access to that host. A host becomes reachable once the attacker compromises one of its adjacent hosts.

Services Each host runs different services such as web services and database services. For instance, within a subnetwork, there are servers running web services that can be accessed outside the company network.

Running services has potential risks since the attacker can take advantage of the vulnerabilities of the services to attack the host. If a service on a host has some known vulnerabilities, the

attacker may perform some exploits and gain privilege on the host. For example, if the attacker detects that an old version service is running on a host that has a severe vulnerability, the attacker can easily exploit the service and gain access to the host.

However, not all services have the same difficulty to compromise. Some vulnerabilities require more time and energy to exploit, while others require less time. Also, the network administrator may notice some vulnerabilities and have put some patches, which can lead to failure of exploit. Considering this, we assigned a value to each service, indicating the difficulty of compromising each service. The greater the value is, the more difficult exploiting the service.

The environments that we use in this thesis are abstraction of communication networks, that is simulated networks. We use two different network structures in our experiments: *passive networks* and *active networks*. They share many similarities in structure only with small differences.

3.1.2 Passive Network

Structure

Passive networks are *passive* since there is no defender involved. The structure of passive networks is the same as the one introduced in the previous section, except that hosts are grouped as subnetworks. This mimics the actual network architecture in companies. Figure 3.1.1 demonstrates the overall structure of passive networks.

Subnetworks A subnetwork structure is shown in passive networks. Subnetworks consist of hosts and are connected by routers, through which they transmit data and communicate. The subnetwork structure indicates the common structure of large networks. For example, a company has multiple departments, and each department has its own network. These networks are connected to build the whole network of this company, and networks for each department become subnetworks. The subnetwork structure makes it convenient for network administrators to monitor, manage and control the network, and makes it easier to conduct defense against malicious attacks.

Hosts within the same subnetwork have direct access to each other. Thus, from the attacker's perspective, compromising a subnetwork is equivalent to compromising any one of the hosts within this subnetwork. Once a subnetwork is compromised, the attacker has access to adjacent subnetworks through routers. The final target of the attacker is to gain access to the subnetwork that the target host lies in and eventually compromise the target host.

Problem

The passive network builds an environment for a typical reinforcement learning problem, where the agent tries to find an optimal action policy to compromise the target host by interacting with the environment and updating its policy with the feedback received. In this scenario, the agent is the attacker, the environment is the simulated network, and the feedback is the reward such as successfully compromising a host. We keep track of the average reward that the attacker receives at the end of each epoch, and consider that the attacker has gained the ability to compromise the network if the attacker can constantly get a positive reward, that is the algorithm converges to a positive reward after training.

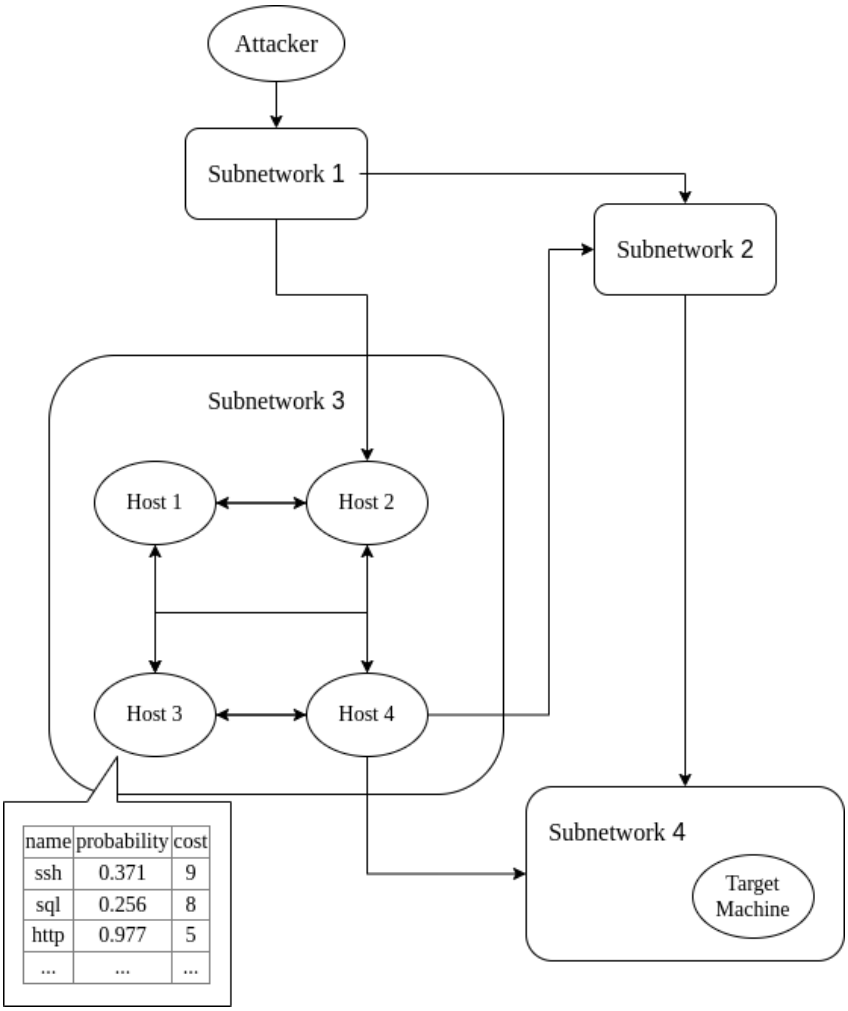


Figure 3.1.1: Structure of passive networks with subnetworks. Hosts are grouped and connected through subnetworks. Hosts within the same subnetwork have access to each other, and compromising the subnetwork is equivalent of compromising any host in the subnetwork.

Assumptions

There are some assumptions in this problem definition:

1. The attacker has no information about the network, but it can scan all hosts adjacent to an already compromised host and get information about them such as existing services.
2. The information of a host does not change whether it has been scanned or compromised by the attacker or not.

3.1.3 Active Network

Active networks are *active* because we introduce a defender into the network. In these networks, instead of having an attacker, we add another agent called *defender*, and two agents take actions simultaneously. The defender can not directly observe the moves of the attacker, but it can speculate the moves by monitoring the changes in the hosts' states. This scheme forms an adversarial model, where two actors encounter each other. This makes the whole environment dynamic since the attacker needs to take different actions according to the defender.

Structure

The structure of active networks is roughly the same as the structure used in [36], and has been fully described in Section 3.1.2. Different from passive networks, active networks have no subnetworks. This is because during our experiments, we find out that our model is only able to deal with small networks, usually with 4 hosts, which can not form meaningful subnetworks. Thus for simplicity, we eliminate the subnetwork structure in active networks.

The structure of active networks is a Directed Acyclic graph (DAG), and each host within has two statuses, the attack status S^A and the defend status S^D. The attack status and the defend status are both arrays, where each element represents the value of a specific service, and they are invisible to each other agents. The attack status is initialized to all zeros, representing that the network has received no attacks at all at step 0. At each step, the attacker chooses to attack a specific service on a certain host, and the corresponding value in the attack status is increased by 1. The defend status is initialized to non-zeros, with each value representing the resistance ability of each service. Similarly, the defender can also increase a certain element in the defend status by 1 at each step, meaning that the defender chooses to defend this specific service. This corresponds to that the defender takes various actions, for example, fixing vulnerabilities, to boost the resistance ability of a certain service.

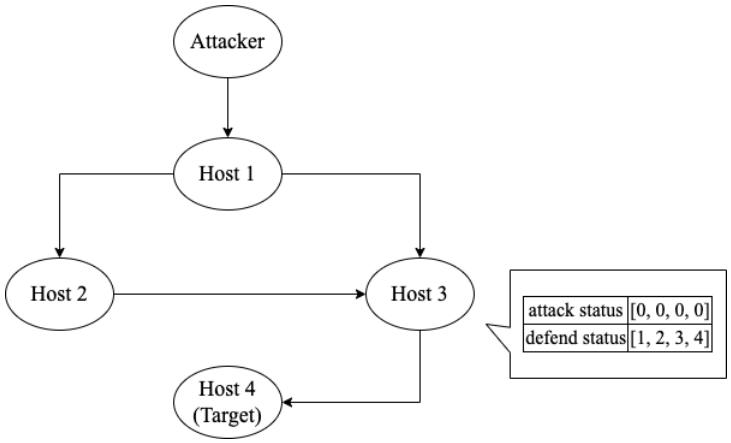


Figure 3.1.2: Structure of active networks. Hosts are connected with each other directly. Each host has a unique attack status and defend status.

For example, the initial attack status for a network containing two hosts with four services is [[0,0,0,0], [0,0,0,0]], and the initial defend status is [[1,2,3,4], [4,3,2,1]]. At step 1, the attacker chooses to attack the second service on the first host, and the defender chooses to defend the third service on the second host. Then at the beginning of step 2, the attack status for this network becomes [[0,1,0,0], [0,0,0,0]], and the defend status becomes [[1,2,3,4], [4,3,3,1]]. A host is compromised by the attacker when one of the services on this host is successfully compromised, which means the value of the attack status is greater than the defend status. In this case, the attack status for the second service on the first host is 1, while the defend status is 2, thus this service is not successfully compromised.

A host also has a separate value called *detect value*, which is a single integer. This value controls the probability that the attacker is detected by the defender when the attacker fails at each step. The greater this value is, the greater chance that the attacker will be detected by the defender.

To limit the action space of the attacker and the defender, we set a maximum value for the attack status and the defend status, represented by the hyperparameter W. Once this value is reached for a certain service, the attacker and the defender can not attack or defend this service anymore.

Defender

The defender aims to defend hosts in the network and tries its best to detect the attacker at each step. Since these two targets must be fulfilled at the same time, the defender can take different actions at a single step, that is to either defend a certain service on a certain host that might be under attack, or increase the detect value to improve the detectability of a certain host. A good defender should be able to balance these two targets and lower the winning ratio of the attacker as much as possible. Using the definition in 2.1, if S is a set of states, A is a set of actions, R is the reward of the attacker, then the objective of defenders is to minimize the total reward that the attacker can achieve:

$$\pi^* = \arg\min_{\pi} \mathrm{E}[\sum_{t=0}^{T} \gamma^t r_{t+1}]. \tag{3.1}$$

In this thesis, to make the defender simpler and make it easier for the attacker to explore the network, we only allow the defender to increase the defend status of the network. This is because, during the exploration phase, a high detect rate will dramatically lower the speed of exploration of the attacker, which leads to slow training of the model.

In this thesis, we use the simplest version of the defender, which is called *defend minimally*. At each time step, the defender will check all defend statuses in the network, choose the defend status that has minimal value, and increase its value by 1. This is a simple method, which ensures that the attacker can not compromise a host easily. The drawback of this defender is that the attacker can easily learn the pattern of the defender and thus learn an efficient method to cope with the defender. Also, the defender can not distinguish whether a host has been compromised and decide whether or not to defend the host afterward. There are also other defenders such as *defend randomly* which chooses a random service to defend, and *trained defender* which is a neural network trained to minimize the winning ratio of the attacker. For simplicity, we only take *defend minimally* as our test defender in this thesis. More complex defender models, such as defenders that use Bayesian inference for maintaining a belief about the attacker's progression [45] are outside of the scope of this thesis.

Problem

The active network constructs the environment of another form of reinforcement learning problem, compared with the passive network, where the underlying state of the environment is constantly changing, and the agent needs to update its policy by feedback not only from the environment but also from another agent. In this scenario, the agent is the attacker, the environment is the simulated network, and another agent is the defender. The objective is to maximize the winning ratio of the attacker at the end of each epoch, which is also used as an indicator to determine whether our model has converged or not.

Assumptions

Besides the assumptions that we make in passive networks, there are some additional assumptions regarding this problem definition:

1. The attacker and defender each take one step in a state, and the attacker moves first.
2. In an episode, the attacker wins if he compromises the target host successfully, and the defender wins if the attacker is detected by the defender or the attacker exceeds the maximum step limit.

3.2 Rewards

The reward is an essential part of the network, which controls the feedback that the attacker and the defender will get when they both take an action.

At each step, the attacker can perform two different actions: scan and exploit, which incur some costs to perform these actions. Also, the attacker can receive rewards when successfully compromising a host, no matter it is the target host or not. In some cases, the attacker can also receive penalties. For instance, if the attacker decides to exploit a service that does not exist or tries to access a host that is unreachable, the attacker will get a penalty instead of a reward. Similarly, the defender can receive a reward if it successfully detects the attacker or gets a penalty if it tries to defend a service that does not exist. In this thesis, we use two different reward schemes, the *detailed reward scheme* and the *final reward only scheme*.

In the first scheme, the attacker gets detailed rewards after each step, and the reward can be defined as follows:

$$R_{detailed} = \begin{cases} R_{compromised} & \text{when a host is compromised} \\ -R_{scan} & \text{when performs scanning} \\ \dots & \end{cases} \tag{3.2}$$

$R_{compromised}$ is the reward when the attacker successfully compromise a network, and R_{scan} is the cost when the attacker performs a scanning. These rewards are constants among active networks in detailed reward scheme, which requires preset rewards as hyperparameters to guide the whole training process and the behavior of the attacker and the defender.

In the second scheme, the attacker only gets rewards if the network reaches terminal states. That is, the attacker only receives a reward if it compromises the target host after each step.

$$R_{detailed} = \begin{cases} 0 & \text{when a host is not compromised} \\ 1 & \text{when a host is compromised} \end{cases} \tag{3.3}$$

This scheme has little feedback for the training, which leads to a slow exploration and training

process since there is nearly no reward shaping, but it is easier to perform due to its simplicity.

Chapter 4

Methods

In this chapter, we introduce our own model, and describe the differences of both structure and training methods from other reinforcement learning algorithms.

4.1 A2C and DDQN

4.1.1 Networks

Test networks are generated with a script. Each network is generated using a specific seed, which controls the randomization of network formation, service probability, service cost, etc. The seed is drawn from the system time when the script runs, and each seed generates a specific network, which makes the network generation reproducible if the script is run at the same time. Following are details of the generation process.

Networks have a different number of subnetworks, which ranges from 6 to 10, and the number of hosts also varies from 30 to 50. All hosts in these networks have 6 kinds of services. The cost of compromising each service is an integer generated randomly within 1 and 10, and the probability of a successful exploration are a random number between 0 and 1. The topology is also randomized, with each edge generated randomly. Note that the generation algorithm ensures a valid path from the start point to the target host, i.e. there is always a solution for a generated network.

4.1.2 States and Rewards

The state of a passive network is the concatenation of all hosts' states within this network. Each host runs different services, and each service has a unique possibility and cost to be compromised. Also, each host has an underlying state, indicating whether this host has been compromised by the attacker. In passive networks, we use *detailed reward scheme*, where all actions the attacker may perform, such as probe or attack, have corresponding rewards. Also, successfully compromising a host gives the attacker an additional reward. Table 4.1.1 shows the rewards and penalties that the algorithm will get when it performs different actions.

Action Type	Action	Reward
Cost	Scan	-5
	Exploit a service	Defined by the network
Reward	Compromise a non-critical host (succeed)	30
	Compromise a critical host	200
Penalty	Exploit an unexisting service	-10
	Try to access an unreachable host	-30
	Compromise a compromised host again	-20
	Compromise an unscanned host	-10
	Exceed maximum actions (fail)	-100

Table 4.1.1: Reward and Penalty for A2C and DDQN

4.1.3 Training

For a single network, we run both A2C and DDQN and record the reward in each episode. The average reward is defined as the average of rewards that the attacker gets in the latest 100 episodes. We also record the total training time that each algorithm takes. The algorithm is considered convergence if it achieves positive rewards in 100 consecutive episodes. In case the algorithm never achieves positive rewards, the maximum episode of training is set as 1500.

We follow the standard training process to train the A2C and DDQN algorithms. During the experiment, we find that the TD method is too unstable for the A2C algorithm to achieve a reasonable result, thus we turn to the MC method instead, and the advantage is calculated according to Equation 2.11. The Actor network is then fed with the next state and keeps interacting with the network until it indicates that the system reaches a terminal state.

4.1.4 Hyperparameters

The hyperparameters for A2C and DDQN are listed in Table 4.1.2.

4.2 MetaNet

To further extend our work in order to find a more general model, we apply some meta-learning ideas into our model. As introduced in Chapter 2, several meta-learning ideas have been proposed to address reinforcement learning problems. In this section, we introduce ideas that we take into our model. Figure 4.2.1 shows the overall structure of MetaNet.

Additional Inputs

Neural Network	A2C		DDQN
	Actor	Critic	
Layer Size	24		
Learning Rate	1e-3	5e-3	5e-3
Batch Size	32		
Gamma	0.5		
Epsilon Decay	0.9999		
Epsilon Minimal	0.001		
Memory Size	1e5		No memory

Table 4.1.2: Hyperparameters for A2C and DDQN

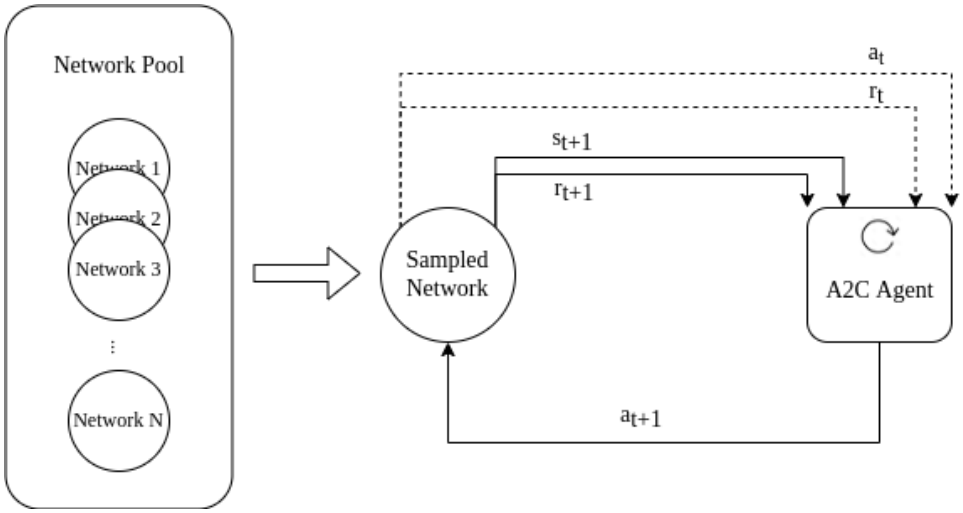


Figure 4.2.1: Structure of MetaNet, which is an A2C network wrapped with an LSTM. The trained network is sampled from the network pool, and the action and reward from the last step is also fed into MetaNet during training. As a comparison, we also propose a structure named MultiNet, which shares the same structure with MetaNet but without the additional input a_t, r_t and the LSTM wrapper.

One principle of meta-learning is to force the model to learn high-level information from inputs. To force our model to learn internal logic between steps and some task-level knowledge, we feed two additional inputs to our model: the action and reward from the last step. At the first step, these two inputs are set to 0. This idea is usually implemented with a LSTM structure serving as a memory.

LSTM Structure

Adding additional memory to neural cells is another method to make the model easier to remember the past experience. LSTM is a good choice here because it can internalize the history of inputs and tune its own weights effectively through backpropogation. The hidden state of each step serves as a memory of the environment, and the agent decides what actions to take according to the hidden state. By using LSTM, the hidden state can serve as an alternative state

of the environment if the actual state is missing. For example, the model with LSTM can still function normally under the circumstance where 80% of the input states have been missing, while the standard model will have a difficult training process.

Training on Different Environments

Another method of building a more general model is to train the model on different environments in a single training process. In this thesis, we redesign the structure of the model and make it able to interact with multiple environments during training. Before each training episode starts, the trainer will randomly select one communication network from all networks as the environment, and the agent starts interacting with the selected network until the end of the episode. In this thesis, since we are using RLlib to build our model and can not manipulate the gradient freely, we simply select one network, finish training one episode and start training on another one. This follows the training process introduced in [20].

Chapter 5

Numerical Results

5.1 Passive Network

In this section, we test the application of reinforcement learning algorithms on passive communication networks. Tested algorithms are A2C and DDQN.

Figure 5.1.1 shows the performance of these two algorithms on networks with 6 subnetworks and 6 services. Both algorithms achieve high average rewards after roughly 400 episodes of training, which indicates that both algorithms manage to find the optimal compromising path after training.

We can see that the number of hosts in the network does not increase the complexity significantly, and each algorithm only takes slightly more episodes on networks with more hosts. This may be due to the fact that a subnetwork is marked as compromised if one of its hosts is compromised. The training time of both algorithms does increase, but the complexity of networks is decided by the number of training episodes. Generally, DDQN takes less time than A2C to converge.

This time we fix the number of hosts and observe how these two algorithms perform on networks with different subnetworks. Figure 5.1.2 shows that DDQN fails on the network with 8 subnetworks (and this is the only failed case on networks with less than 10 subnetworks, the rest test cases are all converged), and they both fail on the network with 10 networks. We can see the training process of A2C is much more smooth, while DDQN is not stable enough, and the time that DDQN takes also rises when the size of networks starts to get larger.

From Figure 5.1.3, we can see that both algorithms are able to find paths for the attacker to achieve its objective given enough training provided that the network is not complicated, but they both fail to find a compromising path once the network reaches 10 subnetworks. Although they have a similar capability, generally DDQN has a faster convergence speed, while its training process is not as stable as A2C.

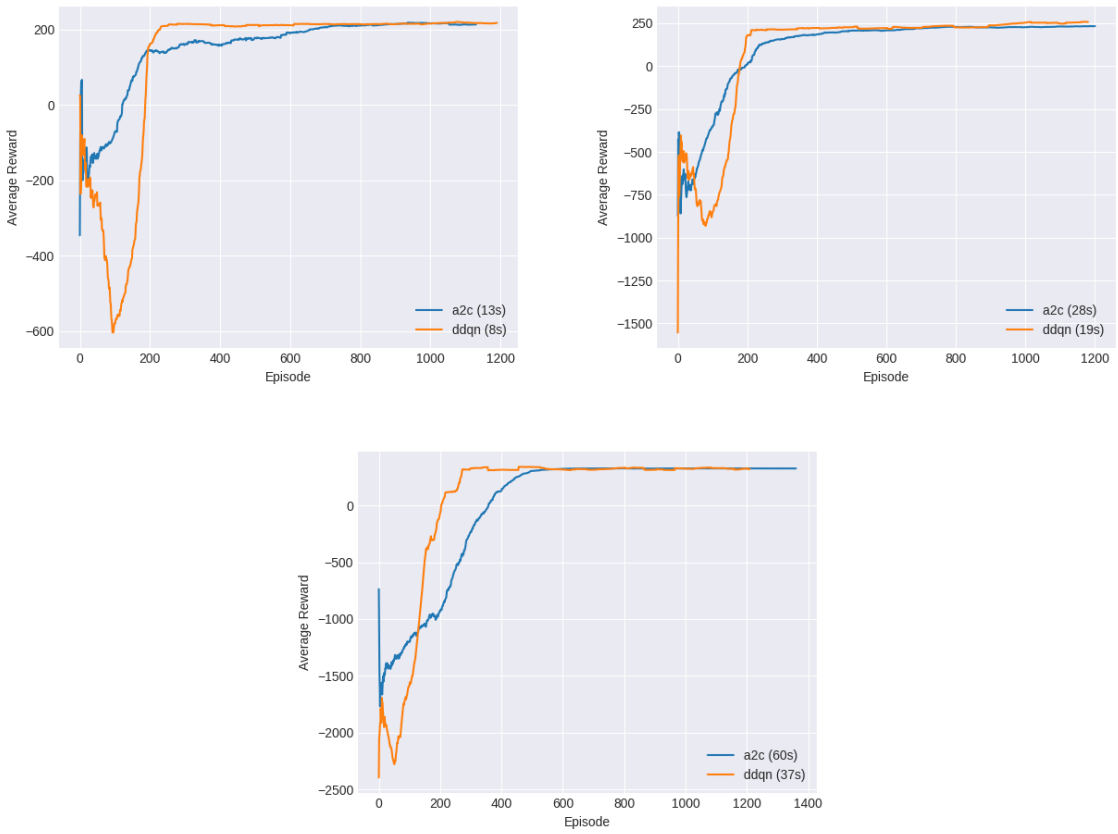


Figure 5.1.1: Performance of A2C and DDQN on networks with 6 subnetworks. The total numbers of hosts (from left to right, from top to bottom) are 30, 40 and 50 respectively. The blue line is A2C and the orange line is DDQN. Both algorithms converge successfully since they achieve positive rewards after enough training, and the number of hosts does not increase the complexity significantly.

5.2 Active Network

In this section, we test the performance of reinforcement learning algorithms on active networks. We take a simple defend strategy, defend minimally, and test its performance against the attacker. Figure 5.2.1 are the winning ratio of the attacker when the attacker uses the A2C algorithm:

We can see that A2C does not converge on all networks, and it fails to converge on Network 1 and Network 3. From the training process, the algorithm fails in the exploitation phase and does not accomplish enough success exploitation. Since the structure of these networks is all generated randomly, the failure may be due to that the structure of Network 1 and Network 3 is hard for successful exploitation.

Finally we test the generalization ability of trained models. Our method is to load the trained model from a checkpoint and apply it to other networks. Table 5.2.1 shows the generalization ability of a model trained on Network 13:

We choose Network 13 because it is also used to train MetaNet in Section 5.3, thus we can compare performance of this model with MetaNet. From Table 5.2.1, we can see that apart from Network 13 which is the network that the model is trained on, the model performs badly

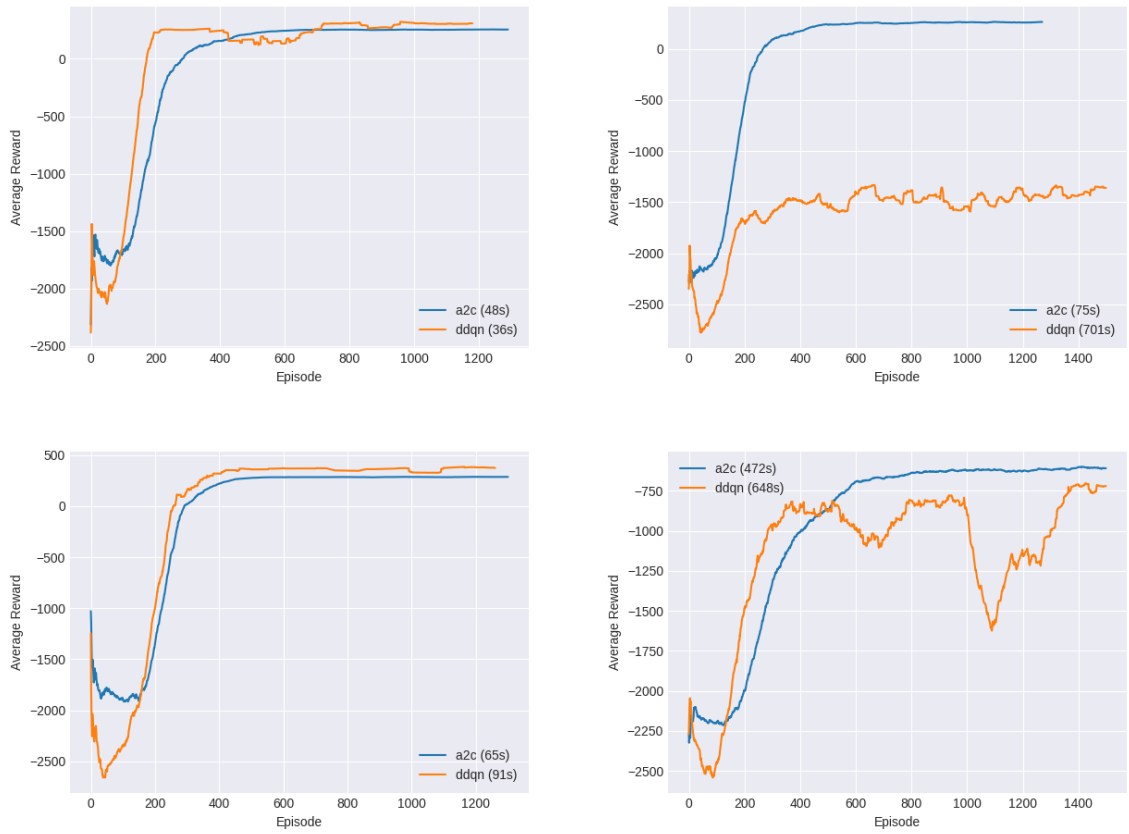


Figure 5.1.2: Performance of A2C and DDQN on networks with 7, 8, 9 and 10 subnetworks (from left to right, from top to bottom). All networks have 30 hosts in total. The blue line is A2C and the orange line is DDQN. When the number of subnetworks reaches 10, both A2C and DDQN are unable to converge since they converge to a negative reward.

Test Networks	Winning Ratio of Model Trained on Network 13
13	0.670
4	0.086
14	0.072
18	0.242
19	0.154

Table 5.2.1: Winning ratio of model trained on Network 13 on other networks

on other networks, which indicates that in this way, our trained model has little generalization ability.

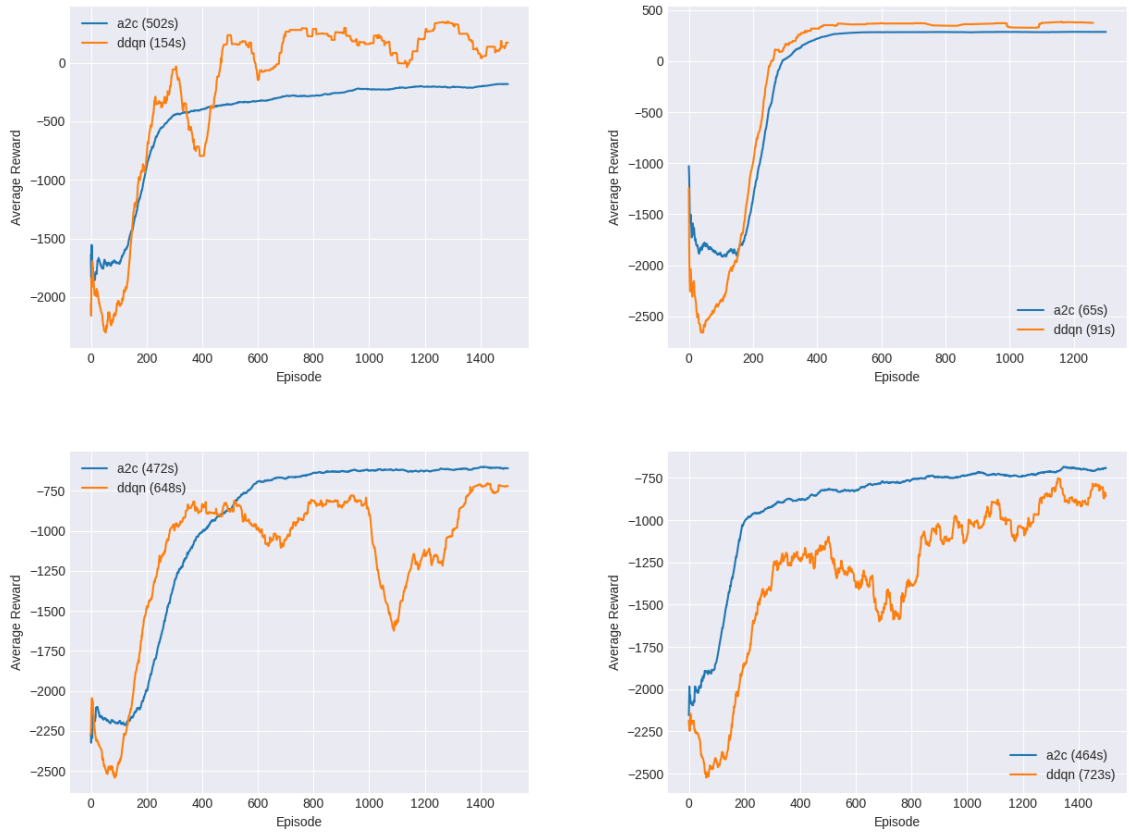


Figure 5.1.3: Stability of A2C and DDQN on different networks. The blue line is A2C and the orange line is DDQN. Generally, DDQN has a faster convergence speed, while its training process is not as stable as A2C.

5.3 MetaNet

5.3.1 Performance on Multiple Networks

Figure 5.3.1 shows the training process of MetaNet and MultiNet on Network 11, 13, 14. The reason why we train our model on Networks 11, 13, 14 instead of Networks 11, 12, 13 is that Network 12 is has a complicated topology, which results in great instability of the training process. Thus in our final experiment, we replace Network 12 with Network 14 and conduct the following experiment.

We train each algorithm for 10,000 episodes, and we can see that the training process is generally unstable, which may be caused by frequently switching between different training networks. However, the overall winning ratio of both algorithms rises along with the training.

Then we test the performance of the trained model on different networks, and the results are shown in Table 5.3.2. Each algorithm interacts with the test network for 500 episodes, and the average reward is the winning ratio of the algorithm. We choose these networks because the results on these networks shows the difference between MetaNet and MultiNet better. On other networks that are not shown here, either they have similar performance or they both achieve zero winning ratios.

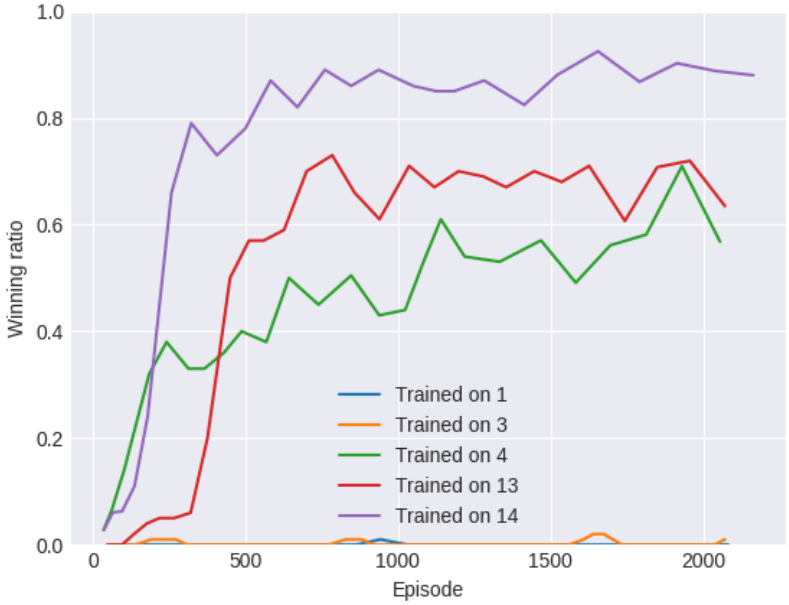


Figure 5.2.1: Winning ratio of A2C on different communication networks. These networks are generated randomly with different seeds. The model achieves reasonable winning ratios on Networks 4, 13 and 14. However, on Network 1 and 3, the model can not find a compromising path after enough training.

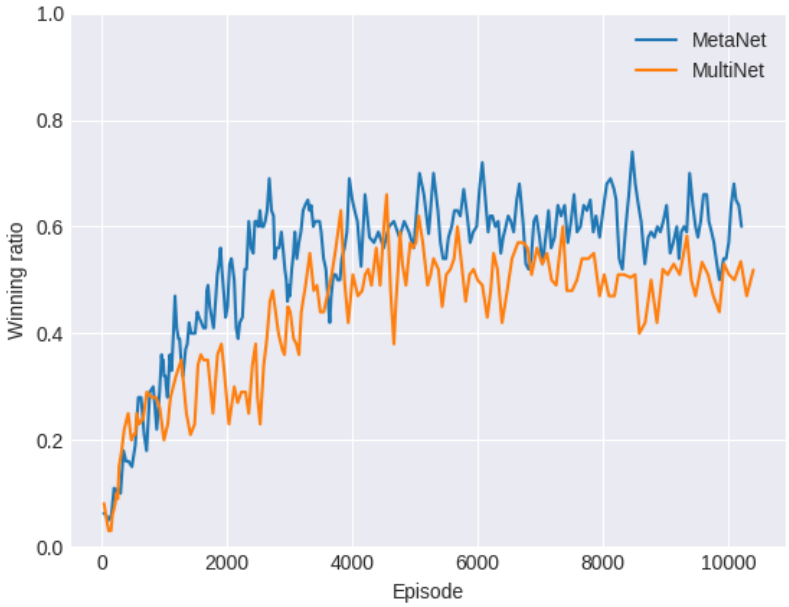


Figure 5.3.1: MultiNet and MetaNet trained on Network 11, 13 and 14. The blue line is MetaNet and the orange line is MultiNet. They have similar training process, but the winning ratio of MetaNet is slightly higher than MultiNet due to the LSTM structure.

First we test the performance of both models on the networks that they are trained on, that is their performance on Networks 11, 13 and 14. We can see that generally they perform well and achieve great winning ratios. However, MultiNet has a zero winning ratio against Network 11.

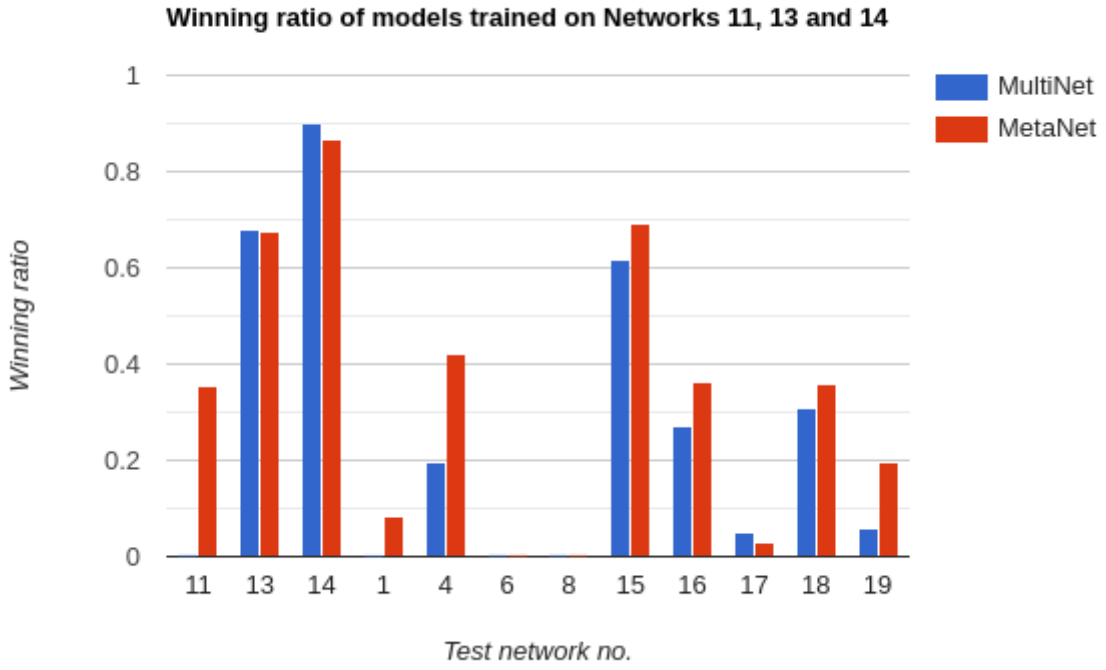


Figure 5.3.2: Winning ratio of MultiNet and MetaNet trained on Network 11, 13 and 14 which are tested on different networks. The red bar stands for MetaNet and the blue bar stands for MultiNet. Overall MetaNet outperforms MultiNet on most of tested networks.

This means that the training process of MultiNet has a great bias toward Network 13 and 14, and the model puts nearly no emphasis on Network 11.

On the networks that both models are not trained on, their performance varies dramatically among different networks. This again indicates that the structure has a great influence on the winning ratio. On some networks, for instance, Networks 6 and 8, they both achieve a zero winning ratio, which means that the policy they learned is not suitable for such network structures. However, training on multiple networks does grant the model some ability to generalize to other networks. Compared with the results in Table 5.2.1, we can see that under most cases, MultiNet and MetaNet have increased their winning ratios. For example, on Network 18, MultiNet and MetaNet improve their performance by 8% and 14% respectively. The reason for this result is that these three networks share similar structures with trained networks, i.e. Networks 11, 13 and 14 than others. Still, the winning ratios on these networks are relatively low, meaning the trained policy is partially applicable to these networks, and further training is required in order to find the optimal path on these networks.

Finally, we compare the performance of MultiNet with MetaNet. Although on some networks they both perform badly, MetaNet achieves better winning ratios on Networks 4, 11 and 19 by 35%, 23% and 13% respectively, and roughly the same winning ratio on the other networks. This indicates that generally, MetaNet performs better than MultiNet, with the help of the LSTM structure and additional inputs.

5.3.2 Performance on Single Network

Besides the generalization ability, MetaNet also brings the model some heuristics about how to explore the environment. Figure 5.3.3 demonstrates the change of winning ratios of training on Networks 12, 18 and 19. The blue one stands for training without pre-training, while the yellow one restores the model that is trained on Network 11, 13 and 14, and then trains on the target network.

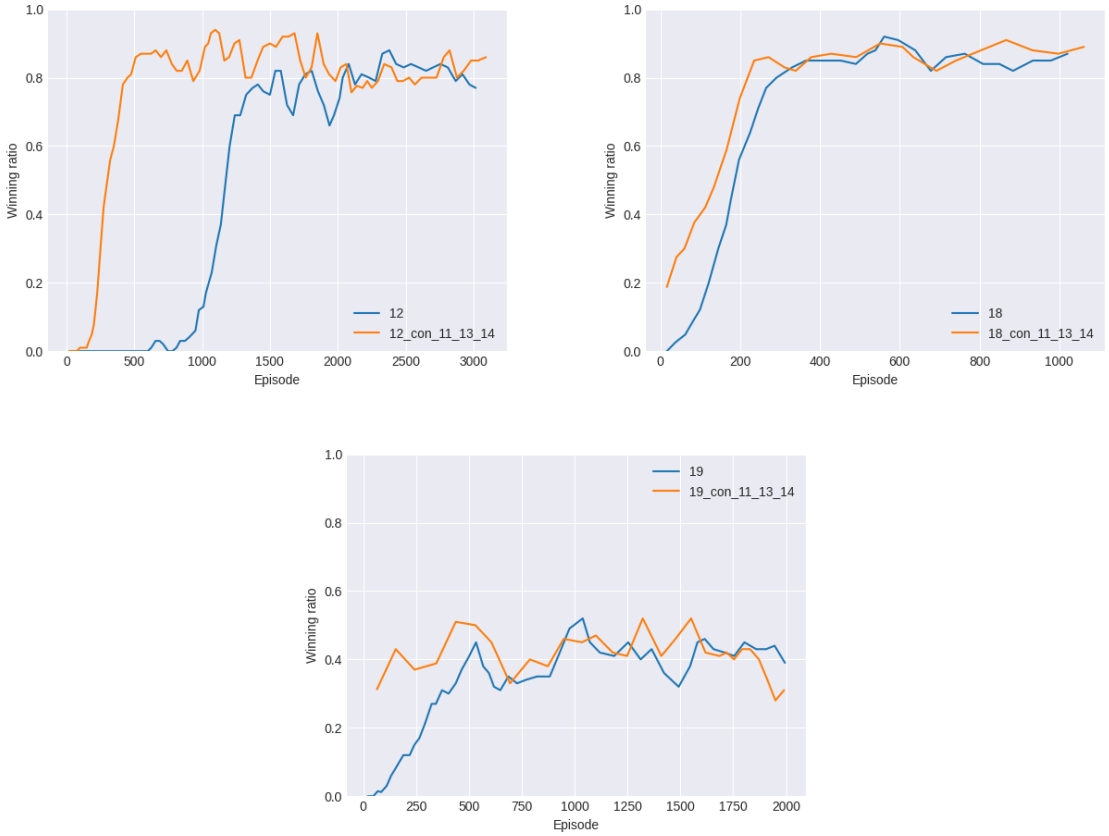


Figure 5.3.3: Comparison of training without pre-training and from a trained model. The blue line is trained without pre-training and the orange line is trained from a trained model. The exploration phase becomes much shorter when training from a trained model.

Training without pre-training always starts with a zero winning ratio, and the winning ratio slightly rises while the agent interacts with the environment. We can see the training starts rather slow, mainly because the complexity of the network that the agent fails to explore the networks. The training only begins once the agent conducts a successful attack during the exploration phase. However, if we take an already trained model, such as the model trained on Networks 11, 13 and 14, and then train the model on the target network, the exploration phase becomes much easier for the agent and the training accelerates quickly.

This experiment shows that our trained model does not only grants the model some generalization ability, but also brings the model some knowledge about exploration, which accelerate the training process on other networks.

Chapter 6

Conclusions

6.1 Discussion

This thesis demonstrates the application of reinforcement learning algorithms as well as meta-learning methods on penetration testing problems. To answer the research questions proposed in 1.2.2: Reinforcement learning algorithms can solve different variants of penetration testing problems, and meta-learning ideas can be used to prove the generalization ability of these algorithms.

We first test the A2C and DDQN algorithms on passive network environments. Within their capabilities, both algorithms gain high winning ratios after enough training. From the results, we can conclude that reinforcement learning algorithms are able to learn the optimal policy for compromising a network without knowing its structure. The algorithms will first randomly explore the network, and once they occasionally perform a successful attack, they move toward that direction until they find the optimal path.

Then we test the performance of the A2C algorithm in active network environments, where a defender tries to defend the attacker from compromising the network. The results show that A2C algorithm again achieves positive and stable rewards on the networks that it is trained on, but it is not able to generalize to other networks. This indicates that although reinforcement learning algorithms are able to solve different variants of penetration testing problems, they have little generalization ability on networks that they have never seen.

To further test the generalization ability, we apply meta-learning ideas to A2C and propose MetaNet, and a huge improvement in generalization ability is observed. The results show that by using LSTM structure and training on multiple networks, we can improve the generalization ability of our model and accelerate training on other networks.

During the experiments, we also observed some drawbacks of machine learning algorithms. The major problem with using reinforcement learning algorithms is that they have too much complexity so we only demonstrate their application on relatively small networks. Another drawback of reinforcement learning algorithms is that they need proper reward shaping to have a smooth exploration phase. In MetaNet, the attacker only gets rewards when it reaches the end of an episode, which leads to a slow exploration phase, especially when the structure of the

network is intricate.

There are also some potential improvements for this thesis. Due to the limitation of RLlib [48] that we use to build our model, we use loop training on different networks rather than using gradient average like what [22] does. This may be one of the causes of unstable training of MetaNet. Also, meta-learning has an important trait that it bonds to a certain distribution, but due to the size of the tested networks, it is not manifested in the results. This could also be a potential research topic in the future.

6.2 Future Work

There are a lot of future works for this project. First, different defenders and reinforcement learning algorithms may be tested to see their behavior on penetration testing problems. In this thesis, we focus on testing if reinforcement learning algorithms are capable of addressing penetration testing problems, and defend minimally is the only defender tested. In the future, other high-level defend algorithms such as neural network-based defenders may also be tested. Also, algorithms other than A2C and DDQN should be tested and compared in different environments. Another limitation of this thesis is that we only test algorithms on small networks which only have 4 hosts and 6 services. In the future, large networks with more hosts may be tested to see the performance of different networks. This may require additional work, for instance, reward shaping to accelerate the exploration phase. During the thesis, we also tried to use MuZero [38] for penetration testing problems but did not go smoothly due to the complexity of the model. This could be great future work for this project.

Bibliography

[1] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM Sigart Bulletin*, vol. 2, no. 4, pp. 160–163, Jul. 1991, ISSN: 0163-5719. DOI: 10.1145/122344.122377.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[3] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, pp. 21–29, 1999.

[4] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *IEEE Symposium on Security and Privacy*, IEEE, 2002, pp. 273–284.

[5] J. Pineau, G. Gordon, S. Thrun, *et al.*, "Point-based value iteration: An anytime algorithm for pomdps," in *International Joint Conference on Artificial Intelligence*, vol. 3, 2003, pp. 1025–1032.

[6] X. Li, W. Cheung, and J. Liu, "Decomposing large-scale pomdp via belief state analysis," in *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2005, pp. 428–434. DOI: 10.1109/IAT.2005.63.

[7] W. G. Halfond, J. Viegas, A. Orso, *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, IEEE Piscataway, NJ, vol. 1, 2006, pp. 13–15.

[8] B. Kordy, S. Mauw, S. Radomirovic, and P. Schweitzer, "Foundations of attack–defense trees," *Lecture Notes in Computer Science*, vol. 6561, pp. 80–95, Sep. 2010. DOI: 10.1007/978-3-642-19751-2_6.

[9] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010. DOI: 10.1109/TKDE.2009.191.

[10] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast internet-wide scanning and its security applications," in *USENIX Security Symposium*, Washington, D.C.: USENIX Association, 2013, pp. 605–620, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric.

[11] I. Koniaris, G. Papadimitriou, and P. Nicopolitidis, "Analysis and visualization of ssh attacks using honeypots," in *Eurocon*, IEEE, 2013, pp. 65–72.

[12] J. L. Obes, C. Sarraute, and G. Richarte, "Attack planning in the real world," *ArXiv*, vol. abs/1306.4044, 2013. DOI: 10.48550/ARXIV.1306.4044.

[13] C. Sarraute, O. Buffet, and J. Hoffmann, "Penetration testing == pomdp solving?" *ArXiv*, vol. abs/1306.4714, 2013. DOI: 10.48550/ARXIV.1306.4714.

[14] F. Holik, J. Horalek, O. Marik, S. Neradova, and S. Zitta, "Effective penetration testing with metasploit framework and methodologies," in *IEEE International Symposium on Computational Intelligence and Informatics (CINTI)*, IEEE, 2014, pp. 237–242.

[15] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas, "Learning to learn by gradient descent by gradient descent," *Advances in neural information processing systems*, vol. 29, 2016.

[16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[17] P. S. Shinde and S. B. Ardhapurkar, "Cyber security analysis using vulnerability assessment and penetration testing," in *World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, 2016, pp. 1–5. DOI: 10.1109/STARTUP.2016.7583912.

[18] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[19] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, *et al.*, "Matching networks for one shot learning," *Advances in neural information processing systems*, vol. 29, 2016.

[20] J. X. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick, "Learning to reinforcement learn," *arXiv*, 2016. DOI: 10.48550/ARXIV.1611.05763.

[21] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2016, pp. 1995–2003.

[22] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 1126–1135.

[23] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017. DOI: 10.1109/tnnls.2016.2582924.

[24] F. Hauer, A. Pretschner, M. Schmitt, and M. Grotsch, "Industrial evaluation of search-based test generation techniques for control systems," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 5–8. DOI: 10.1109/ISSREW.2017.10.

[25] J. Oltsik, C. Alexander, and C. CISM, "The life and times of cybersecurity professionals," *ESG and ISSA: Research Report*, 2017.

[26] D. Shmaryahu, G. Shani, J. Hoffmann, and M. Steinmetz, "Partially observable contingent planning for penetration testing," in *Iwaise: First international workshop on artificial intelligence in security*, vol. 33, 2017.

[27] F. Sung, L. Zhang, T. Xiang, T. Hospedales, and Y. Yang, "Learning to learn: Meta-critic networks for sample efficient learning," *arXiv*, 2017. DOI: 10.48550/ARXIV.1706.09529.

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv*, 2018. DOI: 10.48550/ARXIV.1810.04805.

[29] M. Panfili, A. Giuseppi, A. Fiaschetti, H. B. Al-Jibreen, A. Pietrabissa, and F. D. Priscoli, "A game-theoretical approach to cyber-security of critical infrastructures based on multi-agent reinforcement learning," in *Mediterranean Conference on Control and Automation (MED)*, IEEE, 2018, pp. 460–465.

[30] H. M. Z. A. Shebli and B. D. Beheshti, "A study on penetration testing process and tools," in *IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, 2018, pp. 1–7. DOI: 10.1109/LISAT.2018.8378035.

[31] J. Schwartz and H. Kurniawati, "Autonomous penetration testing using reinforcement learning," *arXiv*, 2019. DOI: 10.48550/ARXIV.1905.05965.

[32] M. Shah, S. Ahmed, K. Saeed, M. Junaid, H. Khan, *et al.*, "Penetration testing active reconnaissance phase–optimized port scanning with nmap tool," in *International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, IEEE, 2019, pp. 1–6.

[33] P. Shi, F. Qin, R. Cheng, and K. Zhu, "The penetration testing framework for large-scale network based on network fingerprint," in *International Conference on Communications, Information System and Computer Engineering (CISCE)*, 2019, pp. 378–381. DOI: 10.1109/CISCE.2019.00089.

[34] T. Y. Zhou, Y. C. Zang, J. H. Zhu, and Q. X. Wang, "Nig-ap: A new method for automated penetration testing," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, pp. 1277–1288, Sep. 2019. DOI: 10.1631/FITEE.1800532.

[35] "Dcms: Cyber security breaches survey 2020," *Computer Fraud & Security*, vol. 2020, no. 4, p. 4, 2020, ISSN: 1361-3723. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1361372320300373.

[36] K. Hammar and R. Stadler, "Finding effective security strategies through reinforcement learning and self-play," in *International Conference on Network and Service Management (CNSM)*, IEEE, 2020, pp. 1–9.

[37] T. A. Olowookere and O. S. Adewale, "A framework for detecting credit card fraud with cost-sensitive meta-learning ensemble approach," *Scientific African*, vol. 8, e00464, 2020.

[38] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[39] "Dcms: Cyber security breaches survey 2021," *Network Security*, vol. 2021, no. 4, p. 4, 2021, ISSN: 1353-4858. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1353485821000362.

[40] K. Hammar and R. Stadler, "Learning intrusion prevention policies through optimal stopping," in *International Conference on Network and Service Management (CNSM)*, 2021, pp. 509–517. DOI: 10.23919/CNSM52442.2021.9615542.

[41] L. Huang, L. Zhang, S. Yang, L. P. Qian, and Y. Wu, "Meta-learning based dynamic computation task offloading for mobile edge computing networks," *IEEE Communications Letters*, vol. 25, no. 5, pp. 1568–1572, 2021. DOI: 10.1109/LCOMM.2020.3048075.

[42] H. Li, W.-C. Chen, A. Levy, C.-H. Wang, H. Wang, P.-H. Chen, W. Wan, H.-S. P. Wong, and P. Raina, "One-shot learning with memory-augmented neural networks using a 64-kbit, 118 gops/w rram-based non-volatile associative memory," in *Symposium on VLSI Technology*, IEEE, 2021, pp. 1–2.

[43] A. M. K. Adawadkar and N. Kulkarni, "Cyber-security and reinforcement learning —a brief survey," *Engineering Applications of Artificial Intelligence*, vol. 114, p. 105 116, 2022, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2022.105116.

[44] C. Greco, G. Fortino, B. Crispo, and K.-K. R. Choo, "Ai-enabled iot penetration testing: State-of-the-art and research challenges," *Enterprise Information Systems*, vol. 0, no. 0, p. 2 130 014, 2022. DOI: 10.1080/17517575.2022.2130014.

[45] Y. Kim and G. Dán, "An active learning approach to dynamic alert prioritization for real-time situational awareness," in *IEEE Conference on Communications and Network Security (CNS)*, 2022, pp. 154–162. DOI: 10.1109/CNS56114.2022.9947246.

[46] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions," *Electronics*, vol. 12, no. 6, p. 1333, 2023.

[47] A. Yang, C. Lu, J. Li, X. Huang, T. Ji, X. Li, and Y. Sheng, "Application of meta-learning in cyberspace security: A survey," *Digital Communications and Networks*, vol. 9, no. 1, pp. 67–78, 2023.

[48] *Rllib: Industry-grade reinforcement learning*. [Online]. Available: https://docs.ray.io/en/latest/rllib/index.html (visited on 09/27/2022).

TRITA-EECS-EX-2024:127