Automated Penetration Testing Using Deep Reinforcement Learning

Zhenguo Hu, Razvan Beuran, Yasuo Tan
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, Japan
zhghu@jaist.ac.jp

***Abstract*—At present, penetration testing is done mostly manually, and relies heavily on the experience of the ethical hackers that are performing it, called "pentesters". This paper presents an automated penetration testing framework that employs deep reinforcement learning to automate the penetration testing process. We plan to use this framework mainly as a component of cybersecurity training activities, to provide guided learning for attack training by making use of the framework to suggest possible strategies. When adding support for actual penetration testing tools, the framework could also be used in defense training, by automatically recreating attacks in the training environment.**

In this paper we present our approach for automated penetration testing, which has two stages. First we use the Shodan search engine to collect relevant server data in order to build a realistic network topology, and employ multi-host multi-stage vulnerability analysis (MulVAL) to generate an attack tree for that topology; traditional search algorithms are used to find all the possible attack paths in that tree and to build a matrix representation as needed by deep reinforcement learning algorithms. As a second stage, we employ the Deep Q-Learning Network (DQN) method to discover the most easy to exploit attack path from the possible candidates. This approach was evaluated by generating thousands of input scenarios, and DQN was able to find the optimal path with an accuracy of 0.86, while also providing valid solutions in the other cases.

***Keywords*-penetration testing; attack tree; deep reinforcement learning; deep Q-learning network**

I. Introduction

With the generalized use of computer networks and the frequent occurrence of security incidents, cybersecurity has become an increasingly prominent problem. An effective way to address this issue is to assess the network security features of a system by employing penetration testing. Penetration testing is an authorized attack methodology used to evaluate the security features of a system, in particular to identify its security weaknesses by conducting ethical attacks on it. The method is well established, and several commercial tools are available to assist the pentesters in performing the more tedious tasks [1]. In the field of cybersecurity education and training, defense training similarly requires that ethical hackers conduct attacks in the training environment, so that trainees can get hands-on experience with handling security incidents. Penetration testing methodology is perfectly suitable for playing the attackers' role in this kind of scenario too.

Generally, penetration testing is performed mostly manually, as pentesters need first to analyze the target system, then exploit the discovered vulnerabilities in different ways in order to penetrate the system and compromise network resources in proof-of-concept attacks. This process is a laborious, time-consuming and complex task that requires a great deal of tacit knowledge, which cannot be easily formalized, and is also prone to human errors [2]. Therefore, in recent years, more and more people tried to use model-based attack planning to generate the attacks by employing a model of the target system. The Core Security company employs this idea commercially since 2010, and in their tool named "Core IMPACT" uses as attack planner a variant of the Metric-FF system [3].

Regarding the use of artificial intelligence (AI) techniques in this area, Boddy et al. first applied AI planning to penetration testing [4], which lead to the inclusion of the cybersecurity domain into the 2008 International Planning Competition. Several years later, the Core Security model [5] has been used to simulate external attacks by making individual "attack actions" correspond to known software vulnerability exploits. However, none of these methods provided a complete penetration testing solution.

The attack tree method could play an important role in ensuring that automated penetration testing is close to human actions. This method for modelling security threats on a given system was proposed by Schneier in 1999, and represents an attack against it in the form of a tree structure [6]. By analyzing the attack tree, we can better understand the relationship between each attack method. Yousefi et al. [7] applied reinforcement learning (RL) to analyze the attack tree, which uses Q-learning to find the attack path, but it still has the problem that the action space and sample space are too small. Compared with reinforcement learning, deep reinforcement learning (DRL) is a more suitable approach for analyzing the attack tree, as it combines deep learning and reinforcement learning and adopts a trial-and-error approach to find an optimal solution.

In this paper we present an automated penetration testing framework that we designed and implemented with the goal of finding the best attack path for a given topology, similarly to a human attacker. Our main contributions are:

- Discuss the use of Shodan to collect actual server data for constructing realistic target networks

DOI 10.1109/EuroS&PW51379.2020.00009

- Introduce our approach of building an attack representation matrix by using the MulVAL and Depth-First Search (DFS) algorithms
- Present the manner in which the Deep Q-Learning Network method is used to analyze the attack matrix and find the optimal attack path
- Evaluate our approach over realistic network scenarios built using Shodan data to show the effectiveness of DQN for automated penetration testing

The paper continues with a research background in Section II, providing a brief overview of the tools, algorithms and techniques that we use. Then we discuss in detail the architecture of the proposed automatic penetration testing framework in Section III. In Section IV we present an experiment we conducted to evaluate the framework effectiveness on a real network topology. Finally, we conclude the paper and propose some future improvements.

II. Background

In this section we shall introduce the tools, algorithms and techniques that we shall later use as components of our automated penetration testing framework that will be discussed in Section III.

A. Shodan

Shodan is an online search engine for Internet-connected devices that has been available since 2009 [8]. Shodan is often used by pentesters and even black-hat hackers to collect information about their potential targets. According to CNN Money, the data base of Shodan is estimated to contain information about 500 million actual network devices, such as their IP addresses, list of running services, and so on [9].

In our research we shall employ Shodan to gather the information needed in order to build realistic network topologies, as it will be discussed in Section III-A. This network topology information is then used as input to the MulVAL algorithm introduced below.

B. Attack Trees

1) Attack Tree Model: Schneier first proposed the attack tree model as a way to express the interdependence between attack behavior and attack steps, and to realize the formalization and documentation of the attack process [6]. Each node of the tree represents an attack behavior or a sub-target, and the root node represents the ultimate target of the attack. Each child node represents either an attack behavior or a child target that should to be finished before the parent node can be attacked.

The nodes of the attack tree are divided into "AND" nodes and "OR" nodes. The label "AND" means that a node can only be targeted after all the child nodes have been targeted. The label "OR" signifies that as long as one of the child nodes has been targeted, the parent node can be targeted as well. In Fig. 1 we show a graph and a text representation of each of the two possible node types, "AND" and "OR".

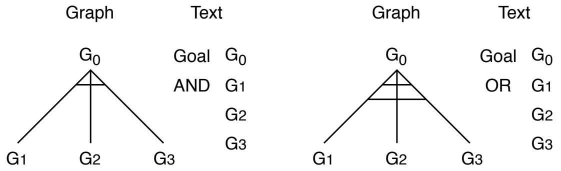

Figure 1. Attack tree node representation.

2) MulVAL: MulVAL is an open-source tool for generating the actual attack tree that corresponds to a given network topology [10]. Depending on the network topology properties, it has been shown that the complexity of the attack tree generation algorithm is between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, hence reasonable for typical small-and-medium size company networks with tens of hosts.

In our framework MulVAL is used to find all the possible paths for a given input network topology, as it will be discussed in Section III-A. A matrix is built according to all the possible paths discovered, then it is simplified using Depth-First Search (DFS) algorithm to make it more suitable for use with the DRL algorithm.

C. Reinforcement Learning

Reinforcement learning (RL) is a type of learning method that maps environment state to actions, so that an agent can get the maximum cumulative reward in the process of interacting with the environment [11]. As an interactive learning method, the main characteristics of RL are trial-and-error and delayed returns [12]. RL agents try to determine strategies that can offer actions corresponding to any given environment state, with the final goal of discovering an optimal strategy so that the cumulative reward received is the largest [11], [13].

Fig. 2 shows the interaction process between the agent and the environment. At each time step t, the agent observes the environment to obtain the state S_t, and then executes action A_t. The environment generates the new state S_{t+1} and the reward R_t depending on A_t. Such a process can be described by Markov Decision Processes (MDP). An MDP is divided into 4 parts, also known as quads, that are denoted by (S, A, P, R), where S represents the state set; A represents the action set; $P(\text{s'}|\ s, a)$ represents the probability of transition to s' after taking the action a in state s; $R(s,\ a)$ represents the reward obtained by taking the action a in state s.

The goal of the strategy is to maximize future cumulative rewards, meaning that the "quality" of the current state can be measured by the future cumulative rewards of that state. Reinforcement learning introduces a reward function

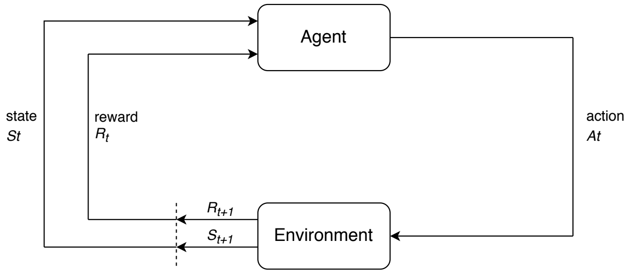

Figure 2. Overview of the reinforcement learning process.

to calculate the reward at any given time t:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \quad (1)$$

where γ is the discount coefficient. Since the farther away from the current state the greater the uncertainty of the reward value is, γ is generally used to account for this uncertainty.

Furthermore, the concept of value function is used to represent the "value" of a state, i.e., the expectation of future cumulative rewards for that state:

$$V^{\pi}(s) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \mid s_t = s\right] \quad (2)$$

In addition, the action-state value function is used to express future cumulative rewards conditioned both by a state and an action:

$$Q^{\pi}(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1+k} \mid s_t = s, a_t = a\right] \quad (3)$$

D. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) refers to a class of techniques that combine deep learning and reinforcement learning. DRL is a general learning method by which the "agent" obtains state information from the environment, selects appropriate actions according to its own strategies, changes the state of the environment, then gets rewards or incentives that express the efficacy of the actions to the agent depending on the new environment state [14]. When applying DRL to penetration testing, the agent plays the role of the pentester, and choose the most effective path in order to obtain the maximum reward.

DRL algorithms are divided into three main classes: value-based functions, strategy-based search and model-based methods. Deep Q-Learning Network (DQN) is a representative value-based method by Mnih et al. [15], [16] that combines a Convolution Neural Network (CNN) with the Q-Learning algorithm [17] in traditional reinforcement learning to create the new DQN model. The input of the DQN model is a simplified matrix that undergoes a non-linear transformation of 3 convolutional layers and 2 fully-connected layers, and finally generates a Q value for each action in the output layer. Fig. 3 shows the architecture of the DQN model.

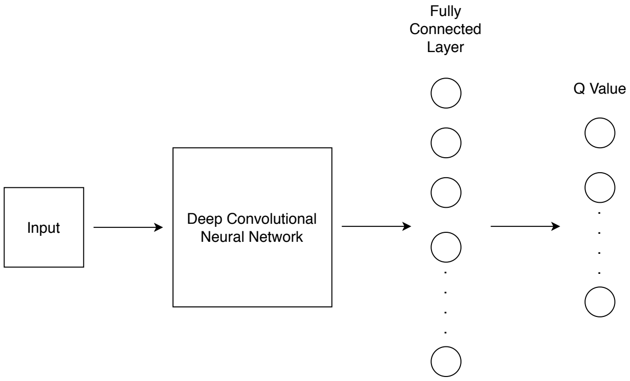

Figure 3. Overview of the DQN model architecture.

In Fig. 4 we introduce the training process of DQN, which improves the traditional Q-learning algorithm in order to alleviate problems such as instability in the representation function of the non-linear network. For instance, DQN uses experience replay to process the transfer samples. At each time step t, the transfer samples obtained by the agent interacting with the environment are stored in the replay buffer unit. During the training process, a small batch of transfer samples is selected randomly and the Stochastic Gradient Descent (SGD) algorithm is used to update the network parameter θ.

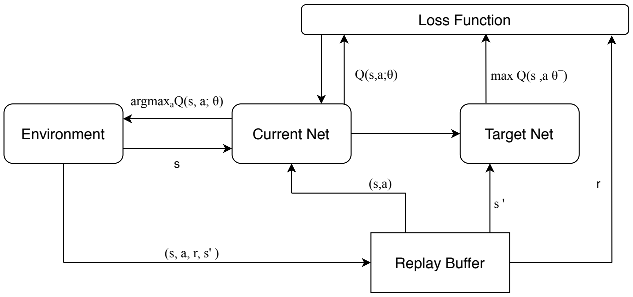

Figure 4. Overview of the DQN training process.

DQN also modifies the manner in which the Q value is computed. Thus, in DQN $Q(s, a \mid \theta_i)$ represents the output of the current value network, and it is used to evaluate the value function of the current state action. $Q(s, a \mid \theta_i^-)$ represents the output of the target value network, and $Y_i = r + \gamma \underset{a'}{max}\, Q(s', a' \mid \theta_i^-)$ is generally adopted as the target Q value.

The parameter θ of the current value network is updated in real time. After every N rounds of iteration, the parameters of the current value network are copied into the target value network. The network parameters are then updated by minimizing the mean square error between the current Q value and the target Q value. The error function is:

$$L(\theta_i) = E_{s,a,r,s'}[(Y_i - Q(s,a \mid \theta_i))^2] \tag{4}$$

and the gradients are computed as follows:

$$\nabla_{\theta_i} L(\theta_i) = \\ E_{s,a,r,s'}[(Y_i - Q(s,a \mid \theta_i))\nabla_{\theta_i} Q(s,a \mid \theta_i)] \tag{5}$$

III. Framework Architecture

In this section we present the automated penetration testing framework that we designed and implemented based on deep reinforcement learning techniques. The framework has three main components (see Fig. 5):

- *Training Data:* Build the training data needed as input by the DQN algorithm
- *DQN Module:* Train the DQN algorithm, then use the trained model for penetration testing
- *Penetration Tools:* Wrapper for external tools used to perform actions on real systems

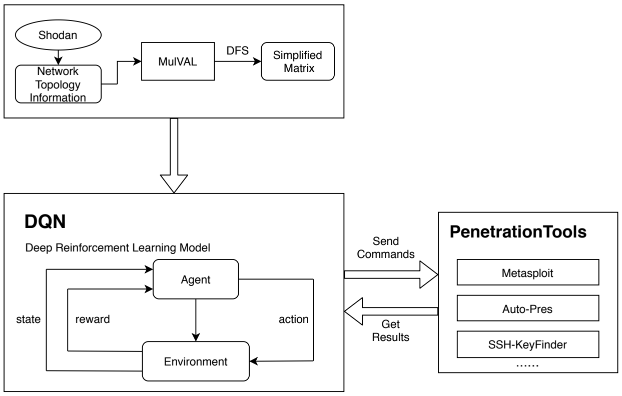

Figure 5. Architecture of the automated penetration testing framework.

In what follows we describe in detail each of the framework components.

A. Training Data

The key to using deep learning is to have training data. Our approach for creating the training data to feed to the DQN model includes three steps:

- Use Shodan to collect network information for modeling a realistic network environment
- Use MulVAL to create the attack tree corresponding to that network environment
- Preprocess the data in order to make it suitable for use in the DQN model

1) Host Dataset: In order to build the training data, we first use Shodan to collect information about real network devices. For example, if we want to get information about real web servers, the Shodan API will return data that includes the actual IP address, used ports and protocols, known vulnerabilities, and so on. An example of raw data collected via Shodan is shown in Fig. 6 (sensitive information, such as IP address, was removed).

```
   "info": "(CentOS)",
   "ip_str": 192.168.1.1,
   "isp": HiNet,
   "os": null,
   "port": 443,
   "product": "Apache httpd",
   "transport": "tcp",
   "version": "2.2.15",
   "vulns": {
      "CVE-2010-1452",
      ...
   }
```

Figure 6. Example of raw data collected via Shodan for a web server.

For each different service, we create a separate service dataset file with all the relevant information concerning a particular network host running that service. In the process of using the collected data we protect privacy by retaining only unidentifiable information, such as open ports or service names. Table I shows some web server profile examples.

Table I
Web Server Profile Examples

Product	Port	Protocol	Vulnerability	OS
Apache	80	https	CVE-2010-1452	CentOS
Nginx	8080	https	CVE-2011-0419	Ubuntu
mt-daapd DAAP	3689	tcp	CVE-2017-9617	FreeBSD

In this way we obtain the real host dataset through Shodan and use it as the basis of DQN training dataset.

2) Vulnerability Dataset: In addition to the host dataset, we also create a vulnerability dataset file. For a set of known vulnerabilities, the vulnerability dataset includes the CVE [18], [19] and Microsoft vulnerability identification numbers, as well as the type, base score and exploitability score components of the CVSS score [20]. In order to include all the relevant information that we require, we combine information from the National Vulnerability Database (NVD) and the Microsoft (MS) Database to create a new dataset. Table II shows a few items from the vulnerability dataset we constructed.

3) DQN Dataset: DQN dataset is a training dataset for Deep Q-Learning network which includes the host dataset collected by Shodan and vulnerability dataset. To generate the dataset needed for the DQN algorithm, we first need to create the attack tree for a given set of network topologies.

Table II
VULNERABILITY DATASET EXAMPLE ENTRIES

CVE ID	MS ID	Type	BaseScore	ExpScore
CVE-2010-0820	MS10-068	Buffer Errors	9.0	8.0
CVE-2014-6321	MS14-066	Code Injection	10.0	10.0
CVE-2017-0022	MS17-022	Information Leak	4.3	8.6

For this purpose, we created a series of network topology templates that are populated with actual data from the host dataset. Fig. 7 shows an example template for a vulnerability configuration of webserver.

```
vulExists(webServer,'CAN-2002-0392',httpd).
vulProperty('CAN-2002-0392',
            remoteExploit, privEscalation).
networkServiceInfo(webServer,
            httpd,tcp,80,apache).
```

Figure 7. Example template for a configuration of webserver.

From this example we can know that the $vulExists$ comes from Shodan, and it shows that the target is a webserver which exists the vulnerability of CAN-2002-0392. The $vulProperty$ comes from Vulnerability dataset and shows the type of CAN-2002-0392 is remote exploit and it can cause the effect of privilege escalation. This detailed network topology information is used by MulVAL to generate the attack tree that corresponds to that topology. An example network topology, and the generated attack tree for that topology are shown later in the paper, in Figs. 8 and 9, respectively.

Next, the attack tree must be converted into a transfer matrix. Yousefi et al. first presented a method for transforming an attack tree into a transfer matrix [7], but it is not suitable for penetration testing, because during the process of penetration testing, other steps in addition to vulnerabilities, such as file access, command execution, etc. are also important. For this reason we propose an improved method for converting the attack tree into a simplified transfer matrix. As a first step, we map all the nodes in the attack tree to a matrix form that includes CVSS score information for the vulnerabilities, and some predefined score for other actions, such as accessing files. At the next step, instead of feeding this matrix directly into the DQN algorithm, we decided to simplify it by using the Depth-First Search (DFS) algorithm. The idea is that the full transfer matrix shows all the possible movements, but it can be simplified if we select only those that can be used to reach the attack goal.

Consequently we use the DFS algorithm to find all the possible paths that can reach the goal, then we create a simplified matrix that contains: (i) the scores for the start node in the first column; (ii) the total scores for the intermediate steps in intermediate columns; (iii) the score for the goal node in the last column. These scores will be further used as reward scores in the DQN algorithm.

B. DQN

DQN is used in our framework to determine the most feasible attack paths through continuous training of the DQN model. The input of the model is the simplified matrix mentioned above, and for the activate function we used the $softmax$ function. The output of the DQN model is the optimal attack path. During the process of learning, the DQN model agent represents an attacker, and the target environment is modelled by the simplified attack matrix. The attacker can move from node to node in the attack matrix until it reaches the final goal, the target server.

The reward corresponding to exploiting each vulnerability used in the DQN model is computed by a vulnerability score that we defined based on components of the Common Vulnerability Scoring System (CVSS):

$$Score_{vul} = baseScore \times \frac{exploitablityScore}{10} \qquad (6)$$

In CVSS, the base score reflects the harmfulness of the vulnerability itself, and the exploitability score reflects the feasibility of exploiting that particular vulnerability. Therefore, we use the exploitability score, which has a maximum value of 10, to weight the significance of the base score considering how feasible it is to use a certain vulnerability.

C. Penetration Tools

In order to make possible the use of our automated penetration testing framework to conduct attacks on real systems, it needs the ability to interact with actual network environments, for instance to run commands, exploit vulnerabilities, and so on. Instead of building our own tools, for this stage we decided to make use of existing penetration testing tools, such as Metasploit [21], similarly to the approach taken by CyPROM [22].

While the implementation of this functionality is still ongoing, we will release it together with the rest of the framework in the near future. Our approach is to create a wrapper for the penetration testing tools, so that the outcome of the DQN model trained as described above can be used to send commands to those penetration tools, which will perform actions on the real target systems. The results of those actions is received as feedback, and used in order to decide how to proceed with a given attack path.

IV. EXPERIMENTAL RESULTS

In order to validate our approach we have conducted a series of experiments by modeling an example network topology and using DQN algorithm to determine the best attack path for that topology.

A. Experiment Scenario

Fig. 8 shows the network topology model that we considered in our experiments. The model represent a small company network which includes a web server, a file server and a workstation. The file server and workstation are in one subnet connected via a router that further connects to a firewall. The web server is in a different subnet and connects via the firewall to the Internet.

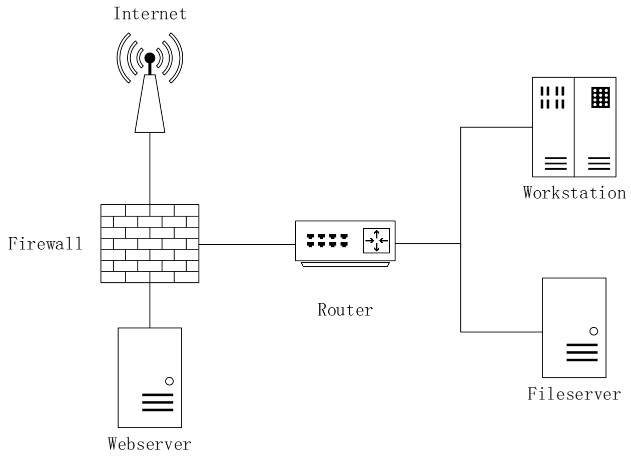

Figure 8. Network topology for the experiment scenario.

From an attack perspective, we make the following assumptions about the attack methods:

- The attacker starts from the Internet and can access the web server via the HTTP and HTTPS protocols.
- The file server and the web server can connect to each other via file protocols such as NFS, FTP, and so on.
- The file server and the workstation can connect to each other via file protocols such as NFS, Samba, and so on.
- The file server and the workstation can access the Internet via the HTTP and HTTPS protocols.

To configure the actual protocol details, we have used data collected using Shodan to initialize the information about vulnerabilities, open ports, products, and protocols for the web server and the file server. As for the workstation, we considered it runs no services, but just uses various transfer protocols. Table III shows an example of the network information configuration for these hosts.

Table III
HOST CONFIGURATION INFORMATION

Host	Vulnerability	Port	Product	Protocol
Web server	CVE-2019-0211	80	Apache	HTTP/HTTPS
File server	CVE-2015-3306	21	-	FTP
Workstation	-	-	-	HTTP/HTTPS/FTP

The vulnerability dataset is used to attach detailed information to each vulnerability, including its type and effect (such as the permission level that can be achieved by exploiting that vulnerability). For every different vulnerability type, the attack tree algorithms will generate a different attack path. In Table IV we give some examples of vulnerability types encountered in our scenario.

Table IV
DETAILED VULNERABILITY INFORMATION

Vulnerability	Type	Effect
CVE-2019-0211	Privilege Escalation	root
CVE-2015-3306	Improper Access Control	user

B. DQN Dataset Generation

Based on the network topology and configuration information that we discussed, MulVAL is used to generate the corresponding attack tree. Fig. 9 displays the main part of the attack tree generated for our example scenario, each node being tagged with a short description of how a certain vulnerability is to be exploited. In this example, the attacker starts from the Internet, which is represented by node 26 located at the top-center of the figure. The attacker's goal is to execute code on the workstation represented by node 1 located at the top-left side of the figure. We assume that the attacker can move in the direction of the arrow until reaching the attack goal.

In order to build the transfer matrix needed by the DQN algorithm, each node needs to be assigned a reward score, as follows:

1) The reward value of the start node (node 26 in our example) is 0.01, and the reward score of the goal node (node 1 in our example) is 100.
2) For every node that exploits a vulnerability (such as node 16 in our example, exploiting the vulnerability CVE-2012-0053 at node 28), we use the $Score_{vul}$ value defined in Eq. 6 as the reward score.
3) For every node that executes some code or accesses files (denoted by $execCode$, $accessFile$ in our example) we define a reward score of 1.5, since such actions are important during the penetration testing process (target node 1 is excluded from this rule).
4) For any other node in the tree we give the reward score 0, and if there is no path between two nodes, the reward score is -1.

C. DQN Training Results

The simplified matrix above becomes the input of the DQN model, which is then trained to determine the total rewards for all the possible paths. For the network topology we presented, we used Shodan data with different vulnerabilities for the considered protocols into the training template so as to create a totoal of 2000 different attack trees as training data, and 1000 different attack trees as validation data. For validation purposes we define that the optimal path in the

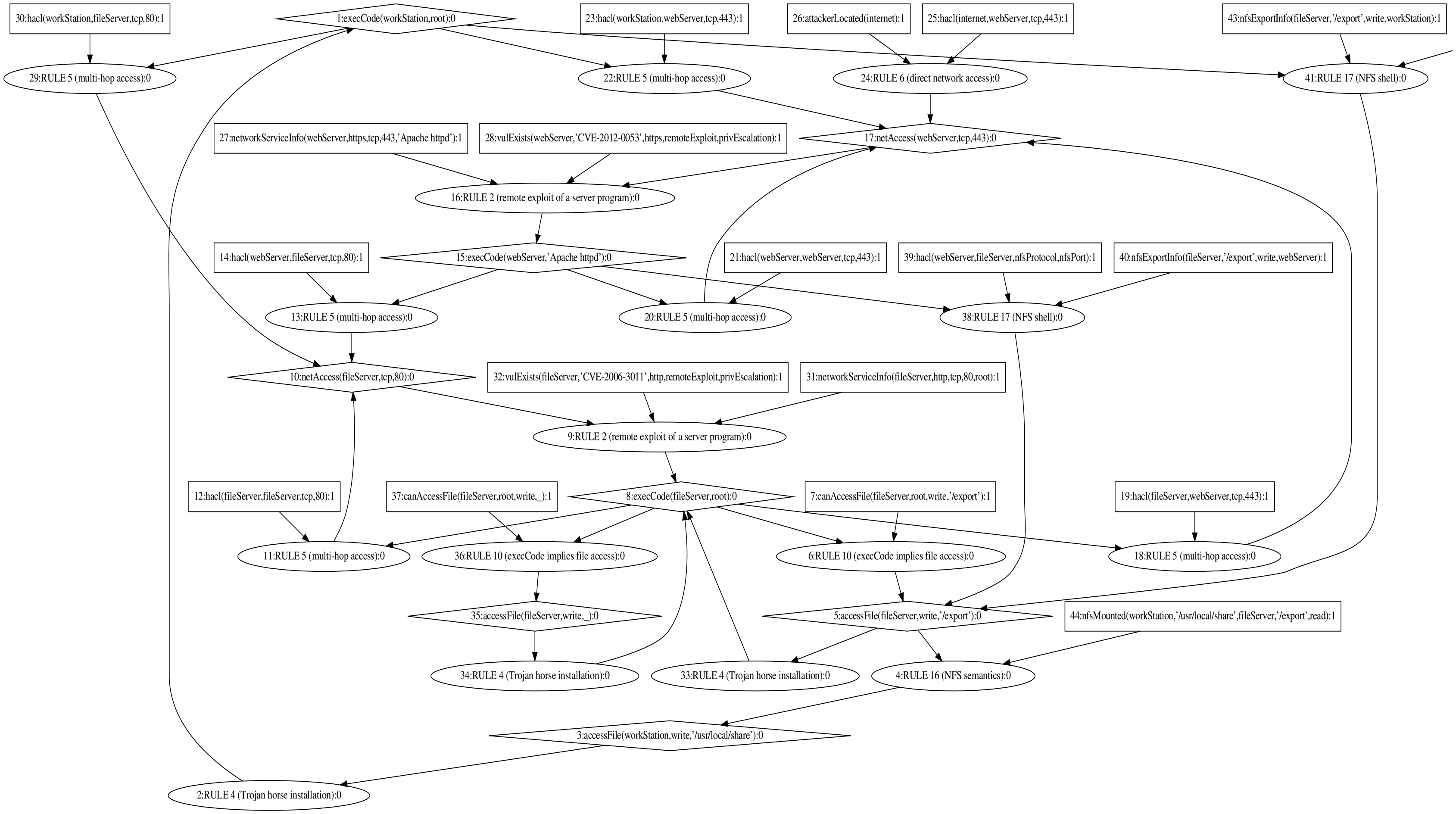
30:hacl(workStation,fileServer,tcp,80):1
1:execCode(workStation,root):0
23:hacl(workStation,webServer,tcp,443):1
26:attackerLocated(internet):1
25:hacl(internet,webServer,tcp,443):1
43:nfsExportInfo(fileServer,'/export',write,workStation):1
29:RULE 5 (multi-hop access):0
22:RULE 5 (multi-hop access):0
24:RULE 6 (direct network access):0
41:RULE 17 (NFS shell):0
27:networkServiceInfo(webServer,https,tcp,443,'Apache httpd'):1
28:vulExists(webServer,'CVE-2012-0053',https,remoteExploit,privEscalation):1
17:netAccess(webServer,tcp,443):0
16:RULE 2 (remote exploit of a server program):0
14:hacl(webServer,fileServer,tcp,80):1
15:execCode(webServer,'Apache httpd'):0
21:hacl(webServer,webServer,tcp,443):1
39:hacl(webServer,fileServer,nfsProtocol,nfsPort):1
40:nfsExportInfo(fileServer,'/export',write,webServer):1
13:RULE 5 (multi-hop access):0
20:RULE 5 (multi-hop access):0
38:RULE 17 (NFS shell):0
10:netAccess(fileServer,tcp,80):0
32:vulExists(fileServer,'CVE-2006-3011',http,remoteExploit,privEscalation):1
31:networkServiceInfo(fileServer,http,tcp,80,root):1
9:RULE 2 (remote exploit of a server program):0
12:hacl(fileServer,fileServer,tcp,80):1
37:canAccessFile(fileServer,root,write,_):1
8:execCode(fileServer,root):0
7:canAccessFile(fileServer,root,write,'/export'):1
19:hacl(fileServer,webServer,tcp,443):1
11:RULE 5 (multi-hop access):0
36:RULE 10 (execCode implies file access):0
6:RULE 10 (execCode implies file access):0
18:RULE 5 (multi-hop access):0
35:accessFile(fileServer,write,_):0
5:accessFile(fileServer,write,'/export'):0
44:nfsMounted(workStation,'/usr/local/share',fileServer,'/export',read):1
34:RULE 4 (Trojan horse installation):0
33:RULE 4 (Trojan horse installation):0
4:RULE 16 (NFS semantics):0
3:accessFile(workStation,write,'/usr/local/share'):0
2:RULE 4 (Trojan horse installation):0

Figure 9. Attack tree generated for the experiment scenario (part of the tree was not shown due to space limitations).

simplified matrix is that with the smallest number of steps when the highest reward is selected. The model accuracy and average step count for the optimal attack path are provided in Table V. From these results we can observe that for this sample network topology the model has a good performance in finding the most feasible path.

Table V
ACCURACY AND AVERAGE STEP COUNT

DRL Framework	Accuracy	Average Step Count
DQN Model	0.863	1.927

Fig. 10 shows the average reward changes of the DQN model for one of the attack paths in the experiment scenario during a total of 100 training iterations. One can notice that the reward is small at first, then gradually rises after about 30 iterations. After about 60 iterations, the reward becomes stable meaning that the DQN model has found the optimal path. In that case the path is given by the sequence of steps "26 → 24 → 17 → 16 → 15 → 13 → 10 → 9 → 8 → 6 → 5 → 4 → 3 → 2 → 1" (cf. Fig. 9). This path not only makes full use of the vulnerabilities of both the web server and the file server, but also adopts simple attack methods that are easy to use via penetration testing tools.

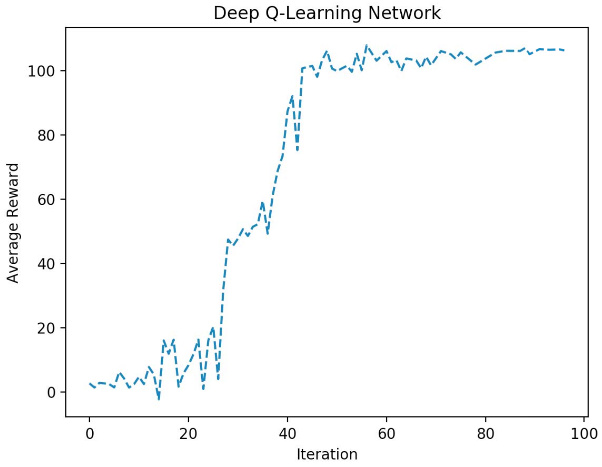

Figure 10. Reward changes for one attack path in the experiment scenario.

D. Discussion

Although the DQN algorithm seems to converge relatively fast when using several sample network topologies, we consider that more work is required to determine the right choice of parameters for the DQN model over a large set of scenarios with different properties.

While have designed our framework for automated penetration testing of actual systems, so far we have only performed training on generic network topology models. As a next step we consider feeding to the DQN algorithm as training model the results of scanning specific actual network topologies, so that it can evaluate the particular characteristics for that topology and suggest an optimal attack path for that topology.

Each node on the most feasible attack path identified by the DQN algorithm contains all the information needed to effectively conduct that step of the attack. Consequently, the path and node information can be used to drive the execution of a penetration testing tool such by Metasploit to conduct that attack. For example, knowing that node 28 represents the CVE-2012-0053 vulnerability, the framework could send a command to Metasploit and use the internal module `msf exploit` to exploit that vulnerability of the web server. We plan to leverage this in the Penetration Tools module mentioned in Section III-C.

Our framework is intended for use in the area of cybersecurity education and training, and it can already be employed for attack training activities, for instance to suggest attack paths that the trainees can then experiment with by themselves in the cyber range, by following a guided learning methodology. Once the Penetration Tools module implementation is finished, we intend to also use this framework to conduct realistic attacks in the training environment in an automated manner. This will reduce the cost of organizing defense training activities, since the support of white-hat hackers to conduct such attacks will no longer be required. Repeating training activities that employ realistic attack methods will lead to a significant improvement of the participants' defense skills.

V. CONCLUSION

In this paper we have proposed an automated penetration testing framework that is based on deep reinforcement learning techniques, in particular Deep Q-Learning Network (DQN). Our approach innovatively combines the Shodan search engine with a variety of vulnerability datasets to gather real host and vulnerability data for building realistic training and validation scenarios. The attack tree methodology is then used to generate attack information for each of the training scenarios, which is then processed and used to train the DQN model. The training uses reward scores assigned to each node mainly based on CVSS score information to determine the most feasible attack path for a given network scenario.

To evaluate the applicability of this approach we have conducted an experiment in which a given network topology was populated with real host and vulnerability data to construct 2000 training and 1000 validation scenarios. Even with such a low number of training scenarios, DQN was able to achieve an accuracy rate of 0.86 in determining the optimal attack path, and provided viable solutions that met our expectations in the other cases.

Our framework can already be used to suggest attack strategies to assist with cybersecurity attack training activities. The only pending item of the framework is the Pentetration Tools module, for which implementation is ongoing. Its completion will make it possible to conduct automated attacks on real network environments, thus greatly simplifying the penetration testing process and assisting in defense training activities.

As future work, we plan to focus on two main directions. On the one hand, we plan to expand the training dataset with additional network topologies, so as to improve the versatility and stability of the DQN model. On the other hand, we consider integrating a network service scanning function into the framework, so that information on real target environments can be automatically provided to the DQN model, leading to more accurate results for actual network topologies.

REFERENCES

[1] B. Burns, E. Markham, C. Iezzoni, P. Biondi, and M. Lynn, *Security Power Tools*. O'Reilly, 2007.

[2] S. Jajodia and S. Noel, "Topological vulnerability analysis: A powerful new approach for network attack prevention, detection, and response," *Algorithms, Architectures and Information Systems Security (Indian Statistical Institute Platinum Jubilee Series)*, pp. 285–305, 2008.

[3] J. Hoffmann, "The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables," *Journal of Artificial Intelligence Research*, vol. 20, pp. 291–341, 2011.

[4] M. Boddy, J. Gohde, T. Haigh, and S. Harp, "Course of action generation for cyber security using classical planning," in *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'05, p. 12–21, 2005.

[5] J. L. Obes, C. Sarraute, and G. G. Richarte, "Attack planning in the real world," *arXiv: Cryptography and Security*, 2013.

[6] B. Schneier, "Attack trees - modeling security threats," *Dr. Dobb's Journal*, vol. 24, 1999.

[7] M. Yousefi, N. Mtetwa, Y. Zhang, and H. Tianfield, "A reinforcement learning approach for attack graph analysis," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 212–217, 2018.

[8] K. Simon, C. Moucha, and J. Keller, "Contactless Vulnerability Analysis using Google and Shodan," *Journal of Universal Computer Science*, vol. 23, pp. 404–430, 2017.

[9] D. Goldman, "Shodan: The scariest search engine on the internet." https://money.cnn.com/2013/04/08/technology/security/shodan/.

[10] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A Logic-based Network Security Analyzer," in *USENIX Security Symposium*, 2005.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[12] H. Zhou, Zhi, *Machine Learning*. Tsinghua University Press, 2016.

[13] M. Cheng-qian, X. Wei, and S. Wei-jie, "Research on reinforcement learning technology: A review," *Command Control & Simulation*, 2018.

[14] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," in *NIPS Deep Learning Workshop 2013*, pp. 201–220, 2013.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[17] C. Watkins, *Learning from delayed rewards*. PhD thesis, Cambridge University, 1989.

[18] B. Cheikes, D. A. Waltermire, and K. Scarfone, "Common platform enumeration: Naming specification version 2.3." National Institute of Standards and Technology, 2011.

[19] M. Karlsson, "The edit history of the national vulnerability database," Master's thesis, Swiss Federal Institute of Technology Zurich, 2012.

[20] T. Grance, M. Stevens, and M. Myers, "Guide to selecting information technology security products." National Institute of Standards and Technology, 2003.

[21] D. Kennedy, J. O'Gorman, D. Kearns, and M. Aharoni, *Metasploit: The Penetration Tester's Guide*. No Starch Press, 2011.

[22] R. Beuran, T. Inoue, Y. Tan, and Y. Shinoda, "Realistic cybersecurity training via scenario progression management," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019.