Automating Penetration Testing using Reinforcement Learning

Stefan Niculae, Daniel Dichiu
Experimental Research Unit
Bitdefender
Bucharest, Romania
{s.niculae, d.dichiu}@bitdefender.com

Kaifeng Yang, Thomas Bäck
Natural Computing Group
Leiden Institute of Advanced Computer Science
Leiden, The Netherlands
{k.yang, t.h.w.baeck}@liacs.leidenuniv.nl

***Abstract*—Penetration testing is the practice of performing a simulated attack on a computer system in order to reveal its vulnerabilities. The most common approach is to gain information and then plan and execute the attack manually, by a security expert. This manual method cannot meet the speed and frequency required for efficient, large-scale security solutions development. To address this, we formalize penetration testing as a security game between an attacker who tries to compromise a network and a defending adversary actively protecting it. We compare multiple algorithms for finding the attacker's strategy, from fixed-strategy to Reinforcement Learning, namely Q-Learning (QL), Extended Classifier Systems (XCS) and Deep Q-Networks (DQN). The attacker's strength is measured in terms of speed and stealthiness, in the specific environment used in our simulations. The results show that QL surpasses human performance, XCS yields worse than human performance but is more stable, and the slow convergence of DQN keeps it from achieving comparable performance, in addition, we find that all of these Machine Learning approaches outperform fixed-strategy attackers.**

***Index Terms*—computer security, penetration testing, machine learning, reinforcement learning, genetic algorithm**

I. Introduction

Cyber-security threats are seeing an ever-increasing level of breadth, sophistication and damaging power (10). Thus, it is becoming increasingly important for security solutions to counteract these threats. When developing a new method, in the field of cyber-security, it is of paramount to be able to validate the approach's effectiveness. Moreover, Artificial Intelligence (AI) techniques also require large amounts of data. One of the most common ways to evaluate security defenses is to purposefully attack the system with the intent of discovering weak points and fixing them before a real attacker can take advantage of them.

Penetration testing (pentesting) is the practice of performing a simulated attack on a computer system. It is a form of ethical hacking which assesses vulnerabilities and reveals security weaknesses. It involves the initial reconnaissance, gathering preliminary information about the system; scanning, revealing exposed ports or targetable software; gaining foothold, obtaining shell access on one or more machines; and deploying the payload, to achieve the desired objective. Additional steps include maintaining access by gaining persistence; widening access through lateral movement to other machines and cleaning up to reduce the chance of detection. Pentests are a crucial part of ensuring a system's security.

Manual pentesting, while effective, cannot meet the requirements of AI security techniques: frequent, on-demand, large-scale and preferably repeatable evaluations, which are used to validate incremental method improvements and search for optimal variations. The alternative, existing automatic pentesting tools fail to simulate the threat of a real attacker. To address these issues, this paper presents a learning-based method aimed at overcoming the limitations of manual testing and surpassing the effectiveness of existing automated tools. Reinforcement Learning (RL) offers algorithms that learn from their interaction with the environment, bettering themselves at reaching a designated objective. The recent roaring success of RL methods in games such as Chess (35) or Go (36) inspired testing them in this different domain, training an attacker. The obtained attacking strategy should be able to compromise a given system faster than randomly attempting exploits and comparable in strength to a real attacker.

The structure of this paper is as follows: Section II provides a summary of some previous approaches for automating pentesting; and describes RL in a general sense, specific algorithms (Q-Learning, Deep Q-Networks, and Extended Classifier System) and recent advancements. Section III describes the problem formalization, namely, what RL algorithms are to solve, the rules they are subject to and how they are evaluated. IV contains the description fo fixed-strategy algorithms as well as how RL algorithms are used in this problem. Experimental results, a comparison of the different attacker algorithms and an analysis of parameter configurations are presented in section V. Finally, conclusions and future directions are discussed in Section VI.

II. Related Work

This paper combines two mostly disjunctive domains: cyber-security (through pentesting) and machine learning (through RL and related methods). Thus, the section follows the clear division between knowledge background and separates them into two subsections.

A. Automatic Pentesting

This subsection briefly describes three papers that were particularly relevant to this paper's model of the pentesting environment. The problems tackled are similar and the ideas

expressed served as good inspiration for modeling our environment. For each paper, the environment definition and methods used to solve it are summarized along which parts were incorporated or continued into the present work.

The literature contains relatively few papers dealing with issues similar to ours. Among those, fewer even, save for ones mentioned below, provide a clear an detailed description of their simulated environment.

Elderman et al. (11) simulate a network of four nodes: one "start" node, one "end" node and two intermediate connected in a diamond topology. In this model, there are two agents: an attacker and a defender. Each node is characterized by a tuple of ten integers $(a_1, a_2, ..., a_{10})$ and $(d_1, d_2, ..., d_{10})$ for the two agents respectively. At each time-step, the agents chose one node and one value to increment. At any point, the attacker may choose to execute an attack on one node, by using one attack value. If it is higher than the defender's value, the attack is successful. Otherwise, it fails and the attacker has a chance of being detected. None of the agents have knowledge of the other's allocation. The game ends when the "end" node has been successfully attacked, or the attacker has been detected.

Even though only a very simple network is simulated in this paper, it highlights the usefulness of regarding the environment as a dynamic system, where action outcomes are influenced by hidden information.

Applebaum et al. (1) underline the importance of pentesting in the security lifecycle and the shortcomings associated with manual execution. The network model is more complex. It includes shared and personal workstations, dynamic machine connections and local and domain admins. There are three participants in a simulation, namely, the attacker, the defender, and the gray agent. The gray agent simulates the behavior of normal users on the system, with the main impact of adding user credentials to new machines by logging in, and opening or closing connections. The defender will analyze new connections based on a fixed probability. The attacker can only see the part of the network which is already compromised. The attacker's capabilities range from reconnaissance to exploits, post-exploit, and cautionary techniques. The authors evaluate multiple strategies for choosing the action to execute: from random to fixed-strategy and classical planning-based ones. Performance was measured in terms of the percentage of machines compromised, credentials obtained and sensitive data exfiltrated. They noted that the running time is much higher for planners than for immediate executors. They also concluded that increased connectivity decreases performance for all strategies, because of the overabundance of options.

The explicit modeling of neutral user behavior, a nuance overlooked by most other approaches, brings the simulation closer to the reality and is included in our model as well. Evaluation metrics are incorporated into our reward definition (details in Section III) and the fixed-strategy agents proved to be a good starting point (details in Section IV). Two ideas suggested in (1)'s future work are adopted and continued here: (1) adding a *do nothing* action, to avoid detection and dynamic vulnerability status (patching or adding new vulnerabilities).

Sarraute et al. (32) target networks with more complex topologies: having asymmetrical connections and clustering themselves into sub-networks. A defender is not explicitly modeled and the attacker's actions are limited to two kinds of scans, a homogenous list of exploits and the option of giving up. Exploits are modeled more in-depth: they may crash the machine and, unsuccessful attempts can reveal further information about the system. Negative rewards are given for action duration risk of duration/detection. To handle the larger network sizes, the authors proposed a "4 Abstraction Level" design: a network is decomposed in sub-networks and the attacker picks its target incrementally: first high-level components then all the way down to individual machines.

As the richer model matches the real world scenario more closely, we use relevant aspects of it, such as detailed exploit properties and the option for the attacker to give up. Decomposing a network into its logical components is an interesting approach, but out of the scope of our project, as we focus on solving smaller networks first.

Out of the other papers consulted, many focused on other aspects of this domain, with little relevance to our formulation. (2) uses manually entered pre- and post-conditions hand-entered manually, which we avoid. Similarly, (7) concludes that if you specify actions granularly enough, a classical planner will try everything and eventually find holes in a specification. In (16), a couple of methods are compared, but ultimately none served as inspiration for our approach. (17) is concerned with creating a repeatable and audit-able environment for cyber attacks and experimentation, and not with solving it.

B. Reinforcement Learning

This subsection starts by briefly describing the type of problems Reinforcement Learning (RL) deals with, follows with descriptions of basic algorithms and finishes with an overview of some useful extensions.

Along with Supervised and Unsupervised Learning, RL is the third main paradigm in Machine Learning. It concerns itself with finding the best strategy for maximizing cumulative reward in a sequential decision-making problem. The most distinctive trait is that a RL algorithm is in charge not only of learning from observations of the environment, but also steering the way new experience flows in. Figure 1 shows the the RL interaction flow. It is composed of an *agent* that selects an action a (from the action space $\mathcal{A}$) based on the observed state s (from state space $\mathcal{S}$) of the *environment*. It then adjusts itself based on the received reward $r \in \mathbb{R}$. This process is repeated until the environment reaches *terminal* state s and the *episode* ends.

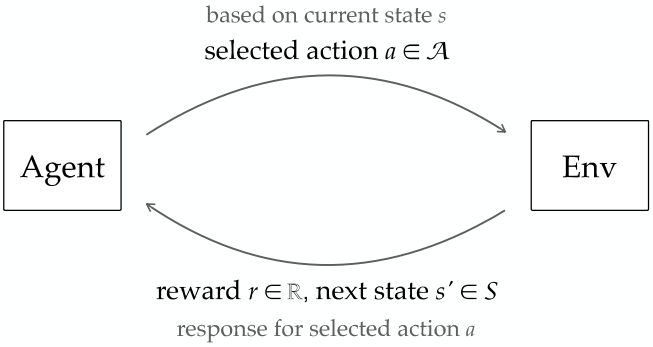

Fig. 1. Reinforcement Learning process

One common RL method, Q-Learning (QL), builds a state-action *value* table $Q(s, a)$. It holds the utility estimates of taking action a in state s, for all actions tried in every encountered state. The pseudocode is presented in Algorithm 1. The key instruction lies on line (10), where the Q value entry is updated towards the observed reward plus estimated future value. The algorithm's parameters are the discount factor $0 < \gamma < 1$, usually close to one and the learning rate (or step size) $0 < \alpha < 1$, usually very small and the exploration probability $0 < \epsilon < 1$. The discount factor controls how much immediate rewards are preferred over delayed ones. Low values produce more hedonistic behavior while high values lead to more visionary behavior. One of the most common action selection strategies is *ϵ-greedy*. When prompted for the next action to take, from state s, the agent acts greedily w.r.t the value estimations by exploiting current information: $argmax_a Q(s, a)$. By acting greedily, the strategy may get stuck in a local optimum. To enable exploration of alternative, potentially better routes, a random action is selected, with probability ϵ.

Algorithm 1: Q-Learning with ϵ-greedy action selection policy

```
1  Initialize Q(s,a) arbitrarily, for all s ∈ S, a ∈ A
2  foreach episode do
3      s ← environment's initial state
4      repeat
5          if exploring (with probability ε) then
6              a ← sampled randomly from A
7          else if exploiting then
8              a ← argmax_i Q(s,i)
9          r, s' ← environment's reaction to taking action
               a
10         Q(s,a) ←
               Q(s,a) + α[r + γ max_i Q(s',i) − Q(s,a)]
11         s ← s'
12     until s is terminal;
13 end
```

The state-action value table can be replaced by an approximate model, such as a Neural Network (5). This algorithm is called Deep Q-Network (DQN) (27) and is able to leverage state similarities to obtain better generalization. There is one input for each state feature and one output for each action. The mechanics are similar to QL. It brings the following changes to Algorithm 1: line (1) is replaced with random network weights initialization. Line (8) changes to the network predicting the action values in state s and taking the index i with the highest value. On line (10) updating $Q(s, a)$ involves training the network on the $(s, \bar{a})$, where the vector $\bar{a}$ is the output of the network on state s, but with the a^{th} position changed to the target value.

A third method, less popular lately, uses a Linear Classifier System (3), which is a rule-based Machine Learning algorithm in which the rule-discovery component is handled by a Genetic Algorithm. States and actions are encoded as binary strings and rules take the form *if in state 010# then take action 101*, where the wildcard character # matches any value. Extended Classifier System (XCS) (9) is a variation designed specifically for RL. The mechanics are, again, similar to QL. Starting from Algorithm 1, line (1) changes to create a new random rule on reaching a previously unencountered state s. Line (8) changes to select the rule that estimates the best reward and is the most accurate. Line (10) involves updating the fitness and statistics of the rule that selected the action. Additionally, genetic operators are applied to the rule-set population to achieve optimization and exploration.

Classical extensions to Q-Learning include decreasing the exploration rate and learning rate as the agent obtains more experience, to enable refinements of built strategy (38). Another modification is using the average of the values of actions in the next state as a future estimation instead of the maximum (this algorithm is known as SARSA), for more cautious behavior – consider the most probable outcome instead of the best one (38). Initial values also have a high impact on the algorithm's behavior: instead of all zeros, they can be sampled from a normal distribution with zero mean and small variance; or can have prior knowledge imbued, by assigning higher values to desired actions. Another enhancement consists of computing return further into the future (38), instead of just one step ahead, which leads to a better estimation of the state's future value.

Besides the aforementioned classical extensions, standard DQN extensions include Double DQN (14), in which a second, identical network is used for predictions, updated periodically to the main network (or slowly, but constantly). This helps stabilize the learning progress and avoid spiraling out by chasing a moving target. Another extension is Dueling DQN (39), in which the value function $Q(s, a)$ is decoupled into state value $V(s)$ (the intrinsic utility of being in state s) and action advantage $A(a)$ (how much better it is to take action a over all possible choices). This has the benefit that the value stream is updated more often, especially in cases of multiple actions.

Other DQN-specific extensions target the modeling network. Bayesian Neural Networks are better equipped to deal with

uncertainty (13) and have their theoretical properties met by interspersing Dropout (37) layers. The Huber Loss function acts like the Mean Squared Error when the difference is small, and like the Mean Absolute Error when the difference is large, making the learning robust to outlier reward values, as does the Batch Normalization (20) technique. More recent DQN extensions include Distributional DQN (4), which estimates a distribution for each state-action value, instead of a single number. Noisy Nets (12) are a way to inject exploration constraints straight into the estimation process, by adding parametric noise to the network weights.

In both tabular and approximate methods, encountered transitions (state, action, reward, next action) can be kept in a *memory buffer* for future revisiting, after value estimates have improved. A particularly effective extension is Prioritizing Experience Replay (33) in which transitions are sampled proportional to the error produced by the model when predicting their values. It favors transitions the model has much to learn from. Another alteration is to show more than just the latest state for deciding on an action, in order to capture temporal relationships. Previous states can have their features concatenated, or, in the case of DQN, Convolutional or Recurrent layers can be employed.

The described ϵ-greedy action selection policy can be exchanged for one that enables more guided exploration. In a Boltzmann policy, instead of sampling randomly, actions are selected proportional to their estimated values, with the tendency to sample uniformly decreasing as more experience is gathered.

The efficiency of algorithms and extensions described above is measured through the prism of common RL benchmarks. One of the most common testbeds is the Atari environment (26), in which the agent learns to play arcade video games. Combining the extensions described has been shown to yield better performance than any individual one (15). It has been shown that the amount of observations the agent is exposed to is paramount to achieved performance (18).

This paper aims not to introduce the pentesting problem as a new RL benchmark, but to apply established RL algorithms in this domain. One common trait of well-performing algorithms on the Atari environment is the use of Convolutional Neural Networks. They bring not only computational efficiency through fewer connections but also allow for deeper architectures. The pentesting environment cannot reap such benefits as the training data (described in III-C) is missing both the homogeneity and the spatial relationships of screen pixels. This holds except for, perhaps, convolution through states history. But the state features adhere to (Partially Observable) Markov Decision Process constraints, meaning they fully describe past events, unlike the velocity of a ball, for example, which cannot be determined from a single static screen frame.

Not much research attention has been dedicated to the joint topic of RL in pentesting. Existing work focuses on describing popular AI techniques and briefly touches on Cyber-Security applications, much less on attacker emulation (8). Another approach uses QL as one of the tested algorithms but focuses on a limited environment (40). Evolutionary Algorithms have been used to find the best order of patching vulnerabilities, however, requiring a human expert to rank the vulnerabilities' impact (22).

III. Environment Definition

This section describes the model of the pentesting problem we designed: environment rules, actors interaction and what their objectives, restrictions and available actions are. We strive to reach an abstraction that is both feasible to implement, so RL methods can be applied, while also staying close to the real world, so the obtained strategy is relevant in a real setting.

The scarcity of previous in-depth approaches in the automated pentesting literature caused a very large portion of the project's development to be spent on environment formalization, and possibly be prone to empirical biases.

We formalize a penetration test as a security game between an attacker and a defender. The game takes place on a network, with a single entry node. At each discrete time-step, the attacker picks one out of the available actions and the defender can counter-act. Additionally, the grey agent has no objective and simulates the behavior of benign users, performing their usual behavior unknowingly. The attacker's objective is to compromise as much of the network, as quickly and quietly as possible, while the defender actively protects it.

As an illustrative example, the final state of a game ending in a successful attack is shown in Figure 2. Not all the steps can be reproduced by a single static snapshot of the network, so a possible path reaching here is described below. The attacker compromised on machines 1 and 7, obtained elevated permissions on machines 2 and 5 and has lost its foothold on machine 0 after a detection. On its way to exfiltrating data from the goal (machine 7), it made use of credentials obtained from machine 1, where it also obtained persistence; cleaned up after performing steps late into the attack on machine 5. The defender is unaware, but suspicious, of which actions the attacker performed, and on what machines. Because of previously getting detected, the attacker triggered the defender to block access to machine 4. The failure of an attack could be caused by the defender blocking key actions on specific machines, leaving the attacker no available moves. The defender cannot possess unlimited blocking resources as they are an abstraction of human intervention. In the real world, they would come at the cost of the security expert's time, computing resources, and the defended system's productivity.

A. Network

The network is modeled after an empirical observation of common enterprise topologies. Connections are bi-directional and are made up of multiple *star* components. The entry point and the goal point are located on diametrically opposed parts

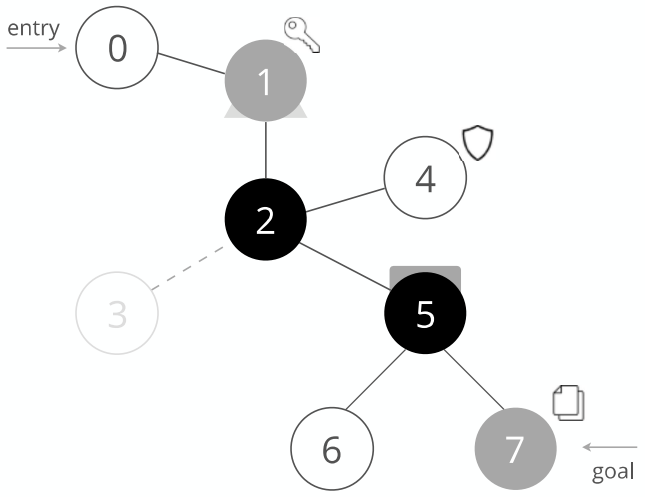

Fig. 2. Network at the end of an attack

of the network, with one of them standing at the end of a line path.

Each machine is assigned one or more local users. Each user belongs to one type (global admin, software developer, non-technical employee). Machine vulnerabilities are governed by assigned user type.

The local users collectively represent the grey agent. It does not act adaptively, and its purpose is to model the noise found in the real environments. At each time-step the agent has a small probability to perform one of:

- Reboot a machine, clearing foothold status obtained by the attacker.
- Log in to another machine, making the admin's credentials available there as well.
- Install new software, adding vulnerabilities to a machine.

B. Attacker

The attacker simulates a pentester, who, in turn, mimics a real hacker. Its actions follow a subset of MITRE's ATT&CK (24) model of cyber adversary behavior:

- Reconnaissance: gathering information prior to exploiting
  - *enumerate* a machine to reveal its connections
  - *scan* a previously discovered machine in order to reveal the exploits it is vulnerable to
- Gaining foothold: exploiting to gain entry or a stronger foothold
  - *exploit* a revealed vulnerability
  - *login* using previously dumped credentials
  - *migrate* from another machine (lateral movement)
  - *escalate* existing session privileges (elevate permissions)
- Post-exploit: complete attack objectives; available only on compromised machines
  - *persist* in order to gain resilience against reboots and detection
  - *dump* credentials of local admins
  - *exfiltrate* sensitive data on the machine
  - *cleanup* to reduce footprint of previous actions
- Auxiliary
  - *wait*, to let the defender's suspicion cool-off
  - perform *evasive maneuvers* to reduce the next action's noise
  - *abandon* when payoff is considered lower than risk

Each action has a number of properties, which naturally lend themselves to costs:

- *reliability*: probability of succeeding
- *duration*: time steps required
- *noise*: chance of detection
- *crash* probability (only for exploits)

To maintain action properties anchored in real facts, we set them according to widely-accepted categorizations. The exploits are parsed from Metasploit (23) (most popular pentesting framework) source code. Their associated costs are an aggregation of NIST's NVD (6) scores, mainly *attack complexity*, *exploitability score*, *impact score*, *user action required* and *publication date*. Other action properties are estimated manually by penetration testers and security researchers.

C. Shown Information

The attacker has access to information that characterizes the environment at the current time step. It does not have access to full information – the process remains partially observable. To avoid giving the agent an unfair advantage over manual pentesters, the automated attacker is shown only the information that a human could easily deduce/find while carrying the pentest.

Designing informative yet distinctive state components is a big challenge. We started by asking experienced pentesters what features they look at when conducting an attack: how do they decide what to do next and how do they assess the value of a machine. Unfortunately, the answers were inconclusive, as they targeted hard to quantify properties, e.g.: "knowing from experience". The most relevant ideas were to show the Operating System, open ports and the number of connections. We designed the environment with these properties in mind, but they are not sufficient to differentiate machines and are not indicative of the machine's value in the attack.

Another way we posed the question was "if one of your (also experienced) pentester colleague was in the middle of an attack, stopped at one point and let you take over, what information would you need to have passed over?". Another way to look at it is describing what information would a complete novice need to look at, in order to perfect its pentesting skills. In the end, we converged to the following state features:

- actions performed, and on what machines
- what actions *available* to be taken next
- machines compromised, connections discovered, user credentials obtained
- action properties (reliability, duration, etc)

Action availability follows the rule of "what would make sense". For example, enumerating the connections of an already enumerated machine is useless; an exploit will not be attempted blindly, before discovering that the machine is vulnerable to it; if foothold has been achieved, there is no use in further scanning for vulnerabilities; etc.

D. Defender

The defender agent models counter-actions done automatically by an anti-virus solution or manually by a security officer. In relationship to the attack's time, possible defender actions are:

- *instant* detection at the time of the attack, based on the action's noise
- *investigate* machines, possibly targeted in the past by the attacker, based on the defender's suspicion
- *prevent* future attack targets by blocking possible targets

On detection, the attacker is kicked off the machine, losing foothold status (gaining persistence circumvents this) and warranting more attention. Additionally, if the detection was the result of an investigation, the entry method is patched. The defender's suspicion level (used in investigation) increases as multiple actions are done in rapid succession; and decreases over time, in periods of low attacker activity. One "attention resource" is earned by the defender after each detection. They are allocated to protect key places, such as valuable information or gateway nodes. One resource blocks a single action on a selected machine.

E. Evaluation

An episode ends when the objective has been reached, such as exfiltrating data from the target machine or compromising a total number of machines. On the other hand, it also ends if there are no more moves available, as a result of efficient defender measures. It can also end prematurely if the attacker decides to give up.

The reward given after each action is the one that guides agent behavior (learning algorithms are briefly described in II-B). By changing actions that are rewarded positively, we can encourage the pursuit of different objectives. By changing actions rewarded negatively, we can warn the agent about effects it should be cautious about.

The main performance metrics are swiftness (time steps taken) and stealthiness (inverse number of times detected). Other, more detailed metrics are: the percentage of machines compromised, percentage of machines data has been exfiltrated from, and percentage of credentials stolen.

IV. Attacker Agents

This section describes algorithms used to find the attacker's strategy. There are two kinds of agents: those operating based on a fixed strategy and those that learn from experience. The latter match the RL algorithms described in II-B, applied in the pentesting environment.

A. Fixed-strategy Agents

The algorithms described in this section do not adapt to the environment's response. They are used as a baseline for evaluating learning agents performance, and can also be used to enhance them with initial knowledge, giving them a head-start and a faster convergence.

The simplest fixed-strategy agent is the Random agent. For any given state, it selects uniformly out of the available actions. Another conceptually simple agent is the Greedy agent. It behaves in accordance with a pre-defined list of preferences. If the most preferred action is available, it always selects that (machine chosen randomly), if not, check the next one, and so on. If none of the preferred actions are available, it acts randomly. The final fixed-strategy is the Finite State Machine (FSM). It acts in pre-defined order. If the action on the first place is available, it selects that (machine chosen randomly) and advances to the next action place. If the action on the current place is unavailable, it acts randomly. The strategies for Greedy and FSM were set after an exhaustive search of possible permutations of a subset of actions.

While fixed-strategy agents provide a lower bound for learning agents' performance, the upper target is given by manual performance. The manual run is comprised of steps picked by a human pentester, having full knowledge of network topology and credential locations. The strategy efficiently navigates towards the goal, in a swift and stealthy manner.

The full list of actions is given in Appendix A.

B. Reinforcement Learning Formalizations

This subsection describes the components used by learning-based agents. They are concrete implementations of environment concepts described in Section III, namely state, action, and rewards.

1) State: Information about the current environment state, as described in Section III-C, is mainly made up of information about already performed and currently available actions. Classical environments feature "localized" state features, that change greatly with interaction. Such as the x and y positions in a 2D maze; the 3D coordinates and velocity of a robot arm or the pixels/RAM content when playing a video game. However, in the case of pentesting, a straightforward "local" state definition is not so evident. One idea is to show information just about the current (i.e.: last acted on) machine. But then the agent would lack the option to operate on any other than that. Another idea would be to show information about the current machine and all its neighbors. Besides padding issues (for all machines other than the most connected ones), this brings the shortcoming that one cannot operate on a machine close to the goal, then search for credentials on a distant machine, close to the entry, and then come back, which could be part of an efficient strategy.

There is also the issue of feature selection. Even though multiple sources information (about *performed*, and *available* actions) can be shown, showing its entirety is not guaranteed to benefit to the learning algorithm. Some of it may be redundant.

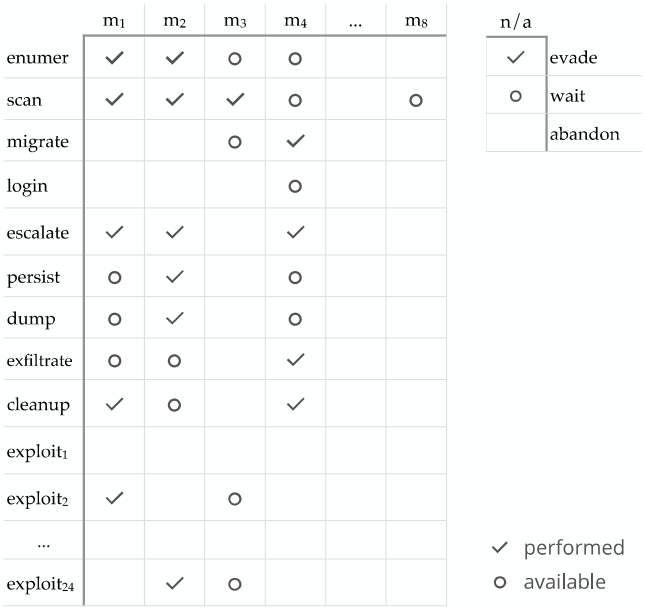

Fig. 3. State features in the middle of an attack

Furthermore, information overload can even degrade agent performance. An algorithm that achieves good performance while looking at only a couple of state features may become overwhelmed and do much worse when presented with the full information. Even though the entirety of information might, in theory, enable reaching higher possible performance. Thus, it might be better to ignore some sources of information or the possibility of taking some actions entirely.

Another step in this direction is to reduce the number of exploits (grouped under the action *exploit* defined in III-B). This offers the agent a single, simplified action: *exploit*. The specific exploit is decided automatically, based on a pre-defined manual ranking.

As a concrete example, consider an environment of 8 machines (max connectivity degree 4) and 24 exploits, with the full information shown. There are 267 possible actions, the cartesian product of (*machine*$_1$, ..., *machine*$_8$) and (*enumerate, scan, login, migrate, escalate, persist, dump, exfiltrate, cleanup, exploit*$_1$, ..., *exploit*$_{24}$) plus the three un-targeted actions *wait, evade, abandon*. The action availability and performed actions vectors have the same length. The restricted representation described in the previous paragraph brings state size (which is the same as the number of moves) down to 52 (5 machines, 9 targeted actions, 1 simplified exploit and 2 un-targeted actions).

The state representation described above is visualized in Figure 3. It shows a matrix where each row represents one action and each column one machine (except for the last column which shows un-targeted actions). For each cell, the agent can know one of the following: it has been performed (successfully); it is available to be chosen as the next action; or it hasn't attempted it and knows nothing about it.

2) Reward: The reward function closely follow the description in III-E. A small positive amount is awarded for gaining foothold (only for the first time on each machine), exfiltrating data and obtaining credentials and a large one is obtained for completing the goal. A large negative penalty is incurred upon detection by the defender, to encourage stealthiness and a small negative reward is applied for each time step, to favor swiftness.

The sparsity of rewards, a cornerstone issue in RL, also came up while designing the reward function. One early formulation consisted of providing a positive reward only when the objective is completed and a negative penalty per time-step. But the agents learned that it is better to wait and incur the small penalty, unknowing that there is a large payout worth exploring for. Another formulation, aimed at guiding the agent towards the goal faster, awarded reward inversely proportional to the goal machine proximity. It worked well, but we felt it provided an unfair advantage to the agent, as a real attacker would not have access to such information.

3) Practical Considerations: To avoid waiting indefinitely, a limit is imposed on the number of maximum consecutive *wait* actions. Also, to speed-up training and avoid dead-ends, the maximum number of moves is capped.

One particularly tricky action was *abandon*. As it causes the episode to end instantly, it has the dangerous effect of throwing away current progress, if chosen as part of a random exploration. For this reason, it was either disabled for fixed-strategy agents or assigned a large negative reward for learning agents, to dissuade unfounded usage.

A very large number of episodes is usually needed for RL algorithms to converge. For this reason, fast simulation of the environment a requirement. As the rules of the environment are computed entirely on boolean arrays, we obtained significant speed-ups (over 40x) when switching the representation from a list of numbers to a single integer, viewed as base two and using bit-wise operators.

C. Learning Agents

We experimented with three kinds of learning agents: tabular Q-Learning (QL), Extended Classifier Systems (XCS) and the Deep Q-Network (DQN), defined in II-B. For each of them, the input state representation is a bit different. As described in the previous subsection, each observation is a list of booleans. To make indexing easier, for QL, the list is treated as a number in base two and converted to base 10 (e.g. (1, 1, 0) will be indexed under 6). The state representation matches the XCS input, as it already expects a binary string. For DQN, the binary numbers are fed in directly.

Figure 4 shows the evolution of total reward received, by each learning agent. An agent's performance is evaluated periodically during training, with exploration functions disabled. Due to the stochasticity of the environment, at each evaluation, 50 episodes are run and their average is presented. The thin line shows the actual reward, while the thick line

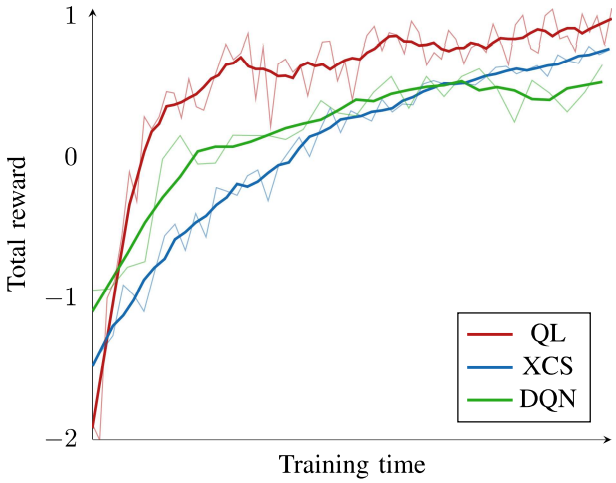

Fig. 4. Training progress of the learning agents

is a rolling average over 5 previous and following periodic evaluations. The first, left-most, points for each agent show performance achieved after training for a single episode. The starting performance differs from one algorithm to another as it is largely influenced by the random initialization of each. Because QL is relatively fast, compared to XCS and DQN, it is able to process more episodes in the same period of time. To make the comparison fair, the agents were ran for approximately the same amount of time (∼40h), in which they have experienced different numbers of episodes (details in Appendix A).

QL improves quickly at first and then refines the built strategy. It produces considerable variance from one episode to another. One interesting fact is that the evaluation is not monotonously increasing: forgoing the best strategy, exploring worse ones enables arriving at a better one in the end. XCS has a steadier improvement and is less noisy. This could be thanks to the natural mapping of state features to rule formation. DQN achieves modest increase and shows relatively high variance during training. It requires much longer training times, which also why it was run for fewer episodes.

Modern RL advancements are focused on DQN and other approximate models, as they perform best in classic settings such as Atari games. In our pentesting environment though, the tabular QL approach achieved better performance, at least when comparing with the same training time. This advantage was only possible after adapting some DQN extensions to QL.

Another significant improvement was brought about by initializing action values according to the Greedy agent preference. Due to the nature of the problem, neither extreme aggression nor complete lack of risk characterizes a good agent. One way an agent assesses this is through future value estimation, by taking either maximum or the average of action values in the next state. To strike a balance between the two, we introduce an *idealization coefficient* in computing the estimate of future value:

$$\text{future} = \text{avg}_a + \eta(\text{max}_a - \text{avg}_a)$$

where $0 \leq \eta \leq 1$ is the idealization coefficient, and *avg* and *max* refer to action value estimates in the next state.

The interpretable η hyper-parameter could give way to generate diversity even among multiple configurations of the same algorithm. Comparable-strength agents with different risk-taking levels can help test the security solutions from multiple angles. Continuing in this direction, we encouraged model parsimony (lower population size, fewer and more shallow network layers, etc), as to increase the possibility to gain insight into the agent's decisions and thus understand how to counter it.

An exhaustive search of possible algorithm configurations was not possible, due to the large number of hyper-parameters, as well as the duration needed to decide whether a certain configuration is promising. As the other extreme, random search, was not a viable alternative either, we opted for a Bayesian Optimization (21) strategy for tuning model configurations. Due to the large computational effort for evaluating a single hyper-parameter setting, high exploitation rate (for the optimization strategy) and only few initial random configurations were utilized. The full list of optimization process parameters is given in Appendix A.

One impediment we encountered was the lack of openly available implementations for Genetic Algorithm methods. Also, development speed would have been greatly increased by had there been mature model experimentation frameworks.

V. Experimental Results

A. Parameter Settings

As opposed to classical RL benchmark problems, where the discount factor γ is set very close to 1, in this setting, lower values performed better. This is indicative of the relatively small number of actions that can be done in an episode. It forces agents to focus on high-value actions, and not waste time on negligible ones.

The η parameter proved to be useful. The best-performing configurations of each agent feature different values of η, but none of them too close to either extreme. This shows that neither full risk nor full caution brings the best results, but a balance between them.

QL excelled with a relatively high learning rate and gradual decay. This benefit could be a direct cause of the fact that its values were not randomly initialized and it was able to continue from where the greedy agent left off. DQN did best with a small network size, both in terms of width and depth. Larger configurations would have had the potential of matching or even surpassing this performance in the long run, but if left for an equal amount of wall time, so could the smaller configuration improve as well. XCS thrived when the wildcard probability was set to lower than a third. This could

be caused by the condensed feature representation, in which each position entails useful information.

For the issue of feature selection, the best-performing agents look at action availability, take into consideration almost all actions, and restrict machines seen to the current and its neighbors. This is likely caused by the fact that the less complex configuration is handled much better, even though it has a lower performance ceiling than the more complex configuration which is easy to get lost in, even though it makes it possible to achieve a higher performance.

The environment and agents were run under Python 3.6 in CentOS 7, on Intel Xeon E5v3 CPUs and Tesla Nvidia K80 GPUs. DQN's neural networks use the Keras (19) implementation, most part of XCS uses an open-source implementation (19), the Bayesian Optimization process uses an open-source implementation as well (29).

We encountered two main challenges in applying RL methods in this research. First, the dissimilarity from classical RL environments and more specifically typical modern benchmarks means that the good performance reported there does not necessarily transfer to this case. One big difference is the large number of actions and the fact that not all of them are available at the same time.

Second, the long training times (upwards of 60 hours) of the agents, coupled with high sensitivity to and large number (for example, more than 20 in XCS' case) of hyper-parameters made it extremely hard to find good configurations. On top of this, GPU computation offered no speed-up. The environment's rules are computed on CPU. QL and XCS use mainly random-access indexing, which GPUs are inefficient at. The small network and few training epochs of DQN make the GPU overhead outweigh computation speed-ups.

B. Agent Comparison

Figure 5 shows the total reward received by each agent. The evaluation is done at the end of training (for learning agents), on the best version of the algorithm (this includes fixed-strategy). Box-plots of 25 evaluations are used in order to showcase not only median results but also performance variance and best/worst cases.

The Random agent displays very high variance, sometimes doing relatively well, sometimes disastrously bad. FSM, while obtaining better overall performance, is still highly unreliable. Greedy, on the other hand, reduces the variance significantly while, at the same time, achieving a higher overall reward. This came as a surprise, as the Greedy algorithm is simpler than FSM, so we expected quite the contrary.

The *Human* agents represent the best performance achieved by the authors in the same environment. It does not represent the performance achieved by the most experienced pentester and is solely used as a relative benchmark for the learning agents. One thing it excels at is high predictability: even though it may not perform optimally, a very similar result is obtained regardless of environmental variations.

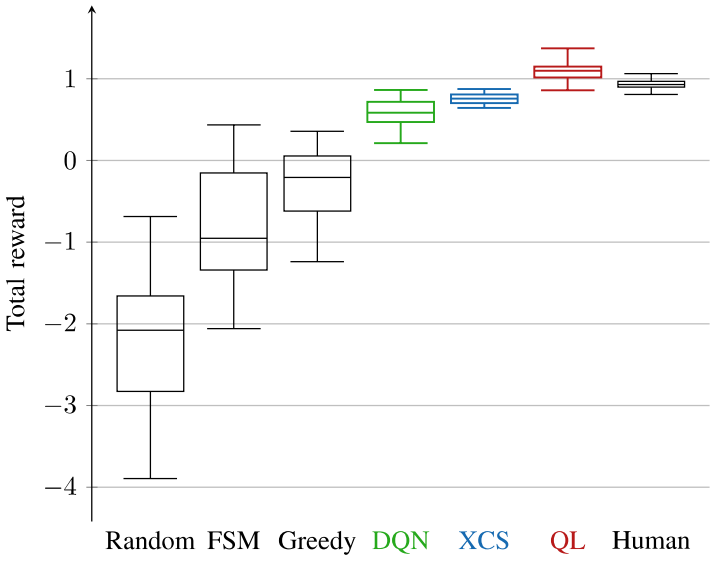

Fig. 5. Distribution of each agent's performance

The best performing agent is QL, which manages to surpass human performance, both on average and for the best run. XCS yields just below human performance but features less variance than QL. This could turn out to be equally valuable: a steady behavior, rather than an erratic one, may come closer to what real experienced attackers display. DQN is the worst performer out of the learning agents, having the most variance, and sometimes even obtaining uncharacteristically low scores. Nevertheless, it performs considerably better than fixed-strategy agents. While DQN has more generalization power and could be able to learn more complex relationships, the large training times, the amount of and the sensitivity to hyper-parameters make it hard to steer.

Each of the fixed-strategy agents performs bad, relatively to learning agents. But if we compare learning strategies among each-other, the disparities become more evident. For example, DQN does much better than even the best fixed-strategy agent (Greedy), but compared to QL, or even XCS, it achieves only moderately good performance.

Upon inspection of the other metrics (described in III-E), we get an overview of how the different agents act. QL is, understandably so, the fastest, while XCS hast the highest number of moves and time steps, perhaps indicative of a methodical approach. DQN has a higher propensity to exfiltrate data and chooses to ignore obtaining admin credentials, while the exact inverse is true for QL. XCS shares similar preferences to QL, which could indicate that these two algorithms have found valuable paths, in the current scenario configuration (detailed in Appendix A. Surprisingly, DQN gets detected fewer times than QL, even though it obtains a lower reward, which suggests that a more aggressive strategy is better, in the given environment. All three algorithms obtain a similar and fairly high percentage of machines compromises, which could be caused by the small network size.

VI. Conclusion and Future Work

This section offers a summary of goals achieved in this paper and discusses possible future directions.

A. Goals Achieved

In this paper, we formalized penetration test as a Reinforcement Learning problem. We measured the performance of multiple fixed-strategy (Random, Greedy, Finite State Machine) and learning-based (Q-Learning, Deep Q-Network, Extended Classifier System) agents. Q-Learning, with some extra techniques applied and greedy agent initialization, performed best, surpassing human performance in the given environment. This work shows how manual penetration testing shortcomings can be overcame by finding attacker strategies through Machine Learning methods.

B. Prospective Next Directions

Other algorithms for learning the attacker's strategy can be experimented with, aiming to mitigate current shortcomings. Methods making use of policy-gradient, such as A3C (25), could better deal with the large number of actions. The issue of dissimilarity from usual algorithm benchmarks could be mitigated by using an algorithm robust to hyper-parameter settings, such as TRPO (34) or ES (31). Generalized Decision Trees (3) can be seen as the natural progress from XCS.

The current work measures the outcome of simulating multiple attacker agents in a single environment, but it would be interesting to note their advantages on various network topologies, defender strength levels, and environment difficulty levels. To this end, a method for automatically generating valid enterprise networks would be useful, which could integrate evaluation tools such as the one presented in (30).

More sophisticated defender algorithms can be tried. Game Theory concepts can be applied, specifically treating the problem as a Stackelberg Security Game (28). Specifically, the allocation of action-blocking "attention resources" can benefit from this. These techniques have been successfully applied in physical security domains such as finding the optimal allocation of airport security officers, given incomplete knowledge of the attacker, and many more points to defend than staff available.

A more faithful representation of the security environment can be pursued. The current formulation models short-term attacks, where longer operation times are explicitly penalized. Starting points would be to change the waiting-duration dynamics and implement remaining ATT&CK actions, such as *collection* (key-logger, webcam, etc) or *command & control* (periodically communicate with an external entity). Measuring long-term impact would enable modeling attacks that focus on stealth and longstanding infiltration.

The final outcome of this project is a way to more easily test security solutions. Developing an efficient simulated attacker is the main problem. Naturally, after satisfactory performance has been achieved in the simulated environment, the next step would be to test its efficiency in the real world. This would involve mapping abstract actions to real commands and translate the system state into feature vectors. Having done this, he simulated defender can be replaced by actual security software, and the agents can be benchmarked against real security solutions.

APPENDIX

This section details the environment which the attacker algorithms have been evaluated in. Network topology and machines information is given in Table I. Costs of each attacker action are given in Table II, and in Table III participating exploits. Table IV lists rewards given in response to attacker actions (*time* is multiplied by action's duration).

TABLE I
NETWORK TOPOLOGY

Machine	Conns	User	OS	#Vulns	Admins
0 entry	1	dev	windows_xp	7	dev1
1	0 2	nontech	windows_8	6	a2, nt1
2	1 3 4	admin	debian_linux	3	a1, a2
3	2	nontech	windows_vista	7	a2, nt2
4	2 5	dev	windows_10	4	a2, d1
5	4 6 7	dev	ubuntu_linux	4	a1, a2, d2
6	5	admin	wserver_2012	3	a1, a2
7 goal	5	admin	wserver_2016	2	a1

TABLE II
PROPERTIES OF ATTACKER ACTIONS

Action	Reliability	Noise	Duration	Reduction
enumerate	0.95	0.075	4	
scan	1	0.025	4	
escalate	0.85	0.15	1	
persist	0.95	0.025	1	0.95
dump	0.9	0.1	1	
exfiltrate	0.9	0.1	3	
cleanup	0.9	0	3	0.95
migrate	0.8	0.1	1	
login	0.95	0.025	1	
evade	0.8	0	3	0.5
wait	1	0	4	
abandon	1	0	0	

TABLE III
AVAILABLE ATTACKER EXPLOITS AND THEIR PROPERTIES

CVE	Reliability	Noise	Duration	Crash
2008-2992	0.875	0.2	4	0
2008-5353	0.99	0.2	1	0.1
2009-3459	0.875	0.2	4	0
2010-0840	0.99	0	1	0.1
2010-0842	0.95	0	1	0.1
2011-2371	0.75	0.2	1	0.1
2011-3556	0.99	0	1	0.1
2011-3659	0.65	0.2	1	0.15
2012-0897	0.75	0	4	0
2012-1533	0.99	0.2	1	0.1
2012-1775	0.75	0.2	4	0
2012-1823	0.99	0	1	0.1
2012-3993	0.99	0.2	2	0
2012-4681	0.99	0.2	1	0.1
2013-0753	0.75	0.2	4	0
2013-0757	0.99	0.2	4	0
2013-1493	0.75	0.2	1	0.1
2013-2465	0.95	0.2	1	0.1
2013-3205	0.75	0.2	4	0
2014-1511	0.99	0.2	2	0
2014-3704	0.99	0	1	0.1
2014-6352	0.99	0.2	4	0
2016-2098	0.99	0	1	0.1
2017-11882	0.25	0.2	4	0.2

TABLE IV
REWARDS GIVEN FOR ATTACKER ACTIONS

Action	Reward
time	-0.01
goal	1
detection	-0.05
foothold	0.1
exfiltrate	0.05

This section lists the parameters we found work best, for all agents. FSM order and Greedy preference is presented next, followed by feature selection results. Table V lists the parameters of the hyper-parameter optimization process and Tables VI, VII, VIII show the parameters of QL, XCS and DQN, respectively. The strategy chosen by the human agent is given in Table IX.

Finite State Machine agent order:

1) login
2) enumerate
3) escalate
4) migrate
5) exfiltrate
6) dump

Greedy agent preference:

1) exfiltrate
2) escalate
3) login
4) enumerate
5) migrate
6) dump

Feature selection:

- shown performed actions: none
- shown actions availability: all
- reduce exploits: yes
- only neighbors: yes
- disallowed actions: *abandon*

TABLE V
BAYESIAN OPTIMIZATION PROCESS PARAMETERS

Parameter	Chosen value
Acquisition function	UCB
κ	2
GP kernel	Matern (generalized RBF)
Matern ν	2.5
Matern α	1e-10
GP optimizer	L-BFGS-B
Initial Observations	16

TABLE VI
Q-LEARNING HYPER-PARAMETERS

Parameter	Found value	Sensible range
Boltzmann τ	2	[0.5, 10]
Discount γ	0.87	[0.5, 0.999]
Exploration ϵ initial	1	[0.2, 1]
Exploration ϵ decay	0.999994	[0.9999, 0.999999]
Exploration ϵ min	0.1	[0.3, 0.0001]
Idealization η	0.6	[0, 1]
Learning rate α initial	0.158	[1e-7, 1]
Learning rate α decay	0.999995	[0.9999, 0.999999]
Learning rate α min	0.0001	[1e-10, 1e-7]
Episodes	150,000	

TABLE VII
XCS HYPER-PARAMETERS

Parameter	Found value	Sensible ranges
accuracy_coefficient	0.1	(0, 1]
accuracy_power	5	(0, 100)
crossover_probability	0.85	(0, 0.95)
deletion_threshold	50	[0, 500]
discount_factor	0.9	[0.5, 0.999]
do_action_set_subsumption	yes	{yes, no}
do_ga_subsumption	yes	{yes, no}
eps_decay	0.999985	[0.9999, 0.999999]
eps_min	0.01	[0.3, 0.0001]
error_threshold	0.01	[0, 1]
exploration_probability	0.65	[0.2, 1]
fitness_threshold	0.1	[0, 1]
ga_threshold	20	[0, 100]
idealization_factor	0.9	[0, 1]
initial_error	1e-5	(0, 0.1]
initial_fitness	1e-5	(0, 10]
initial_prediction	1e-5	(0, 1]
learning_rate	0.1	[1e-7, 1]
lr_decay	0.999986	[0.9999, 0.999999]
lr_min	0.01	[1e-10, 1e-7]
max_population_size	200	[10, 5000]
minimum_actions	1	[1, 100]
mutation_probability	0.1	[0.01, 0.9]
subsumption_threshold	25	[0, 100]
wildcard_probability	0.2	[0.05, 0.9]
episodes	50,000	

TABLE VIII
DQN HYPER-PARAMETERS

Parameter	Found value	Sensible options
batch_normalization	yes	{yes, no}
batch_size	32	[1, 8192]
discount	0.9	[0.5, 0.999]
double	yes	{yes, no}
dueling	yes	{yes, no}
exploration_anneal_steps	20,000	[5,000; 25,000]
exploration_min	0.05	[0.3, 0.0001]
exploration_q_clip	(-1,000; + 1,000)	$[0.1, 10000]^2$
exploration_start	1	[0.2, 1]
exploration_temp	2	[0.5, 10]
exploration_temp_min	0.2	[0.001, 1]
hidden_activation	relu	{sigmiod, tanh, selu }
hidden_dropout	0.4	[0, 0.9]
history_len	1	[1, 5]
idealization	0.7	[0, 1]
input_dropout	0.2	[0, 0.6]
layer_sizes	(128, 64)	$[8, 1024]^{[1,5]}$
loss	logcosh	{mse, mae, logcosh}
lr_decay	0.99993	[0.9999, 0.999999]
lr_init	0.1	[1e-7, 1]
lr_min	0.0005	[1e-10, 1e-7]
memory_size	50,000	[5,000; 1,000,000]
multi_steps	2	[1, 8]
n_epochs	1	[1, 10]
out_activation	softmax	{linear, softmax}
policy	max-boltzmann	{eps-greedy, max-boltzmann}
prioritize_replay	yes	{yes, no}
priority_exp	0.01	(0, 1)
priority_shift	0.1	(0, 5]
q_clip	(-10,000; +10,000)	$[0.1, 100000]^2$
streams_size	32	[4, 512]
target_update_freq	1,000	[100; 5,000]
weights_init	lecun_uniform	{uniform, normal, lecun_uniform}
episodes	25,000	

TABLE IX
STEPS TAKEN BY THE "HUMAN" AGENT

Action	Machine
evade	-
exploit	0
enumerate	0
scan	1
escalate	0
dump	0
migrate	1
enumerate	1
scan	2
exploit	2
enumerate	2
login	4
enumerate	4
scan	5
exploit	5
persist	5
escalate	5
dump	5
enumerate	5
login	7
escalate	7
exfiltrate	7

REFERENCES

[1] Andy Applebaum, Doug Miller, Blake Strom, Henry Foster, and Cody Thomas. 2017. Analysis of Automated Adversary Emulation Techniques. , Article 16 (2017), 12 pages.

[2] Andy Applebaum, Doug Miller, Blake Strom, Chris Korban, and Ross Wolf. 2016. Intelligent, Automated Red Team Emulation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, New York, NY, USA, 363–373.

[3] Thomas Back, David B. Fogel, and Zbigniew Michalewicz (Eds.). 1997. *Handbook of Evolutionary Computation* (1st ed.). IOP Publishing Ltd., Bristol, UK, UK.

[4] Marc G. Bellemare, Will Dabney, and Rémi Munos. 2017. A Distributional Perspective on Reinforcement Learning. *CoRR* abs/1707.06887 (2017). arXiv:1707.06887

[5] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[6] Harold Booth, Doug Rike, and Gregory A Witte. 2013. *The National Vulnerability Database (NVD): Overview*. Technical Report.

[7] Josip Bozic and Franz Wotawa. Planning the Attack! Or How to use AI in Security Testing?. In *IWAISe: First International Workshop on Artificial Intelligence in Security*. 50.

[8] Anna L Buczak and Erhan Guven. A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys & Tutorials* 18, 2 (????), 1153–1176.

[9] M. V. Butz and S. W. Wilson. 2002. An algorithmic description of XCS. *Soft Computing* 6, 3 (01 Jun 2002), 144–153.

[10] European Commission. 2017. Press release - State of the Union 2017 - Cybersecurity: Commission scales up EU's response to cyber-attacks. (2017).

[11] Richard Elderman, Leon J. J. Pater, Albert S. Thie, Madalina M. Drugan, and Marco Wiering. 2017. Adversarial Reinforcement Learning in a Cyber Security Simulation. (2017).

[12] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2017. Noisy Networks for Exploration. *CoRR* abs/1706.10295 (2017). arXiv:1706.10295

[13] Yarin Gal and Zoubin Ghahramani. 2016. Dropout As a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 1050–1059.

[14] Hado van Hasselt. 2010. Double Q-learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2 (NIPS'10)*. Curran Associates Inc., USA, 2613–2621.

[15] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. *ArXiv e-prints* (Oct. 2017). arXiv:cs.AI/1710.02298

[16] Joerg Hoffmann. 2015. Simulated Penetration Testing: From "Dijkstra" to "Turing Test++". (2015).

[17] Hannes Holm and Teodor Sommestad. 2016. Sved: Scanning, vulnerabilities, exploits and detection. In *Military Communications Conference, MILCOM 2016-2016 IEEE*. IEEE, 976–981.

[18] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. *CoRR* abs/1803.00933 (2018). arXiv:1803.00933

[19] Aaron Hosford. 2015. XCS Library. (2015).

[20] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167

[21] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

[22] Jüri Kivimaa and Toomas Kirt. 2011. Evolutionary algorithms for optimal selection of security measures. In *European Conference on Cyber Warfare and Security*. Academic Conferences International Limited, 172.

[23] David Maynor and Thomas Wilhelm. 2007. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research* (1st ed.). Syngress Publishing.

[24] MITRE. 2017. ATT&CK Adversarial Tactics, Techniques, and Common Knowledge. (2017).

[25] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *CoRR* abs/1602.01783 (2016). arXiv:1602.01783

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533.

[28] Thanh Hong Nguyen, Debarun Kar, Matthew Brown, Arunesh Sinha, Albert Xin Jiang, and Milind Tambe. 2016. Towards a science of security games. In *Mathematical Sciences with Multidisciplinary Applications*. Springer, 347–381.

[29] Fernando Nogueira. 2015. Bayesian Optimization Library. (2015).

[30] Emilie Purvine, John R. Johnson, and Chaomei Lo. 2016. A Graph-Based Impact Metric for Mitigating Lateral Movement Cyber Attacks. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense (SafeConfig '16)*. ACM, New York, NY, USA, 45–52.

[31] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864* (2017).

[32] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. 2012. POMDPs Make Better Hackers: Accounting for Uncertainty in Penetration Testing. *CoRR* abs/1307.8182 (2012).

[33] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized Experience Replay. *CoRR* abs/1511.05952 (2015). arXiv:1511.05952

[34] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015. Trust Region Policy Optimization. *CoRR* abs/1502.05477 (2015). arXiv:1502.05477

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[36] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1929–1958.

[38] Richard S. Sutton, Andrew G. Barto, and Harry Klopf. 2016. Reinforcement Learning: An Introduction Second edition , in progress.

[39] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[40] Key whan Chung, Charles A. Kamhoua, Kevin A. Kwiat, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2016. Game Theory with Learning for Cyber Security Monitoring. *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)* (2016), 1–8.