POMDP + Information-Decay: Incorporating Defender's Behavior in Autonomous Penetration Testing

Abstract

Penetration testing (pen-testing) aims to assess vulnerabilities in a computer network by emulating possible attacks. Autonomous pen-testing allows frequent and regular pen-testing to be performed, which is increasingly necessary as networks become larger and more complex. Autonomous pen-testing is a planning under uncertainty problem, where the uncertainty is caused by partial observability of the network, lack of reliability of attack tools, and possible changes in the network that are triggered by the network administrator (the defender). Autonomous pen-testing approaches that account for the first two causes of uncertainty have been developed based on the mathematically principled framework, the Partially Observable Markov Decision Process (POMDP). However, they do not account for the third type of uncertainty. Work that accounts for the defender's actions do not account for both partial observability and unreliability of the attack tools. ***This paper proposes a POMDP-based autonomous pen-testing framework that accounts for the defender's behaviour.*** Key to our model is the observation that the defender's actions can be abstracted into two types: Network analysis, which does not alter the network, and Counter-attacks, which alter the network. This observation enables us to represent the defender's behavior as a single variable: An information decay factor. This decay factor is based on the expected time the defender takes to move from analysing the network to performing counter-attack(s), and therefore represents the decay of a pen-tester's knowledge about the network. We propose D-PenTesting, which assumes the decay factor is known prior to execution, and LD-PenTesting, which learns the decay factor as it attempts to break into the network. Simulation tests on two benchmark scenarios indicate that D-PenTesting and LD-PenTesting outperform existing POMDP-based pen-tester and is more robust than one that incorporates a POMDP-based defender.

Introduction

Penetration testing (pen-testing) aims to identify vulnerabilities in a computer network by emulating a real attack. It is essential to ensure network security. However, due to the highly skilled and expensive specialists required, pen-testing is performed infrequently, if at all, which is unacceptable as networks become larger and more complex. These difficulties have led to the proposal of autonomous pen-testing. In this paper, we focus on a component of autonomous pen-testing: The problem of deciding which tools to use when, assuming the pen-tester is a single agent.

The above problem is a planning under uncertainty problem, where uncertainty is caused by (*i*) partial observability of the network properties and its vulnerabilities, (*ii*) the lack of reliability of the tools used, and (*iii*) the possible changes to the system triggered by the network administrator (aka., the defender). The first two causes of uncertainty have been modeled in various prior work (Hoffmann 2015), with robust approaches based on the mathematically principled framework Partially Observable Markov Decision Process (POMDP) proposed since 2011 (Sarraute, Buffet, and Hoffmann 2011). A POMDP agent acknowledges that due to uncertainty in sensing and actions, the agent never knows its exact state, and therefore, it estimates its current state as a distribution over the state space, called beliefs, and decides the best action to perform with respect to beliefs, rather than single states. Such a framework fits well for modelling the first two types of uncertainty in pen-testing problem. However, incorporating the defender's behavior is more difficult. In fact, to this end, autonomous pen-testing relies on the game theoretic framework (Liang and Xiao 2012). However, current game theoretic approaches do not account for partial observability of the network or unreliable attack tools.

This paper proposes a POMDP-based pen-tester that takes into account possible network changes triggered by the defender. To this end, we develop a compact defender model based on the observation that a pen-tester's knowledge about the defender's actions and strategies are only gained via observations about changes in the network. Therefore, we can abstract the defender's actions into two types: Network analysis, which does not change the network, and Counter-attack, which alters the network properties. Based on this abstraction, we model the defender's behavior as the time it takes to switch from performing network analysis to counter-attacks. This time can naturally be modelled as a Markovian Arrival Process (Asmussen 2010; Neuts 1979). In this paper, we adopt the simplest Markovian Arrival Process, which is the Bernoulli process, and model the defender's behaviour simply as the parameter for this Bernoulli process. We refer to this Bernoulli parameter as the decay factor, as it resembles a decay in the pen-tester's knowledge about the network.

Utilising the above model, we propose a POMDP-based pen-tester, called D-PenTesting, that incorporates the defender by augmenting the uncertainty in the effect of actions with the decay factor. Since the decay factor may differ according to a particular defender, it's value is generally unknown in advance. Hence, we also propose a Bayesian Reinforcement Learning (BRL) pen-tester, called LD-PenTesting, that learns the decay factor as it attempts to break into the network. LD-PenTesting frames the learning problem as BRL and solves it as yet another POMDP. We tested both D-PenTesting and LD-PenTesting on two pen-testing scenarios that were introduced in previous work on POMDP-based pen-testing (Sarraute, Buffet, and Hoffmann 2012) and stochastic-game-based pen-testing (Lye and Wing 2005). In both scenarios, D-PenTesting and LD-PenTesting outperform existing POMDP-based pen-testing methods and are more robust than an autonomous pen-testing agent that incorporates a POMDP-based defender.

Related Work and Background

POMDP Formally, a POMDP is a tuple $\langle S, A, T, O, Z, R, \gamma \rangle$ (Kaelbling, Littman, and Cassandra 1998). At each time-step, a POMDP agent is in a state $s \in S$, executes an action $a \in A$, perceives an observation $o \in O$, and moves to the next state $s' \in S$. The next state is distributed according to $T(s, a, s')$, which is a conditional probability function $P(s'|s, a)$ that represents uncertainty in the effect of actions. The observation o perceived depends on the observation function Z, which is a conditional probability function $P(o|s, a)$ that represents uncertainty in sensing. The notation R is a state-action dependent reward function, from which the objective function is derived. The notations $\gamma \in (0, 1)$ is the discount factor to ensure that an infinite horizon POMDP problem remains a well-defined optimization problem.

The solution to a POMDP problem is an optimal policy π^*, that maps beliefs to actions in order to maximize the expected total reward, i.e. $V^*(b) = \max_{a \in A} \left[R(b, a) + \gamma \sum_{o \in O} Z(b, a, o) V^*(\tau(b, a, o)) \right]$, where $R(b, a) = \sum_{s \in S} R(s, a) b(s)$ and $Z(b, a, o) = \sum_{s' \in S} Z(s', a, o) \sum_{s \in S} T(s, a, s')$. The function τ computes the updated belief after the agent executes a from b and perceives o.

Bayesian Reinforcement Learning (BRL) Reinforcement Learning (RL) is a machine learning framework that finds the best action to perform in order to balance model learning and the achievement of the agent's goal. It learns the model just enough to accomplish the goal well, rather than learning the best model possible. BRL is an approach to RL that represents model uncertainty stochastically and leverages Bayesian inference to learn the model from observations. Many approaches have been proposed to solve BRL (Ghavamzadeh et al. 2015). A principled approach is to cast it as a POMDP, where the state space consists of a joint product between the state space of the system and the model parameters being learned. The POMDP beliefs then represent uncertainty on both the system's state and on the model, while the policy represents the best action to perform, so as to simultaneously optimise model learning and goal achievement.

Pen-testing A good review of developments with respect to pen-testing and AI planning can be found in (Hoffmann 2015) and for game theory and cyber security in (Liang and Xiao 2012; Merrick et al. 2016).

Early research and success in autonomous pen-testing came with the development of attack graphs, which involve modelling individual attack actions in terms of preconditions and post conditions (Swiler and Phillips 1998; Lippmann and Ingols 2005; Ammann, Wijesekera, and Kaushik 2002). Combining attack graphs with AI planning made it possible to generate attack paths through a target system in order to identify vulnerabilities and has been applied in commercial systems (Boddy et al. 2005; Lucangeli, Sarraute, and Richarte 2010). However, the classical planning approach requires complete knowledge of the network topology and host configurations. Additionally, it assumes that host configurations are static and actions are monotonic; meaning once an attack asset is gained, it cannot be lost. These restrictions make classical planning limited when trying to model the uncertainty inherit in realistic pen-testing.

Framing the pen-testing problem as a POMDP has enabled the modeling of uncertainty, including partial observability, in autonomous pen-testing (Sarraute, Buffet, and Hoffmann 2011). This approach removed the requirement that each hosts configuration be known at the start of an attack. Additionally, by incorporating this POMDP based pen-tester into a hierarchical model, it was shown that it could scale to large networks and handle actions that may cause a host to crash (Sarraute, Buffet, and Hoffmann 2012). Simpler models for planning under uncertainty have also been proposed for autonomous pen-testing that have better scaling properties (Durkota and Lisỳ 2014; Shmaryahu et al. 2018). Although able to more realistically model pen-testing compared to classical planning, these approaches still assume monotonic actions, static host configurations and the absence of a defender.

In order to model both the attacker and defender, a number of studies have used a game theoretic approach to network security (Lye and Wing 2005; Alpcan and Basar 2006; Nguyen, Alpcan, and Basar 2009; Liang and Xiao 2012). However, these studies have either framed the problem as a fully observable stochastic game or have not accounted for both partial observability of the network and stochastic actions.

Problem Formulation

Define the POMDP model of a D-PenTesting agent with decay factor $d \in [0, 1]$ as $\mathcal{P}_d = \langle S, A, T_d, O, Z, R, \gamma \rangle$.

The state space is $S = S_1 \times S_2 \times \cdots \times S_n$, where each state variable represents a property of the network and n is a finite natural number that represents the number of network properties the agent might alter. This network property can be relatively primitive such as, whether the ssh port of a machine is open, or relatively high level, such as whether the network has been compromised.

The action space is $A = A_1 \cup A_2 \cup \cdots \cup A_n \cup A_{n+1}$, where A_i for $i \in [1, n]$ represents the set of actions whose pre- and post-conditions only relate to the network property represented by the state variable S_i, while A_{n+1} is the set of actions whose pre- and/or post-conditions relate to a combination of multiple state variables.

Although the action space in this model consists of only the pen-tester's actions, the transition function T_d considers both the pen-tester's and defender's actions. A defender's action is implicitly accounted by the decay factor d, which represents the probability that the value of a state variable is changed by the defender. Before describing the exact transition function, we will first describe the model and assumptions of the defender in D-PenTesting.

A D-PenTesting agent only considers the defender's actions that might alter network properties that the pen-tester agent might alter, i.e., properties represented by at least one of the state variables in S. More precisely, D-PenTesting assumes that the set of possible actions that the defender might perform is $A^{def} = A_1^{def} \times A_2^{def} \times \cdots \times A_n^{def}$, where each variable A_i^{def} for $i \in [1, n]$ has two values: An action to assess whether the network property represented by S_i has been altered by an attacker and a patching action that nullifies the attacker action(s) on S_i. The first action does not change the state of the D-PenTesting agent, while the second obviously, might.

D-PenTesting then assumes a typical defender's strategy, which starts by analysing the network leading to performing a patch that nullifies the attacker's actions. However for simplicity, D-PenTesting assumes that the effect of the defender's actions on a state variable is independent from its effect on different state variables. The defender's move from analysing to patching the network properties related to S_i ($i \in [1, n]$) can naturally be cast as a Markovian Arrival Process.

In this paper, we adopted the simplest Markovian Arrival Process, which is the Bernoulli process where the success probability p_i is the probability that at the current time-step, the defender nullifies the attacker's action related to S_i. Assuming the defender has the same capability in analysing whether an attack related to different state variables has happened, D-PenTesting can use the same success probability for all $i \in [1, n]$, which is denoted as the decay factor d.

Now, we can describe the transition function T_d in more detail. When the action performed is $a \in \bigcup_{i=1}^{n} A_i$, then we can assume conditional independence: $T_d(\prod_{i=1}^{n} S'_i \mid \prod_{i=1}^{n} S_i \,, a) = \prod_{i=1}^{n} T_d(S'_i \mid S_i \,, a)$. Furthermore, if the action $a \in A_i$ is performed, then for any $j \in [1, n], j \neq i$, the transition is affected only by the defender's action, i.e.:

$$T_d(S'_j = s'_j \mid S_j = s_j, a) = \begin{cases} d \cdot \frac{1}{|S'_j|-1} & s'_j \neq s_j \\ 1-d & \text{Otherwise} \end{cases} \quad (1)$$

The first line in eq.(1) represents the case when the value of S_j is changed by the defender. The probability that the defender made the change is d. In the simplest case, the exact value the defender changes S_j into is assumed to be uniform. However, it is also simple enough to incorporate different distributions over the values of S_j if additional information is known. The transition $T_d(S'_i \mid S_i, a)$, however, will be affected only by the reliability of action a carried out by the pen-tester, i.e., the defender's action is not counted. The reason is that, for the above action, D-PenTesting assumes that the observation it perceives immediately after the action is performed would be informative enough so that it can ignore the assumed defender's behavior. When $a \in A_{n+1}$, the decay factor is added to the conditional probability function that reflects the reliability of a.

To perceive an observation about a property of the network, D-PenTesting must probe the network with the appropriate tools. When an action $a \in \bigcup_{i=1}^{n} A_i$ is performed, then the agent perceives information about S_i only. But, when the action performed is $a \in A_{n+1}$, the agent may perceive information about multiple state variables.

High reward is given whenever the network is compromised and small cost is given for each step. Reward can also be used to reduce the chances of being caught.

Note that the decay factor d in T_d does not directly affect the observation and reward functions. Rather, it affects the state of the network and the pen-tester's understanding about the state of the network, which is reflected in the belief.

Now, the extremely simplified defender model may cast doubts on the ability of D-PenTesting to handle deceptions such as honeypots. Interestingly, the simple defender model does not prevent a D-PenTesting agent from avoiding deceptions, as long as the agent can perceive observations on whether a component is a possible deception or not. If such observations exist, even if it is imperfect, avoiding deception, such as honeypots, is a matter of setting the appropriate penalty to discourage D-PenTesting from entering.

Learning The Decay Factor

The decay factor d essentially represents how fast a defender is able to analyse and act to nullify the work of an attacker. This speed will likely differ from one defender to another, and therefore needs to be learned while the pen-tester assesses the network.

The above learning problem can naturally be framed as a BRL problem. BRL balances the effort for model learning (in this case, learning the decay factor) and accomplishing the task (in this case, breaking into the network) by learning a good enough model for making good decisions, rather than learning the best model possible. We call this BRL-based pen-testing agent, LD-PenTesting. To compute a good strategy for an LD-PenTesting agent, we cast the BRL framing of the problem as yet another POMDP and solve the POMDP approximately using existing solvers.

Suppose $\mathcal{P}_d = \langle S, A, T_d, O, Z, R, \gamma \rangle$ is the POMDP model of the D-PenTesting agent. Then, the corresponding LD-PenTesting agent is defined as a POMDP $\mathcal{P} = \langle S, A, T, O, Z, R, \gamma \rangle$, where:

- $\mathcal{P}.S = \mathcal{P}_d.S \times D$, where D is the model parameter, which in this case is the set of possible values for the decay factor. To reduce computational complexity, we discretise D up to a certain resolution, denoted as δ. This state space definition means that the belief of the LD-PenTesting agent represents the uncertainty of the pen-tester in both the state of the network and the defender's behavior, where the defender's behaviour refers to the defender's speed in moving from analysing to patching the network.

- $\mathcal{P}.A = \mathcal{P}_d.A$. The transition function differs slightly. Assuming the model parameter does not change over time, the transition function of $\mathcal{P}$ can be computed as:

$$\begin{aligned}\mathcal{P}.T(\langle s,d\rangle, a, \langle s',d'\rangle) &= P(s',d' \mid s,d,a)\\ &= P(s' \mid d's,d,a)\cdot P(d' \mid s,d,a)\\ &= P(s' \mid d',s,d,a)\cdot \Delta_{dd'}\\ &= \mathcal{P}_d.T_d(s,a,s')\cdot \Delta_{dd'}\end{aligned}$$

where $\Delta_{dd'}$ is the Kronecker Delta function, $\Delta_{dd'} = 1$ whenever $d = d'$ and 0 otherwise. Note that although we assume the parameter does not change, the pen-tester's understanding about the model parameter might change over time, which is reflected in the belief of the pen-tester.

- $\mathcal{P}.O = \mathcal{P}_d.O$, $\mathcal{P}.Z(\langle s,d\rangle,a,o) = \mathcal{P}_d.Z(s,a,o)$, $\mathcal{P}.R(\langle s,d\rangle,a) = \mathcal{P}_d.R(s,a)$. As described in the previous section, the observation space and the observation and reward functions are not affected directly by the decay factor. LD-PenTesting learns the model parameter indirectly, in the sense that the transition function under different decay factors results in different states, leading to different observations being perceived. Given the same action and a different decay factor, the probability of perceiving the particular observation differs and this differing probability will result in a different Bayesian estimate about the state of the network and the decay factor.

The simplified model of the defender enables us to construct a compact POMDP representation of the BRL problem. In this paper, for comparison purposes, we solve this BRL using an off-line POMDP solver. However, both D-PenTesting and LD-PenTesting are general enough to be used with any POMDP solver, including on-line solvers such as those found in (Silver and Veness 2010; Somani et al. 2013; Kurniawati and Yadav 2013), in a straightforward manner, which will improve scalability of autonomous pen-testing.

Experiment Scenarios

We test our approach using two different scenarios in simulation. The first is based on previous work that modeled pen-testing as a POMDP and included uncertainty but did not incorporate the actions of the defender (Sarraute, Buffet, and Hoffmann 2012). The second is based on work that modeled a cyber security operation as a stochastic game, incorporating actions of both the pen-tester and the defender, but without incorporating uncertainty due to partial observability (Lye and Wing 2005). We extend both scenarios so that each incorporates uncertainty and the defenders actions. Table 1 shows the POMDP properties for each scenario.

Table 1: POMDP problem size for each scenario and agent. The Min steps column shows the minimum possible number of actions for agent to reach the goal.

Scenario, pen-testing agent	$\lvert S\rvert$	$\lvert A\rvert$	$\lvert O\rvert$	Min steps
1, D	3072	20	6	1
1, LD ($\delta = 0.1$)	30720	20	6	1
2, D	128	8	2	5
2, LD ($\delta = 0.1$)	1280	8	2	5

Table 2: Pen-tester exploit actions for scenario 1. The target columns show the OS and port number that each exploit targets (i.e. the exploit preconditions).

Exploit name	Target OS	Target Port
vsftpd-234-exploit	linux	21
IIS-ftp-exploit	windows	21
openBDSD-ssh-exploit	openBSD	22
IIS-http-exploit	windows	80
apache-openBSD-exploit	openBSD	80
rsh-exploit	linux	512
unreal-ircd-3281-exploit	linux	3632
manageEngine9-exploit	windows	8020
tomcat-exploit	windows	8282
elastic-search-exploit	windows	9200

Experiment Scenario 1

The first scenario involves a single host where the goal of the pen-tester is to exploit the host (Sarraute, Buffet, and Hoffmann 2012).

State The state for this scenario is $S = E \times T \times C_1 \times \cdots \times C_n$, where E represents whether the host has been successfully exploited, T represents if the pen-tester has given up the attack, and C_i for $i \in [1, n]$ represents a component of the hosts configuration. Both E and T variables are fully observable and the scenario ends when either becomes *true*

The host configuration components include the OS of the host and each of the ports targeted by the pen-testers available exploits (see table 2 for all ports included). The OS can be either *linux*, *windows* or *openBSD* and each port can be either *open* or *closed*.

Pen-tester

- **Actions:** Along with a *terminate* action which voluntarily ends the attack, we consider two types of actions: scans and exploits. Scan actions allow the pen-tester to acquire information about the configuration of the host. The agent has a *scan-OS* action, for scanning the hosts OS, along with a *port-scan* action for each port included in the state, for detecting if a port is open or closed. Table 2 shows each of the exploit actions in our scenario.
- **Transition:** Exploit actions are deterministic and, given their preconditions are met, will always successfully ex-

ploit the host. For example, the *vsftpd-234-exploit* requires *os=linux* and *port 21=open*, if these state conditions are met this exploit will succeed. The uncertainty for the pen-tester comes from having no knowledge of the initial configuration of the host and uncertainty due to the actions of the defender. The latter, we represent by the decay factor. In D-PenTesting the decay factor is fixed, while in LD-PenTesting it is learned.

- **Observations and Observation function:** The exploit and terminate actions affect the fully observable part of the state, namely the *exploited* and *terminated* state variables and thus receive their observations directly from fully observable changes in the state. The OS and port scan actions return an accurate observation of the state variable that they target. OS scans will return an observation of the OS $\in \{linux, windows, openBSD\}$ and port scans will return an observation of the state of a port $\in \{open, closed\}$. In this way they can be considered deterministic.
- **Rewards:** The reward function is conditional on the action performed and whether an end state has been reached. In line with experiments run in the original study we use a cost of 10 for both port scans and exploits and a cost of 50 for an OS scan. The attacker receives zero reward for using the *terminate* action and moving into the *terminated* state, while they receive a large reward of +9000 for successfully exploiting the host.
- **Initial belief:** For this scenario the pen-tester starts with no knowledge of the host configuration. The belief is uniformly distributed across the possible values for each state variable. For example, initial belief for OS will be $(0.33, 0.33, 0.33)$ for *linux*, *windows* and *openBSD*, respectively.

Defender In order to provide an opponent to evaluate our approach against, we extend the original paper and incorporate a defender into our scenario. The strategy and actions of the defender are not incorporated into the planning of the pen-tester at all and are used only during evaluation.

- **Actions:** The defender has an action for controlling each aspect of the host configuration. Specifically, for each port the defender has an *open-port* and *close-port* action. For the OS the defender has an action to change the host to a specific OS: *change-os-linux*, *change-os-windows* and *change-os-openBSD*. We also include a *do-nothing*, that has no effect on the state.
- **Strategies:** We test our approach against two different defender strategies: no defender and random defender. The random defender simply chooses an action uniformly at random from all available actions at each time step. This strategy is used to test how well each pen-tester can handle random changes to the state. Ideally, we would also test our approach against a more sophisticated defender strategy, unfortunately this would require significant modifications to the underlying problem formulation and make comparison with the original approaches infeasible.

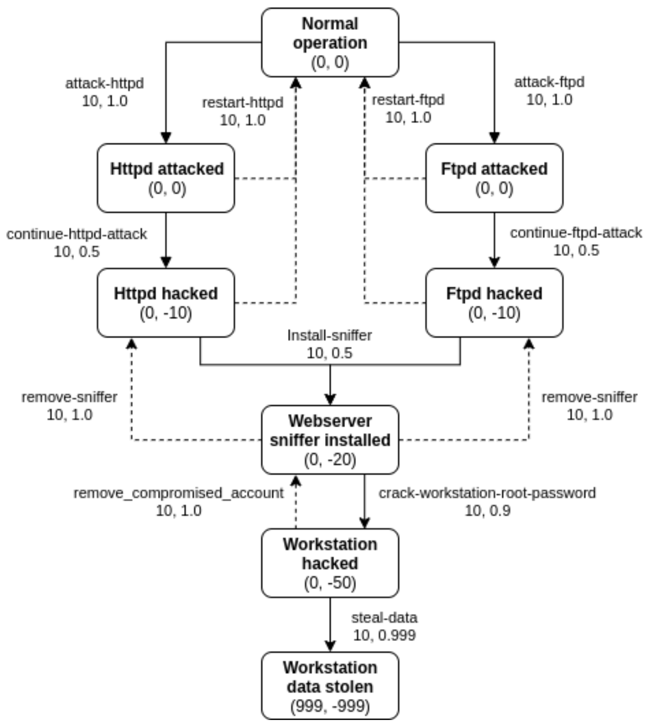

Figure 1: Graph representation for scenario 2. Each node, excluding the "Normal operation" node, is a binary state variable in the scenario and represents a different stage of the attack. The "Normal operation" node represents the state where there is no attack being performed (i.e. where all state variables have a value of *false*). The values in brackets within each node are the reward for the pen-tester and defender respectively. The solid lines are edges corresponding to actions by the pen-tester, while dashed lines correspond to actions by the defender. Each action has the cost and success probability under the actions name.

Experiment Scenario 2

The second scenario is based on a stochastic-game involving a pen-tester and defender operating on a small network of two machines (Lye and Wing 2005). We specifically focus on the *stealing confidential data* scenario (scenario 3, section 4.3) from the original paper, where the goal of the pen-tester is to steal data from the workstation machine. Figure 1 shows a high-level graph for the pen-tester and defender for this scenario. The original scenario was defined for a stochastic-game problem, where the state was fully observable with changes in the available action space used to handle cases where the defender may not have knowledge of the attacker. We modify the scenario to make the state partially observable.

State The state represents the current state of an attack on the network. Each state variable is binary (true or false) and corresponds to a different stage in the attack. The state variables for the scenario can be seen in figure 1, where each node in the graph, except the "Normal Operation" node, corresponds to a state variable. The "Normal Operation" node simply represents the case where all state variables are *false*. The scenario ends when the pen-tester steals the data from the workstation, i.e. *Workstation data stolen = true*.

Pen-tester

- **Actions and Transition Function:** Figure 1 shows each of the possible pen-tester actions, their cost and the effect they have on the state (solid lines). The pen-tester also has a *do-nothing* action which is not shown on the graph and which has a cost of 0 and has no effect on the state. Each action has associated preconditions, cost, and probability of success. The preconditions define the necessary state required for the action to have a chance to succeed. For example, the *install-sniffer* action requires that either *http-hacked=true* or *ftpd-hacked=true*, as shown by the edge for the *install-sniffer* action coming from these two states. The cost of an action represents the abstract cost of performing the action (i.e. in time, chance of detection, etc). The probability of success defines the chance that the action will succeed, given its preconditions are satisfied, and models the stochastic nature of the network environment. The cost and success probability values were chosen to match the original study.
- **Observations and Observation Function:** For each action performed, the pen-tester receives an observation; *success* if the action succeeded and *failure* if it failed.
- **Rewards:** In addition to the cost associated with performing each action the pen-tester receives a reward of +999 for successfully stealing the workstation data and 0 for any other state.

Defender As was the case for scenario 1, the strategy and actions of the defender are used only for evaluation and are not incorporated into the planning of the pen-tester, with the exception of Oracle-PenTesting which we discuss in the next section.

- **Actions and Observations:** All defender actions, with the exception of *monitor*, are shown in figure 1, along with their costs and success probabilities. The *monitor* action has no effect on the state but allows the defender to make a noisy observation of the current state. This is similar to how an intrusion detection system (IDS) alerts system administrators of a potential attack. We chose a success probability of 0.9 for the *monitor* action to account for false positives and negatives common to an IDS. The *monitor* action has a cost of 1.
- **Rewards:** The goal of the defender is to prevent the pen-tester from stealing the data from the workstation. With that in mind, the defender receives different rewards for each state depending on how close the state is to the pen-tester stealing the data. Figure 1 shows the reward corresponding to each stage of the attack.
- **Strategies:** As with scenario 1, we test each pen-tester agent against no defender and random defender strategies. We also test against a defender policy generated by solving the defender POMDP (which we hereby refer to as the defender POMDP strategy).

Results and Discussion

We ran experiments on the two scenarios described above in order to: (*i*) find the parameters for D-PenTesting and LD-PenTesting agents and analyse the sensitivity of performance with respect to parameters, and (*ii*) compare performance of D-PenTesting and LD-PenTesting agents against other pen-testing agents.

Experimental Setup

Experiments were run on a machine with an Intel Xeon Silver 2.1 GHz CPU and 128 GB of RAM. We used the Approximate POMDP Planning toolkit (APPL) (Du et al. 2014), which provides an efficient C++ implementation of the SARSOP algorithm (Kurniawati, Hsu, and Lee 2008).

For planning, using a discount factor of $\gamma = 0.95$, we ran the offline SARSOP solver until the time horizon of 1 hour or the target precision of $\epsilon = 0.001$ was reached.

We evaluated the performance of each agent by running simulations using APPL. We ran a total of 1000 simulations for each experiment with each simulation running for a maximum of 100 time steps. At each time step during the simulations, each player performed an action, with the pen-tester always going first, followed by the defender. For each simulation, the pen-tester won if they managed to reach the goal within the time step limit, otherwise the defender won.

We compare our D-PenTesting and LD-PenTesting agents against the POMDP pen-tester approach with no decay factor (hereby referred to as POMDP-PenTesting) (Sarraute, Buffet, and Hoffmann 2011; 2012). For scenario 2 we also compare against Oracle-PenTesting.

Oracle-PenTesting In order to determine how well D-PenTesting and LD-PenTesting perform, we also compare them against a stronger pen-tester agent: Oracle-PenTesting. We augment the POMDP-PenTesting agent by adding the defender's action as a fully observable variable to the agents state. During planning, the Oracle-PenTesting agent has access to the defender POMDP agent's policy and incorporates this into its transition function. Note, however, that Oracle-PenTesting remains a POMDP agent, and not a Partially Observable Stochastic Game agent. During each simulation, at each step we set the fully observable defender action state variable of the Oracle-PenTesting agent to be the next action of the defender. This is done for all defender strategies (no defender, random and POMDP). In this way Oracle-PenTesting has full information about the defenders next action along with the effects of the action on the next state.

Unfortunately, we could only include the Oracle-PenTesting agent as a comparison for scenario 2. Since, as discussed above, for scenario 1 we were unable to create a POMDP defender and therefore there is no defender policy to incorporate into the Oracle-PenTesting agent planning step. Additionally, incorporating a random defender strategy would not be a meaningful comparison as the Oracle-PenTesting agent is designed to counter a specific defender policy.

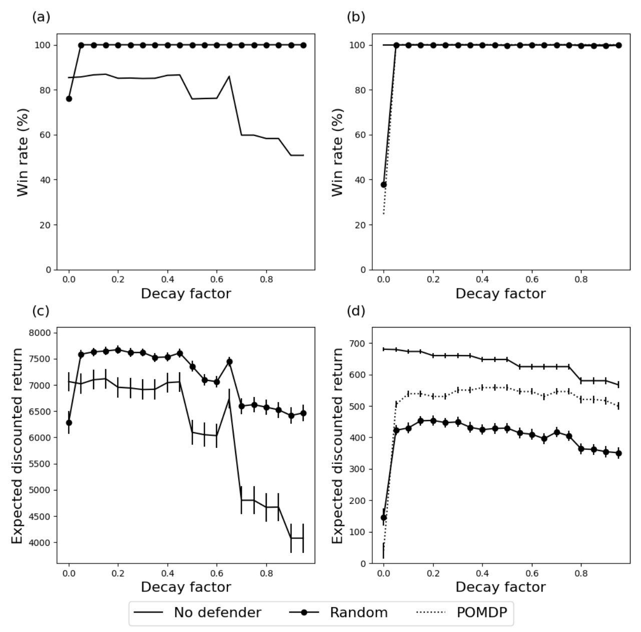

Figure 2: Performance of D-PenTesting with decay factor $\in [0, 0.95]$ at intervals of 0.05, for each defender strategy. (a) and (c) show results for scenario 1 while (b) and (d) shows results for scenario 2. (a) and (b) shows the pen-tester win rate, while (c) and (d) shows expected discounted return ($\pm$ 95% CI), of D-PenTesting against each defender strategy. When the decay factor = 0.0, this is the same as POMDP-PenTesting. Note there is no POMDP defender for scenario 1; (a) and (c).

Parameter Selection

D-PenTesting We evaluated the performance of D-PenTesting using different decay factors, against each defender strategy, on both scenarios. We ran experiments for multiple decay factors $\in [0, 0.95]$ at an interval of 0.05. For these experiments, POMDP-PenTesting is equivalent to when the decay factor is 0.0. Figure 2 shows the performance of D-PenTesting for both scenarios.

When no defender is present, for scenario 1 performance of D-PenTesting is consistent up to a decay factor of 0.45, with performance declining significantly for decay factors above 0.45. For scenario 2 with no defender, the win rate is 100% for all tested decay factors. However, the performance in terms of expected discount return declines as the decay rate increases. The decline is quite slow when the decay factor is below 0.5, with decline occurring faster for higher decay factors.

Against the random defender, for both scenarios, we see a similar trend to when no defender is present. The expected discounted return is fairly constant up to a decay factor of around 0.5, before it begins to decline more rapidly.

This trend in performance can be attributed to overly conservative behaviour of D-PenTesting for high decay factors. We have observed that for larger decay factors ($>= 0.5$), where the agent expects faster changes to the state, the D-PenTesting agent becomes increasingly conservative resulting in many repeated actions. This results in many wasted actions and a decline in performance. For scenario 1, for decay factors $>= 0.7$, this even resulted in agents repeatedly performing the same subset of actions leaving them unable to exploit the host in the cases when the host was not vulnerable to the subset of actions performed.

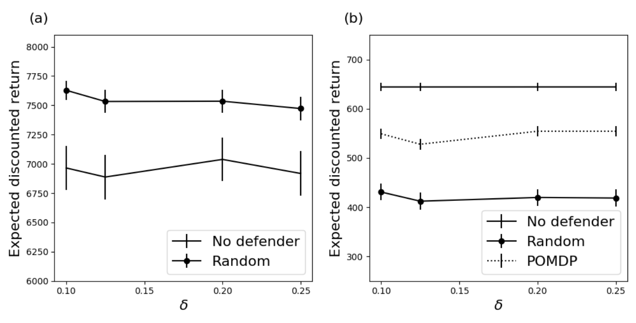

Figure 3: Performance of LD-PenTesting against each defender strategies for different decay discretization resolution, δ, values (0.1, 0.125, 0.2 and 0.25). Plots (a) and (b) shows results for scenario 1 and scenario 2, respectively.

LD-PenTesting To evaluate LD-PenTesting, we ran experiments using four different decay discretization resolution values of δ: 0.1, 0.125, 0.2 and 0.25. These were chosen based on the results from D-PenTesting, where we observed that performance for both scenarios tended to be consistent over decay factor intervals of 0.1 or larger.

Figure 3 shows the performance of LD-PenTesting for both scenarios. The expected discounted return was relatively consistent across the different δ values, against all defender strategies, across both scenarios. This shows that at least for the scenarios tested, the δ has little effect on the possible performance.

Parameter Sensitivity Based on the above results, we can see that both D-PenTesting and LD-PenTesting are quite insensitive to parameter selection. For D-PenTesting, performance was reasonably consistent for decay factors $\in [0.05, 0.45]$. For LD-PenTesting, performance was consistent for each δ value tested.

Pen-Tester Comparison

Figure 4 shows a comparison of the performance of each pen-tester agent for both scenarios. For D-PenTesting we chose the best decay factor for each defender strategy. For LD-PenTesting we show results for the agent with $\delta = 0.1$.

Against no defender, performance of LD-PenTesting is slightly worse than that of the best D-PenTesting agent, which can be explained by the cost of learning in LD-PenTesting. When there is no defender the correct decay factor is 0.0. For the LD-PenTesting agent it cannot observe the decay factor directly and instead infers it through the transition dynamics, hence it will always have some uncertainty over the correct value. This uncertainty incurs a learning cost in the form of more repeated actions. This cost is emphasised when the planning horizon is longer; as it is in scenario 2 where the difference in performance is most notable. However, even with this extra cost, against no defender, the expected discounted return of LD-PenTesting is only slightly lower than that of D-PenTesting and POMDP-PenTesting and win rate is comparable across all three agents for both scenarios.

For the random and POMDP defenders, performance was comparable between D-PenTesting and LD-PenTesting. The consistency of performance between these two approaches can be explained by their insensitivity to parameter selection. This meant that learning a good decay factor would be relatively easy for the LD-PenTesting agent since the range of good decay factors is quite large.

When compared with POMDP-PenTesting, when there is no defender, we see comparable performance with D-PenTesting and LD-PenTesting, except for scenario 2 where the performance of LD-PenTesting is slightly worse (discussed above). Against the random and POMDP defenders D-PenTesting and LD-PenTesting significantly outperform POMDP-PenTesting. This illustrates the superior advantage D-PenTesting and LD-PenTesting have over POMDP-PenTesting in the presence of a defender and dynamic changes to the state.

When compared with Oracle-PenTesting we see that, in terms of expected discounted return, D-PenTesting and LD-PenTesting outperform Oracle-PenTesting against the random defender. When we incorporate the defender policy into the Oracle-PenTesting agent we essentially fit the pen-tester agent to a specific defender policy, hence why the Oracle-PenTesting agent performs significantly worse against the random defender than against the POMDP defender (i.e. the policy it had planned with is wrong). The better performance of D-PenTesting and LD-PenTesting versus Oracle-PenTesting, against a random defender, suggests that D-PenTesting and LD-PenTesting agents can be more robust to different defender behaviour than an agent designed to counter a specific defender strategy.

As expected, Oracle-PenTesting outperforms D-PenTesting and LD-PenTesting against the POMDP defender. However, both D-PenTesting and LD-PenTesting are still able to achieve a 100% win rate and a good expected discounted return that is significantly higher than that of POMDP-PenTesting. This shows that D-PenTesting and LD-PenTesting are able to perform well against a reasonable defender strategy. This further supports that incorporating decay into the pen-tester model can help improve agent robustness against different defender strategies.

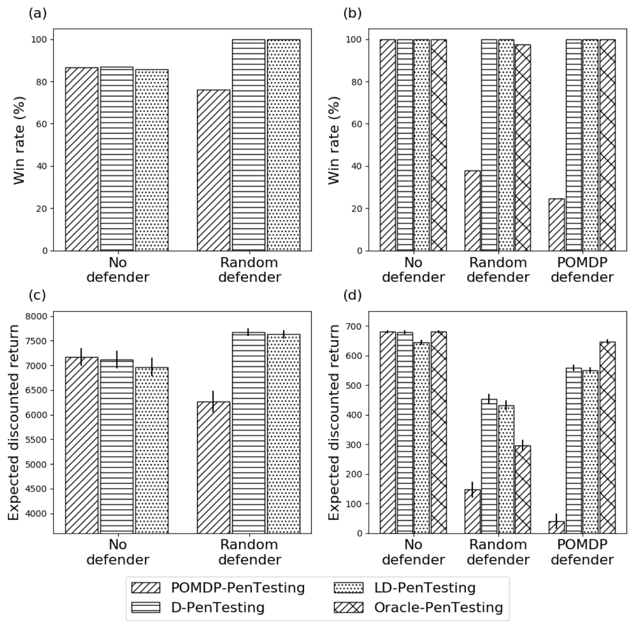

Figure 4: Comparison between the different pen-testing agents for scenario 1 ((a) and (c)) and scenario 2 ((b) and (d)). For the D-PenTesting agent the best decay factor for each defender strategy is shown (scenario 1: no defender = 0.15, random = 0.2, and scenario 2: no defender = 0.05, random = 0.2, POMDP = 0.45), for the LD-PenTesting agent we show results for policy with decay discretization resolution of δ = 0.1 for both scenarios. (a) and (b) shows the win rate, and (c) and (d) shows expected discounted return ($\pm$ 95% CI), for each pen-tester agent against each defender strategy.

Summary

This paper presented POMDP-based pen-testing models, D-PenTesting and LD-PenTesting. They have been designed to handle the three main sources of uncertainty in autonomous pen-testing: partial observability of the network and its vulnerabilities, lack of reliability of pen-testing tools, and possible changes triggered by the defender. Based on an observation that a pen-tester's knowledge about the defender's actions and strategies are gained in an indirect manner, via the network, we abstract the defender's actions into two types: Network analysis, which does not change the network, and counter-attacks, which alter the network properties. This abstraction enables us to model the defender's behavior as a single variable, namely the decay factor. D-PenTesting incorporates this variable in its transition function, assuming the decay factor is known a priori. LD-PenTesting learns the decay factor as it attempts to break into the network, by framing the problem as BRL, solved as yet another POMDP problem. Experiments on two benchmark scenarios indicate D-PenTesting and LD-PenTesting outperform the POMDP-based pen-testing approach that does not account for the defender's behavior and is more robust than autonomous pen-testing that assumes a POMDP-based defender.

Future work abounds. This work indicates that even a simple defender's model can substantially improve the capability of autonomous pen-testing. We are interested in better understanding the limits of this approach by testing it against more powerful defenders, such as (Ahmadi et al. 2018). Another avenue is to apply the proposed concept to on-line POMDP solvers for scalability. Last but not least is to remove assumptions, such as the memory-less assumption of the Bernoulli Process without substantially increasing the complexity of the defender's model.

References

Ahmadi, M.; Cubuktepe, M.; Jansen, N.; Junges, S.; Katoen, J.-P.; and Topcu, U. 2018. The partially observable games we play for cyber deception.

Alpcan, T., and Basar, T. 2006. An intrusion detection game with limited observations. In *12th Int. Symp. on Dynamic Games and Applications, Sophia Antipolis, France*, volume 26. Citeseer.

Ammann, P.; Wijesekera, D.; and Kaushik, S. 2002. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 217–224. ACM.

Asmussen, S. 2010. Applied probability and queues (stochastic modeling and applied probability series vol. 51).

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of action generation for cyber security using classical planning. In *ICAPS*, 12–21.

Du, Y.; S.W.Png; H.Kurniawati; S.C.W.Ong; W.S.Lee; and D.Hsu. 2014. Approximate POMDP planning toolkit (APPL). http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/.

Durkota, K., and Lisỳ, V. 2014. Computing optimal policies for attack graphs with action failures and costs. In *STAIRS*, 101–110.

Ghavamzadeh, M.; Mannor, S.; Pineau, J.; Tamar, A.; et al. 2015. Bayesian reinforcement learning: A survey. *Foundations and Trends® in Machine Learning* 8(5-6):359–483.

Hoffmann, J. 2015. Simulated penetration testing: From" dijkstra" to" turing test++". In *Twenty-Fifth International Conference on Automated Planning and Scheduling*.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101(1-2):99–134.

Kurniawati, H., and Yadav, V. 2013. An Online POMDP Solver for Uncertainty Planning in Dynamic Environment. In *Proc. Int. Symp. on Robotics Research*.

Kurniawati, H.; Hsu, D.; and Lee, W. S. 2008. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and systems*, volume 2008. Zurich, Switzerland.

Liang, X., and Xiao, Y. 2012. Game theory for network security. *IEEE Communications Surveys & Tutorials* 15(1):472–486.

Lippmann, R. P., and Ingols, K. W. 2005. An annotated review of past papers on attack graphs. Technical report, Massachusetts Ins. of Tech. Lexington Lincoln Lab.

Lucangeli, J.; Sarraute, C.; and Richarte, G. 2010. Attack planning in the real world. In *Workshop on Intelligent Security (SecArt 2010)*.

Lye, K.-w., and Wing, J. M. 2005. Game strategies in network security. *International Journal of Information Security* 4(1-2):71–86.

Merrick, K.; Hardhienata, M.; Shafi, K.; and Hu, J. 2016. A survey of game theoretic approaches to modelling decision-making in information warfare scenarios. *Future Internet* 8(3):34.

Neuts, M. F. 1979. A versatile markovian point process. *Journal of Applied Probability* 16(4):764–779.

Nguyen, K. C.; Alpcan, T.; and Basar, T. 2009. Security games with incomplete information. In *2009 IEEE International Conference on Communications*, 1–6. IEEE.

Sarraute, C.; Buffet, O.; and Hoffmann, J. 2011. Penetration testing== pomdp solving? In *Working Notes for the 2011 IJCAI Workshop on Intelligent Security (SecArt)*, 66.

Sarraute, C.; Buffet, O.; and Hoffmann, J. 2012. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Shmaryahu, D.; Shani, G.; Hoffmann, J.; and Steinmetz, M. 2018. Simulated penetration testing as contingent planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*.

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

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

Swiler, L. P., and Phillips, C. 1998. A graph-based system for network-vulnerability analysis. Technical report, Sandia National Labs., Albuquerque, NM (United States).