Partially Observable Contingent Planning for Penetration Testing

Dorin Shmaryahu and **Guy Shani**
Information Systems Engineering
Ben Gurion University, Israel

Joerg Hoffmann and **Marcel Steinmetz**
Department of Computer Science
Saarland University, Germany

Abstract

Penetration Testing (pentesting), where network administrators automatically attack their own network to identify and fix vulnerabilities, has recently received attention from the AI community. Smart algorithms that can identify robust and efficient attack plans may imitate human hackers better than simple protocols. Classical planning methods for pentesting model poorly the real world, where the attacker has only partial information concerning the network. On the other hand POMDP-based approaches provide a strong model, but fail to scale up to reasonable model sizes. In this paper we offer a middle ground, allowing for partial observability and non-deterministic action effects, by modeling pentesting as a partially observable contingent problem. We experiment with a real network of a large organization, showing our solver to scale to realistic problem sizes. We also experiment with sub-sampled networks, comparing the expected reward of a contingent plan graph to that of a POMDP policy.

1 Introduction

Penetration testing (pentesting) is a popular technique for identifying vulnerabilities in networks, by launching controlled attacks (Burns et al. 2007). A successful, or even a partially successful attack reveals weaknesses in the network, and allows the network administrators to remedy these weaknesses. Such attacks typically begin at one entrance point, and advance from one machine to another, through the network connections. For each attacked machine a series of known exploits is attempted, based on the machine configuration, until a successful exploit occurs. Then, this machine is controlled by the attacker, who can launch new attacks on connected machines. The attack continues until a machine inside the secure network is controlled, at which point the attacker can access data stored inside the secured network, or damage the network.

In automated planning the goal of an agent is to produce a plan to achieve specific goals, typically minimizing some performance metric such as overall cost. There are many variants of single agent automated planning problems, ranging from fully observable, deterministic domains, to partially observable, non-deterministic or stochastic domains. Automated planning was previously suggested as a tool for conducting pentesting, exploring the two extreme cases — a classical planning approach, where all actions are deterministic, and the entire network structure and machine configuration are known, and a POMDP approach, where machine configuration are unknown, but can be noisily sensed, and action outcomes are stochastic.

The classical planning approach scales well for large networks, and has therefore been used in practice for pentesting. However, the simplifying assumptions of complete knowledge and fully deterministic outcomes results in an overly optimistic attacker point-of-view. It may well be that a classical-planning attack has a significantly lower cost than a real attack, identifying vulnerabilities that are unlikely to be found and exploited by actual attackers. MDPs(Durkota *et al.* 2015; Hoffmann 2015) provide a slightly more realistic description of the problem, allowing for actions to fail. Still, like classical planning, MDPs do not measure the partial information that an attacker may have, and the sensing actions that may be needed.

The POMDP approach on the other hand (Sarraute *et al.* 2011), models the problem better, and can be argued to be a valid representation of the real world. One can model the prior probabilities of various configurations for each machine as a probability distribution over possible states, known as a belief. Probing actions, designed to reveal configuration properties of machines are modeled as sensing actions, and a probability distribution can be defined for the possible failure in probing a machine. The success or failure of attempting an exploit over a machine can be modeled as a stochastic effect of actions.

This approach, however, has two major weaknesses — first, POMDP solvers do not scale to the required network size and possible configurations. Second, a POMDP requires accurate probability distributions for initial belief, sensing accuracy, and action outcomes. In pentesting, as in many other applications, it is unclear how the agent can reliably obtain these distributions. In particular, identifying an accurate probability distribution over the possible OS for the machines in the network. Prior work (Sarraute *et al.*) has devised only a first over-simplifying model of "software updates".

In this paper we suggest an intermediate model between classical planning and POMDPs. We replace the POMDP definition with partially observable contingent planning, a qualitative model where probability distributions are re-

placed with sets of possible configurations or action effects (Albore *et al.* 2009; Muise *et al.* 2014; Komarnitsky and Shani 2014). Solvers for this type of models scale better than POMDP solvers, and can be used for more practical networks. As these models require no probabilities, we avoid the guesswork inherent in their specification.

Contingent planners attempt to find a plan tree (or graph), where nodes are labeled by actions, and edges are labeled by observations. This plan tree is a solution to the problem if all leaves represent goal states.

We experiment with a network of a large organization, with the real vulnerabilities that were found in a scan of that network, showing that our contingent planner computes a plan graph for this network. In addition, we compare plan graphs to POMDP policies for much smaller networks that were sub-sampled from the real network data that we collected.

2 Networks and Pentesting

We begin by providing a short background on pentesting.

We can model networks as directed graphs whose vertices are a set M of *machines*, and edges representing connections between pairs of $m \in M$. Like previous work in the area, we assume below that the attacker knows the structure of the network. But this assumption can be easily removed in our approach. We can add sensing actions that test the outgoing edges from a controlled host to identify its immediate neighbors. From an optimization prespective, though, not knowing anything about the network structure, makes it difficult to create smart attacks, and the attacker is forced to blindly tread into the network. It might well be that some partial information concerning the network structure is known to the attacker, while additional information must be sensed. We leave discussion of interesting forms of partial knowledge to future work.

Each machine in the network can have a different *configuration* representing its hardware, operating system, installed updates and service packs, installed software, and so forth. The network configuration is the set of all machine configurations in the network.

Machine configuration may be revealed using sensing techniques. For example, if a certain series of 4 TCP requests are sent at exact time intervals to a target machine, the responses of the target machine vary between different versions of Windows (Lyon 2009). In many cases several different such methods must be combined to identify the operating system. Sending such seemingly innocent requests to a machine to identify its configuration is known as fingerprinting. Not all the properties of a target machine can be identified. For example, one may determine that a certain machine runs Windows XP, but not which security update is installed.

Many configurations have *vulnerabilities* that can be *exploited* to gain control over the machine, but these vulnerabilities vary between configurations. Thus, to control a machine, one first probes it to identify some configuration properties, and based on these properties attempts several appropriate exploits. As the attacker cannot fully observe the configuration, these exploits may succeed, giving the attacker full control of the target machine, or fail as some undetectable configuration property made this exploit useless.

The objective of penetration testing (pentesting) is to gain control over certain machines that possess critical content in the network. We say that a machine m is *controlled* if it has already been hacked into, and the attacker can use it to fingerprint and attack other machines. A *reached* machine m is connected to a controlled machine. All other machines are *not reached*. We assume that the attacker starts controlling the internet, and all machines that are directly connected to the internet are reached.

We will use the following (small but real-life) situation as an illustrative example (Sarraute *et al.*):

Example 2.1. The attacker has already hacked into a machine m', and now wishes to attack a reached machine m. The attacker may try one of two exploits: *SA*, the "Symantec Rtvscan buffer overflow exploit"; and *CAU*, the "CA Unicenter message queuing exploit". SA targets a particular version of "Symantec Antivirus", that usually listens on port 2967. CAU targets a particular version of "CA Unicenter", that usually listens on port 6668. Both work only if a protection mechanism called *DEP* ("Data Execution Prevention") is disabled. The attacker cannot directly observe whether DEP is enabled or not.

If SA fails, then it is likely that CAU will fail as well because DEP is enabled. Hence, upon observing the result of the SA exploit, the attacker learns whether DEP is enabled. The attacker is then better off trying other exploits else. Achieving such behavior requires the attack plan to observe the outcomes of actions, and to react accordingly. Classical planning which assumes perfect world knowledge at planning time cannot model such behaviors.

2.1 POMDPs for Pentesting

Partially observable Markov decision proccesses (POMDPs) (Sondik 1978) were previously suggested as a strong modeling tool for pentesting (Sarraute *et al.*).

A POMDP is a tuple $\langle S, A, \Omega, tr, O, R, b_0 \rangle$, where S is a set of states, A is a set of actions, Ω is a set of possible observations, $tr(s, a, s')$ is the probability of transitioning from a state s to a state s' using action a, $O(a, s', o)$ is the probability of observing $o \in \Omega$ after executing action a, arriving at state s', $R(s, a, s')$ is the reward (or cost) for executing action a in state s arriving at state s', and b_0 is the initial belief — a probability distribution over the possible initial states.

Sarraute et al. model pentesting using a POMDP where the states are the possible configuration of the network. That is, a state defines for each machine in the network its operating system (OS), open ports, running software, and vulnerabilities. The initial belief is hence a probability distribution over the possible network configurations.

There are sensing actions, that do not change the state of the world, that is, $tr(s, a_{sense}, s) = 1$, but provide information about certain machine properties, such as its OS, through the observations distribution. The states change when using exploit actions, resulting in a new state where an attacked machine becomes controlled by the attacker. In addition, there is a special *terminate* action, that moves the POMDP to a terminal state.

Sarraute et al. set costs to all actions, a reward when taking control of any machine, and a larger reward when taking control of some important machines. They experiment with deterministic POMDPs, where actions have deterministic effects, and observations are also deterministic, but POMDPs can also be used to model sensing noise, and stochastic success of exploits.

Sarraute et al. show that the constructed POMDPs can be solved by a POMDP solver (Kurniawati *et al.* 2008) only for small problems.

3 Contingent Planning Model and Language

A contingent planning problem is a tuple $< P, A_{act}, A_{sense}, \phi_I, G >$, where P is a set of propositions, A_{act} is a set of actuation actions, and A_{sense} is a set of sensing actions. An actuation action is defined by a set of preconditions — propositions that must hold prior to executing the actions, and effects — propositions that hold after executing the action. A sensing action a_{sense} has preconditions, but no effects. Instead, a_{sense} reveals the value of a proposition. ϕ_I is a propositional formula describing the set of initially possible states. $G \subset P$ is a set of goal propositions.

In our pentesting application, P contains propositions describing machine configuration, such as $OS(m_i, winxp)$, denoting that machine m_i runs the OS Windows XP. Similarly, $SW(m_i, IIS)$ represents the existence of the software IIS on machine m_i. In addition, the proposition $controlling(m_i)$ denotes that the attacker currently controls m_i, and the proposition $hacl(m_i, m_j)$ denotes that machine m_i is directly connected to machine m_j.

The set A_{sense} in our pentesting model represents the set of possible queries that one machine can launch on another, directly connected machine, probing it for various properties, such as its OS, software that runs on it, and so forth. Each such sensing action requires as precondition only that the machines will be connected, and reveals the value of a specific property. In some cases there are certain "groups" of operating systems, such as Windows XP with varying service packs and updates installed. In this case we can allow one property for the group ($OS(m_i, winxp)$) and another property for the version, such as ($OSVersion(m_i, winxp_sp1)$) which may not be observable by the attacker.

The set A_{act} in our pentesting model contains all the possible exploits. We create an action $a_{e,m_{source},m_{target}}$ for each exploit e and a pair of directly connected machines m_{source}, m_{target}. If an exploit e is applicable only to machines running Windows XP, then $OS(m_{target}, winxp)$ would appear in the preconditions. Another precondition is $controlling(m_{source})$ denoting that the attacker must control m_{source} before launching attacks from it. The effect of the action can be $controlling(m_{target})$, but we further allow the effect to depend on some hidden property p that cannot be sensed. This is modeled by a conditional effect $\langle p, controlling(m_{target})\rangle$ denoting that if property p exists on m_{target} than following the action the attacker controls m_{target}.

Belief states in contingent planning are sets of possible states, and can often be compactly represented by logic formulas. The initial belief formula ϕ_I represents the knowledge of the attacker over the possible configurations of each machine. For example $oneof(OS(m_i, winxp), OS(m_i, winnt4), OS(m_i, win7))$ states that the possible operating systems for machine m_i are Windows XP, Windows NT4, and Windows 7.

Like Sarraute et al., we assume no non-determinism, i.e., if all properties of a configuration are known, then we can predict deterministically whether an exploit will succeed. We do allow for non-observable properties, such as the service pack installed for the specific operating system. We support actions for sensing whether an exploit has succeeded. Hence, observing the result of an exploit action reveals information concerning these hidden properties.

Example 3.1. We illustrate the above ideas using a very small example, written in a PDDL-like language for describing contingent problems (Albore *et al.* 2009).

We use propositions to describe the various properties of the machines and the network. For example, the predicate *(hacl* $?m_1$ $?m_2$*)* specifies whether machine m_1 is connected to machine m_2, and the predicate *(HostOS ?m ?o)* specifies whether machine m runs OS o. While in this simple example we observe the specific OS, we could separate OS type and edition (say, Windows NT4 is the type, while Server or Enterprise is the edition). We can then allow different sensing actions for type and edition, or allow only sensing of type while edition cannot be directly sensed.

We define actions for probing certain properties. For example, the *probe-os* action:

```
(:action ping-os
    :parameters (?src - host ?target - host ?o - os)
    :precondition (and (hacl ?src ?target)
                       (controlling ?src)
                       (not(controlling ?target))
    :observe (HostOS ?target ?o)
)
```

allows an attacker that controls host s connected to an uncontrolled host t, to probe it to identify whether it's OS is o. We allow for a similar probe action for installed software.

The *exploit* action attempts to attack a machine exploiting a specific vulnerability:

```
(:action exploit
    :parameters (?src - host ?target - host ?o - os ?sw - sw
                 ?v - vuln)
    :precondition (and (hacl ?src ?target)
                       (controlling ?src)
                       (not(controlling ?target))
                       (HostOS ?target ?o)
                       (HostSW ?target ?sw)
                       (Match ?o ?sw ?v))
    :effect (when (ExistVuln ?v ?target) (controlling ?target))
)
```

The preconditions specify that the machines must be connected, that the OS is o and the software sw is installed, and that the vulnerability v which we intend to exploit matches the specific OS and software.

The success of the exploit depends on whether the vulnerability exists on the target machine, which manifests in the conditional effect. The attacker cannot directly observe whether a specific vulnerability exists, but can use the *CheckControl* action to check whether the exploit has succeeded:

```
(:action CheckControl
    :parameters (?src - host ?target - host)
    :precondition (and (hacl ?src ?target ?p)
                       (controlling ?src))
    :observe (controlling ?target)
)
```

The initial state of the problem describes the knowledge of the attacker prior to launching an attack:

```
(:init
1:    (controlling internet)
2:    (hacl internet host0)
      (hacl internet host1)
      (hacl host1 host2)
      (hacl host0 host2)
      ...
3:    (oneof (HostOS host0 winNT4ser) (HostOS host0 winNT4ent))
      (oneof (HostOS host1 win7ent) (HostOS host1 winNT4ent))
      ...
4:    (oneof (HostSW host0 IIS4) (HostSW host1 IIS4))
      ...
5:    (Match winNT4ser IIS4 CVE-X-Y)
      ...
6:    (or (ExistVuln CVE-X-Y host0) (ExistVuln CVE-Z-W host0))
      ...
)
```

We state that initially the attacker controls the "internet" only (part 1). In this case the structure of the network is known, described by the *hacl* statements (part 2). Then, we describe which operating systems are possible for each of the hosts (part 3). Below, we specify that either $host0$ or $host1$ are running the software IIS (part 4). We describe which vulnerability is relevant to a certain OS-software pair (part 5), and then describe which vulnerabilities exit on the various hosts (part 6).

The above specification may allow for a configuration where no vulnerability exists on a host (machine) that matches the host OS and software. Hence, none of the exploits will work for that specific host.

4 Contingent Plan Trees for Pentesting

A solution to a contingent planning problem is a plan tree, where nodes are labeled by actions. A node labeled by an actuation action will have only a single child, and a node labeled by a sensing action has two children, and each outgoing edge to a child is labeled by a possible observation.

An action a is applicable in belief state b, if for all $s \in b$, $s \models pre(a)$. The belief state b' resulting from the execution of a in b is denoted $a(b)$. We denote the execution of a sequence of actions $a_1^n =< a_1, a_2, ..., a_n >$ starting from belief state b by $a_1^n(b)$. Such an execution is valid if for all i, a_i is applicable in $a_1^{i-1}(b)$.

Plan trees can often be represented more compactly as plan graphs (Komarnitsky and Shani 2014; Muise *et al.* 2014), where certain branches are unified. This can lead to a much more compact representation, and to scaling up to larger domains. Still, for ease of exposition, we discuss below plan trees rather than graphs.

In general contingent planning, a plan tree is a solution, if every branch in the tree from the root to a leaf, labeled by actions a_1^n, $a_1^n(b_I) \models G$. In pentesting, however, it may not be possible to reach the goal in all cases, because there may be network configurations from which the target machine simply cannot be reached. To cater for this, we need to permit plan trees that contain *dead-ends*. We define a dead-end to be a state from which there is no path to the goal, given *any* future sequence of observations. That is, any plan tree starting from a dead-end state would not reach the goal in any of its branches. For example, a dead-end state arises if no exploit is applicable for the goal machine. It is clearly advisable to stop the plan (the attack) at such states. On the other hand, if a state is not a dead-end, then there still is a chance to reach the target so the plan/attack should continue.

There is hence need to define contingent plans where some of the branches may end in dead-ends. A simple solution, customary in probabilistic models, is to introduce a give-up action which allows to achieve the goal from any state. Setting the cost of that action (its negative reward) controls the extent to which the attacker will be persistent, through the mechanism of expected cost/expected reward.

In a qualitative model like ours, it is not as clear what the cost of giving up (effectively, of flagging a state as "dead-end" and disregarding it) should be. It may be possible to set this cost high enough to force the plan to give up only on dead-ends as defined above. But then, the contingent planner would effectively need to search all contingent plans not giving up, before being able to give up even once.

We therefore employ here a different approach, allowing the planner to give-up on s iff it can prove that s is a dead-end. Such proofs can be lead by classical-planning dead-end detection methods, like relaxation/abstraction heuristics, adapted to our context by determinizing the sensing actions, allowing the dead-end detector to choose the outcome. In other words, we employ a sufficient criterion to detect dead-end states, and we make the give-up action applicable only on such states. As, beneath all dead-ends, eventually the pentest will run out of applicable actions, eventually every dead-end will be detected and the give-up enabled.

In general, this definition would not be enough because the planner could willfully choose to move into a dead-end, thereby "solving" the task by earning the right to give up. This cannot happen, however, in the pentesting application, as all dead-ends are *unavoidable*, in the following sense. Say N is a node in our plan tree T, and denote by $[N]$ those initial states from which the execution of T will reach N. If N is a dead-end, then every $I \in [N]$ is unsolvable, i.e., there does not exist any sequence of A_{act} actions leading from I to the goal. In other words, any dead-end the contingent plan may encounter is, in the pentesting application, inherent in the initial state.

5 Network Data Acquisition

To test our approach we created realistic models using data obtained from scanning the network of a large organization, containing several subnets. Using the machine configurations and existing exploits discovered using the scan, we can create real world models that allow us to provide an empirical evaluation of our approach. We now provide some explanations of the model and the network, unfortunately omitting many details due to confidentiality restrictions.

To collect the needed information for our models, we began by running a scan of the various subnets using the Nessus scanner[1]. Nessus starts its scan from a given computer, and identifies all reachable hosts from that computer, including desktops, gateways, switches, and more.

As Nessus does not actually launches attacks to control a host, a Nessus scan identifies only hosts that are directly logically reachable from the source machine where the scan is running, possibly through several switches and gateways. We hence executed several such scans, each from a different subnet within the organization, as well as one scan from outside the organization network.

The resulting scans contain the set of machines that are visible from each source machine. The machines inside a subnet are all visible to each other. Hence, we assume that all machines within a subnet can directly access the machines that the representative source machine can access. Only a part of the machines outside the subnet are visible from within the subnet, due, e.g., to firewall restrictions. We model the accessibility of machines identified through the scans as direct edges in the network graph. That is, machine m_1 is connected in our model to machine m_2, if m_2 is visible from m_1 or vice versa.

In addition, Nessus reveals for each identified host its operating system. The network contained hosts running Windows and Linux (with a few versions of each operating system). Nessus also identifies softwares with potential vulnerabilities that run on the machines. Our model contains about 50 such software, including well known applications such as *openssh*, *tomcat*, *pcanywhere*, ftp services, and many more.

Nessus identifies potential vulnerabilities in the scanned machines. These vulnerabilities may not actually exist, but the only way to know is by performing an exploit for that vulnerability, which of course we did not do. As we explain above, we model the uncertainty about the existence of the vulnerability directly in our model. The agent must attempt an exploit and check afterwards whether the exploit was successful, and hence, whether the vulnerability actually exists.

Nessus finds vulnerabilities of varying importance. For the purpose of this experiment we ignored all the lesser vulnerabilities, which do not allow an attacker control of the system. We remain with about 60 serious types of vulnerabilities that exist in the network. We remove from the network all hosts that do not run any software for which a serious vulnerability exists, remaining with about 35 hosts.

For constructing our pnetesting goal, we took two random hosts from the innermost subnet, and set them as the target hosts. The problem goal is to gain control over one of these two hosts. To add uncertainty into the model, we specify in the initial belief for each host, aside from the true operating system and the real applications running on it, one more potential operating system, and 3 more possible applications.

[1] `https://www.tenable.com/products/nessus-vulnerability-scanner`

5.1 POMDP Model

We use the above data to create a POMDP formulation of the network, following the overall guidelines set by Sarraute et al (), by differing on some details to be more compatible with our contingent planning modeling. We follow Sarraute et al, defining a state for each possible configuration of all network machines. We use deterministic vulnerability exploit actions that take control of a given machine. These action result also in a deterministic success or failure observation.

Our sensing actions support only binary observations. Hence, we replace Sarraute's *ProbeOS(host)* sensing action with a set of sensing actions *ProbeOS(host,os)* sensing actions. The Nessus output relates a vulnerability to a software, rather than a port. We hence do not use *ProbePort* actions, but rather *ProbeSW(host,software)* actions. We ignore ports in our problem formalization, but open ports can also be taken into account using a slightly more complex description. The network connectivity structure is embedded into the transition and observation probabilities. Probing a machine that has no controlled neighbor results always in a *false* observation. Attempting an exploit on a machine that has no controlled neighbor does not change the state, even if the exploited vulnerability exists on the host.

Our reward structure is substantially different than suggested before. First, the Nessus output contains costs for the exploits that were identified (Lai and Hsia 2007). We use these costs for the exploit actions in our model. Probing a host for its operating system can be done by only listening to the network traffic from that host (Yarochkin *et al.* 2009). We hence set a very low cost (0.1) for *ProbeOS(host,os)* actions. Probing for running software is more costly, because it requires sending requests to that software awaiting responses. We hence set the cost of software probing to be equal to the least costly exploit (1 in our data).

Finally, we focus on modeling a goal directed approach, where the attacker goal is to take control over some target machine, perhaps containing some important information. We model this by rewarding the attacker only when terminating (using the *terminate* action), when one of the target machine is controlled, using a large reward (1000). There can be deadend states from which the target machine cannot be controlled (for example, when it has no vulnerabilities). The agent receives no penalty when terminating at a deadend state. We add, however, a substantial penalty (-1000) for terminating when no target machine is controlled, in a state which is not a deadend. We consider these rewards to be the weakest part of our modeling approach, because they are not supported by the data, but induced by us to motivate the agent towards a desirable behavior. A deeper investigation into setting such rewards is left for future work.

6 Empirical Study

We now provide an empirical study of the contingent planning approach to modeling penetration testing.

To obtain a plan tree (graph) for the pentesting contingent problems we use a modification of the offline planner CPOR (Komarnitsky and Shani 2014), which constructs an efficient plan graph by identifying states for which a solution was already computed. CPOR uses an online solver as a heuristic. We replace this component by a domain-specific heuristic.

Our heuristic analyzes the network graph, and identifies the next accessible host closest to the target machines. Then, the heuristic probes that host attempting to discover its operating system and running software. Finally, the heuristic attempts possible exploits for the identified operating system and software. Once the host is controlled, or if all exploits have failed, the heuristic chooses the next host to attack.

We augment CPOR with a simple deadend detection mechanism. If all paths from the set of controlled hosts to the target machines contain a host for which all possible exploits have failed, then there is no possible path to the target machines, and an unavoidable deadend is declared.

6.1 Real Network

When running the contingent planner, we avoid traversing branches that correspond to these additions, and cannot be reached in reality. For example, if the true operating system of host m_1 is *Windows 7*, we may add *Windows NT* as a second operating system. The planner then uses a ping action to sense the operating system. For instance, the planner may choose to probe for *Windows NT* first. In reality, the host was running *Windows 7*, and hence, the attacker will receive a *false* observation for this action. We allow the planner to execute the probe action, but then traverse only the branch where the observation conformed to the true one. That is, in this example we will only traverse the *false* child, ignoring the *true* child. This better simulates the attacks of a real attacker over this real network, ignoring impossible branches. As such, although the plan s contain many ping actions, our plan tree branches only on the success or failure of exploits.

We ran our contingent planner over the resulting problem. The planner computed a plan graph in 116 seconds, containing 1453 nodes. The equivalent POMDP with 35 hosts, 2 operating system, 50 software, and 60 vulnerabilities, assuming 25 software for each operating system, and 5 average vulnerabilities per software, has a state space of about $2^{35\times2\times25\times5} = 2^{8750}$, which is far beyond the capabilities of POMDP solvers. Even factored POMDP solvers (Shani *et al.* 2008; Veiga *et al.* 2014) do not scale up to these sizes.

6.2 Performance over Smaller Networks

While our planner scales well to large network, the quality of the policy computed by the planner is also important. This can be done by comparing the expected reward from executing the policy to the expected reward of a POMDP policy.

As POMDP solvers cannot scale up to the real network, we create a set of tiny networks that can be handled by the SARSOP solver (Kurniawati *et al.* 2008), using the data gathered by our network scan. First, we sample a set of n connected machines from the various subnets of the network. We sample a set of m software for each machine, from its real set of running software, following the frequency of the software given an operating system in our scan. We then sample k vulnerabilities for each software, again following the frequency of vulnerabilities in our data. Each software and vulnerability can either exist on a machine or not. This process provides a set of configurations, and we assume that each machine can have any of these possible configurations.

We compute the probabilities of a configuration following the distribution of operating systems, software, and vulnerabilities in our scan. The probability of a machine running operating system o, software s, and vulnerability v, is computed using $pr(o) \times pr(s|o) \times pr(v|s)$, where the probabilities are the maximum likelihood estimators from the data, normalized to the reduced sample in the particular instance.

			POMDP				Contingent	
n	m	k	$\|S\|$	$\|A\|$	Time	$E(R)$	Time	$E(R)$
2	4	6	203	22	0.75	291.271	0.009	256.143
2	5	7	493	25	5.1	265.753	0.009	247.434
3	2	2	125	19	0.25	393.337	0.12	315.614
3	2	4	4913	24	587	269.729	0.18	244.834
3	4	6	8323	32	3349	139.232	0.19	114.514
3	5	7	20,213	38	N/A	N/A	0.3	104.446

Table 1: Comparing expected discounted reward and time (secs), over small networks sampled from the real network distributions. n, k, m are the number of machines, software, and vulnerabilities, respectively. $|S|$ and $|A|$ are the number of states and actions in the POMDP problem.

Table 1 compares the expected discounted reward of our plan graph to that of the SARSOP policy for the equivalent POMDP model. As can be seen, the expected reward of the contingent planning solution is lower than the expected reward of POMDP solution. This can be attributed in part to the heuristic in our planner, that intentionally ignores costs and probabilities. Adding these factors to the heuristic selection of actions is left for future research.

On the other hand, the POMDP solver fails even on very small networks, with only 3 machines, 4 software, and 6 vulnerabilities. This is clearly far below acceptable model sizes.

7 Conclusion and Future Work

We suggest contingent planning as an alternative for modeling pentesting. This model allows for partial observability of various properties, such as a machine operating system and installed software, that can be sensed by probe actions. Thus, contingent planning offers a richer model than classical planning, while being able to scale up better than POMDP-based approaches. We show that our approach scales to real network sizes far beyond the capabilities of current POMDP solvers, and compare its expected reward to that of a POMDP over smaller sub-sampled networks.

In the future we intend to create smarter heuristics for ordering actions given states, to achieve better expected rewards. In addition, we intend to experiment with a factored representation of the POMDP problem, to try scaling up to

more reasonable problem sizes.

References

Alexandre Albore, Héctor Palacios, and Hector Geffner. A translation-based approach to contingent planning. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1623–1628, 2009.

Burns et al. *Security Power Tools*. O'Reilly Media, 2007.

Karel Durkota, Viliam Lisý, Branislav Bosanský, and Christopher Kiekintveld. Optimal network security hardening using attack graph games. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 526–532, 2015.

Jörg Hoffmann. Simulated penetration testing: From "dijkstra" to "turing test++". In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pages 364–372, 2015.

Radimir Komarnitsky and Guy Shani. Computing contingent plans using online replanning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2322–2329, 2014.

Hanna Kurniawati, David Hsu, and Wee Sun Lee. SARSOP: efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems IV, Eidgenössische Technische Hochschule Zürich, Zurich, Switzerland, June 25-28, 2008*, 2008.

Yeu-Pong Lai and Po-Lun Hsia. Using the vulnerability information of computer systems to improve the network security. *Computer Communications*, 30(9):2032–2047, 2007.

Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

Christian J. Muise, Vaishak Belle, and Sheila A. McIlraith. Computing contingent plans via fully observable non-deterministic planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 - 31, 2014, Québec City, Québec, Canada.*, pages 2322–2329, 2014.

Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. POMDPs make better hackers: Accounting for uncertainty in penetration testing.

Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Penetration testing == POMDP solving? In *SecArt'11*, 2011.

Guy Shani, Pascal Poupart, Ronen I. Brafman, and Solomon Eyal Shimony. Efficient ADD operations for point-based algorithms. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, pages 330–337, 2008.

Edward J. Sondik. The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304, 1978.

Tiago Veiga, Matthijs TJ Spaan, Pedro U Lima, Carla E Brodley, and Peter Stone. Point-based pomdp solving with factored value function approximation. In *AAAI*, pages 2513–2519, 2014.

Fedor V Yarochkin, Ofir Arkin, Meder Kydyraliev, Shih-Yao Dai, Yennun Huang, and Sy-Yen Kuo. Xprobe2++: Low volume remote network information gathering tool. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 205–210. IEEE, 2009.