Simulated Penetration Testing as Contingent Planning

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

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

Abstract

In penetration testing (pentesting), network administrators attack their own network to identify and fix vulnerabilities. Planning-based simulated pentesting can achieve much higher testing coverage than manual pentesting. A key challenge is for the attack planning to imitate human hackers as faithfully as possible. POMDP models have been proposed to this end, yet they are computationally very hard, and it is unclear how to acquire the models in practice. At the other extreme, classical planning models are scalable and simple to obtain, yet completely ignore the incomplete knowledge characteristic of hacking. We propose contingent planning as a new middle ground, feasible in both computation burden and model acquisition effort while allowing for a representation of incomplete knowledge. We design the model, show how to adapt available solvers, and show how to acquire the model from real network scans in practice. We experiment on real networks and show that our approach scales to practical input sizes.

1 Introduction

Penetration testing (pentesting) identifies vulnerabilities in networks, by launching controlled attacks (Burns et al. 2007). A successful attack reveals weaknesses in the network, which the network administrators can then remedy. 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. The attack continues until the attacker succeeds in controlling a critical machine, allowing to access sensitive data or inflict critical damage.

AI planning was previously suggested as a tool for conducting pentesting. In particular, the two extreme cases have been explored — classical planning, where the entire network structure and machine configuration are known to the attacker (Boddy et al. 2005; Lucangeli, Sarraute, and Richarte 2010); and Partially observable Markov decision processes (POMDP), where machine configurations are initially unknown to the attacker and can be partially sensed (Sarraute, Buffet, and Hoffmann 2012).

The classical planning approach scales well, and has been used commercially. However, the simplifying assumptions of complete knowledge and 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) have been proposed as a slightly more realistic model, yet they still do not reflect the partial information attackers typically have, nor the sensing which is an integral part of real-world attacks.

The POMDP approach, in contrast, can be argued to be a valid representation of the real world. One can model the attacker's prior knowledge about the network configuration as a probability distribution over possible states, known as a belief. Probing actions, designed to reveal configuration properties of machines, can be modeled as sensing actions.

However, the POMDP approach also has several major weaknesses. POMDP solvers do not scale anywhere near the size of realistic input (Sarraute et al. (2012) propose to use the POMDP for single machines only, and assemble an approximate attack for the overall network from that). Further issues pertain to model acquisition. The standard reward maximization models are questionable in practice as it is unclear how to quantify the importance of different assets in the network, and therewith capture attacker utility. Humans find such quantification notoriously hard to do, and different reward choices may affect the result produced in complex and unpredictable ways. Finally, and most importantly, the POMDP model requires an accurate probability distribution for the initial belief. In pentesting, as in many other applications, it is unclear how to reliably obtain that distribution. In particular, this pertains to identifying an accurate probability distribution over the possible OS for the machines in the network. Prior work (Sarraute, Buffet, and Hoffmann 2012) has devised only a crude over-simplifying model of "software updates", for generating synthetic benchmarks. Again, the choices made in the design of the initial belief may significantly affect the result, hampering the practical use and added value of the analysis for a human network administrator.

Here we introduce an intermediate model, between classical planning and POMDPs. We replace POMDPs with partially observable contingent planning, a qualitative model where probability distributions are replaced with sets of possible configurations or action effects, and rewards are replaced with goal-directed behavior (e.g. (Weld, Anderson, and Smith 1998; Hoffmann and Brafman 2005; Albore, Palacios, and Geffner 2009)). Contingent planners attempt to find a plan tree (or graph) of actions, where edges are labeled by observations and all leaves attain goal states. Solvers for this type of model scale much better than POMDP solvers, and can be used for more practical networks. As these models require neither probabilities nor rewards, we avoid the need for their specification.

Contingent planning naturally models an attacker who initially has qualitative knowledge about the network configuration, and improves that knowledge during the attack using the outcomes of attempted exploits, as well as explicit sensing actions. In particular, like POMDPs and unlike any other planning-based pentesting model yet proposed, the contingent planning model is able to consciously mix exploits and sensing actions similarly to real attackers.

We formalize our model as a qualitative variant of the partially-observable Canadian Hacker Problem previously sketched by Hoffmann (2015). We then provide a concrete encoding in terms of established contingent-planning PDDL. We show how to acquire this model from network scans in practice. The semantics of our model slightly differs from standard contingent planning, in its handling of unsolvable initial states (network configurations in which the target machine is unreachable). We show how to adapt an available contingent planner to handle this semantics.

We experiment with real networks of two organizations, with the real vulnerabilities that were found in a scan of these networks. We show that our contingent planner succeeds in computing plans for the networks. In addition, we compare contingent plans to POMDP policies for much smaller networks that were sub-sampled from the real network data that we collected. The results show that our method finds attack policies of lower yet reasonable quality, many orders of magnitude faster than the POMDP model.

We emphasize that the experiments over real networks constitute a contribution on their own. Previous literature on automated planning for pentesting has evaluated methods over artificially generated toy benchmarks only.

2 Pentesting and POMDPs

We summarize pentesting and its properties. We briefly introduce POMDPs and previous POMDP pentesting models.

2.1 Networks and Pentesting

We model networks as directed graphs whose vertices are a set M of *machines*, and edges representing connections between pairs of machines. Like previous work in the area, we assume that the attacker knows the structure of the network. Each machine in the network can have a different *configuration* specifying its hardware, operating system, installed software, installed updates, service packs, etc. The network configuration is the conjunction of all machine configurations. The configuration may be partially 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 such methods must be combined to identify the operating system. Sending such seemingly innocent requests to identify a machine's 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.

We say that a machine m is *controlled* if the attacker can run a malware on it, and can use it to fingerprint and attack other machines. Many configurations have *vulnerabilities* that can be *exploited* to gain control over the machine, but these vulnerabilities vary between configurations. Thus, to gain control over a machine, one typically 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 (e.g. a certain security update) made this exploit useless.

The objective of *penetration testing (pentesting)* is to control certain machines containing important assets in the network. A *reached* machine m is connected to a controlled machine. All other machines are *not reached*. The attacker starts controlling the internet, and all machines that are directly connected to the internet are reached.

2.2 Properties of Pentesting

Previous work on planning models of pentesting has made a number of observations – modeling assumptions justified in the pentesting application – that are also important here. Hoffmann (2015) comprehensively discusses and motivates these observations. In what follows, we briefly point out those observations relevant in our context.

The first and most basic observation is that infinite looping behaviors are not useful in pentesting. There is a finite number of machines to gain control over, and actions naturally have a strictly positive cost. In practice, a pentest stops either when the attack succeeds, or when the attacker decides to give up. It is thus natural to frame pentesting as a Stochastic Shortest Path (SSP) problem.

Furthermore, as argued by Hoffmann (2015), the following restrictions are reasonable: (i) The only effect of exploits is control of the target machine, not the network structure nor its configuration. (ii) Each exploit either succeeds or fails, where the latter outcome has no effect. (iii) The outcome of every action (exploits and sensing) depends deterministically on the network configuration. We will refer to these restrictions by their numbers (i) – (iii).

2.3 POMDPs

Partially observable Markov decision proccesses (POMDPs) (Sondik 1978) are a well known model of probabilistic planning under partial observability. 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, and Ω is a set of possible observations. The transition function $tr(s, a, s')$ provides the probability of transitioning from state s to state s' using action a. The observation function $O(a, s', o)$ specifies the probability of observing $o \in \Omega$ after executing action a, arriving at state s'. The reward function $R(s, a, s')$ provides the reward for executing action a in state s arriving at state s'; rewards may be negative, corresponding to costs. Finally, b_0 is the initial *belief*, a probability distribution over states.

We do not discuss POMDPs in detail (optimal solutions, methods for obtaining them) as this is not relevant to understand our contribution. Suffice it to say that tr and O induce a transition system over beliefs, and that the solution is a *policy* π mapping beliefs to actions. There is a variety of optimization objectives whose applicability depends on context. In pentesting, due to the properties outlined above, it makes sense to assume an absorbing terminal state – reachable either by achieving the goal or by a dedicated give-up action – and to maximize the expected undiscounted reward given π.

Sarraute et al. (2012) introduced a POMDP model of pentesting. The states s are the possible configurations of the network, encompassing the network structure and software. Exploit actions have preconditions on the network configuration, and may either succeed or fail in gaining control over a target machine; sensing actions provide information about configuration aspects. A give-up action always allows the attack to terminate, at high cost. In short, Sarraute et al.'s model is a straightforward encoding of what we outlined above, satisfying in particular the properties discussed.

Hoffmann subsequently observed that, given (i) - (iii), pentesting is closely related to the Canadian Traveler Problem (e.g. (Papadimitriou and Yannakakis 1991; Eyerich, Keller, and Helmert 2010)), where the task is to traverse a map whose individual road connections may be blocked. The only major difference is that, in pentesting, being "at" a location is replaced by controlling a machine, so that, rather than moving from one location to the next, the "traveler" accumulates controlled machines. Hence, Hoffmann introduced the *Canadian Hacker Problem (CHP)* as an abstract model, and in particular the *partially observable CHP (POCHP)*.

In POCHP, the input consists of a graph G with start node n_0 and target nodes N_T, where each edge is labeled with a conjunctive precondition over configuration propositions C; exploit actions attempt to traverse edges while sensing actions observe elements of their labels: and an initial belief specifies a probability distribution over the truth value assignments to C. This abstract model is naturally made concrete – formalized and solved – as a POMDP, and indeed Sarraute et al.'s model does exactly that.

A major downside of POMDP models for pentesting is the need to acquire the initial belief distribution, and the rewards and/or the cost of the give-up action. These difficulties are irrelevant when considering a qualitative model instead.

In the following sections, we give the background on contingent planning; we design a qualitative variant of POCHP as an abstract model based on contingent planning with modified semantics; we design a concrete model based on established contingent-planning PDDL; finally we explain how we adapted available contingent-planning solvers to suit the desired semantics, for our experiments.

3 Contingent Planning

A contingent planning task is a tuple $\Pi = \langle P, A_{\mathsf{act}}, A_{\mathsf{sense}}, \phi_I, G\rangle$. Here, P is a set of propositions, A_{act} is a set of *actuation actions*, A_{sense} is a set of *sensing actions*, ϕ_I is a formula over P specifying the initial belief (see below), and $G \subseteq P$ is the goal. An actuation action a_{act} is defined by a precondition $\mathsf{pre}(a_{\mathsf{act}})$ and an effect $\mathsf{eff}(a_{\mathsf{act}})$, where $\mathsf{pre}(a_{\mathsf{act}}) \subseteq P$ and $\mathsf{eff}(a_{\mathsf{act}})$ is a set of conditional outcomes (c_l, l). In such a conditional outcome, the *condition* c_l is a formula over P, and the *outcome literal* l a literal over P; the conjunction of outcome literals in $\mathsf{eff}(a_{\mathsf{act}})$ is satisfiable (i.e., there are no contradictory effects).

A sensing action a_{sense} has a precondition like actuation actions; instead of an effect, it defines a proposition $\mathsf{obs}(a_{\mathsf{sense}}) \in P$ whose value is observed when executing a_{sense}.

The semantics of states and actuation actions is defined as in classical planning (details omitted for brevity). The outcome state of applying action a to state s is denoted $a(s)$.

A belief b is a set of states. The *initial belief* is $b_0 := \{s \mid s \models \phi_I\}$. Action a is applicable at belief b if $\mathsf{pre}(a) \subseteq s$ for all $s \in b$. In that case, for $a \in A_{\mathsf{act}}$ the outcome belief is defined as $a(b) := \{a(s) \mid s \in b\}$; for $a \in A_{\mathsf{sense}}$ there are two outcome beliefs, namely $a(b, 1) := \{s \mid s \in b, s \models \mathsf{obs}(a_{\mathsf{sense}})\}$ and $a(b, 0) := \{s \mid s \in b, s \not\models \mathsf{obs}(a_{\mathsf{sense}})\}$. For a *goal belief* b, $G \subseteq s$ for all $s \in b$.

Plans for a contingent problem are *action trees* T, i.e., edge-labeled trees whose nodes are actions, where each $a \in A_{\mathsf{act}}$ has a single (unlabeled) outgoing edge, and each $a \in A_{\mathsf{sense}}$ has two outgoing edges labeled with $true$ and $false$ respectively. An action tree T *solves* a belief b if (1) T is empty and b is a goal belief; or the root of T, a, is applicable in b and either (2a) $a \in A_{\mathsf{act}}$ and the tree rooted at a's child solves $a(b)$, or (2b) $a \in A_{\mathsf{sense}}$ and the tree rooted at a's child labeled with 1 (respectively 0) solves $a(b, 1)$ (respectively $a(b, 0)$). A *plan* for a task Π is an action tree that solves b_0.[1]

We remark that the semantics just specified is known as *strong plans* (Cimatti et al. 2003). Alternative semantics are *weak plans*, where the goal is achieved in at least one terminal state; and *strong cyclic plans*, that ascertain the goal in all terminal states but may include loops. Strong cyclic plans are not relevant here because, matching property (iii), our actions are deterministic. Weak plans are not of interest either as, with property (ii), they default to a classical-planning pentest model (where exploit preconditions are assumed/sensed to be true). That said, as we shall discuss in depth momentarily, a pentest typically contains possible initial configurations from which the goal is not reachable at all. We will adapt the strong planning semantics above to cater for such cases, permitting the attacker to ignore them.

4 The Qualitative POCHP

We detail a qualitative version of POCHP (in difference to Hoffmann's sketch) as the formal basis for our approach. We

[1]Action trees can often be represented more compactly as a directed acyclic graphs (Komarnitsky and Shani 2014; Muise, Belle, and McIlraith 2014). For ease of exposition, we only discuss trees here.

refer to our model as Q-POCHP. Like POCHP, Q-POCHP is an abstract model, intended for the purpose of specification and discussion. For practical purposes, we will encode Q-POCHP as a concrete contingent-planning model below.

An instance of Q-POCHP is defined by an edge-labeled directed graph with nodes N and edges E. There is a start node $n_0 \in N$, and a set of target nodes $N_T \subseteq N$. There can be several, differently labeled edges, between two given nodes. Any one edge may be *open* or *blocked*, and the agent does not initially know which edges are open.

The nodes correspond to machines, where n_0 is initially controlled by the attacker and N_T are the critical machines. The pentest should check whether the attacker can gain control over any one $n_T \in N_T$ (the interpretation of N_T is disjunctive). The edges $n \xrightarrow{\psi} n'$ encode, in addition to the network structure (connectivity), the possible exploits between connected machines n and n'. The edge specifies that, under a configuration condition ψ depending on the vulnerability in question (see below), the attacker can run an exploit from n to gain control over n'. The exploit is considered *possible*, hence the edge appears in the graph; but the agent does not know whether or not the required vulnerability actually exists on n'. The latter – whether the edge is open or blocked – depends on the configuration of n'.

We assume a set C of machine *configuration propositions*. We associate each node n with its *configuration* $\phi(n)$, a conjunction of atoms from C, modeling n's properties. We associate each edge (n, n') with its *condition* $\psi(n, n')$, also a conjunction of atoms from C, modeling the configuration properties required on n' for the respective vulnerability to be present. We identify atom conjunctions with sets.[2]

Edge conditions are known to the agent, reflecting that exploits and their prerequisites are known to the attacker. As such, the edge conditions are part of the input, in the form of edge labels. Node labels, however, are unknown to the agent, as they encode the machine configuration properties, unknown prior to the attack. Instead, the agent has qualitative prior knowledge — a set L_0^N of *node-labelings* deemed to be possible, where a node-labeling is a function $l^N : N \mapsto 2^C$. We do not discuss a compact representation of L_0^N here, at the level of the abstract model. In our concrete model, we specify L_0^N as part of the initial belief ϕ_I.

Say the agent controls a node n connected through an (open or blocked) edge $n \xrightarrow{\psi} n'$ to a node n'. Then the agent may sense the *observable* properties of n', one at a time. The agent may also attempt to seize control of n', through the vulnerability represented by $n \xrightarrow{\psi} n'$. If the edge is open, n' becomes controlled. Otherwise, the agent knows that there exists a property $p \in \psi(n, n')$ such that $p \notin \phi(n')$.

Optionally, one can define costs for exploits and sensing. We assume unit costs here, for simplicity of presentation, and as even the acquisition of costs is an issue in practice.

Overall, the syntax of Q-POCHP thus is as follows:

[2]One could allow arbitrary formulas over C as conditions, but conjunctions of atoms suffice for practical modeling, and result in correspondingly simpler PDDL syntax.

Definition 1 (Q-POCHP Syntax)**.** *A Q-POCHP task is a tuple* $\Pi = \langle N, E, n_0, N_T, C, O, L_0^N, l^E \rangle$*, where* $O \subseteq C$ *are the observable configuration propositions,* $l^E : E \mapsto 2^C$ *is the edge-labeling, and all other elements are as above.*

We specify the semantics following the definitions for contingent planning given before, in terms of transitions over states and beliefs. We next introduce the basic notations, then define a suitable concept of solutions.

A *state* s in a Q-POCHP task is a pair (N_s, l_s^N) where $N_s \subseteq N$ is the set of controlled nodes and l_s^N is the node-labeling. An edge $e = (n, n')$ is *active* at s if $n \in N_s$. In that case, the agent can apply an action a_e exploiting e. The outcome state, denoted $a_e(s)$, is $(N_s \cup \{n'\}, l^N)$ if $l^E(e) \subseteq l_s^N(n')$ (success), and is s otherwise (failure).

We assume that the attacker does not immediately see whether or not an exploit has succeeded, but can use a sensing action to make that observation. So there are two types of sensing actions, $a_{n'}$ sensing for $n' \in N$ whether or not n' is controlled ($n' \in N_s$?), and $a_{n',o}$ sensing for $(n', o) \in N \times O$ whether or not n' has property o ($o \in l_s^N(n')$).

A *belief* b is a set of states. The *initial belief* is $b_0 := \{(\{n_0\}, l_0^N) \mid l_0^N \in L_0^N\}$. An edge $e = (n, n')$ is active at a belief b if it is active in all $s \in b$. In that case, the exploit action a_e, as well as the sensing actions $a_{n'}$ and $a_{n',o}$, are applicable at b. The outcome belief of a_e is $a_e(b) := \{a(s) \mid s \in b\}$ exactly as in contingent planning. The outcome belief $a_{n'}(b, 1)$ contains those $s \in b$ where n' is controlled, and $a_{n'}(b, 0)$ contains those $s \in b$ where that is not so; similarly for $a_{n',o}(b, 1)$ and $a_{n',o}(b, 0)$. A *goal belief* is one where there exists $n \in N_T$ s.t. $n \in N_s$ for all $s \in b$.

It remains to define what a solution, a plan, for a Q-POCHP task is. Here we cannot simply adopt the contingent planning semantics, due to the existence of *unsolvable* configurations, i.e., network configurations given which no target node can be reached. If we adopted the strong-plan semantics given above, then such Q-POCHP tasks would have no plan. Yet this – reporting that there is no strong plan – would be useless for pentesting analysis.

One can tackle this problem by introducing a give-up action, akin to SSP variants of POMDP pentesting as designed previously by Sarraute et al. (2012). This would, however, incur the model-acquisition issue of defining the cost for giving up – we cannot assign unit cost to this as otherwise the attack plan would always simply give up.

The approach we propose here instead is to permit the attacker to ignore unsolvable network configurations, i.e., to simply not tackle these in the plan. From the point of view of the system administrator using the pentest analysis, the plan then specifies what the attacker will do in all cases relevant to the target, in other words a maximally successful attack.

Precisely, we say that a node-labeling l^N is *solvable* if there exists a path $n_0, n_1, \dots, n_k$ where $n_k \in N_T$ and, for every $0 \leq i < k$, $(n_i, n_{i+1}) \in E$ and $l^E(n_i, n_{i+1}) \subseteq l^N(n_{i+1})$. A state s is solvable if its node labeling l_s^N is solvable. Observe that solvability here is *static*, in the sense that the state-changing actions – the exploits – do not affect the node-labeling, in line with observation (i). Planner decisions have no impact on configuration solvability, so unsolvable configurations can be naturally handled outside the plan.

Specifying the semantics of Q-POCHP is now simple. Action trees are defined exactly as in contingent planning, and we say that an action tree T *solves* a Q-POCHP belief b under the exact same conditions (1) for empty trees (leaf nodes), (2a) for exploit actions, and (2b) for sensing actions. It just remains to define what a plan for the task is:

Definition 2 (Q-POCHP Semantics)**.** *A* plan *for a Q-POCHP task* Π *is an action tree that solves the belief* $\{s_0 \mid s_0 \in b_0, s_0 \textit{ is solvable}\}$.

Note that, while elegant and simple, it is unclear how to make this definition operational. Given a specific node-labeling l^N, one can test easily whether or not l^N is solvable (remove blocked edges from the graph and test the reachability of N_T from n_0).

We need, though, to identify the subset of solvable l^N within the initial belief, which in practice will be represented compactly. We will tackle this below, when describing our modified planning tools for contingent pentesting, by observing that Definition 2 is equivalent to a restricted form of allowing to give up. Prior to that, we introduce our concrete contingent-planning model.

5 Concrete PDDL Model

We now briefly review a PDDL specification for a Q-POCHP task, formulated as a contingent planning problem under partial observability, using standard contingent PDDL syntax (Albore, Palacios, and Geffner 2009; Bonet and Geffner 2011).

First, the machine connectivity structure is captured by $(hacl\ ?src\ ?target)$ predicates, specifying that machine $?target$ can be directly reached from machine $?src$. These connectivity predicates are initially known and static. We use a predicate of the form $(controlling\ ?m)$ to capture the set of machines controlled by the attacker. The node n_0 in our modeling represents a public machine outside the organization network, and is the only one initially controlled.

The set C of machine configuration properties is modeled by a set of propositions modeling the operating system and software running on a machine. We use predicates of the form $(HostOS\ ?m\ ?os)$ and $(HostSW\ ?m\ ?sw)$ to model which OS and software a specific machine runs. One could add additional such predicates capturing other configuration properties, such as specific installed updates, open ports, and so forth. These properties are initially unknown, and can be sensed by probing from a controlled neighboring machine:

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

Recall that edges $e \in E$ in a Q-POCHP capture not only connectivity, but also a possible existence of a vulnerability on the target machine. In the Q-POCHP this is captured by the edge labeling $n \xrightarrow{\psi} n'$, where ψ captures the configuration condition, i.e., combination of OS and software required for a vulnerability to apply. To model this in a reusable manner, we model the requirements for a vulnerability of a given type independent of a machine, using predicates of the form $(Match\ ?os\ ?sw\ ?v)$ specifying whether a given vulnerability can exist in a given os-software combination.

Even though a target machine configuration matches a specific vulnerability, the vulnerability may not exist on the target machine due to, e.g., a security patch that was installed and cannot be probed from outside the machine. We capture this by $(ExistVuln\ ?v\ ?m)$ predicates, specifying whether a vulnerability exists on a machine. These predicates are initially unknown, and cannot be directly sensed, modeling the inability to probe for all information. An edge $(n, n') \in E$ is hence open if $(hacl\ n\ n') \wedge (ExistVuln\ v\ n')$, where v is the vulnerability associated with the edge.

In the Q-POCHP semantics this means that we model directly only the observable properties $O \subseteq C$ (operating system and running software), and the unobservable properties are modeled by the $(ExistVuln\ ?v\ ?m)$ predicates.

An attacker controlling a machine can attempt to exploit vulnerabilities to control neighboring machines only if the vulnerability may exist on the machine configuration. Only after probing for the observable machine configuration properties, the attacker can launch an *exploit* action:

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

We follow the Q-POCHP definition where the attacker does not know whether an attack has succeeded. This is modeled in the conditional effect of the exploit action, which achieves control only if the vulnerability exists. Hence, following an exploit on a machine m the value of $(controlling\ m)$ becomes unknown. We add a sensing action to check whether an attacker controls a machine:

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

We use a CNF formula (in addition to *oneof* clauses), standard in contingent planning problems, to express the initial belief — the set of possible states. Unit clauses represent known facts, such as the network connectivity structure. Disjunction and *oneof* clauses capture partial knowledge, such as the possible operating system on a given machine.

For example, a possible initial belief may be:

```
(: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)) ...
)
```

specifying, e.g., that machine *host0* runs either Windows NT 4 Server, or Windows NT 4 Enterprise edition, and that one of the machines *host0* or *host1* runs Internet Information Services (IIS). This captures partial information that is initially known to the attacker, such as that a specific subnet contains at least one IIS server. The *internet* object captures the starting point of the attacker (n_0).

We use a disjunction to specify whether a vulnerability exists on a machine, allowing a machine to have multiple vulnerabilities. It is possible, however, that the vulnerability that exists will not match the operating system and the software on that machine, and hence, cannot be exploited. Thus, an attacker cannot launch an exploit for that vulnerability, and unless other vulnarabilities exist can never achieve control over the machine. This can lead to unsolvable initial states as previously discussed.

6 Adapting Contingent Planners

We now introduce our contingent pentesting planner. We begin by showing how to tackle the non-standard Q-POCHP semantics as per Definition 2, then we detail the concrete tools we implemented for our empirical evaluation.

6.1 Operational Q-POCHP Semantics

As previously discussed, the Q-POCHP semantics is not immediately operational, as it is unclear how to identify and compactly represent the subset of solvable initial states among the initial belief in the concrete model above. We tackle this by observing that such identification can be done as part of the search for a contingent plan. Namely, we allow the planner to *give up* – to have non-goal beliefs as leaf nodes in the plan – but only when the belief in question consists entirely of unsolvable states. This criterion is equivalent to solving only the solvable initial states to begin with (i.e. to the Q-POCHP semantics), because solvability is *static*: it depends only on the configuration, cf. property (iii), which cannot be modified by the attacker's actions, cf. property (i).

We make this formal in the Q-POCHP framework. Let $\Pi = \langle N, E, n_0, N_T, C, O, L_0^N, l^E \rangle$ be a Q-POCHP task. We say that a belief b in Π is *hopeless* if all $s \in b$ are unsolvable, i.e., if all the node-labels l_s^N of these s do not open any path from n_0 to a target node $n_T \in N_T$.

Definition 3 (Q-POCHP Give-Up-Hopeless Semantics)**.** *A* give-up-hopeless plan *for a Q-POCHP task* Π *is an action tree that solves* b_0 *when allowing empty* T *not only at goal beliefs* b*, but also at hopeless beliefs* b*.*

Theorem 1. *Let* Π *be a Q-POCHP task, and let* T *be any action tree in* Π*. We have:*

1. *If* T *is a give-up-hopeless plan for* Π*, then* T *is a plan for* Π*.*
2. *If* T *is a plan for* Π*, then a give-up-hopeless plan* T' *for* Π *can be constructed in poly-time in the size of* Π *and* T*.*

Proof sketch: This holds because state-solvability is static: if s is any state reachable from some $s_0 \in b_0$ in Π, then s is unsolvable if and only if s_0 is. The first claim then holds because, starting from $\{s_0 \mid s_0 \in b_0, s_0 \text{ is solvable}\}$, the hopeless beliefs will be empty and, therefore, trivially goal beliefs. For the second claim, we merely have to extend each leaf node of T with a sensing action for each target node, distingushing the goal states from the unsolvable ones. ■

Given Theorem 1, we can find plans for Q-POHCP tasks by finding give-up-hopeless plans. This is straightforward in principle: modify the search for a contingent plan to treat beliefs known to be hopeless like goal beliefs. Clearly, the outcome will be a give-up-hopeless plan.

Of course, in general, we cannot test easily whether or not a belief is hopeless. But there are many methods allowing to prove unsolvability in planning, in particular fast sufficient criteria for doing so. Furthermore, in the specific case of Q-POCHP and our concrete PDDL model, given the known connectivity structure of the network, a belief is proven hopeless once there exists a set of edges in the connectivity graph that are already known to be blocked, and that form a cut between the target machines and those controlled by the attacker. We discuss in the next section our concrete solver design.

Some words are in order regarding prior work related to Definition 3 and Theorem 1. Hoffmann et al. (2012) also define a contingent-planning semantics allowing unsolvable leaf nodes. But they do so to handle the specifics of the business processes their planner generates, and their semantics is very specific to that application.

Daum et al. (2016) compute a contingent plan solving the maximum possible subset of initial states. But in their context (undoability checking), solvability is not static, so that Theorem 1 does not hold; and their solution is very different, allowing to give up anywhere but at a very high cost.

One can view the requirement, set by Definition 3, to prove a belief hopeless before giving up, as an epistemic goal on the knowledge of the planning agent, constituting a very particular case of epistemic planning (e.g. (Bolander and Andersen 2011; Ågotnes et al. 2014)). However, epistemic planners cannot currently handle problems of this magnitude, and specifying hopeless beliefs in current syntax is difficult. Some tools for planning under uncertainty use quantifications of the planning agent's uncertainty (such as belief size), for search guidance (Cimatti and Roveri 2000; Bryce, Kambhampati, and Smith 2006; Albore, Palacios, and Geffner 2009).

6.2 Q-POCHP Solver Implementation

We now describe our special purpose contingent solver for Q-POHCP problems, built on top of the CPOR offline contingent planner (Komarnitsky and Shani 2014). CPOR starts at the initial belief. If the goal can be reached using no sensing actions, then CPOR plans a path to the goal. Otherwise, CPOR uses a heuristic to decide on the next sensing action to apply at the current belief, possibly following a sequence of non-sensing actions. Then, CPOR expands both child beliefs. CPOR employs a mechanism for identifying equivalent belief states for which a plan tree has already been computed, thus reducing a plan tree to a more compact plan graph, avoiding the need to repeatedly compute identical plan tree fragments. CPOR maintains beliefs using regression (Brafman and Shani 2016), regressing queries through the action-observation sequence in the current tree branch.

Our planner maintains, in addition to the data structures required by CPOR, the current knowledge of the Q-POHCP graph — the set of edges and their status (open, closed, or unknown), and the set of machines and their properties.

We replace the CPOR heuristic, with a heuristic computed over this graph. Specifically, we identify a shortest path from a controlled machine to a goal machine, using Dijkstra's algorithm (Dijkstra 1959), assuming that all unknown edges are open. Let m be the first machine on this path. We now focus our attention in attacking m to gain control over it.

Given the set of possible vulnerabilities V_m for m (as specified in the initial belief), we identify the property (operating system or software) which is required for the largest number of vulnerabilities in V_m, and sense for it. After observing the value for that property, if there is a property in V_m that matches the machine known properties, we attempt to exploit it. Otherwise, we sense for another property.

Once an exploit has succeeded in gaining control over a machine, we recompute a shortest path, and repeat the process of attacking a machine. If we have failed to gain control over m, that is, we have exhausted all possible vulnerabilities in V_m, and all exploits failed, then we check whether the goal machines may still be reachable. If not, that is, if all edges from controlled machines to uncontrolled machines are closed, then we declare the current belief to be hopeless.

7 Model Acquisition

Understandably, modern organizations are concerned about revealing information concerning their network configuration, which might be useful for malicious hackers. It is not surprising, thus, that there is currently no publicly available network data containing all the required information, including network connectivity and each machine configuration and relevant software. 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 explain the model acquisition process, and then review some properties of the acquired models.

7.1 Network Scanning

There are many existing applications that allow scanning a network from the outside or from within to obtain a list of hosts and their properties. We use below the Nessus scanner (www.tenable.com) that scans all visible hosts in an IP range. A scan reveals the following properties for the identified machines — the operating system, some installed software, and a list of possible vulnerabilities, following the CVE convention (cve.mitre.org/).

The scans contained about 50 identified software, including well known applications such as *openssh*, *tomcat*, *pcanywhere*, ftp services, and many more. The scans revealed hundreds of vulnerability types. Following the CVSS convention (www.first.org/cvss/), we filter out all vulnerabilities of medium and low severity, remaining with about 60 types of high or critical severity vulnerabilities. Hosts with no vulnerabilities are not interesting for us, and were removed from the model.

However, a Nessus scan does not attack hosts, installing remote agents on other machines. Hence, the machines that are not directly visible from the scanning host, such as machines behind firewalls, will not appear in the scans.

To discover all machines in the network we run a scan from one host in each subnet, allowing us to identify all the hosts within the subnet, and also all hosts that are visible from within the subnet. In addition, we run one more scan from outside the organization, identifying hosts, such as the DMZ, that are directly accessible from the outside.

The scan also identifies some information that is redundant for us, such as switches and gates in the system. We filter out this information.

7.2 Q-POCHP Model

Given the scans, we can create the Q-POCHP model. First, the nodes of the Q-POCHP are the hosts identified in the scans, in addition to an "internet" host representing an arbitrary host outside the organization. For each subnet s, we assume that all hosts within s are connected. For hosts in other subnets, we assume that all hosts in other subnets that were visible during the scan from the machine within s are visible to all other hosts within s.

Nessus reveals the true operating system and software running on a machine, but an attacker must probe the machines, in a similar manner to what Nessus does, in order to achieves this information.

The model, hence, does not specify the true operating system and software of each host, but rather introduces uncertainty, by allowing each machine in subnet s to run any of the operating systems found within s. We add limited uncertainty concerning the running software. We add to the software that were identified by Nessus, a number of sampled software running on other machines within the same subnet.

This process adds limited structured uncertainty into the model, simulating the partial knowledge of an attacker. For example, an attacker may have partial knowledge about which operating systems are possible within a subnet, but not which operating system each host is running.

Finally, we set the possible vulnerabilities of each host to be those detected by the scan, avoiding adding additional vulnerabilities. The possible vulnerabilities identified by the scan may not exist on the machine, because Nessus does not attempt to attack the machine by exploiting a vulnerability. We follow this, allowing for states where these vulnerabilities may or may not exist in the model definition.

			POMDP				Contingent - Heuristic		Contingent - Random	
Machines	Software	Vulnerabilities	$\|S\|$	$\|A\|$	Time	$E(R)$	Time	$E(R)$	Time	$E(R)$
2	4	6	203	22	0.75	291.271	0.009	256.143	0.251	60.48
2	5	7	493	25	5.1	265.753	0.009	247.434	0.187	-0.66
3	2	2	125	19	0.25	393.337	0.12	315.614	0.214	-3.778
3	2	4	4913	24	587	269.729	0.18	244.834	0.316	-14.141
3	4	6	8323	32	3349	139.232	0.19	114.514	0.174	-0.736
3	5	7	20,213	38	N/A	N/A	0.3	104.446	0.541	6.54

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

7.3 POMDP Model

The POMDP model extends the Q-POCHP, adding probabilities and costs. We follow the guidelines set by Sarraute et al (2012). We define a state for each possible configuration of all network machines. The configuration probabilities are based on their frequencies within a subnet in the scan.

We use deterministic vulnerability exploit actions that take control of a given machine, also resulting in a deterministic success or failure observation. 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. An exploit on a machine that has no controlled neighbor does not change the state, even if the vulnerability exists on the host.

The Nessus output contains costs for the exploits that were identified (Lai and Hsia 2007), used 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).

We reward the attacker (1000) only when terminating when one of the target machine is controlled. The agent receives no penalty when terminating at a deadend state. We add 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.

8 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 our special purpose solver described in Section 6.2, using the network based heuristic, and goal reachability analysis mechanism. The experiments are run on a machine with an Intel Core i5 - 5300U CPU at 2.3 GHz and 8 GB of RAM.

Real networks: We ran our planner over the scanned networks of two organizations. Table 2 shows the network statistics, the runtime, and the number of nodes in the resulting plan graph. The generated models exhibit a state space well beyond the ability of current POMDP solvers.

Domain	Hosts	OS	SW	Vul	$\|S\|$	Time	Nodes
Org1	35	2	50	60	2^{8750}	116	1435
Org2	95	3	10	30	2^{4275}	3270	3270

Table 2: Statistics of real organization networks: number of hosts, operating systems (OS), software (SW), vulnerabilities (Vul), POMDP states ($|S|$), planner runtime (secs), and the number of nodes in the plan graph.

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, Hsu, and Lee 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 probabilities are the maximum likelihood estimators from the data, normalized to the reduced sample in the particular instance.

Table 1 compares the expected reward of our plan graph to that of the SARSOP policy. 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. 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.

9 Conclusion and Future Work

We suggest contingent planning as a tool for modeling pentesting, supporting partial observability of various properties, such as the operating system and installed software, that can be sensed by probe actions. 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 beyond the capabilities of current POMDP solvers, and compare its expected reward to that of a POMDP over smaller networks.

In the future we intend to create smarter heuristics for ordering actions given states, to achieve better expected rewards. We also intend to construct automated methods to draw conclusions from the plan graph, such as which vulnerabilities to fix first.

Acknowledgments: We thank the reviewers for their useful comments. This work was partially supported by the Cyber Security Research Center at Ben-Gurion University of the Negev, by ISF Grant 933/13, and by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Center of Ben-Gurion University of the Negev.

References

Ågotnes, T.; Lakemeyer, G.; Löwe, B.; and Nebel, B. 2014. Planning with epistemic goals (dagstuhl seminar 14032). *Dagstuhl Reports* 4(1):83–103.

Albore, A.; Palacios, H.; and Geffner, H. 2009. 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*, 1623–1628.

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of action generation for cyber security using classical planning. In *ICAPS'05* (2005).

Bolander, T., and Andersen, M. B. 2011. Epistemic planning for single and multi-agent systems. *Journal of Applied Non-Classical Logics* 21(1):9–34.

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In Walsh, T., ed., *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, 1936–1941. AAAI Press/IJCAI.

Brafman, R. I., and Shani, G. 2016. Online belief tracking using regression for contingent planning. *Artif. Intell.* 241:131–152.

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Planning graph heuristics for belief space search. 26:35–99.

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

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. 147(1–2):35–84.

Daum, J.; Torralba, Á.; Hoffmann, J.; Haslum, P.; and Weber, I. 2016. Practical undoability checking via contingent planning. 106–114.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1(1):269–271.

Durkota, K.; Lisý, V.; Bosanský, B.; and Kiekintveld, C. 2015. 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*, 526–532.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-quality policies for the Canadian traveler's problem. In *AAAI'10*.

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *ICAPS'05* (2005), 71–80.

Hoffmann, J.; Weber, I.; and Kraft, F. M. 2012. SAP speaks PDDL: Exploiting a software-engineering model for planning in business process management. 44:587–632.

Hoffmann, J. 2015. 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.*, 364–372.

2005.

Komarnitsky, R., and Shani, G. 2014. 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.*, 2322–2329.

Kurniawati, H.; Hsu, D.; and Lee, W. 2008. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *RSS IV*.

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

Lucangeli, J.; Sarraute, C.; and Richarte, G. 2010. Attack planning in the real world. In *SecArt'10*.

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

Muise, C. J.; Belle, V.; and McIlraith, S. A. 2014. 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.*, 2322–2329.

Papadimitriou, C. H., and Yannakakis, M. 1991. Shortest paths without a map. *Theoretical Computer Science* 84:127–150.

Sarraute, C.; Buffet, O.; and Hoffmann, J. 2012. POMDPs make better hackers: Accounting for uncertainty in penetration testing. In *AAAI'12*, 1816–1824.

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

Weld, D. S.; Anderson, C.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In Mostow, J., and Rich, C., eds., *Proceedings of the 15th National Conference of the American Association for Artificial Intelligence (AAAI'98)*, 189–197.

Yarochkin, F. V.; Arkin, O.; Kydyraliev, M.; Dai, S.-Y.; Huang, Y.; and Kuo, S.-Y. 2009. Xprobe2++: Low volume remote network information gathering tool. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, 205–210. IEEE.