Constructing Plan Trees for Simulated Penetration Testing

Dorin Shmaryahu
Information Systems Engineering
Ben Gurion University
Israel

Abstract

Penetration Testing (pentesting), where network administrators automatically attack their own network to identify and fix their vulnerabilities, has recently received attention from the AI community. Smart algorithms that can identify robust and efficient attack plans can imitate human hackers better than simple protocols. Current 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 more realistic model of the problem, allowing for partial observability and non-deterministic action effects, by modeling pentesting as a partially observable contingent problem. We suggest several optimization criteria, including worst case, best case, and fault tolerance. We experiment with benchmark networks, showing contingent planning to scale up to large networks.

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.

The POMDP approach on the other hand, 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. Pinging 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 pinging 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, how to identify 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”, which the authors admit themselves is not suitable and may adversely affect the usefulness of the pentesting result (”garbage in, garbage out”). One might consider research into obtaining better distributions, e.g. by statistics from data, but this is wide open, and in any case the scalability weakness remains.

A possible simple approach to defining such probabilities is to use a uniform distribution. However, a solution to a POMDP defined using a uniform distribution can be arbitrarily bad. Consider, for example, a case where there exists a large set of configurations that are easy to penetrate, such as a variety of old, unupdated operating systems. All these configurations may be very rare in the network, yet still exist on some machines, and are hence represented in the model. Assuming a uniform distribution over possible configurations, an attacker may believe that these vulnerable configurations are as frequent as any other configuration, and may hence attempt a long sequence of exploits which will work only for these faulty configurations. In such a case, the performance of the agent measured over the uniform POMDP, may be arbitrarily far from its performance in practice.

As an intermediate model between classical planning and POMDPs, MDP models of pentesting have been suggested (Durkota *et al.* 2015; Hoffmann 2015). These somewhat simplify the issue of obtaining the probabilities, now corresponding to ”success statistics” for exploits. Yet even this data is not easy to come by in practice, and scalability may still be problematic given that solving factored MDPs is notoriously hard (a thorough empirical investigation has yet to be conducted).

In this paper we suggest another, different, intermediate model between classical planning and POMDPs. We replace the POMDP definition with partially observable contingent planning, a qualitative model where probability distributions are replaced 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. In pentesting, one is also interested in finding better attacks, i.e. in ranking the set of possible plan trees by some measurable quantity. For example, an attacker may be interested in attacks that, at the worst case, take no more than a certain amount of time. An important research question is, hence, to define possible optimization criteria for attack plan trees. Then, one must design algorithms dedicated to these optimization criteria.

We focus here on the first question — possible optimization criteria for ranking contingent plan trees. We suggest a number of such criteria, including best and worst case, budget constrained plans, and fault-tolerant planning (Domshlak 2001). We also consider deadends, which arise in pentesting as some machine configurations cannot be penetrated, leaving no opportunity to the attacker to reach its goal. We discuss how to define and compare contingent plans under such unavoidable deadends.

We demonstrate empirically that different heuristics produce different plan trees, and that these plan trees can be compared using our optimization criteria, to prefer on heuristic over another. We leave the construction of optimal and approximate contingent planners for future research.

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 pings 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.

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. Sensing actions also have preconditions, but instead of effects they reveal the value of a set of propositions. 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 for 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 $hacl(m_i, m_j, p)$ denotes that machine m_i is directly connected to machine m_j through port p.

The set A_{sense} in our pentesting model represents the set of possible queries that one machine can launch on another, directly connected machine, pinging 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 pinging certain properties. For example, the *ping-os* action:

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

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

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

```
(: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 ?s)
                       (Match ?o ?sw ?v))
    :effect (when (ExistVuln ?v ?t) (controlling ?t))
)
```

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 vuln1)
        ...
6:      (or (ExistVuln vuln1 host0) (ExistVuln vuln2 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 Plan Trees and Optimization Criteria

We now formally define solutions to a contingent planning problem. We discuss deadends that arise in pentesting, and then turn our attention to a discussion of optimization criteria.

4.1 Contingent Plan Trees

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 an observation action will have multiple children, and each outgoing edge to a child will be 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. Matters change if we impose a budget limit on the attack, in which case the dead-ends encountered depend on which decisions are taken. We define an according plan quality criterion as part of the next subsection.

4.2 Optimization Criteria for Contingent Plans

General contingent planning follows the satisfying planning criterion, that is, one seeks any solution plan tree. It is possible, though, to consider cases where one plan tree is preferable to another, and construct algorithms that seek better, or even the best possible plan tree.

When we assume that the environment is modeled as a POMDP, and we know all the probability distributions, an obvious optimization criterion is the expected discounted reward (or cost) from executing a plan tree in the environment, and can be estimated by running multiple trials and computing the average discounted reward (ADR). In this paper, however, we focus on cases where these distributions are unknown. Without the specified distributions one cannot accurately estimate expected reward. Any attempt to use a different distribution, such as a uniform distribution, which may be arbitrarily far from the true distribution, may result in quality estimation that is arbitrarily far from reality.

We hence revert to other possible optimization criteria. Perhaps the most trivial optimization criteria under unknown probability distributions is the best case scenario, or the worst case scenario. In the best case scenario we compare plan trees based on the length of the shortest branch leading to a goal state. In the worst case scenario we compare the length of the longest branch leading to a goal state, preferring plan trees with shorter worst case branches. This may be somewhat different than the naive definition of a worst case, as a complete failure is obviously worse (less desirable) than a success after a lengthy sequence of actions. In our case, as the deadends in the plan trees are unavoidable, the naive worst case — a complete failure — is identical in all plan trees. We thus choose to ignore branches ending with deadends when considering worst case analysis.

While well defined, best and worst case optimization may not be sufficiently expressive. A best case scenario is too optimistic, assuming that all attack attempts will be successful. A worst case scenario is over pessimistic, assuming that all attack attempts, but the last one, will fail. We would like to define finer optimization criteria.

Budget Optimization One possible such criterion assumes attacks on a budget — that is, the attacker is allowed only a certain predefined number of actions (or total cost) in a branch. When the budget runs out, the attacker is not allowed any additional actions, and hence, a deadend occurs. Setting a budget prior to attacking seems like a reasonable requirement from an attacker. For example, if action costs represent the time it takes for each action, the attacker may wish to restrict attention only to attacks that require less than a certain amount of time.

Now, given two plan trees that respect a given budget, we can compare them on two possible criteria — the best case scenario and the set of solved network configurations. The worst case scenario is less interesting here as it will probably be identical to the budget.

The set of network configurations where the attacker has reached the goal under the budget is now interesting, because deadends induced by the budget may well be avoidable. That is, one can choose different attack plans, that may lead to the goal faster and hence will result in less deadends. However, simply counting the number of network configurations for which the goal has been reached is undesirable under our qualitative assumptions. For example, it may well be that plan tree τ_1 solves only for a single configuration c, while another plan tree τ_2 solves for all configurations but c. Still, it may be that the (unknown) probability of c is 0.9, making τ_1 preferable to τ_2. As we do not know these probabilities, we cannot make such comparisons.

We can hence only declare plan tree τ_1 to be better than plan tree τ_2 if the set of solved configurations ofτ_1 is a strict superset of the set of solved configurations of τ_2. As contingent planners typically maintain some type of belief over the set of possible network configurations in each search node, such computations are feasible. For example, if the belief is maintained by a logic formula, as we do, then each goal leaf g has a logic formula ϕ_g defining the belief at that leaf. We can check whether

$$\bigvee_{g \in G(\tau_1)} \phi_g \models \bigvee_{g \in G(\tau_2)} \phi_g \qquad (1)$$

$$\bigvee_{g \in G(\tau_2)} \phi_g \not\models \bigvee_{g \in G(\tau_1)} \phi_g \qquad (2)$$

where $G(\tau)$ is the set of goal leaves in plan tree τ.

Fault Tolerance Optimization Another possible optimization is by extending the ideas of fault-tolerance planning to pentesting. In fault-tolerance planning (Domshlak 2001), assuming that certain actions may fail with some low probability, a solution achieves the goal under the assumption that no more than k failures will occur. The underlying assumption is that the probability of more than k failures is so small, that we can ignore it. A failure in our case can be defined in one of two ways — either that we will ping a machine for a given property (say, $OS(m_i, winxp)$) and receive a negative response. Alternatively, we may declare a failure only when we attempt an exploit, and it fails to achieve control of a machine (due to some unobserved property).

With that view in mind, we can compare solution plan trees, focusing only on branches that contain exactly k failures. As having no more than k failures is an optimistic assumption, it is reasonable to check the worst case under this optimistic assumption. That is, of the branches of the plan tree that have the lowest probability that we care about, we compare the longest branches. Looking at the best case — the shortest branch when having no more than k failures, is identical to the overall best case scenario, ignoring failures all together.

A complementing approach assumes no less than k failures at each branch. This assumption is more appropriate where the probability of failure is sufficiently large, such that the probability of completing a task without any failure is very low. In such cases, we again compare only branches with exactly k branches, and as no less than k failures is a pessimistic assumption, we compare the best case scenario — the shortest branch with exactly k failures. Again, the worst case is less interesting as it is identical to the overall worst case.

5 Research plan

We now discuss the next steps on our research agenda. We have four different designed items:

1. Improving the configuration of the network.
2. Getting real network data.
3. Finding additional heuristics.
4. Empirical validation.

5.1 Improving the Configuration of the Network

Currently we believe that we provide a stronger and more realistic model of the problem than previous approaches, that can scale up to reasonably sized networks. However, we consider changing some of the possible configuration, making the model even more realistic.

We intend to add OS families — generalizing the OS configuration under the assumption that we can not sense a specific Os, but only a general category of an Os. As we explain above, the sensing procedure for machine configuration is known as fingerprinting. In the real world this process may identify the OS family and not the specific one. For example, if a certain machine runs Windows XP SP3, the observation about the target machine OS may be Windows XP.

We can support hidden connections between machines (sensing for connections). That way we eliminate the need for planner knowledge over the hosts connection. We have doubts about hiding those connection. Making the planner work without previous knowledge will make all states and action to seem equally valuable — choosing to attack one machine is equivalent to choosing to attack another. For example, our current heuristics choose to attack a machine that is closest to the target machine, which cannot be done when network connections are unknown.

5.2 Getting real network data

We are working on getting some real data network configuration using Nmap (Network Mapper) tool to gathering all configuration data (OS, Softwares, connections and vulnerabilities) from existing network. The Nmap tool is a free and open source utility for network discovery and security auditing. Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what applications those hosts are offering, what operating systems they are running. First we need to run Nmap on a small network showing there is no risk to execute it in a larger network, and then hopefully we could run Nmap on the university network.

5.3 Identifying Useful Heuristics

Our planning algorithm is a best first contingent planner. We use a heuristic to determine which state and action to expand next.

We are interested in generating a variety of plan trees, given different heuristics. Currently, our planner supports 4 different heuristics. The first two heuristics, random and LIFO are not useful as they cannot create attack graphs for large networks. We will use them only as a baseline to show that the other heuristics are better given the optimization criteria. Our main goal is to develop new heuristics that will be scalable to larger networks and will produce better plan trees.

Random Heuristic This heuristic gives each state a random heuristic value. It preform badly and is not scalable to larger networks of more than 5 hosts. This heuristic will be our baseline for comparing plans trees in small networks.

LIFO Heuristic A LIFO heuristic gives the highest value to the last state. The last state to arrive is the next state the planner will expand. This heuristic also preforms badly and is not scalable to larger networks of more than 5 hosts.

Shortest Path Heuristic This heuristic chooses the next host to attack among the reachable hosts closest to the goal host. Attack actions for the chosen host are selected at random. This heuristic is more scalable than the previous two, handling networks with up to 16 hosts.

Shortest Path and Action Selection Heuristic We chose the next host to attack from the hosts closest to the goal host. When choosing actions, we first ping a host for its operating system, and then we ping it only for software that, combined with the observed OS, may have a vulnerability. If a possible vulnerability has been detected, we attempt an exploit, followed by a sensing action to check if control was gained over the attacked host. This simple heuristic, proves to be highly effective for this application, and we manage to produce attack graphs for networks with 80 hosts and more.

5.4 Empirical Study

We now review the experiments that we want to conduct in order to show the contribution of using contingent planning for pentesting .We want to demonstrate that the criteria we suggest can be used to differentiate between various plan trees (graphs), helping us to select a better algorithm. First, we will generate a number of networks of varying sizes using the generator of Hoffman and Steinmetz(Sarraute *et al.* ; Steinmetz *et al.*).

We experiment with a simple greedy best first contingent planner that uses a heuristic to determine which state and action to expand next. In addition, we use a mechanism for detecting repeated plan tree nodes, converting the plan tree into a plan graph. We augment this algorithm with a domain specific deadend detection mechanism, checking whether there is still a path from the attack source ("the internet") to the target host.

We will employ several domain specific heuristics (as we explained above), that leverage the network graph. Generating several different plan graphs. We can compare the runtime and scalabilty of the various heuristics.

We will run the heuristics over various network sizes. This allows us to employ the optimal solution criteria to conclude which plan is better. We can calculate the best and worst case in the fault tolerance scenario, running the planner with different values of k and compare the best (shortest) and worst (longest) path to goal. We will say that a plan graph 'A' is better than another plan graph 'B' if in a given k failures the longest path to the goal in 'A' is shorter than the longest path to goal in 'B'.

We also need to compare between the sets of initial states where the goal can be reached in each plan graph. Let $s(A)$ be the set of states for which the goal can be reached in plan graph A and $s(B)$ to be the set of such states in plan graph B. If $s(B) \subset s(A)$ we can say that plan graph A is better than plan graph B. Identifying the set of initial states for which there is a solution, and comparing sets is not a trivial problems, and we must identify efficient methods for doing that.

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.

Carmel Domshlak. Fault tolerant planning: Complexity and compilation. volume 22, pages –, 2001.

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.

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.

Marcel Steinmetz, Jörg Hoffmann, and Olivier Buffet. Revisiting goal probability analysis in probabilistic planning.