Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

POMDPs Make Better Hackers: Accounting for Uncertainty in Penetration Testing

Carlos Sarraute
Core Security & ITBA
Buenos Aires, Argentina
carlos@coresecurity.com

Olivier Buffet
INRIA
Nancy, France
buffet@loria.fr

Jörg Hoffmann
Saarland University
Saarbrücken, Germany
hoffmann@cs.uni-saarland.de

Abstract

Penetration Testing is a methodology for assessing network security, by generating and executing possible hacking attacks. Doing so automatically allows for regular and systematic testing. A key question is how to generate the attacks. This is naturally formulated as planning under uncertainty, i.e., under incomplete knowledge about the network configuration. Previous work uses classical planning, and requires costly pre-processes reducing this uncertainty by extensive application of scanning methods. By contrast, we herein model the attack planning problem in terms of partially observable Markov decision processes (POMDP). This allows to reason about the knowledge available, and to intelligently employ scanning actions as part of the attack. As one would expect, this accurate solution does not scale. We devise a method that relies on POMDPs to find good attacks on individual machines, which are then composed into an attack on the network as a whole. This decomposition exploits network structure to the extent possible, making targeted approximations (only) where needed. Evaluating this method on a suitably adapted industrial test suite, we demonstrate its effectiveness in both runtime and solution quality.

Introduction

Penetration Testing (short *pentesting*) is a methodology for assessing network security, by generating and executing possible attacks exploiting known vulnerabilities of operating systems and applications (e.g., (Arce and McGraw 2004)). Doing so automatically allows for regular and systematic testing without a prohibitive amount of human labor, and makes pentesting more accessible to non-experts. A key question is how to automatically generate the attacks.

A natural way to address this issue is as an *attack planning* problem. This is known in the AI Planning community as the "Cyber Security" domain (Boddy et al. 2005). Independently (though considerably later), the approach was put forward also by Core Security (Lucangeli, Sarraute, and Richarte 2010), a company from the pentesting industry. In that form, attack planning is very technical, addressing the low-level system configuration details that are relevant to vulnerabilities. Herein, we are concerned exclusively with this setting. We consider regular automatic pentesting as done in Core Security's "Core Insight Enterprise" tool. We will use the term "attack planning" in that sense.

Lucangeli et al. (2010) encode attack planning into PDDL, and use off-the-shelf planners. This already is useful—in fact, it is currently employed commercially in Core Insight Enterprise, using a variant of Metric-FF (Hoffmann 2003). However, the approach is limited by its inability to handle uncertainty. The pentesting tool cannot be up-to-date regarding all the details of the configuration of every machine in the network, maintained by individual users.

Core Insight Enterprise currently addresses this by extensive use of *scanning* methods as a pre-process to planning, which then considers only *exploits*, i.e., hacking actions modifying the system state. The drawbacks of this are that (a) this pre-process incurs significant costs in terms of running time and network traffic, and (b) even so, since scans are not perfect, a residual uncertainty remains (Metric-FF is run based on the configuration that appears to be most likely). Prior work (Sarraute, Richarte, and Lucangeli 2011) has addressed (b) by associating each exploit with a success probability. This is unable to model dependencies between the exploits, and it still requires extensive scanning (to obtain realistic success probabilities) so does not solve (a). Herein, we provide the first solution able to address both (a) and (b), intelligently mixing scans with exploits like a real hacker would. The basic insight is that penetration testing can be naturally modeled in terms of solving a POMDP.

We encode the incomplete knowledge as an uncertainty of state, thus modeling the possible network configurations in terms of a probability distribution. Scans and exploits are deterministic in that their outcome depends only on the state they are executed in. Negative rewards encode the cost (the duration) of scans and exploits; positive rewards encode the value of targets attained. The model incorporates firewalls, detrimental side-effects of exploits (crashing programs or entire machines), and dependencies between exploits relying on similar vulnerabilities.

POMDP solvers fail to scale to large networks. This is not surprising—even the input model grows exponentially in the number of machines. We show how to address this based on exploiting network structure. We view networks as graphs whose vertices are fully-connected subnetworks, and whose arcs encode the connections between these, filtered by firewalls. We decompose this graph into biconnected components. We approximate the attacks on these components by combining attacks on individual subnetworks. We approximate the latter by combining attacks on individual machines. The approximations are conservative, i.e., they never overestimate the value of the policy returned. Attacks on individual machines are modeled and solved as POMDPs, and the solutions are propagated back up. We evaluate this approach based on the test suite of Core Insight Enterprise, showing that, compared to a global POMDP model, it vastly improves runtime at a small cost in attack quality.

We next discuss some preliminaries. We then describe our POMDP model, our decomposition algorithm, and our experimental findings, before concluding the paper.

Preliminaries

We fill in some details on network structure and penetration testing. We give a brief background on POMDPs.

Network Structure

Networks can be viewed as directed graphs whose vertices are given by the set M of *machines*, and whose arcs are connections between pairs of $m \in M$. However, in practice, these network graphs have a particular structure. They tend to consist of *subnetworks*, i.e., clusters N of machines where every $m \in N$ is directly connected to every $m' \in N$. By contrast, not every subnetwork N is connected to every other subnetwork N', and typically, if such a connection does exist, then it is filtered by a *firewall*.

From the perspective of an attacker, the firewalls filter the connections and thus limit the attacks that can be executed when trying to hack into a subnetwork N' from another subnetwork N. On the other hand, once the hacker managed to get into a subnetwork N, access to all machines within N is easy. Thus a natural representation of the network, from an attack planning point of view, is that of a graph whose vertices are subnetworks, and whose arcs are annotated with firewalls F. We herein refer to this graph as the *logical network* LN, and we denote its arcs with $N \xrightarrow{F} N'$.

We formalize firewalls as sets of rules describing which kinds of communication (e.g., ports) are disallowed. Thus smaller sets correspond to "weaker" firewalls, and the *empty firewall* blocks no communication at all.

We remark that, in our POMDP model, we do not provide for privilege escalation, or obtaining passwords. This can instead be modeled at the level of LN. Different privilege levels on the same machine m can be encoded via different copies of m. If controlling m allows the retrieval of passwords, then m can be connected via empty firewalls to the machines m' who can be accessed by using these passwords, more precisely to high-privilege copies of these m'.

Penetration Testing

Uncertainty in pentesting arises because it is impossible to keep track of all the *configuration* details of individual machines, i.e., exactly which versions of which programs are installed etc. However, it is safe to assume that the pentesting tool knows the structure of the network, i.e., the graph LN and the filtering done by each firewall: changes to this are infrequent and can easily be registered.

The objective of pentesting is to gain control over certain machines (with critical content) in the network. At any point in time, each machine has a unique *status*. A *controlled* machine m has already been hacked into. A *reached* machine m is connected to a controlled machine, i.e., either m is in a subnetwork N one of whose machines is controlled, or m is in a subnetwork N' with a LN arc $N \xrightarrow{F} N'$ where one of the machines in N is controlled. All other machines are *not reached*. The algorithm starts with one controlled machine, denoted here by $*$.[1] We will use the following (small but real-life) situation as a running example:

Example 1 *The attacker has already hacked into a machine m', and now wishes to attack a machine m within the same subnetwork. The attacker knows 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.*

If SA fails, then it is likely that CAU will fail as well (because DEP is enabled). The attacker is then better off trying something 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 accomplish this.

Furthermore, port scans—observation actions testing whether or not a particular port is open—should be used only if one actually intends to execute a relevant exploit. Here, if we start with SA, we should scan only port 2967. We accomplish such behavior through the use of POMDPs. By contrast, to reduce uncertainty, classical planning requires a pre-process executing *all* possible scans. In this example, there are only two—ports 2967 and 6668—however in general there are many, causing significant network traffic and waiting time.

POMDPs

POMDPs are usually defined (e.g., (Monahan 1982; Kaelbling, Littman, and Cassandra 1998)) by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, T, O, r, b_0 \rangle$. If the system is in state $s \in \mathcal{S}$ (the *state space*), and the agent performs an action $a \in \mathcal{A}$ (the *action space*), then that results in (1) a transition to a state s' according to the *transition function* $T(s, a, s') = Pr(s'|s, a)$, (2) an observation $o \in \mathcal{O}$ (the *observation space*) according to the *observation function* $O(s', a, o) = Pr(o|s', a)$ and (3) a scalar *reward* $r(s, a, s')$. b_0, the *initial belief*, is a probability distribution over $\mathcal{S}$.

The agent must find a decision *policy* π choosing, at each step, the best action based on its past observations and actions so as to maximize its future gain, which we measure

[1] For simplicity, we will notate $*$ as a separate vertex in LN. If $*$ is part of a subnetwork N, this means to turn $N \setminus \{*\}$ into a separate vertex in LN, connected to $*$ via the empty firewall.

here through the total accumulated reward. The expected value of an optimal policy is denoted with V^*.

The agent typically reasons about the hidden state of the system using a *belief state* b, a probability distribution over $\mathcal{S}$. For our experiments we use SARSOP (Kurniawati, Hsu, and Lee 2008), a state of the art point-based algorithm, i.e., an algorithm approximating the value function as the upper envelope of a set of hyperplanes, corresponding to a selection of particular belief states (referred to as "points").

POMDP Model

A preliminary version of our POMDP model appeared at the SecArt'11 workshop (Sarraute, Buffet, and Hoffmann 2011). The reader may refer to that paper for a more detailed example listing complete transition and observation models for some actions, and exemplifying the evolution of belief states when applying these actions. In what follows, we keep the description brief in the interest of space.

States

Several aspects of the problem—notably the network structure and the firewall filtering rules—are known and static. POMDP variables encoding these aspects can be compiled out in a pre-process, and are not included in our model.

The states capture the status of each machine (controlled/reached/not reached). For non-controlled machines, they also specify the software configuration (operating system, servers, open ports, …). We specify the vulnerable programs, as well as programs that can provide information about these (e.g., the protection mechanism "DEP" in our running example is relevant to both exploits). The states also indicate whether a given machine or program has crashed.

Finally, we introduce one special *terminal* state into the POMDP model (of the entire network, not of individual machines). That state corresponds to giving up the attack, when for every available action (if any) the potential benefit is not worth the action's cost.

Example 2 *The states describe the attacked machine m. For simplicity, we assume that the exploits here do not risk crashing the machine (see also next sub-section). Apart from the terminal state and the state representing that m is controlled, the states specify which programs ("SA" or "CAU") are present, whether they are vulnerable, and whether "DEP" is enabled. Each application is listening on a different port, so a port is open iff the respective application is present, and we do not need to model ports separately. Thus we have a total of* 20 *states:*

```
1 terminal      3 m_none              12 m_DEP_none
2 m_controlled  4 m_CAU               13 m_DEP_CAU
                5 m_CAU_Vul           14 m_DEP_CAU_Vul
                6 m_SA                15 m_DEP_SA
                7 m_SA_CAU            16 m_DEP_SA_CAU
                8 m_SA_CAU_Vul        17 m_DEP_SA_CAU_Vul
                9 m_SA_Vul            18 m_DEP_SA_Vul
               10 m_SA_Vul_CAU        19 m_DEP_SA_Vul_CAU
               11 m_SA_Vul_CAU_Vul    20 m_DEP_SA_Vul_CAU_Vul
```

In short, the states for each machine m essentially are tuples of status values for each relevant program. Global system states then are tuples of these machines-states, with one entry for each $m \in M$. The state space enumerates these tuples. In other words, the state space is factored in a natural way, by programs and machines. An obvious option is, thus, to model and solve the problem using factored POMDPs (e.g., (Hansen and Feng 2000)). We did not try this yet; our POMDP model generator internally enumerates the states, and feeds the ground model to SARSOP.[2]

The factored nature of our problem also implies that the state space is huge. In a realistic setting, the set C of possible configuration tuples for each machine $m \in M$ is very large, yielding an enormous state space $|\mathcal{S}| = O(|C|^{|M|})$. In practice, we will run POMDPs only on single machines, i.e., $|M| = 1$.

Actions

To reach the terminal state, we need a *terminate* action indicating that one gives up on the attack.

There are two main types of actions, *scans* and *exploits*, which both have to be targeted at reachable machines. Scans can be OS detection actions or port scans. In most cases, they have no effect on the state of the target machine. Their purpose is to gain knowledge about a machine's configuration, by an observation that typically allows to prune some states from the belief (e.g., observing that the OS must be some Windows XP version). Exploits make use of a vulnerability—if present—to gain control over a machine. The outcome of the exploit is observed by the attacker, so a failed exploit may, like a scan, yield information about the configuration (e.g., that a protection mechanism is likely to be running). For a minority of exploits, a failed attempt crashes the machine.

For all actions, the outcome is deterministic: which observation is returned, and whether an exploit succeeds/fails/crashes, is uniquely determined by the target machine's configuration.

Example 3 *In our example, there are five possible actions:*

```
m_exploit_SA
m_exploit_CAU
m_scan_port_2967
m_scan_port_6668
terminate
```

The POMDP model specifies, for each state in Example 2, the outcome of each action. For example, `m_exploit_SA` *succeeds if and only if SA is present and vulnerable, and DEP is disabled. Hence, when applied to either of the states 9, 10, or 11,* `m_exploit_SA` *results in state 2, and returns the observation* `succeeded`*. Applied to any other state,* `m_exploit_SA` *leaves the state unchanged, and the observation is* `failed`*.*

The outcomes of actions also depend on what firewall (if any) stands between the pentester and the target. If the firewall filters out the relevant port, then the action is unusable: its transition model leaves the state unchanged, and no observation is returned. For example, if a firewall F filters out

[2]Note that this approach enables certain non-trivial optimizations: some of the states in Example 2 could be merged. If DEP is enabled, then it does not matter whether or not CAU/SA are vulnerable. For brevity, we do not discuss this in detail here.

port 2967, then m_scan_port_2967 and m_exploit_SA are unusable through F, but can be employed as soon as a machine behind F is under control.

Rewards

No reward is obtained when using the *terminate* action or when in the terminal state.

The instant reward of any scan/exploit action depends on the transition it induces in the present state. Our simple model is to additively decompose the instant reward $r(s, a, s')$ into $r(s, a, s') = r_e(s, a, s') + r_t(a) + r_d(a)$. Here, (i) $r_e(s, a, s')$ is the value of the attacked machine in case the transition (s, a, s') corresponds to a successful exploit, and is 0 for all other transitions; (ii) $r_t(a)$ is a cost that depends on the action's duration; and (iii) $r_d(a)$ is a cost that reflects the risk of detection when using this action. (iii) is orthogonal to the risk of crashing a program/machine, which as described we model as a possible outcome of exploits. Note that (ii) and (iii) may be correlated; however, there is no 1-to-1 correspondence between the duration and detection risk of an exploit, so it makes sense to be able to distinguish these two. Finally, note that (i) results in summing up rewards for successful exploits on different machines. That is not a limiting assumption: one can reward breaking into *[m_1 OR m_2]* by introducing a new virtual machine, accessible at no cost from each of m_1 and m_2.

Example 4 *In our example, we set $r_e = 100$ in case of success, 0 otherwise; $r_t = -10$ for all actions; and $r_d = 0$ (no risk of detection). We will see below what effect these settings have on an optimal policy.*

Since all actions are deterministic, there is no point in repeating them on the same target through the same firewall—this will not produce new effects or bring any new information. In particular, positive rewards cannot be received multiple times. Thus cyclic behaviors incur infinite negative costs. This implies that the expected reward of an optimal policy is finite even without discounting.[3]

Designing the Initial Belief

Penetration testing is done at regular time intervals. The initial belief—our knowledge of the network when we start the pentesting—depends on (a) what was known at the end of the previous pentest, and on (b) what may have changed since then. We assume for simplicity that knowledge (a) is perfect, i.e., each machine m at time 0 (the last pentest) is assigned one concrete configuration $I(m)$. We then compute the initial belief as a function $b_0(I, T)$ where T is the number of days elapsed since the last pentest. The uncertainty in this belief arises from not knowing which software updates were applied. We assume that the updates are made independently on each machine (simplifying, but reasonable given that updates are controlled by individual users).

A simple model of updates (Sarraute, Buffet, and Hoffmann 2011) encodes the uncertain evolution of each program independently, in terms of a Markov chain. The states

[3]In fact, the problem falls into the class of *Stochastic Shortest Path Problems* (Bertsekas and Tsitsiklis 1996).

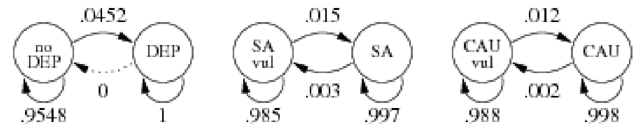


Figure 1: The three independent Markov chains used to model the update mechanism in our example network.

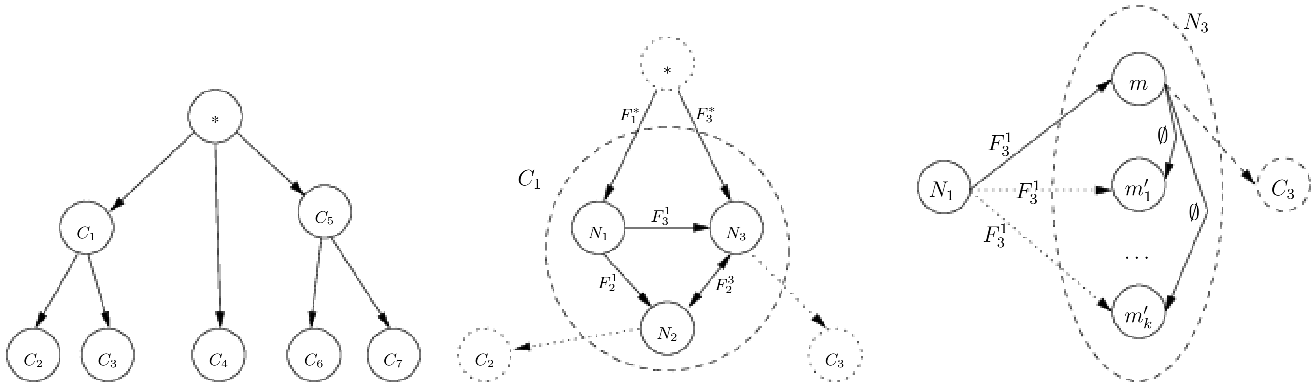
in each chain correspond to the different versions of the program, and the transitions model the possible program updates (with estimated probabilities that these updates will be made). The initial belief then is the distribution resulting from this chain after T steps.

Example 5 *In our running example, the three components in the single machine are DEP, CAU and SA. They are updated via three independent Markov chains, each with two states, as illustrated in Figure 1. The probabilities indicate how likely the machine is to transition from one state to another during one day. Say we set $T = 30$, and run the Markov chains on the configuration I in which m has DEP disabled, and both SA and CAU are vulnerable to the attacker's exploit. In the resulting initial belief $b_0(I, T)$, DEP is likely to be enabled; the weight of states 12–20 in Example 2 is high in b_0 ($> 70\%$).*

Here, we use this simple model as the basic building block in a method taking into account that version x of program A may need version y or z of program B. We assume that programs are organized in a hierarchical manner, the operating system being at the root of a directed acyclic graph, and a program having as its parents the programs it directly depends on. This yields a Dynamic Bayesian Network, where each conditional probability distribution is derived from a Markov chain $Pr(X_t = x' | X_{t-1} = x)$ filtered by a compatibility function $\delta(X = x, parent_1(X) = x_1, \ldots, parent_k(X) = x_k)$, that returns 1 iff the value of X is compatible with the parent versions, 0 otherwise. This model of updates is reasonable, but of course still not realistic; future work needs to investigate such models in detail.

We now illustrate how reasoning with the probabilities of the initial belief results in the desired intelligent behavior.

Example 6 *Say we compute the initial belief $b_0(I, T)$ as in Example 5. Since the weight of states 12–20 is high in b_0, if* m_exploit_SA *fails, then the success probability of* m_exploit_CAU *is reduced to the point of not being worth the effort anymore, and the attacker (the optimal policy) gives up, i.e., would try a different attack not prevented by DEP. Namely, consider $Pr(CAU^+|2967^+)$, i.e., the probability of* m_exploit_CAU *succeeding, after observing that port 2967 is open. This corresponds to the weight of (A) states 8 and 11 in Example 2, within the states (B) 6–11 plus 15–20. That weight (A/B) is about 20%. Thus the expected value of* m_exploit_CAU *in this situation is about $100 * 0.2$ [success reward] -10 [action cost] $= 10$, cf. Example 4, so the action is worthwhile. By contrast, say that* m_exploit_SA *has been tried and failed. Then (A) is reduced to state 8 only, while (B) still contains (in particular) all the DEP states 15–20. The latter states have a lot of weight, and thus $Pr(CAU^+|2967^+, SA^-)$ is only about 5%. Given this, the expected value of* m_exploit_CAU *is negative, and it is better to apply* terminate *instead.*



(a) LN as tree of components C. (b) Paths for attacking C_1. (c) Attacking N_3 from N_1, using m first.

Figure 2: Illustration of Levels 1, 2, and 3 (from left to right) of the 4AL algorithm.

4AL Decomposition Algorithm

As hinted, POMDPs do not scale to large networks (cf. the experiments in the next section). We now present an approach using decomposition and approximation to overcome this problem, relying on POMDPs only to attack individual machines. The approach is called *4AL* since it addresses network attack at 4 different levels of abstraction. 4AL is a POMDP solver specialized to attack planning as addressed here. Its input are the logical network LN and POMDP models encoding attacks on individual machines. Its output is a policy (an attack) for the global POMDP encoding LN, as well as an approximation of the value of the global value function. We next overview the algorithm, then fill in some technical details. To simplify the presentation, we will focus on the approximation of the value function, and outline only briefly how to construct the policy.

4AL Overview and Basic Properties

The four levels of 4AL are: (1) *Decomposing the Network*, (2) *Attacking Components*, (3) *Attacking Subnetworks*, and (4) *Attacking Individual Machines*. We outline these levels in turn before providing technical details. Figure 2 provides illustrations.

- **Level 1:** Decompose the logical network LN into a tree of biconnected components, rooted at $*$. In reverse topological order, call Level 2 on each component; propagate the outcomes upwards in the tree.

Every graph decomposes into a unique tree of biconnected components (Hopcroft and Tarjan 1973). A biconnected component is a sub-graph that remains connected when removing any one vertex. In pentesting, intuitively this means that there is more than one possibility (more than one path) to attack the subnetworks within the component, requiring to reason about the component as a whole (which is the job of Level 2). By contrast, if removing subnetwork X (e.g., N_2 in Figure 2 (b)) makes the graph fall apart into two separate sub-graphs (C_2 vs. the rest of LN, compare also Figure 2 (a)), then *all* attacks from $*$ to one of these sub-graphs (C_2 here) must first traverse X (N_2 here). Thus the overall expected value of the attack can be computed by (1) computing the value of attacking that sub-graph (C_2) alone, and (2) adding the result as a *pivoting reward* to the reward of breaking into X (N_2). In other words, we "propagate the outcomes upwards" in the tree displayed in Figure 2 (a).

It is important to note that this tree decomposition will typically result in a huge reduction of complexity. Biconnected components in LN arise only from clusters of more than 2 subnetworks sharing a common (physical) firewall machine. Such clusters tend to be small. In the real-world test scenario used by Core Security and in our experiment here, there is only one cluster, of size 3. In case there are no clusters at all, LN is a tree and 4AL Level 2 trivializes completely.

- **Level 2:** Given component C, consider, for each rewarded subnetwork $N \in C$, all paths P in C that reach N. Backwards along each P, call Level 3 on each subnetwork and associated firewall. Choose the best path for each N. Aggregate these path values over all N, by summing up but disregarding rewards that were already accounted for by a previous path in the sum.

In case a biconnected component C contains more than one subnetwork, to obtain the best attack on C, in general we have no choice but to encode the entire component as a POMDP. Since that is not feasible, Level 2 considers individual "attack paths" within C. Any single path P is equivalent to a sequence of attacks on individual subnetworks; these attacks are evaluated using Level 3. We consider the rewarded vertices N in separation, enumerating the attack paths and choosing a best one. The values of the best paths are aggregated over all N in a conservative (pessimistic) manner, by accounting for each reward at most once. A strict under-estimation occurs in case the best paths for some rewarded vertices are not disjoint: then these attacks share some of their cost, so a combined attack has a higher expected reward than the sum of independent attacks.

In Figure 2 (b), N_2 and N_3 have a pivoting reward because they allow to reach the components C_2 and C_3 respectively. If the best paths for both N_2 and N_3 go via N_1 (because the firewall F_3^* is very strict), then these paths are not disjoint, duplicating the effort for breaking into N_1.

Obviously, enumerating attack paths within C is exponential in the size of C. This is the only point in 4AL—apart of course from calls to the POMDP solver—that has worst-case exponential runtime. In practice, biconnected components

Algorithm 1: Level 1 (Decomposing the Network)

Input: LN: Logical Network.
Output: Approximation V of expected value V^* of attacking LN from controlled machine $*$.

/* Decompose LN into tree DLN of biconnected components, rooted at $*$; see text for ``clean-up''. */
$DLN \leftarrow$HopcroftTarjan(LN);
Set tree root to $*$ and *clean-up* LN and DLN;
$C_1, \ldots, C_k \leftarrow$ a topological ordering of DLN;
Intitialize pivoting reward $pr(N)$ for all $N \in LN$ to 0;
for $i = k, ..., 1$ **do**
  /* Call Level 2 to attack each component. */
  $V(C_i) \leftarrow$Level2(C_i, pr);
  /* Propagate expected reward. */
  $N \leftarrow$ the parent of C_i in LN;
  $pr(N) \leftarrow pr(N) + V(C_i)$;
return $pr(*)$

Algorithm 2: Level 2 (Attacking Components)

Input: Biconnected component C, reward function pr.
Output: Approximation V of expected value V^* of attacking C, given its parent is controlled and its pivoting rewards are pr.

$R \leftarrow 0$;
/* Account for each rewarded vertex N. */
while $\exists N \in C$ *s.t.* $r(N) > 0$ *or* $pr(N) > 0$ **do**
  $P \leftarrow \langle\rangle$; $R(P) \leftarrow 0$; $P(N) \leftarrow P$;
  /* Maximize over all simple paths (no repeated vertices) from an entry vertex to N. */
  foreach *simple path* P *of the form* $\xrightarrow{F_0} N_1 \xrightarrow{F_1} N_2 \ldots \xrightarrow{F_{k-1}} N_k = N$ *where* $N_1, \ldots, N_k \in C$ *and* $N_1 \in C_*$ **do**
    /* Propagate rewards along P, calling Level 3 for attack on each subnetwork. */
    $R(P) \leftarrow 0$;
    for $i = k, ..., 1$ **do**
      $R(P) \leftarrow$ Level3$(N_i, F_{i-1}, pr(N_i), R(P))$;
    $P(N) \leftarrow$ arg max$(R(P(N)), R(P))$;
  $R \leftarrow R + R(P(N))$;
  $r(N_i), pr(N_i) \leftarrow 0$ for all vertices N_i on $P(N)$;
return R

Algorithm 3: Level 3 (Attacking Subnetworks)

Input: Firewall F, subnetwork N, rewards pR, $pathR$.
Output: Approximation V of expected value V^* of attacking N through F, given F is reached, N's pivoting reward is pR, and the path reward behind N is $pathR$.

$R \leftarrow 0$;
/* Maximize over reward obtained when hacking first into a particular machine $m \in N$. */
foreach $m \in N$ **do**
  $R(m) \leftarrow r(m)$;
  /* After breaking m, we can pivot behind N, and reach all $m \neq m' \in N$ without F. */
  $R(m) \leftarrow R(m) + pR + pathR$;
  foreach $m \neq m' \in N$ **do**
    $R(m) \leftarrow R(m)$+Level4$(m', \emptyset, r(m'))$;
  $R \leftarrow \max(R,$ Level4$(m, F, R(m)))$;
return R

Algorithm 4: Level 4 (Attacking Individual Machines)

Input: Firewall F, machine m, reward R.
Output: Approximation V of expected value V^* of attacking m through F, given m is reached and the current reward of breaking it is R.

if (m, F, R) *is cached* **then**
  return $V(m, F, R)$
$M \leftarrow$createPOMDP(m, F, R);
$V \leftarrow$solvePOMDP(M);
Cache (m, F, R) with V;
return V

Figure 3: 4AL algorithm, pseudo-code.

are typically small, cf. the above.

- **Level 3:** Given a subnetwork N and a firewall F through which to attack N, for each machine $m \in N$ approximate the reward obtained when attacking m first. For this, modify m's reward to take into account that, after breaking m, we are behind F: call Level 4 to obtain the values of all $m' \neq m$ with an empty firewall; then add these values, plus any pivoting reward, to the reward of m and call Level 4 on this modified m with firewall F. Maximize the resulting value over all $m \in N$.

Consider Figure 2 (c). When attacking N (here, N_3) from some machine behind the firewall F (here, F_3^1), we have to choose which machine inside N to attack. Given we commit to one such choice m, the attack problem becomes that of breaking into m and afterwards exploiting the direct connection to any $m \neq m' \in N$, and any descendant network (here, C_3) we can now pivot to. As described, that can be dealt with by combining attacks on individual machines with modified rewards. (The pivoting reward for descendant networks is computed beforehand by Levels 1 and 2.)

Like Level 2, Level 3 makes a conservative approximation. It fixes a choice of which $m \in N$ to attack. By contrast, the best strategy may be to switch between different $m \in N$ depending on the success of the attack so far. For example, if one exploit is very likely to succeed, then it may pay off to try this on all m first, before trying anything else.

- **Level 4:** Given a machine m and a firewall F, model the single-machine attack planning problem as a POMDP, and run an off-the-shelf POMDP solver. Cache known results to avoid duplicate effort.

This last step should be self-explanatory. The POMDP model is created as described earlier. Note that Level 3 may, during the execution of 4AL, call the same machine with the same firewall more than once. For example, in Figure 2 (c), when we switch to attacking m'_1 instead of m, the call of Level 4 with m'_k and an empty firewall is repeated.

Summing up, 4AL has low-order polynomial runtime except for the enumeration of paths within biconnected components (Level 2), and solving single-machine POMDPs (Level 4). The decomposition at Level 1 incurs no information loss. Levels 2 and 3 make conservative approximations, so, if the POMDP solutions are conservative (e.g., optimal), then the overall outcome of 4AL is conservative as well.

Technicalities

To provide a more detailed understanding of 4AL, we now discuss pseudo-code for the algorithm, provided in Figure 3. Consider first Algorithm 1. It should be clear how the overall structure of the algorithm corresponds to our previous discussion. It calls the linear-time algorithm by Hopcroft and Tarjan (1973) (hereafter, HT) to find the decomposition. The loop $i = k, \dots, 1$ processes the components in reverse topological order. The pivoting reward function pr encodes the propagation of rewards upwards in the tree; this should be self-explanatory apart for the expression "the parent" of C_i in LN. The latter relies on the fact that, after "clean-up" (line 2), each component has exactly one such parent.

To explain the clean-up, note first that HT works on undirected graphs; when applying it, we ignore the direction of the arcs in LN. The outcome is an undirected tree of biconnected components, where the *cut vertices*—those vertices removing which makes the graph break apart—are shared between several components. In Figure 2 (b), e.g., N_2 prior to the clean-up belongs to both, C_1 and C_2. The clean-up sets the root of the tree to $*$, and assigns each cut-vertex to the component closest to $*$ (e.g., N_2 is assigned to C_1); $*$ itself is turned into a separate component. Re-introducing the direction of arcs in LN, we then prune vertices not reachable from $*$. Next, we remove arcs that cannot participate in any non-redundant attack path starting in $*$. Since moving *towards* $*$ in the decomposition tree necessarily leads any attack back to a vertex it has visited (broken into) already, after such removal the arcs between components form a directed tree as in Figure 2 (a). Each non-root component C_i (e.g., C_3) has exactly one parent component C in the cleaned-up tree (e.g., C_1). The respective subnetwork $N \in C$ (e.g., N_3) is a cut vertex in LN. Thus, as claimed above, N is the *only* vertex, in LN, that connects into C_i.

Obviously, all attacks on C_i must pass through its parent N. Further, the vertices and arcs removed by clean-up cannot be part of an optimal attack. Thus Level 1 is loss-free. To state this—and the other properties of 4AL—formally, we need some notations. We will use V^* to denote the real (optimal) expected value of an attack, and V to denote the 4AL approximation. The attacked object is given as the argument. For example, $V^*(LN)$ is the expected value of attacking LN; $V(C, pr)$ is the outcome of running 4AL Level 2 on component C and pivoting reward function pr.

Proposition 1 *Let LN be a logical network. Say that, for all calls to 4AL Level 2 made by 4AL Level 1 when run on LN, we have $V(C, pr) = V^*(C, pr)$. Then $V(LN) = V^*(LN)$. If $V(C, pr) \leq V^*(C, pr)$ for all calls to 4AL Level 2, then $V(LN) \leq V^*(LN)$.*

Consider now Algorithm 2. Our previous description was imprecise in omitting the additional algorithm argument pr. This integrates with the algorithm by being passed on, for every subnetwork on the paths we consider (line 7), to Algorithm 3 which adds it to the reward obtained for hacking into that subnetwork (Algorithm 3 line 4).

R aggregates the values (lines 1, 9), over all rewarded subnetworks N. This aggregation is made conservative by removing all rewards—pivoting rewards as well as the own rewards of the individual machines involved—that have already been accounted for (line 10). Regarding the individual machines, Algorithm 2 uses the shorthands (a) $r(N) > 0$ (line 2) and (b) $r(N) \leftarrow 0$ (line 10); (a) means that there exists $m \in N$ so that $r(m) > 0$; (b) means that $r(m) \leftarrow 0$ for all $m \in N$. Regarding pivoting rewards, note that line 10 of Algorithm 2 modifies the function pr maintained by Algorithm 1. This does not lead to conflicts because, at the time when Algorithm 1 calls Algorithm 2 on component C, all descendants of C in LN have already been processed, and thus in particular Algorithm 1 will make no further updates to the value of $pr(N)$, for any $N \in C$.

By C_* (line 4) we denote the set $\{N \in C \mid \exists N' \in LN, N' \notin C : (N', N) \in LN\}$ of subnetworks that serve as an entry into C (e.g., N_1 and N_3 for C_1 in Figure 2 (b)). Note in line 4 that the path P starts with a firewall F_0. To understand this, consider the situation addressed. The algorithm assumes that the parent N of C ($*$, for component C_1 in Figure 2 (b)) is under control. But then, to break into C, we still need to traverse an arc from N into C. F_0 is the firewall on the arc chosen by P (F^*_1 or F^*_3 in Figure 2 (b)).

The calls to Level 3 (line 7) comprise the network N_i to be hacked into, the firewall F_{i-1} that must be traversed for doing so, the pivoting reward of N_i, as well as the ongoing path reward $R(P)$ which gets propagated backwards along the path. Clearly, this is equivalent to the sequence of attacks required to execute P, and harvesting all pivoting rewards associated with such an attack. Thus, with the conservativeness of the aggregation across the subnetworks N, we get:

Proposition 2 *Let C be a biconnected component, and let pr be a pivoting reward function. Say that, for all calls to 4AL Level 3 made by 4AL Level 2 when run on (C, pr), we have $V(F, N, pR, pathR) \leq V^*(F, N, pR, pathR)$. Then $V(C, pr) \leq V^*(C, pr)$.*

Algorithms 3 and 4 should be self-explanatory, given our previous discussion. Just note that the pivoting reward pR is represented by the arc from m to C_3 in Figure 2 (c), which is accounted for by simply adding it to the value of m (Algorithm 3 line 4). The path reward $pathR$ (not illustrated in Figure 2 (c)) is also added to the value of m (Algorithm 3 line 4). Max'ing over attacks on the individual machines m is, obviously, a conservative approximation because attack strategies are free to choose m. Thus:

Proposition 3 *Let F be a firewall, let N be a subnetwork, let pR be a pivoting reward, and let $pathR$ be a path reward. Say that, for all calls to 4AL Level 4 made by 4AL Level 3 when run on $(F, N, pR, pathR)$, we have*

$V(F, m, R) \leq V^*(F, m, R)$. *Then* $V(F, N, pR, pathR) \leq V^*(F, N, pR, pathR)$.

Policy Construction

At Level 1, the global policy is constructed from the Level 2 policies simply by following the tree decomposition: starting at the tree root, we execute the Level 2 policies for all reached components (in any order); once a hack into a component succeeds, the respective children components become reached. At Level 2, i.e., within a bi-connected component C, the policy corresponds to the set of paths P considered by Algorithm 2. Each P is processed in turn. For each node N in P (until failure to enter that subnetwork), we call the corresponding Level 3 policy.

At Level 3, i.e., considering a single subnetwork N, our policy simply attacks the machine $m \in N$ that yielded the maximum in Algorithm 3. The policy first attacks m through the firewall, using the respective Level 4 policy. In case the attack succeeds, the policy attacks the remaining machines $m' \in N$ in any order (i.e., for each m', we perform the associated Level 4 policy until termination). At Level 4, the policy is the POMDP policy returned by our POMDP solver.

Experiments

We evaluated 4AL against the "global" POMDP model, encoding the entire attack problem into a single POMDP. The experiments are run on a machine with an Intel Core2 Duo CPU at 2.2 GHz and 3 GB of RAM. The 4AL algorithm is implemented in Python. To solve and evaluate the POMDPs generated by Level 4, we use the APPL toolkit.[4]

Test Scenario

Our test scenario is based on the network structure shown in Figure 5. The attack begins from the Internet ($*$ is the cloud in the top left corner). The network consists of three areas—*Exposed, Sensitive, User*—separated by firewalls. Internally, each of Exposed and Sensitive is fully connected (i.e., these areas are subnetworks), whereas User consists of a tree of subnetworks separated by empty firewalls. Only two machines are rewarded, one in Sensitive (reward 9000) and one in a leaf subnetwork of User (reward 5000). The cost of port scans and exploits is 10, the cost of OS detection is 50. We allow to scale the number of machines $|M|$ by distributing, of every 40 machines, the first one to Exposed, the second one to Sensitive, and the remaining 38 to User. The exploits are taken from Core Security's database. The number of exploits $|E|$ is scaled by distributing these over 13 templates, and assigning to each machine m one such template as $I(m)$ (the known configuration at the time of the last pentest). The initial belief $b_0(I, T)$, where T is the time elapsed since the last pentest, is then generated as outlined.

The fixed parameters here (rewards, action costs, distribution of machines over areas, number of templates) are estimated based on practical experiences at Core Security. The network structure and exploits are realistic, and are used for industrial testing in that company. The main weakness of

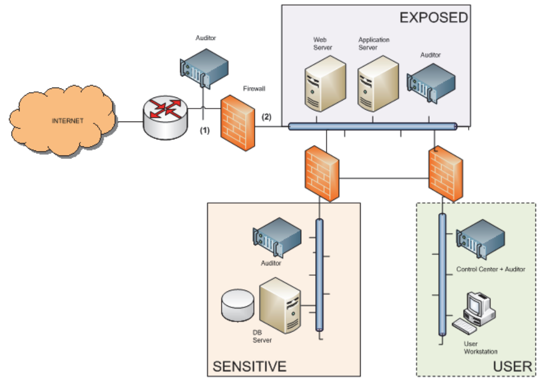


Figure 5: Network structure in our test suite.

the scenario lies in the approximation of software updates underlying $b_0(I, T)$. Altogether, the scenario is still simplified, but is natural and does approach the complexity of real-world penetration testing.

For lack of space, in what follows we scale only $|M|$ and $|E|$, fixing $|T| = 50$. The latter is realistic but challenging: pentesting is typically performed about once a month; smaller T are easier to solve as there is less uncertainty.

Approximation Loss

Figure 4 (a) shows the relative loss of quality when running 4AL instead of a global POMDP solution, for values of $|E|$ and $|M|$ where the latter is feasible. We show $quality(global\text{-}POMDP) - quality(4AL)$ in percent of $quality(global\text{-}POMDP)$. Policy quality here is estimated by running 2000 simulations. That measurement incurs a variance, which is almost stronger than the very small quality advantage of the global POMDP solution. The maximal loss for any combination of $|E|$ and $|M|$ is 14.1% (at $|E| = 7$, $|M| = 6$), the average loss over all combinations is 1.96%. The average loss grows monotonically over $|M|$, from -1.14% for $|M| = 1$ to 4.37% for $|M| = 6$. Over $|E|$, the behavior is less regular; the maximum average loss, 5.4%, is obtained when fixing $|E| = 5$.

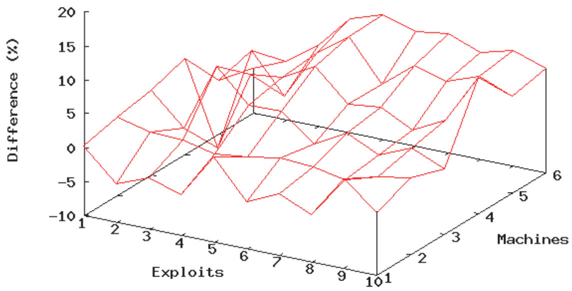
Scaling Up

Figure 4 (b) shows the runtime of 4AL when scaling up to much larger values of $|E|$ and $|M|$. The scaling behavior over $|M|$ clearly reflects the fact that 4AL is polynomial in that parameter, except for the size of biconnected components (which is 3 here). Scaling E yields more challenging single-machine POMDPs, resulting in a sometimes steep growth of runtime. However, even with $|M|$ and $|E|$ both around 100, which is a realistic size in practice, the runtime is always below 37 seconds.

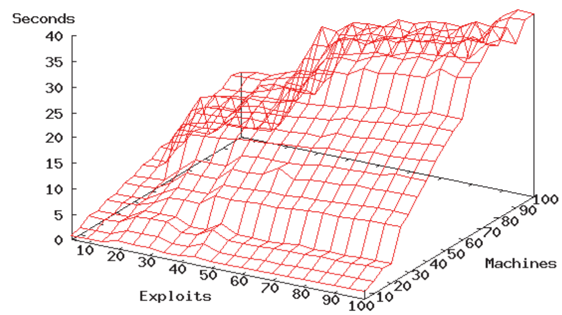
Conclusion

We have devised a POMDP model of penetration testing that allows to naturally represent many of the features of this application, in particular incomplete knowledge about the network configuration, as well as dependencies between different attack possibilities, and firewalls. Unlike any previous methods, the approach is able to intelligently mix scans with

[4]APPL 0.93 at http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/



(a) Attack quality comparison.



(b) Runtime of 4AL.

Figure 4: Empirical results for 4AL compared to a global POMDP model.

exploits. While this accurate solution does not scale, large networks can be tackled by a decomposition algorithm. Our present empirical results suggest that this is accomplished at a small loss in quality relative to a global POMDP solution.

An important open question is to what extent our POMDP+decomposition approach is more cost-effective than the classical planning solution currently employed by Core Security. Our next step will be to answer this question experimentally, comparing the attack quality of 4AL against that of the policy that runs extensive scans and then attaches FF's plan for the most probable configuration.

4AL is a domain-specific algorithm and, as such, does not contribute to the solution of POMDPs in general. At a high level of abstraction, its idea can be understood as imposing a template on the policy constructed, thus restricting the space of policies explored (and employing special-purpose algorithms within each part of the template). In this, the approach is somewhat similar to known POMDP decomposition approaches (e.g., (Pineau, Gordon, and Thrun 2003; Müller and Biundo 2011)). It remains to be seen whether this connection can turn out fruitful for either future work on attack planning, or POMDP solving more generally.

The main directions for future work are to devise more accurate models of software updates (hence obtaining more realistic designs of the initial belief); to tailor POMDP solvers to this particular kind of problem, which has certain special features, in particular the absence of non-deterministic actions and that some of the uncertain parts of the state (e.g. the operating systems) are static; and to drive the industrial application of this technology. We hope that these will inspire other researchers as well.

Acknowledgments. Work performed while Jörg Hoffmann was employed by INRIA, Nancy, France.

References

Arce, I., and McGraw, G. 2004. Why attacking systems is a good idea. *IEEE Computer Society - Security & Privacy Magazine* 2(4).

Bertsekas, D., and Tsitsiklis, J. 1996. *Neurodynamic Programming*. Athena Scientific.

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

Hansen, E., and Feng, Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS'00)*.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Hopcroft, J., and Tarjan, R. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM* 16:372–378.

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

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

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

Monahan, G. 1982. A survey of partially observable Markov decision processes. *Management Science* 28:1–16.

Müller, F., and Biundo, S. 2011. HTN-style planning in relational POMDPs using first-order FSCs. In *Proceedings of the 34th German Conference on AI (KI'11)*, 216–227.

Pineau, J.; Gordon, G.; and Thrun, S. 2003. Policy-contingent abstraction for robust robot control. In *Proceedings of the 19th Conference on Uncertainty in Articifical Intelligence (UAI'03)*, 477–484.

Sarraute, C.; Buffet, O.; and Hoffmann, J. 2011. Penetration testing == POMDP solving? In *Proceedings of the 3rd Workshop on Intelligent Security (SecArt'11)*.

Sarraute, C.; Richarte, G.; and Lucangeli, J. 2011. An algorithm to find optimal attack paths in nondeterministic scenarios. In *ACM Workshop on Artificial Intelligence and Security (AISec'11)*.