Minimization and Reliability Analyses of Attack Graphs

Somesh Jha[1] Oleg Sheyner[2] Jeannette M. Wing[2]

February 2002

CMU-CS-02-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Parts of this report will appear in our paper accepted by the IEEE Symposium on Security and Privacy, Oakland, May 2002; and parts are in a paper submitted to the Computer Security Foundations Workshop, Nova Scotia, June 2002.

[1] Computer Sciences Department, University of Wisconsin, Madison, WI 53706. E-mail: jha@cs.wisc.edu

[2] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. E-mails: {sheyner,wing}@cs.cmu.edu

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOD, ARO, or the U.S. Government.

Keywords: attack graph, model checking, minimization analysis, reliability analysis, Markov Decision Processes, network vulnerability, security

Abstract

An attack graph is a succinct representation of all paths through a system that end in a state where an intruder has successfully achieved his goal. Today Red Teams determine the vulnerability of networked systems by drawing gigantic attack graphs by hand. Constructing attack graphs by hand is tedious, error-prone, and impractical for large systems. By viewing an attack as a violation of a safety property, we can use model checking to produce attack graphs automatically: a successful path from the intruder's viewpoint is a counterexample produced by the model checker. In this paper we present an algorithm for generating attack graphs using model checking.

Security analysts use attack graphs for detection, defense, and forensics. In this paper we present a minimization technique that allows analysts to decide which minimal set of security measures would guarantee the safety of the system. We provide a formal characterization of this problem: we prove that it is polynomially equivalent to the minimum hitting set problem and we present a greedy algorithm with provable bounds. We also present a reliability technique that allows analysts to perform a simple cost-benefit analysis depending on the likelihoods of attacks. By interpreting attack graphs as Markov Decision Processes we can use a standard MDP value iteration algorithm to compute the probabilities of intruder success for each attack the graph.

We illustrate our work in the context of a small example that includes models of a firewall and an intrusion detection system.

important to automate. When evaluating the security of a network, it is not enough to consider the presence or absence of isolated vulnerabilities. A large network builds upon multiple platforms and diverse software packages and supports several modes of connectivity. Inevitably, such a network will contain security holes that have escaped notice of even the most diligent system administrator.

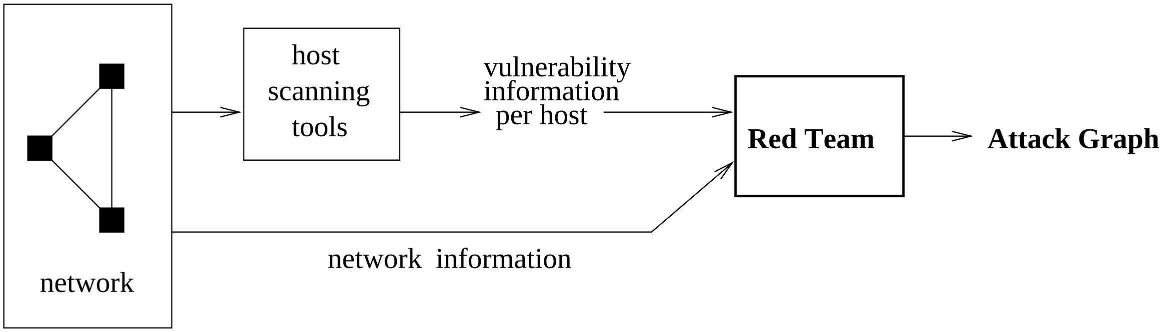

Figure 1: Vulnerability Analysis of a Network

To evaluate the vulnerability of a network of hosts, a security analyst must take into account the effects of interactions of local vulnerabilities and find global vulnerabilities introduced by interconnections. A typical process for vulnerability analysis of a network is shown in Figure 1. First, scanning tools determine vulnerabilities of individual hosts. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, the analyst produces an *attack graph*. Each path in an attack graph is a series of exploits, which we call *atomic attacks*, that leads to an undesirable state (e.g., a state where an intruder has obtained administrative access to a critical host).

1.1 Attack Graphs and Intrusion Detection

Attack graphs can serve as a basis for detection, defense, and forensic analysis. To motivate our study of the generation and analysis of attack graphs, we discuss the potential applications of attack graphs to these areas of security.

Detection
System administrators are increasingly deploying intrusion detections systems (IDSs) to detect and combat attacks on their network. Such systems depend on software sensor modules that first detect suspicious events and activity and then issue alerts. Setting up the sensors involves a trade-off between sensitivity to intrusions and the rate of false alarms in the alert stream. When the sensors are set to report all suspicious events, the sensors frequently issue alerts for benign background events. Frequent false alarms results in administrators turning off the IDS entirely. On the other hand, decreasing sensor sensitivity reduces their ability to detect real attacks.

To address this trade-off, many intrusion detection systems employ heuristic algorithms to correlate alerts from a large pool of heterogeneous sensors. Valdes and Skinner [VS01] describe a probabilistic approach to alert correlation. Successful correlation of multiple alerts increases the chance that the suspicious activity indicated by the alerts is in fact malicious.

Attack graphs can enhance both heuristic and probabilistic correlation approaches. Given a graph describing all likely attacks (i.e., sequences of attacker actions), an IDS can match individual alerts to attack edges in the graph. Matching successive alerts to individual paths in the attack graphs dramatically increases the likelihood that the network is under attack. This on-line vigilance allows the IDS to predict attacker goals, aggregate alarms to reduce the volume of alert information to be analyzed, and reduce the false alarm rates. Knowledge of attacker goals and likely next steps helps guide defensive response.

our models are expressive enough to reflect the administrator's choice of security policy for an IDS and his choice of network configuration. Attack graphs enable an administrator to perform several kinds of analyses to assess their security needs: marking the paths in the attack graph that an IDS will detect; determining where to position new IDS components for best coverage; exploring trade-offs between different security policies and between different software/hardware configurations; and identifying the worst-case scenarios and prioritizing defense strategy accordingly.

Forensics
After a break-in, forensic analysis is used to find probable attacker actions and to assess damage. If legal action is desired, analysts seek evidence that a sequence of sensor alerts comprises a coherent attack plan, and is not merely a series of isolated, benign events. This task becomes even harder when the intruders obfuscate attack steps by slowing down the pace of the attack and varying specific steps. We can construct a convincing argument as to the malicious intent of intruder actions by matching data extracted from IDS logs to a formal reference model based on attack graphs [Ste].

Given that attack graphs can be used to perform a variety of analysis, we can use them to answer the following kinds of questions, of particular interest to system administrators:

Question 1: What successful attacks are undetected by the IDS?

Question 2: If all measures for protecting a network are deployed, does the system become safe?

Question 3: Given a set of measures M, what is the smallest subset of measures M' whose deployment makes the system safe?

Answers to these questions, can help a system or network administrator choose the best upgrade strategy. We address these questions in Section 5.

When we are modeling a system operating in an unpredictable environment, certain transitions in the model represent the system's reaction to changes in the environment. We can think of such transitions as being outside of the system's control—they occur when triggered by the environment. When no empirical information is available about the relative likelihood of such environment-driven transitions, we can model them only as nondeterministic "choices" made by the environment. Moreover, for new vulnerabilities data for estimating likelihoods might not be available. However, sometimes empirical data make it possible to assign probabilities to environment-driven transitions. We would like to take advantage of such quantitative information added appropriately to attack graphs. In this context, a system administrator might be interested in answering the following question:

Question 4: The deployment of which security measure(s) will increase the likelihood of thwarting an attacker?

The system administrator can use the answer to question 4 to perform a quantitative evaluation of various security fixes. We address this question in Section 6.2.

1.2 Our Contributions

Constructing attack graphs is a crucial part of performing vulnerability analysis of a network of hosts. Currently, *Red Teams* produce attack graphs by hand, often drawing gigantic diagrams on floor-to-ceiling whiteboards. Doing this by hand is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes.

The main contributions of our work, some of which have appeared in an earlier paper [SHJ+02] are:

- We demonstrate how model checking can be applied to generate attack graphs automatically. We show that the attack graphs produced by our method are *exhaustive*, i.e., covering all possible attacks, and *succinct*, i.e., containing only relevant states and transitions (see Section 3.2).

model represents the state of the system between atomic attacks. A typical transition from state s_1 to state s_2 corresponds to an atomic attack whose preconditions are satisfied in s_1 and whose effects hold in state s_2. An *attack* is a sequence of state transitions culminating in the intruder achieving his goal. The entire attack graph is thus a representation of all the possible ways in which the intruder can succeed.

- We prove that finding a *minimum* set of atomic attacks that must be removed to thwart an intruder is NP-complete. Beyond the proof sketched in our earlier paper [SHJ$^+$02], here we further explore the complexity of this problem. Section 5.2.1 proves that the problem is polynomially equivalent to the minimum hitting set problem where the collection of sets is represented as a labeled directed graph. This reduction provided us with additional insight, enabling us to find a greedy algorithm with provable bounds, which can be used to answer questions 1, 2, and 3.

- We present an algorithm to compute the *reliability*—fined as the likelihood of an intruder not succeeding– of a networked system. An advantage of our algorithm is that it allows *incomplete information*, i.e., probabilities of all transitions need not be provided. To our knowledge, previous metrics in the area of security require complete information. We can use this algorithm an answer question 4 precisely.

We present related work in Section 2. Section 3 describes our model and our algorithm to generate attack graphs. We give details of an example networked system in Section 4 and use it throughout the paper for illustrative purposes. In Section 5 we present a minimization analysis to help administrators decide what measures to deploy to thwart attacks. In Section 6 we present a reliability analysis over *probabilistic attack graphs* based on the value iteration algorithm defined for Markov Decision Processes; this analysis can help administrators determine how deployment of one measure can decrease the likelihood of certain attacks. Finally, we present a brief summary and directions for future work in Section 7.

2 Related Work

Phillips and Swiler [PS98] propose a concept of attack graphs similar to the one we describe. However, they model only attacks. Since we have a generic state machine model, we can simultaneously model not just attacks, but also seemingly benign system events (e.g., link failures and user errors) and even system administrator recovery actions. Therefore, our attack graphs are more general than the one proposed by Phillips and Swiler. They also built a tool for generating attack graphs [SPEC00]; it constructs the attack graph by forward exploration starting from the initial state. In our work, we use a symbolic model checker (i.e., NuSMV) that works backward from the goal state to construct the attack graph. A major advantage of the backward algorithm is that vulnerabilities that are not relevant to the safety property (or the goal of the intruder) are never explored; this technique can result in significant savings in space. In fact, Swiler *et al.* [SPEC00] refer to the advantages of the backward search in their paper. Finally, the post-facto analysis suggested by Phillips and Swiler is also different from the ones we present in this paper. We plan to incorporate their analysis into our tool suite.

Dacier [Dac94] proposes the concept of privilege graphs, where each node represents a set of privileges owned by the user and arcs represent vulnerabilities. Privilege graphs are then explored to construct attack state graphs, which represent different ways in which an intruder can reach a certain goal, such as root access on a host. Dacier proposes a metric, called the *mean effort to failure* or METF, based on the attack state graphs. Orlato *et al.* [ODK99] describe an experimental evaluation of this framework. At the surface our notion of attack graphs seems similar to Dacier's. However, as in the case with Phillips and Swiler, Dacier takes an "attack-centric" view of the world; again, our attack graphs are more general. From the experiments conducted by Orlato *et al.* it appears that even for small examples the space required to construct attack state graphs becomes prohibitive. Model checking has made significant advances in representing large state spaces. Therefore, by basing our algorithm on model checking we leverage off those advances and can hope to represent large attack graphs. The analytical analysis proposed by Dacier can also be performed on

Ritchey and Ammann [RA01] also used model checking for vulnerability analysis of networks. They used the unmodified model checker SMV [SMV]. Therefore, they could only obtain one counter-example or one attack corresponding to a intruder's goal. In contrast, we modified the model checker NuSMV to produce complete attack graphs, which represents all possible attacks. We also described analyses that can be performed on these attack graphs. These analyses cannot be meaningfully performed on single attacks.

3 Generating Attack Graphs using Model Checking

First, we formally define *attack graphs*, the data structure used to represent all possible attacks on our networked system. We restrict our attention to attack graphs representing violations of safety properties[1].

Definition 1 Let *AP* be a set of atomic propositions. An *attack graph* or *AG* is a tuple $G = (S, \tau, S_0, S_s, L)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \to 2^{AP}$ is a labeling of states with a set of propositions true in that state.

Unless stated otherwise, we assume that the transition relation τ is total. We define an *execution fragment* as a finite sequence of states $s_0 s_1 \ldots s_n$ such that $(s_i, s_{i+1}) \in \tau$ for all $0 \leq i < n$. An execution fragment with $s_0 \in S_0$ is an *execution*, and an execution whose final state is in S_s is an *attack*, i.e., the execution corresponds to a sequence of atomic attacks leading to the intruder's goal state. Intuitively, S_s denotes all states where the intruder has achieved his goal, e.g., obtaining root access on a critical host.

Next we turn our attention to algorithms for automatic generation of attack graphs and properties that we can guarantee of them. Starting with a description of a network model M and a security property p, the task is to construct an attack graph representing all executions of M that violate p—these are the successful attacks. For the kinds of attack graph analyses suggested in Section 1, it is essential that the graphs produced by the algorithms be *exhaustive* and *succinct*. An attack graph is exhaustive with respect to a model M and correctness property p if it covers all possible attacks in M leading to a violation of p, and succinct if it only contains those states and transitions of M that lead to a state violating p.

3.1 Reachability Analysis

If we restrict ourselves to safety properties, an attack graph may be constructed by performing a simple state-space search. Starting with the initial states of the model M, we use a graph traversal procedure (e.g., depth first search) to find all reachable *success* states where the safety property p is violated. The attack graph is the union of all paths from initial states to success states.

While this algorithm has the advantage of simplicity, it handles only safety properties and may run into the state explosion problem for non-trivial models. Model checking has dealt with both of these issues with some success, so we will consider algorithms based on that technology.

3.2 Model Checking Algorithm

Model checking is a technique for checking whether a formal model M of a system satisfies a given property p. In our work, we use the model checker NuSMV [NuS], for which the model M is a finite labeled transition system and p is a property expressed in *Computation Tree Logic* (*CTL*). For now, we consider only safety properties, which in CTL have the form $\mathbf{AG}f$ (i.e., $p = \mathbf{AG}f$, where f is a formula in propositional logic). If the model M satisfies the property p, NuSMV reports "true." If M does not satisfy p, NuSMV produces a *counter-example*. A single counter-example shows an execution that leads to a violation of the property. In this section, we explain how to construct attack graphs for safety properties using model checking.

[1]We say more on liveness properties in Section 7.

$S_0 \subseteq S$ – set of initial states
$L : S \rightarrow 2^{AP}$ – labeling of states with propositional formulas
$p = \mathbf{AG}(\neg unsafe)$ (a safety property)

Output:

attack graph $G_p = (S_{unsafe}, R^p, S_0^p, S_f^p, L)$

Algorithm: $GenerateAttackGraph(S, R, S_0, L, p)$

(* Use model checking to find the set of states S_{unsafe} that violate the safety property $\mathbf{AG}(\neg unsafe)$. *)

$S_{unsafe} \;=\; modelCheck(S, R, S_0, L, p).$

(* Restrict the transition relation R to states in the set S_{unsafe} *)

$R^p \;=\; R \cap (S_{unsafe} \times S_{unsafe}).$
$S_0^p \;=\; S_0 \cap S_{unsafe}.$
$S_f^p \;=\; \{s | s \in S_{unsafe} \wedge s \models unsafe\}.$
$return(S_{unsafe}, R^p, S_0^p, S_f^p, L).$

Figure 2: Algorithm for Generating Attack Graphs

Attack graphs depict ways in which the system can reach an unsafe state (or, equivalently, a successful state for the intruder). We can express the property that an unsafe state cannot be reached as:

$$\mathbf{AG}(\neg unsafe)$$

When this property is false, there are unsafe states that are reachable from the initial state. The precise meaning of *unsafe* depends on the application. For example, in the network security example given in Section 4, the property given below is used to express that the privilege level of the intruder on the host with index 2 should always be less than the root (administrative) privilege.

$$\mathbf{AG}(\textit{network.adversary.privilege[2]} < \textit{network.priv.root})$$

We briefly describe the algorithm (see Figure 2) for constructing attack graphs for the property $\mathbf{AG}(\neg unsafe)$. The first step is to determine the set of states S_r that are reachable from the initial state. Next, the algorithm computes the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using an iterative algorithm derived from a fix-point characterization of the **AG** operator [CGP00]. Let R be the transition relation of the model, i.e., $(s, s') \in R$ if and only if there is a transition from state s to s'. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R^p that represents the edges of the attack graph. Therefore, the attack graph is $(S_{unsafe}, R^p, S_0^p, S_f^p, L)$, where S_{unsafe} and R^p represent the set of nodes and edges of the graph respectively; $S_0^p = S_0 \cap S_{unsafe}$ is the set of initial states; and $S_f^p = \{s | s \in S_{unsafe} \wedge s \models unsafe\}$ is the set of success states.

In symbolic model checkers, such as NuSMV, the transition relation and sets of states are represented using BDDs [Bry86], a compact representation for boolean functions. There are efficient BDD algorithms for all operations used in the algorithm shown in Figure 2.

3.3 Attack Graph Properties

We can show that an attack graph G generated by the algorithm in Figure 2 is *exhaustive* (Lemma 1(a)) and *succinct* with respect to states and transitions (Lemmas 1(b) and 1(c)).

an attack in G that contains s.

(c) succinct-transition. A transition $t = (s_1, s_2)$ of the input model (S, R, S_0, L) is in the attack graph G if and only if there is an attack in G that includes t.

Proof:

(a) exhaustive. ($\Rightarrow$) Let $e = s_0 t_0 \ldots t_{n-1} s_n$ be a (finite) execution of the input model such that s_n is an *unsafe* state. To prove that e is an attack in G, it is sufficient to show (1) $s_0 \in S_0^p$, (2) $s_n \in S_s^p$, and (3) for all $0 \leq k \leq n$, $s_k \in S$ and $t_k \in R^p$.

Since *unsafe* holds at s_n and for all k there is a path from s_k to s_n in the input model, by definition every s_k along $e1$ violates **AG**$(\neg unsafe)$. Therefore, by construction, every s_k is in S_{unsafe} and every t_k is in R^p. (1) and (2), and (3) follow immediately.

($\Leftarrow$) Suppose that $e = s_0 t_0 \ldots t_{n-1} s_n$ is an attack in the attack graph G. By construction, all states and transitions of e are also states and transitions in the input model. Since e is an attack, $s_0 \in S_0^p$ and $s_n \in S_s^p$. Therefore, $s_0 \in S_0$ and $s_n \in S$. So e is an execution of the input model, its first state is an initial state of the model, and p is false in its final state. It follows that e violates the property **AG**$(\neg unsafe)$.

(b) succinct-state. ($\Rightarrow$) By construction of the algorithm in Figure 2, all states generated for the attack graph are reachable from an initial state, and all of them violate **AG**$(\neg unsafe)$. Therefore, for any such state s in the input model, there is a path e_1 from an initial state to s, and there is a path e_2 from s to an *unsafe* state.

The concatenation of e_1 and e_2 is an execution e of the input model that violates **AG**$(\neg unsafe)$. By lemma 1a, e is an attack in G. Since e contains s, the proof is complete.

($\Leftarrow$) If there is an attack in G that contains s, then trivially s is in G.

(c) succinct-transition. ($\Rightarrow$) By lemma 1b, there is an attack $e_1 = q_0 t_0 \ldots s_1 \ldots t_{m-1} q_m$ that contains state s_1 and an attack $e_2 = r_0 u_0 \ldots s_2 \ldots u_{n-1} r_n$ that contains state s_2. So the following attack includes both states s_1 and s_2 and the transition t: $e = q_0 t_0 \ldots s_1 t s_2 \ldots u_{n-1} r_n$.

($\Leftarrow$) If there is an attack in G that contains t, then trivially t is in G.

□

4 A Simple Intrusion Detection Example

Consider the example network shown in Figure 3. There are two target hosts, ip_1 and ip_2, and a firewall separating them from the rest of the Internet. As shown, each host is running two of three possible services (ftp, sshd, a database). An intrusion detection system (IDS) monitors the network traffic between the target hosts and the outside world. There are four possible atomic attacks, identified numerically as follows: (0) sshd buffer overflow, (1) ftp .rhosts, (2) remote login, and (3) local buffer overflow. If an atomic attack is *detectable*, the intrusion detection system will trigger an alarm; if an attack is *stealthy*, the IDS misses it. The ftp .rhosts attack needs to find the target host with two vulnerabilities: a writable home directory and an executable command shell assigned to the ftp user name. The local buffer overflow exploits a vulnerable version of the xterm executable.

In this section, we construct a finite state model of the example network so that each state transition corresponds to a single atomic attack by the intruder. A state in the model represents the state of the system between atomic attacks. A typical transition from state s_1 to state s_2 corresponds to an atomic attack whose preconditions are satisfied in s_1 and whose effects hold in state s_2.

The intruder launches his attack starting from a single computer, ip_a, which lies outside the firewall. His eventual goal is to disrupt the functioning of the database. For which, the intruder needs root access on the database host ip_2.

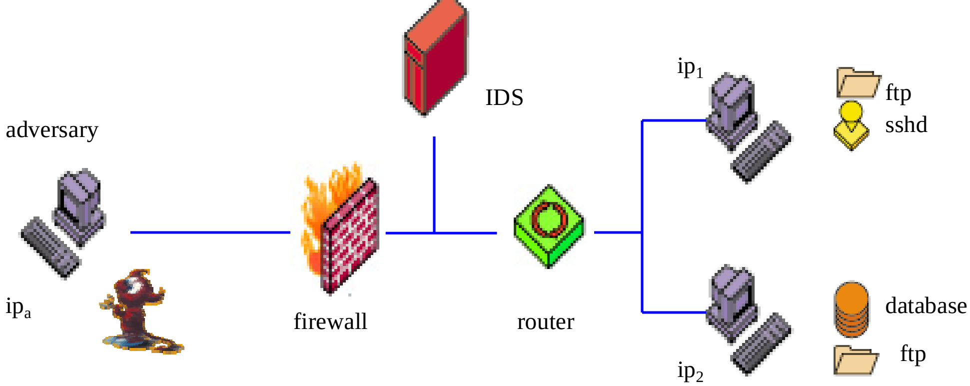

Figure 3: Example Network

4.1 States of the Finite State Machine Model

The Network

We model the network as a set of facts, each represented as a relational predicate. The state of the network specifies services, host vulnerabilities, connectivity, and a remote login trust relationship between hosts. There are six boolean variables for each host, specifying whether any of the three modeled services are running and whether any vulnerabilities are present on that host.

variable	**meaning**
ssh_h	ssh service is running on host h
ftp_h	ftp service is running on host h
data_h	database is running on host h
wdir_h	ftp home directory is writable on host h
fshell_h	ftp user has executable shell on host h
xterm_h	xterm executable is vulnerable to overflow on host h

Connectivity is expressed as a ternary relation $R \subseteq Host \times Host \times Port$, where $R(h_1, h_2, p)$ means that host h_2 is reachable from host h_1 on port p. The constants *sp* and *fp* will refer to the specific ports for the ssh and ftp services, respectively. Slightly abusing notation (by overloading R), we write $R(h_1, h_2)$ when there is a network route from h_1 to h_2. We model trust as a binary relation *RshTrust* $\subseteq Host \times Host$, where *RshTrust*$(h_1, h_2)$ indicates that a user may log in from host h_2 to host h_1 without authentication (i.e., host h_1 "trusts" host h_2).

The Intruder

The function *plvl*$_A$: *Hosts* $\rightarrow$ $\{$*none, user, root*$\}$ gives the level of privilege that intruder A has on each host. There is a total order on the privilege levels: *none* $<$ *user* $<$ *root*.

Several state variables specify which attack the intruder will attempt next:

variable	**meaning**
attack	attack type
source	source host
target	target host
strain	stealthy/detectable attack

detectable, it will trigger an alarm when executed on a host or network segment monitored by the IDS; if an attack is *stealthy*, the IDS does not detect it.

We specify the IDS with a function *ids: Host* $\times$ *Host* $\times$ *Attack* $\rightarrow \{d, s, b\}$, where *ids*(h_1, h_2, a) = *d* if attack a is *d*etectable when executed with source host h_1 and target host h_2; *ids*(h_1, h_2, a) = *s* if attack a is *s*tealthy when executed with source host h_1 and target host h_2; and *ids*(h_1, h_2, a) = *b* if attack a has *b*oth detectable and stealthy strains, and success in detecting the attack depends on which strain is used. When h_1 and h_2 refer to the same host, *ids*(h_1, h_2, a) specifies the intrusion detection system component (if any) located on that host. When h_1 and h_2 refer to different hosts, *ids*(h_1, h_2, a) specifies the intrusion detection system component (if any) monitoring the network path between h_1 and h_2. In addition, a global boolean variable specifies whether the IDS alarm has been triggered by any previously executed atomic attack.

4.2 Initial States

Initially, there is no trust between any of the hosts; the trust relation *Tr* is empty. The connectivity relation R is shown in the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). Each entry is a triple of boolean values. The first value is 'y' if h_1 and h_2 are connected by a physical link, the second value is 'y' if h_1 can connect to h_2 on the ftp port, and the third value is 'y' if h_1 can connect to h_2 on the sshd port.

R	ip_a	ip_1	ip_2
ip_a	y,n,n	y,y,y	y,y,n
ip_1	y,n,n	y,y,y	y,y,n
ip_2	y,n,n	y,y,y	y,y,n

We use the connectivity relation to reflect the firewall rule sets as well as the existence of physical links. For the table above, the firewall is open and does not place any restrictions on the flow of network traffic.

Initially, the intruder has *root* privileges on his own machine ip_a and no privileges on the other hosts.

The paths between (ip_a, ip_1) and between (ip_a, ip_2) are monitored by the single network-based IDS. The path between (ip_1, ip_2) is not monitored. There are no other host-based intrusion detection components. The IDS detects the remote login attack and the detectable strains of the sshd buffer overflow attack.

4.3 Transitions

Our model has nondeterministic state transitions. If the current state of the network satisfies the **preconditions** of more than one atomic attack rule, the intruder nondeterministically "chooses" one of those attacks. The state then changes according to the **effects** clause of the chosen attack rule. The intruder repeats this process until his goal is achieved.

We model four atomic attacks. Throughout the description, S is used to designate the source host and T the target host. Recall that $R(S, T, p)$ denotes that host T is reachable from host S on port p.

Sshd Buffer Overflow

This remote-to-root attack immediately gives a remote user a root shell on the target machine.

intruder preconditions

$plvl_A(S) \geq$ user	*User-level privileges on host S*
$plvl_A(T) <$ root	*No root-level privileges on host T*

network preconditions

ssh_T	*Host T is running sshd*
$R(S, T, sp)$	*Host T is reachable from S on port* sp

intruder effects

$plvl_A(T) =$ root	*Root-level privileges on host T*

network effects

$\neg\text{ssh}_T$	*Host T is not running sshd*

end

Ftp .rhosts

Using an ftp vulnerability, the intruder creates an .rhosts file in the ftp home directory, creating a remote login trust relationship between his machine and the target machine.

attack ftp-rhosts **is**

intruder preconditions

$plvl_A(S) \geq$ user	*User-level privileges on host S*

network preconditions

ftp_T	*Host T is running ftp*
$R(S, T, fp)$	*Host T is reachable from S on port* fp
wdir_T	*Ftp directory writable on host T*
fshell_T	*Ftp user has been assigned a valid shell on host T*
$\exists X.\neg RshTrust(X, T)$	*No rsh trust for some host X and T*

intruder effects

none

network effects

$\forall X.RshTrust(X, T)$	*Rsh trust between all hosts and T*

end

Remote Login

Using an existing remote login trust relationship between two machines, the intruder logs in from one machine to another, getting a user shell without supplying a password. This operation is usually a legitimate action performed by regular users, but from the intruder's viewpoint, it is an atomic attack.

attack rsh-login **is**

intruder preconditions

$plvl_A(S) =$ user	*User-level privileges on host S*
$plvl_A(T) =$ none	*No privileges on host T*

network preconditions

$RshTrust(S, T)$	*Rsh trust between S and T*
$R(S, T)$	*Host T is reachable from S*

intruder effects

end

Local Buffer Overflow

If the intruder has acquired a user shell on the target machine, the next step is to exploit a buffer overflow vulnerability on a `setuid root` file to gain root access.

attack local-setuid-buffer-overflow **is**
  intruder preconditions
    $plvl_A(T) =$ user — *User-level privileges on host T*
  network preconditions
    xterm$_T$ — *There is a vulnerable xterm executable*
  intruder effects
    $plvl_A(T) =$ root — *Root-level privileges on host T*
  network effects
    none
end

It is easy to see that each atomic attack strictly increases either the intruder's privilege level on the target host or remote login trust between hosts. This means that *the attack graph has no cycles*.

From our finite model we can now automatically construct attack graphs that demonstrate how the intruder can violate various security properties. Suppose we want to generate all attacks that demonstrate how the intruder can gain root privilege on host ip_2 and remain undetected by the IDS. The following CTL formula expresses the safety property that *the intruder on host ip_2 always has privilege level below root* or *is detected*:

$$\mathbf{AG}(\textit{network.adversary.privilege[2]} < \textit{network.priv.root} \mid network.detected)$$

Figure 4 shows the attack graph produced by our tool for this property. Each node is labeled by an attack id number, which corresponds to the atomic attack *to be attempted next*; a flag S/D indicates whether the attack is stealthy or detectable by the intrusion detection system; and the numbers of the source and target hosts (ip_a corresponds to host number 0).

Any path in the graph from the root node to a leaf node shows a sequence of atomic attacks that the intruder can employ to achieve his goal while remaining undetected. For instance, the path highlighted by dashed-boxed nodes consists of the following sequence of four atomic attacks: overflow sshd buffer on host 1, overwrite .rhosts file on host 2 to establish rsh trust between hosts 1 and 2, log in using rsh from host 1 to host 2, and finally, overflow a local buffer on host 2 to obtain root privileges.

We have also expanded the example described above by adding two additional hosts, four additional atomic attacks, several new vulnerabilities, and flexible firewall configurations. For this larger example the attack graph has 5948 nodes and 68364 edges.

5 Minimization Analysis

Once we have an attack graph generated for a specific network with respect to a given safety property, we can utilize it for further analysis. A system administrator has available to him a set of *measures*, such as deploying additional intrusion detection tools, adding firewalls, upgrading software, deleting user accounts,

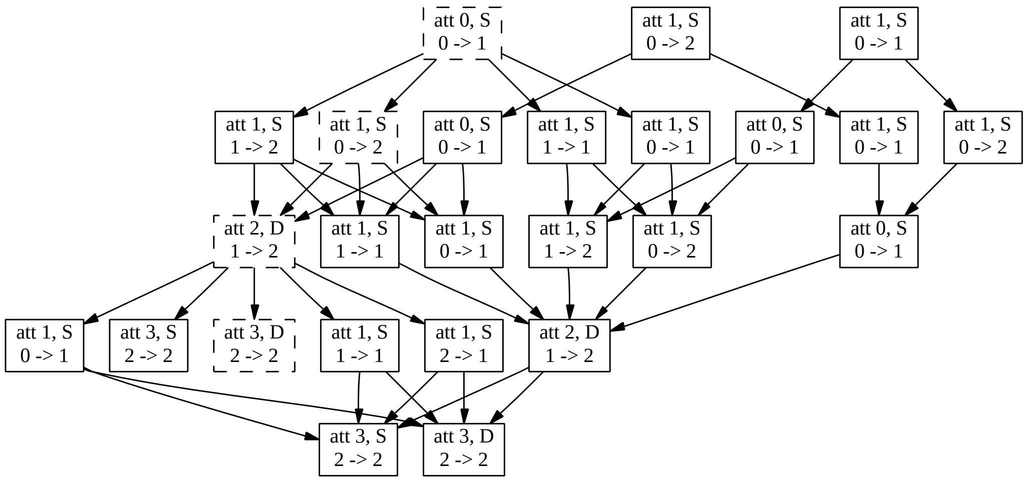

Figure 4: Attack Graph

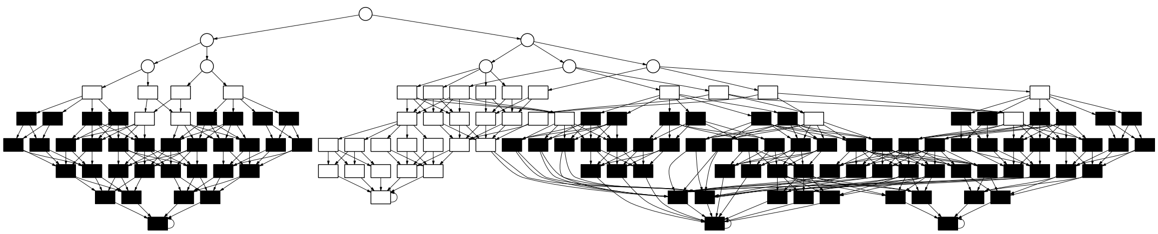

Figure 5: Attack Graph Analysis

set of atomic attacks they thwart. It helps us answer questions such as 1, 2, and 3 posed in Section 1.1. Let us look at each question in turn since they suggest different solution approaches.

5.1 Minimal Subsets of Atomic Attacks to Thwart

Suppose we want to find a minimal set, A, of atomic attacks that must be prevented to guarantee the adversary cannot achieve his goal. A system analyst can use this information in deciding to choose one measure m_1, which eliminates this minimal set of attacks over another measure, m_2, perhaps cheaper than m_1, but ineffective with respect to A.

A naive solution is as follows:

1. Make only a subset of the atomic attacks available to the intruder.
2. Run the model checking algorithm to determine if the adversary can succeed.
3. Do Steps 1 and 2 for all possible non-empty subsets of atomic attacks.

Clearly this solution is exponential in the number of atomic attacks. For our example, however, the number is small, and we can easily determine this minimal set. As a by-product of determining this set, we can easily answer the first question posed in Section 1.

Question 1: What successful attacks are undetected by the IDS?

Answer: To answer this question, we modify the model slightly. For simplicity, we nondeterministically decide which subset to consider initially, before any attack begins; once the choice is made, the subset of available atomic attacks remains constant during any given attack. We ran the model checker on the modified model with the invariant property that says the intruder never obtains root privilege on host ip_2:

$$\mathbf{AG}(\textit{network.adversary.privilege[2]} < \textit{network.priv.root})$$

The post-processor marked the states where the intruder has been detected by the IDS. The result is shown in Figure 5. The white rectangles indicate states where the attacker had not yet been detected by the intrusion detection system. The black rectangles are states where the intrusion detection system has sounded an alarm. Thus, white leaf nodes are desirable for the attacker because his objective is achieved without detection. Black leaf nodes are less desirable—the attacker achieves his objective, but the alarm goes off.

The resolution of which atomic attacks are available to the intruder happens in the circular nodes near the root of the graph. The first transition out of the root (initial) state picks the subset of attacks that the intruder will use. Each child of the root node is itself the root of a disjoint subgraph where the subset of atomic attacks chosen for that child is used. Note that the number of such subgraphs descending from the root node corresponds to the number of subsets of atomic attacks with which the intruder can be successful—the model checker determines that for any other possible subset, there is no possible successful sequence of atomic attacks.

The root of the graph in Figure 5 has two subgraphs, corresponding to the two subsets of atomic attacks that will allow the intruder to succeed. In the left subgraph the sshd buffer overflow attack is not available to the intruder; it can be readily seen that the intruder can still succeed, but cannot do so while remaining undetected by the IDS. In the right subgraph, all attacks are available. Thus, the entire attack graph implies that all atomic attacks other than the sshd attack are indispensable: the intruder cannot succeed without them. That is, for no other subset of atomic attacks can the intruder succeed in achieving his goal. The analyst can use this information to guide decisions on which network defenses can be profitably upgraded.

The white cluster in the middle of the figure is isomorphic to the attack graph presented in Figure 4; it shows attacks in which the intruder can achieve his objective without detection (i.e., all paths by which the intruder reaches a white leaf in the graph).

$$\mathbf{AG}(\neg unsafe)$$

Let $\mathcal{A}$ be the set of atomic attacks, and $G = (S, E, s_0, s_s, L)$ be the attack graph, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L : E \rightarrow \mathcal{A} \cup \{\epsilon\}$ is a labeling function where $L(e) = a$ if an edge $e = (s \rightarrow s')$ corresponds to an atomic attack a, otherwise $L(e) = \epsilon$. Edges labeled with ϵ represent system transitions that do not correspond to an atomic attack. Moreover, as demonstrated below additional ϵ edges can be also introduced by our construction. Without loss of generality we can assume that there is a single initial and success state. For example, consider an attack graph with multiple initial states $s_0^1, \cdots, s_0^j$ and success states $s_s^1, \cdots, s_s^u$. We can add a new initial state s_0 and a new success state s_s with ϵ-labeled edges (s_0, s_0^m) $(1 \leq m \leq j)$ and (s_s, s_s^t) $(1 \leq t \leq u)$.

Suppose we are also given a finite set of measures $M = \{m_1, \cdots, m_k\}$ and a function $covers : M \rightarrow 2^{\mathcal{A}}$. An atomic attack $a \in covers(m_i)$ if adopting measure m_i removes the atomic attack a.

We are now ready to address the question of what measures a system administrator should deploy to ensure the system is safe. Again, there is a naive solution, that is, to try all possible subsets of measures $M' \subseteq M$ and determine which of those make the system safe. We discuss this approach in the context of question 2:

Question 2: If all measures for protecting a network are deployed, does the system become safe?

Answer: A network administrator wants to find out whether adopting measures from a set $M' \subseteq M$ will make the network safe. This question can be answered in linear time using the attack graph G. First, we define $covers(M')$ as $\bigcup_{m \in M'} covers(m)$. Next, we remove all edges e from G such that $L(e) \in covers(M')$. The network is safe iff the success state s_s is not reachable from the initial state s_0. This simple reachability question can be answered in time that is linear in the size of the graph.

As the set of measures grows (and as the set of atomic attacks grows), we really would like to have the system administrator choose the smallest subset of measures that would guarantee the networked system is safe. We address this decision in the context of question 3:

Question 3: Given a set of measures M, what is the smallest subset of measures M' whose deployment makes the system safe?

Answer: A network administrator wishes to find a subset $M' \subseteq M$ of smallest size, such that adopting the measures in the set M' will make the network safe. Unfortunately, this problem is NP-complete, but we develop good approximation algorithms. We proceed in two steps:

Step 1: Finding a small set of atomic attacks.

In this step, we find a set of atomic attacks whose removal makes the network safe. As described in the previous section, checking every possible subset of attacks is exponential in the number of attacks. In an earlier conference paper [SHJ+02], we show that finding the *minimum* set of atomic attacks which must be removed to thwart an intruder is in fact *NP*-complete. We repeat part of the proof below (see Lemma 2). We also demonstrated how a *minimal* set can be found in polynomial-time.[2] In this paper, we further explore the complexity of this problem. Section 5.2.1 proves that the problem of finding a minimum set of attacks is polynomially equivalent to the minimum hitting set problem, where the collection of sets is represented as labeled directed graph. This reduction provided us with additional insight. This additional insight enabled us to find a greedy algorithm with provable bounds.

[2] In the conference paper we showed the reduction to the *minimum cover* problem [GJ79, Page 222]; here we show it to the *minimum hitting set* problem.

is a function, where $covers(m_i)$ represents the set of atomic attacks that are removed by adopting the measure m_i. With each attack a in the set A', we associate a set of measures $M(a)$ which is $\{m_i \mid a \in covers(m_i)\}$. The set of attacks A' defines a collection $C_{A'}$ of subsets of M. We wish to find the smallest subset $M' \subseteq M$ such that for all $a \in A'$ there exists an $m_i \in M'$ such that $a \in covers(m_i)$, or equivalently $M' \cap M(a) \neq \emptyset$. This is known as the minimum hitting set problem, which is NP-complete, but good approximation algorithms exist to solve this problem (see Section 5.2.2)

5.2.1 The Minimum Critical Attack Sets and the Minimum Hitting Set Problem

This section addresses the first step in the answer to question 3. Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L : E \to \mathcal{A} \cup \{\epsilon\}$ is a labeling function.

Given a state $s \in S$, a set of attacks C is *critical* with respect to s if and only if the intruder cannot reach his goal from s when the attacks in C are removed from his arsenal. Equivalently, C is critical with respect to s if and only if every path from s to the success state s_s has at least one edge labeled with an attack $a \in C$.

A critical set corresponding to a state s is *minimum* (denoted $M(s)$) if there is no critical set $M'(s)$ such that $|M'(s)| < |M(s)|$. In general, there can be multiple minimum sets corresponding to a state s. Of course, all minimum critical sets must be of the same size.

A critical set of an attack graph $G = (S, E, s_0, s_s, L)$ is defined as a critical set corresponding to the initial state s_0. Therefore, the *Minimum Critical Set of Attacks (MCSA) problem* is the problem of finding a minimum critical set of attacks $M(s_0)$. The decision version of the problem is defined as follows: given an attack graph $G = (S, E, s_0, s_s, L)$ and a positive integer K, is there a critical set of attacks $A \subseteq \mathcal{A}$ such that $|A| \leq K$?

Lemma 2 Assume that we are given an attack graph $G = (S, E, s_0, L)$ and an integer k. The MCSA problem of determining whether there is a critical set $C(s_0)$ such that $|C(s_0)| \leq k$ is *NP*-complete.

Proof: *First, we prove that the problem is in* NP. *Guess a set $C \subseteq \mathcal{A}$ with size $\leq k$. We need to check that C is a critical set of attacks. This can be accomplished in polynomial time using the reachability algorithm described before (see answer to question 2). Therefore, the problem is in* NP.

Next, we prove that the problem is NP-*hard. The reduction is from the* minimum hitting set *problem, details as given in the remainder of this section.*

Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$. A *path* π is sequence of states $q_1, \cdots, q_n$, such that $q_i \in S$ and $(q_i, q_{i+1}) \in E$. A *complete path* starts from the initial state s_0 and ends in the success state s_s. The label of a path $\pi = q_1, \cdots, q_n$ (abusing notation, we will denote it also as $L(\pi)$) is a subset of a set of attacks $\mathcal{A}$

$$\bigcup_{i=1}^{n-1} \{L(q_i, q_{i+1})\} \setminus \{\epsilon\} .$$

$L(\pi)$ represents the set of atomic attacks used on the path π. A set of attacks $A \subseteq \mathcal{A}$ is called *realizable* in the attack graph G iff there exists a complete path π in G such that $L(\pi) = A$. In other words, an intruder can use the set of attacks A to start from the initial state and reach the success state. The set of all realizable sets in an attack graph G is denoted by $Rel(G)$. The following lemma is easy to prove and follows straight from the definitions.

Lemma 3 Assume that we are give an attack graph $G = (S, E, s_0, s_s, L)$. A set of attacks A is critical iff

$$\forall A' \in Rel(G). A' \cap A \neq \emptyset .$$

In other words, all realizable sets have a non-empty intersection with a critical set A.

in C?

Lemma 3 proves that the problem of finding whether the attack graph G has a critical set of size $\leq K$ *is the* hitting set problem with $C = Rel(G)$, $S = \mathcal{A}$, and K.

Next suppose we have an instance (C, S, K) of the hitting set problem. We will construct an attack graph $G' = (S', E', s_0', s_s', L')$, where $L' : E' \to S \cup \{\epsilon\}$, i.e., the set of attacks used in the attack graph G' is S. Moreover, the set of realizable sets $Rel(G')$ of the graph G' is the collection C. A critical set of size $\leq K$ of the attack graph G' is a hitting set for the collection C. Next, we describe the construction of G'. Let $C = \{C_1, \cdots, C_m\}$ be the collection of sets and $S = \{s_1, \cdots, s_n\}$ be the set. We make m copies $S^1, \cdots, S^m$ of the set S. The set of elements in S^i will be denoted by $\{s_1^i, \cdots, s_n^i\}$. The set of states S' in the attack graph G' is

$$\{s_0', s_s'\} \cup S^1 \cup \cdots \cup S^m .$$

The initial state is s_0' and the final state is s_s'. The set of edges E' and the labeling function L' are defined as follows:

- There is an edge from s_0' to every state in the set $\{s_1^1, s_1^2, \cdots, s_1^m\}$, and label of the edge (s_0', s_i^1) is s_1 if $s_1 \in C_i$, otherwise it is ϵ.
- For all $1 \leq i \leq m$ and $1 \leq j \leq n-1$, there is an edge (s_j^i, s_{j+1}^i), and the label of edge (s_j^i, s_{j+1}^i) is s_{j+1} if $s_{j+1} \in C_i$, otherwise it is ϵ.
- There is an edge from every state in the set $\{s_n^1, s_n^2, \cdots, s_n^m\}$ to the state s_s', and labels of all these edges is ϵ.

The sizes of the sets S' and E' in the attack graph G' are $mn + 2$ and $2m + mn$ respectively. It is easy to see that $Rel(G')$ is equal to C, and $S' \subseteq S$ is a critical set of the attack graph G' iff S' is a hitting set for the collection C. Since the size of G' is polynomial in the size of the instance of the hitting set problem and the hitting set problem is NP-complete, the MCSA problem is NP-hard. Lemma 2 proves that MCSA is in NP. Therefore, MCSA is NP-complete. The next example illustrates our construction.

Note: The discussion above also proves that the problem of finding a minimum set of measures whose adoption will make the network safe is also NP-complete. One can simply take the set of measures M to be the set of attacks $\mathcal{A}$.

Example 1 We give a short example to illustrate the reduction. Consider a set $S = \{s_1, s_2, s_3\}$. Suppose that the collection C consists of the following subsets:

$$\begin{aligned} C_1 &= \{s_1, s_2\} \\ C_2 &= \{s_2, s_3\} \\ C_3 &= \{s_2\} \end{aligned}$$

The attack graph G' corresponding to this problem is shown in Figure 6. The set of attacks is $\{s_1, s_2, s_3\}$. The set of realizable sets $Rel(G')$ is exactly the collection C. The set of attacks $\{s_1, s_2\}$ is critical because every path from s_0' to the success state s_s' uses at least one edge with the label in the set $\{s_1, s_2\}$. Moreover, $\{s_1, s_2\}$ is a hitting set for the collection $C = \{C_1, C_2, C_3\}$.

The above discussion proves that the problem of finding critical sets in attack graph is *polynomially equivalent* to finding hitting sets for a collection, with one caveat–the collection of sets C is represented as an attack graph. *An attack graph can be an exponentially succinct representation of a collection of sets.* Figure 7 shows an attack graph of linear size whose set of realizable sets is the power set of $\{s_1, \cdots, s_n\}$. Therefore, the minimum critical set problem is polynomially equivalent to the hitting set problem where the collection of sets C is represented as a labeled directed graph.

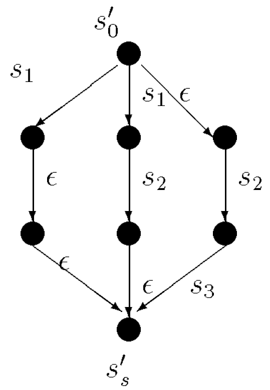

Figure 6: Attack graph corresponding to the collection C.

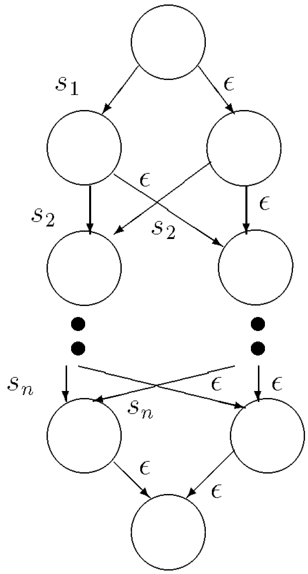

Figure 7: Attack graph representing an exponential number of realizable sets.

(C, S, K) be an instance of the hitting set problem. Let S' and C' be initially the empty set. The greedy algorithm executes the following step until $C' = C$.

- Pick an element s out of the set $S \setminus S'$ that covers the maximum number of sets in the collection $C \setminus C'$. An element s is said to cover a set $S_1 \subseteq S$ iff $s \in S_1$.
- Let s be the element picked in the previous step and $C(s)$ be the collection of sets in C covered by s. Update S' and C' as follows:

$$S' \leftarrow S' \cup \{s\}$$
$$C' \leftarrow C' \cup C(s)$$

Let H_d be the d-th harmonic number $\sum_{i=1}^{d} \frac{1}{i}$. Let $C(s)$ be the number of sets in the collection C that are covered by the element s.

Lemma 4 *GREEDY-HITTING-SET* is a polynomial-time $\rho(n)$-approximation algorithm, where $\rho(n) = H(\max_{s \in S}\{|C(s)|\})$.

The proof of the lemma follows from the equivalence between the minimum hitting set and the minimum cover problem [ADP80] and the proof of the approximation factor $\rho(n)$ for the greedy algorithm for the minimum cover problem [CLR85]. Using the equivalence between the problems of finding a minimum critical set and a minimum hitting set, we can construct a greedy procedure (called *GREEDY-CRITICAL-SET*) for finding a critical set for the attack graph. Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L : E \rightarrow \mathcal{A} \cup \{\epsilon\}$ is a labeling function. Moreover, assume that we can compute in polynomial time the function $\mu_G : \mathcal{A} \rightarrow \aleph$, where $\mu_G(a)$ is the number of realizable sets in the attack graph G that contain the attack a. Formally, $\mu_G(a)$ is equal to

$$|\{A' | a \in A' \text{ and } A' \in Rel(G)\}| \; .$$

Initially, let A' be the empty set and $G' = G$. The greedy algorithm *GREEDY-CRITICAL-SET* executes the following step until G' is empty.

- Pick and element a from the set $\mathcal{A} \setminus A'$ that maximizes $\mu_{G'}(a)$.
- Let a be the element picked in the previous step. Update A' and G' as follows:

$$A' \leftarrow A' \cup \{a\}$$
Remove all edges labeled with a from G'

Lemma 5 *GREEDY-CRITICAL-SET* is a polynomial-time $\rho(n)$-approximation algorithm, where $\rho(n) = H(\max_{a \in \mathcal{A}}\{\mu_G(a)\})$.

Next, we explore conditions when the function μ_G can be computed in polynomial time. Assume that the attack graph G is a DAG. An argument for this was given in Section 4.3. Moreover, assume that each atomic attack is *used only once* on a path from the initial state s_0 to the success state s_s. This is not a unreasonable assumption because the attack graph edges are labeled with instantiations of attack templates shown in Section 4.3, e.g., a local-setuid-buffer-overflow attacks on two different hosts are distinct in the attack graph. Such attack graphs are called *use-once* DAGs. The following lemma is easy to prove.

Lemma 6 For an attack graph that is a use-once DAG, the function μ_G can be computed in time that is linear in size of the attack graph.

of thwarting an attack? If we have probabilities available to us, we can annotate attack graphs to help system administrators answer such questions.

In our work, we do not require that all transitions be given probabilities; in general, our annotated attack graphs can have a mix of probabilistic and nondeterministic state transitions. We pursue the implications of this general kind of attack graph in this section.

In general, we also do not require probabilities to be numeric; they can be symbolic, e.g., “high,” “medium,” or “low,” and even partially ordered. In an earlier paper [JW01], we discuss an analysis that uses symbolic probabilities; in this paper, however, we restrict ourselves to numeric values.

6.1 Probabilistic Attack Graphs

Suppose that the graph has a state s with only two outgoing transitions. In a regular attack graph, the choice of which transition to take when the system is in state s is nondeterministic. However, we may have some empirical data that enables us to estimate that whenever the system is in state s, on average it will take one of the transitions four times out of ten and the other transition six remaining times. We can place probabilities 0.4 and 0.6 on the corresponding edges in the attack graph. Intuitively, the probability of the transition $s \to s'$ represents the likelihood that the atomic attack corresponding to the transition will succeed. We call a state with known probabilities for outgoing transitions *probabilistic*. When we have assigned all known probabilities in this way, we are left with an attack graph that has some probabilistic and some nondeterministic states in it. We call such mixed attack graphs *probabilistic attack graphs*. We use probabilistic attack graphs to evaluate the reliability of a network. Note that probabilities of all the transitions might not be available because of lack of data, e.g., a new type of atomic attack.

Since the attack graph includes only those states and transitions that can lead to success states, it excludes some transitions that exist in the complete model M. These excluded transitions can have non-zero probability, so that the sum of probabilities of transitions from a probabilistic state will be less than 1. To address this problem, we must model the rest of M in some way. We add a “catch-all” *escape* state s_e to the attack graph. A probabilistic state s in the attack graph will have a transition to s_e if and only if in M there is a transition from s to some state *not* in the attack graph. The probability of going from s to s_e will be 1 minus the sum of the probabilities of going to other states. There are no transitions out of s_e except a self-loop (which preserves the totality of the transition relation τ).

In an attack graph containing the escape state s_e attacks are allowed to terminate in s_e. We will call them *escape attacks*, or attacks that were pre-empted by the intruder before he reached his goal.

6.1.1 Definition of PAGs

Definition 3 A *probabilistic attack graph* or PAG is a tuple $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$, where S_n is a set of nondeterministic states, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state ($s_e \notin S_s$), $S = S_n \cup S_q$ is the set of all states, $\tau \subseteq S \times S$ is a transition relation, $\pi : S_q \to S \to \Re$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \to 2^{AP}$ is a labeling of states with a set of propositions true in that state.

A probabilistic attack graph distinguishes between nondeterministic states (set S_n) and probabilistic states (set S_q). Moreover, the sets of nondeterministic and probabilistic states are disjoint ($S_n \cap S_q = \emptyset$). The function π specifies probabilities of transitions from probabilistic states, so that for all transitions $s_1 \to s_2 \in \tau$ such that $s_1 \in S_q$, we have $P(s_1 \to s_2) = \pi(s_1)(s_2) > 0$. Thus, $\pi(s)$ can be viewed as a probability distribution on next states. Intuitively, when the system is in a nondeterministic state s_n, we have no information about the relative probabilities of the possible next transitions. When the system is in a probabilistic state s_q, it will choose the next state according to probability distribution $\pi(s_q)$.

Let $G = (S, \tau, S_0, S_s, L)$ be the attack graph and P a function that assigns probabilities to transitions. The probabilities can be loosely interpreted as the probability of the atomic attack corresponding to the transition succeeding. We are interested in finding the reliability of the attack graph, i.e., the probability that the intruder will not succeed. We can view G as a Markov chain with S as its state space and $P(s_1 \to s_2)$ as its transition probability. Let $U : S \to \Re^+$ be the steady state probability of the Markov chain (see [Dur95] for definitions and technical conditions). In this case, the reliability of the attack graph G is given by the following expression:

$$1 - \sum_{s \in S_s} U(s)$$

In other words, the reliability is the probability that in the "long run" the Markov chain will not be in a state in the set S_s.

In general, however, we do not have probabilities assigned to all transitions; thus in Section 6.2 we show how to perform similar reliability analysis on probabilistic attack graphs in the presence of nondeterministic states. The justification of our approach relies on converting a probabilistic attack graph (PAG) into an alternating probabilistic attack graph (APAG) and then interpreting the result as a Markov Decision Process; we give this construction and interpretation in Section 6.3; we give the proof of correctness of the MDP value iteration algorithm applied to PAGs in Section 6.4. Sections 6.3 and 6.4 can be skipped upon a first reading.

6.2 Reliability Analysis of PAGs

Assume that we are given a PAG $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$. Intuitively, we are interested in finding out the probability that the intruder will reach a success state starting from one of the initial states. As shown above, in the absence of nondeterministic states we can compute this metric by using the steady state probabilities of the Markov chain. In the presence of nondeterministic states the intruder will choose transitions in order to maximize his probability of succeeding. For example, if an intruder reaches a nondeterministic state s with transitions to $s_1, \cdots, s_k$, he will choose to transition to state s_i $(1 \leq i \leq n)$ which will maximize his probability of reaching a success state. This idea can be "formalized" using concepts from the theory of Markov Decision Processes [Alt99, Put94].

6.2.1 Value Iteration for PAGs

Given a state s, the set of successors of s is denoted by $succ(s)$. Formally, $succ(s)$ is equal to $\{s' | (s, s') \in \tau\}$. First, we define a *value function* $V : S \to \Re^+$. For all $s \in S_s$, $V(s) = 1.0$. For all states $s \in S \setminus S_s$ the value function is iterated according to the following equations until convergence.

$$V(s) = \begin{cases} \max_{s' \in succ(s)} V(s') & \text{if } s \in S_n \setminus S_s \\ \sum_{s' \in succ(s)} P(s \to s') V(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

Let $V^\star$ be the value function after convergence. Intuitively, $\sum_{s \in S_0} V^\star(s)$ is the probability for the intruder to reach a success state if he "breaks" the nondeterminism to maximize the probability of succeeding. Therefore, the worst case reliability of the network is $1 - \sum_{s \in S_0} V^\star(s)$. This algorithm is known as *value iteration*. The justification of the value iteration algorithm as applied to PAGs is presented in Section 6.4.

6.2.2 Example Revisited

We implemented the value iteration algorithm in our attack graph post-processor and ran it on a slightly modified version of the intrusion detection example from Section 4. In the modified example, each attack has both detectable and stealthy variants. The intruder chooses which atomic attack to try next, and he has a certain probability of picking a stealthy or a detectable variant. We assigned imaginary probabilities of picking a stealthy attack variant as follows: 0.2 for sshd buffer overflow, 0.5 for ftp .rhosts, 0.05 for the

In this setup, the computed probability of intruder success is 0.2, and his best strategy is to attempt sshd buffer overflow on host ip_1, and then conduct the rest of the attack from that host. The only possibility of detection is the sshd buffer overflow attack itself, since the IDS does not see the activity between hosts ip_1 and ip_2.

Given this context, a system administrator can answer the following question:

Question 4: The deployment of which security measure(s) will increase the likelihood of thwarting an attacker?

Answer: Installing an additional IDS component to monitor the network traffic between hosts ip_1 and ip_2 reduces the probability of the intruder remaining undetected to 0.025; installing a host-based IDS on host ip_2 reduces the probability to 0.16. Other things being equal, this is an indication that the former remedy is more effective.

6.3 Alternating Probabilistic Attack Graphs and Markov Decision Processes

In this section we show that probabilistic attack graphs can be reduced to Markov Decision Processes (without the reward function). We then demonstrate how we can assign a reward function to attack graphs such that standard MDP algorithms can be used to compute reliability metric of the network being modeled.

Definition 4 *[Alt99, Put94]* A Markov Decision Process is a tuple $(\mathbf{X}, A, \mathcal{P}, c)$ where

- $\mathbf{X}$ is a finite state space. Generic notation for MDP states will be x, y, z.
- A is a finite set of actions. $A(x) \subseteq A$ denotes the actions that are available at state x. Set $\mathcal{K} = (x, a) : x \in \mathbf{X}, a \in A(x)$ is the set of state-action pairs. A generic notation for an action will be a.
- $\mathcal{P} : \mathbf{X} \times A \times \mathbf{X}$ are the transition probabilities; thus, $\mathcal{P}(xay)$ (also written as $\mathcal{P}_{xay}$) is the probability of moving from state x to y if action a is chosen.
- $r : \mathcal{K} \rightarrow \Re$ is an immediate reward. Cost may be equivalently viewed as a negative reward. We will freely use the term cost to mean negative reward, and vice versa.

An *execution fragment* (also known as history in the traditional MDP literature) of an MDP is a sequence $x_0 a_1 x_1 \ldots a_n x_n$ of alternating states and actions such that the sequence begins and ends with a state, and for all $0 < k \leq n$, $a_k \in A(x_{k-1})$ and $0 < \mathcal{P}(x_{k-1}, a_k, x_k) \leq 1$. Given an execution fragment $e = x_0 a_1 x_1 \ldots a_n x_n$, the probability of the execution fragment (denoted by $P(e)$) is given by the following expression:

$$\prod_{k=1}^{n} P(x_{k-1}, a_k, x_k)$$

It is possible to convert a probabilistic attack graph into an MDP such that the behaviors of the PAG and the MDP are identical. To explain the conversion procedure, we define a restricted kind of probabilistic attack graph.

Definition 5 An *alternating probabilistic attack graph* or *APAG* is a tuple $G = (S_n, S_q, s_e, S, \tau_n, \tau_q, \pi, S_0, S_s, L)$, where S_n is a set of nondeterministic states, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state, $S = S_n \cup S_q$ is the set of all states, $\tau_n \subseteq S_n \times S_q$ is a set of nondeterministic transitions, $\tau_q \subseteq S_q \times S_n$ is a set of probabilistic transitions, $\pi : S_q \rightarrow S_n \rightarrow \Re$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \rightarrow 2^{AP}$ is a labeling of states with a set of propositions true in that state.

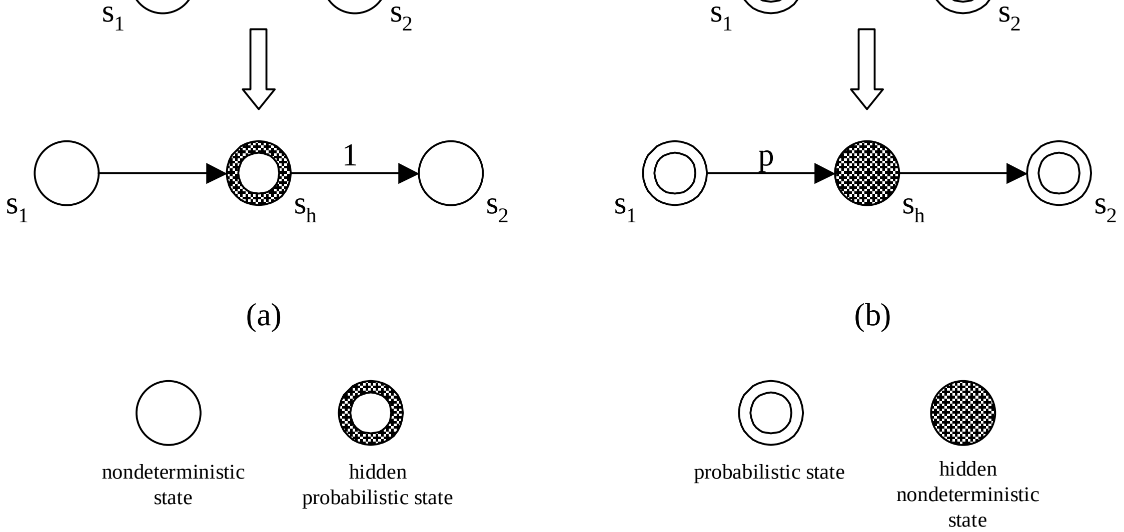

Figure 8: Converting PAG to APAG

An alternating probabilistic attack graph (APAG) *does not have any transitions between two nondeterministic or between two probabilistic states*. In other words, a nondeterministic state has transitions to probabilistic states only, and vice versa. An execution of an APAG will always have strictly alternating nondeterministic and probabilistic states.

Next we describe an algorithm that converts a PAG $G_p = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ into an APAG $G_p^A = (S_n^A, S_q^A, s_e, S, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ that has equivalent behaviors. The algorithm works by adding *hidden* states and transitions to the graph such that every execution becomes strictly alternating, yet does not change its *observable* (non-hidden) components.

We start with $S_n^A = S_n$, $S_q^A = S_q$, $\tau_n^A := \emptyset$, $\tau_q^A := \emptyset$, $\pi^A := 0.0$, and $L^A = L$. Next,

1. Whenever τ has a transition from probabilistic state s_1 to nondeterministic state s_2, we add the transition to τ_p^A and its probability to π^A.

2. Whenever τ has a transition from nondeterministic state s_1 to probabilistic state s_2, we add the transition to τ_n^A.

3. Whenever τ has a transition between two nondeterministic states s_1 and s_2, we add a hidden probabilistic state s_h to S_q^A, an observable transition $s_1 \to s_h$ to τ_n^A, and a hidden transition $s_h \to s_2$ to τ_p^A, assigning the latter probability 1.0 in π^A (Figure 8a). We also set $L^A(s_h) = L(s_1)$.

4. Whenever τ has a transition between two probabilistic states s_1 and s_2, we add a hidden nondeterministic state s_h to S_n^A, a hidden transition $s_h \to s_2$ to τ_n^A, and an observable transition $s_1 \to s_h$ to τ_p^A, assigning the latter the original probability p of going from s_1 to s_2 (Figure 8(b)). We also set $L^A(s_h) = L(s_1)$.

Let G_p be a PAG and G_p^A be the corresponding APAG. An execution fragment $e = s_0 s_1 \cdots s_n$ in G_p^A is called *proper* if the start and end states (s_0 and s_n) are observable states. Let e be a proper execution fragment of G_p^A. We define e^{obs} by removing hidden states and hidden transitions from e, i.e., restricting the execution to observable states and transition. Consider an execution fragment $e = s_0 s_1 \cdots s_n$. Let $S_P(e)$ be the set of probabilistic states in the set $\{s_0, \cdots, s_{n-1}\}$. Define the probability of an execution fragment e (denoted by

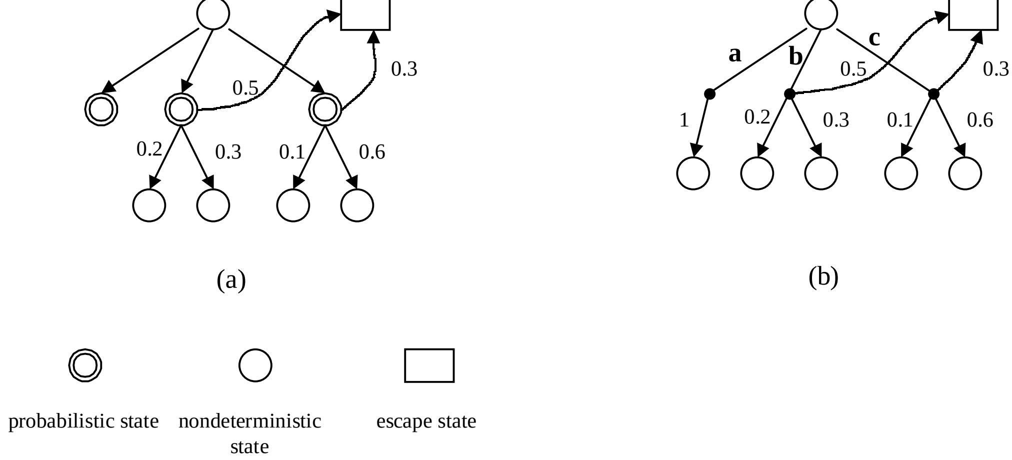

Figure 9: Converting an APAG to a MDP

$P(e)$) as

$$\prod_{s_i \in S_P(e)} P(s_i \to s_{i+1}) \ .$$

In other words, the probability of an execution fragment is the product the probabilities of the probabilistic transitions in it. The following lemma follows straight from the construction.

Lemma 7 Let G_p be a PAG and G_p^A be the corresponding APAG. Let e be a proper execution fragment of G_p^A. The following three statements are true:

1. e^{obs} is an execution fragment of G_p.
2. $P(e) = P(e^{obs})$, where the first probability is interpreted in G_p^A and the second probability is interpreted in G_p.
3. For all execution fragments e_1 of G_p there exists proper execution fragment e in G_p^A such that $e = e_1^{obs}$.

Lemma 7 clearly shows that there is a one-to-one correspondence (given by obs) between proper execution fragments of a APAG and corresponding execution fragments of a PAG. Moreover, this correspondence preserves probabilities. We have shown that APAGs have the same expressive power as PAGs, so hereafter we consider them interchangeable.

An APAG $G = (S_n, S_q, s_e, S, \tau_n, \tau_q, \pi, S_0, S_s, L)$, has a direct interpretation as an MDP $M_G = (\mathbf{X}, A, \mathcal{P}, c)$, where $\mathbf{X} = S_n$, $A = \tau_n$. That is, each action in the MDP represents a transition from a nondeterministic to a probabilistic state. Further, let $x, y \in \mathbf{X}$ and $a \in A(x)$, so that a represents a transition from x to some probabilistic state s_q in the APAG. Then we have $\mathcal{P}(x, a, y) = \pi(s_q)(y)$.

It is preferable to have all APAG success states represented explicitly as MDP states, so that we can reason about attacks in the MDP context. For this reason, we add a hidden nondeterministic state (and a transition thereto) to every probabilistic success state in the APAG. We omit proofs of equivalence of an APAG before and after this modification.

Figure 9(a) shows an example APAG, with the corresponding MDP shown in Figure 9(b). The nondeterministic transitions from the root node in the APAG are represented by the MDP actions **a**, **b**, and **c**. The leftmost leaf in the APAG is a probabilistic success state; in the MDP it is represented by the appended hidden nondeterministic state.

the reward function r depending on the questions we are trying to answer.

Let $e = s_0^n s_1^p s_1^n \cdots s_1^{n-1} s_n^p s_n^n$ be an execution fragment of the APAG G, where s_k^n and s_k^p represent nondeterministic and probabilistic states respectively. Let $mdp(e) = e^{mdp} = s_0^n t_1^n s_1^n \ldots s_n^n$, where t_i^n is the action that corresponds to the transition $s_{i-1}^n \rightarrow s_i^p$. Notice that in $mdp(e)$ probabilistic states do not occur. The proof of the following lemma follows straight from the construction.

Lemma 8 Let G be a APAG and M_G be the corresponding MDP. Let e be an execution fragment of G and $mdp(e)$ be the corresponding execution fragment in the MDP M_G. The following statements are true.

1. $mdp(e)$ is an execution fragment of the MDP M_G.
2. $P(e) = P(mdp(e))$, where $P(e)$ and $P(mdp(e))$ are interpreted in G and M_G respectively.
3. For all execution fragments e_m in the MDP M_G, there exists an execution fragment e in G such that $mdp(e) = e_m$.

6.4 Correctness of the Value Iteration Algorithm for Attack Graphs

Let $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ be a PAG, and $G^A = (S_n^A, S_q^A, s_e, S^A, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ be the corresponding APAG. Recall that the APAG G^A is obtained from the PAG G by adding hidden states whenever there is a transition between two nondeterministic or probabilistic states (see Section 6.3). An APAG $G = (S_n, S_q, s_e, S, \tau_n, \tau_p, \pi, S_0, S_s, L)$ has a direct interpretation as an MDP $M_G = (\mathbf{X}, A, \mathcal{P}, r)$, where $\mathbf{X} = S_n$, $A = \tau_n$. That is, each action in the MDP represents a transition from a nondeterministic to a probabilistic state. Further, let $x, y \in \mathbf{X}$ and $a \in A(x)$, so that a represents a transition from x to some probabilistic state s_q in the APAG. Then we have $\mathcal{P}(x, a, y) = \pi(s_q)(y)$. We first demonstrate that the *value iteration algorithm* (or *VI* for short) on the APAG G^A is simply a transformed version of the value iteration algorithm on the corresponding MDP M_G with an appropriate reward function r. After that, we prove that the value iteration algorithm on the PAG and the corresponding APAG converge to the same value. The advantage of this approach is that all the technical results in the context of value iteration in MDPs can be directly applied to value iteration in PAGs [Put94, Chapter 9].

6.4.1 Correspondence Between Value Iteration in MDPs and APAGs

Consider a MDP $M = (\mathbf{X}, A, \mathcal{P}, r)$. A *value function* is positive real valued function $V : \mathbf{X} \rightarrow \Re^+$. The value iteration algorithm uses the following equation to update the function V:

$$V(x) \quad = \quad \max_{a \in A(x)} [r(x, a) + \sum_{y \in X} \mathcal{P}(x, a, y) V(y)]$$

Technical conditions that guarantee the convergence of the value iteration algorithm can be found in [Put94, Chapter 9].

Let G^A be a APAG and M_G be the corresponding MDP. Recall that we assumed that all success states in G^A are nondeterministic states so that they are explicitly represented in the MDP M_G. Before we proceed, we need to slightly modify the MDP M_G. We add a new state s_{new} and action a_{new} to the MDP M_G. The only action allowed from s_{new} is a_{new} ($A(s_{new}) = \{a_{new}\}$) and $P(s_{new}, a_{new}, s_{new}) = 1.0$ (so by definition $P(s_{new}, a_{new}, s) = 0.0$ if $s \neq s_{new}$). Moreover, we add the action a_{new} to the action set corresponding to the success states S_f and for all $s \in S_f$ we have $P(s, a_{new}, s_{new}) = 1.0$ (so by definition $P(s, a_{new}, s') = 0.0$ if $s' \neq s_{new}$). We have the following reward function r

$$r(s, a) \quad = \quad \begin{cases} 1.0 & \text{if } s \in S_s \text{ and } a = a_{new} \\ 0.0 & \text{otherwise} \end{cases}$$

state s_{new} and 1.0 to a state in the set S_f. For states that are not in the set $\{s_{new}\} \cup S_s$ the value function V changes according to the following equation:

$$\begin{aligned} V(x) &= \max_{a \in A(x)} \sum_{y \in X} P(x, a, y) V(y) \\ &= \max_{s_q \in succ(x)} \sum_{y \in X} P(s_q \to y) V(y) \end{aligned}$$

The second equation follows from the construction of the MDP M_G from the APAG G^A. Recall that actions in the MDP correspond to the transitions from nondeterministic to probabilistic states. Next we extend the value function V to probabilistic states S_q by defining $V(s)$ (for all $s \in S_q$) as

$$\sum_{y \in X} P(s \to y) V(y) \ .$$

Notice that in an APAG only successors of a probabilistic state s are nondeterministic state, so $V(y)$ is well defined. Using this definition the value iteration algorithm can be re-written as:

$$V(s) = \begin{cases} \max_{s' \in succ(s)} V(s') & \text{if } s \in S_n \setminus S_s \\ \sum_{s' \in succ(s)} P(s \to s') V(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

The value iteration (VI) equation given above was obtained by transforming the VI equation for the corresponding MDP. Moreover, the equation we obtain is exactly the VI equation for an APAG that was provided earlier (see Section 6.2).

6.4.2 Correspondence Between Value Iteration in MDPs and PAGs

Let $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ be a PAG, and $G^A = (S_n^A, S_q^A, s_e, S^A, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ be the corresponding APAG. Recall that G^A is obtained from G by adding hidden states whenever there is a transition between two nondeterministic or probabilistic states (see Figure 8). Suppose there is a transition between two nondeterministic states s_1 and s_2 in G. In G^A, we add a new probabilistic state s_h and add transitions $s_1 \to s_h$ and $s_h \to s_2$, where the probability of the transition $s_h \to s_2$ is 1.0. Consider the i-th iteration of the VI algorithm in G. In this case, the value $V(s_2)$ in the $(i-1)$-the iteration is used to update the value of the state s_1. Now consider the value iteration algorithm in G^A. The value $V(s_h)$ of the hidden state s_h in the $(i-1)$-th iteration is used to update the value of $V(s_1)$ in the i-th iteration. It is easy to see that $V(s_h)$ in the $(i-1)$-th iteration is $V(s_2)$ in the $(i-2)$-th iteration. Therefore, hidden states *add a delay of* 1 *in the value iteration algorithm*. The case for transition between two probabilistic states is analogous.

Consider a PAG $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$. The equation for the value iteration algorithm without delay is:

$$V^i(s) = \begin{cases} 1.0 & \text{if } s \in S_s \\ \max_{s' \in succ(s)} V^{i-1}(s') & \text{if } s \in S_n \setminus S_s \\ \sum_{s' \in succ(s)} P(s \to s') V^{i-1}(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

We have added the iteration index i to the VI algorithm so that we can refer to it in the proof. The value iteration algorithm with the delay is:

$$V_1^i(s) = \begin{cases} 1.0 & \text{if } s \in S_s \\ \max\{\max_{s' \in succ(s) \cap S_n} V_1^{i-2}(s'), \max_{s' \in succ(s) \cap S_q} V_1^{i-1}(s')\} & \text{if } s \in S_n \setminus S_s \\ \sum_{s' \in succ(s) \cap S_q} P(s \to s') V_1^{i-2}(s') \ + \ \sum_{s' \in succ(s) \cap S_n} P(s \to s') V_1^{i-1}(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

Initially, both sequences start with the value functions V^0 and V_1^0 that assign 1.0 to states in S_f and 0.0 to all other states. Notice that in the value iteration algorithm for V_1^i there is delay of 1 added (the $(i-2)$-th value

$s \in S$ and $i \geq 2$:

$$V^i(s) \geq V_1^i(s) \geq V^{i-2}(s)$$

The equation given above directly follows from the monotonicity property and the equations that define value iteration.

Suppose V converges to $V_\star$ pointwise, i.e., for all $s \in S$, $V(s) \to V_\star(s)$. Next we prove that for all $s \in S$, if $V^i(s) \to V_\star(s)$, then $V_1^i(s) \to V_\star(s)$. This proves that V_1 also converges to $V_\star$. By definition of convergence, for all $\epsilon > 0$, there exists a positive integer $N(\epsilon)$ such that for all $i > N(\epsilon)$ we have

$$|V_\star(s) - V^i(s)| \; < \; \epsilon \; .$$

Assume that we are given a $\beta > 0$. It is easy to see that the limit $V_\star(s) \geq V^i(s)$ for all i (this follows from the fact that $V^i(s)$ is a monotonic sequence). Therefore, we have the following inequality

$$|V_\star(s) - V_1^i(s)| \quad \leq \quad |V_\star(s) - V^{i-2}(s)| \; .$$

The equation given above follows from the inequality $V_1^i(s) \geq V^{i-2}(s)$ for all s. Since $V^i(s) \to V_\star(s)$, there exists an $N(\beta)$ such that if $i > N(\beta)$, then

$$|V_\star(s) - V^i(s)| \; < \; \beta \; .$$

By the argument given above $|V_\star(s) - V_1^i(s)| < \beta$ for $i > N(\beta) + 2$. This proves that $V_1^i(s) \to V_\star(s)$. Conversely assume that V_1 converges to $V'_\star$. Using the inequality given below it is easy to prove that $V^i(s) \to V'_\star(s)$.

$$|V'_\star(s) - V^i(s)| \quad < \quad |V'_\star(s) - V_1^i(s)|$$

Therefore, we prove that the value iteration algorithm with and without delay converge to the same value. The VI algorithm with delay is essentially the VI algorithm on the APAG G^A, which was derived from the VI algorithm on the corresponding MDP. Therefore, the correctness of the VI algorithm on the PAG G follows.

7 Summary of Contributions and Future Work

Our foremost contribution is the automatic generation of attack graphs. Our key insight is that an attack is equivalent to a counterexample produced by off-the-shelf model checkers; the attack/counterexample is a witness to a violation of a safety property. By a small, but critical enhancement to an existing model checker, i.e., NuSMV, we can easily produce attack graphs automatically; moreover, these graphs are succinct and exhaustive. A by-product of this part of our work is showing, by example, what level of abstraction is appropriate for modeling attacks. We use simple state machine specifications to model not just intruder behavior (by a set of atomic attacks), but also normal system behavior, system administrator recovery actions, and connectivity (communication) between subsystems.

Our second most important contribution is support for a range of formal analyses of attack graphs. Security analysts use attack graphs informally for attack detection, defense, and forensics. In this paper, we explain how they can now use our minimization analysis technique on attack graphs to more precisely answer questions like "Which security measure should I deploy in order to thwart this set of attacks?" and "Which set of security measures should I deploy to guarantee the safety of my system?" To do reliability analysis, we annotate attack graphs with probabilities and then interpret them as Markov Decision Processes (MDP). Then, by using MDP algorithms such as value iteration, security analysts can more precisely answer questions like "Will deploying this intrusion detection system increase or decrease the likelihood of thwarting this type of attack?"

On the theoretical front, we have so far restricted our work to only safety (invariant) properties. To exploit the full power of model checking, we need a method of generating attack graphs for more general classes

$$\mathbf{AG}(server.user.request \rightarrow \mathbf{AF}(server.user.acesss))$$

This property would not be true if the server can be disabled using a denial-of-service attack. Another such liveness property is that a legitimate user's transaction will finish despite intruder interference. We plan to explore generation of attack graphs for universally quantified fragments of Computational Tree Logic and Linear Temporal Logic.

On the practical front, we plan to conduct larger case studies to illustrate the usefulness of automatically generating attack graphs. To make our tool suite more usable by security experts and system administrators, we see the value of building a library of specifications of atomic attacks. Our hope is that increasing this arsenal of specifications outpaces the growth in the arsenal of known attacks; we can potentially discover new, unexpected attacks, and hence identify new network vulnerabilities. Finally, we also intend to build a tool that merges our work on attack graphs with existing intrusion detection technologies. The tool is intended help security analysts evaluate and enhance the security of a network.

References

[ADP80] G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among convex optimization problems. *Journal of Computational System Sciences*, 21:136–153, 1980.

[Alt99] Eitan Altman. *Constrained Markov Decision Processes*. Chapman & Hall/CRC, 1999.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CGP00] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CLR85] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1985.

[Dac94] M. Dacier. *Towards Quantitative Evaluation of Computer Security*. PhD thesis, Institut National Polytechnique de Toulouse, December 1994.

[Dur95] Richard Durrett. *Probability: Theory and Examples*. Duxbury Press, 1995. 2nd edition.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[JW01] Somesh Jha and Jeannette M. Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, May 2001.

[NuS] NuSMV. Nusmv: a new symbolic model checker. `http://afrodite.itc.it:1024/ nusmv/`.

[ODK99] R. Ortalo, Y. Dewarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, September/October 1999.

[PS98] C.A. Phillips and L.P. Swiler. A graph-based system for network vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, 1998.

[Put94] M. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, NY, 1994.

[RA01] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 156–165, May 2001.

[SMV] SMV. Smv: a symbolic model checker. `http://www.cs.cmu.edu/ modelcheck/`.

[SPEC00] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, June 2000.

[Ste] Peter Stephenson. Using formal methods for forensic analysis of intrusion events - a preliminary examination. White Paper, available at http://www.imfgroup.com/Document Library.html.

[VS01] Alfonso Valdes and Keith Skinner. Probabilistic alert detection. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2001.