Automated Generation and Analysis of Attack Graphs

Oleg Sheyner
Computer Science Department
Carnegie Mellon University
oleg@cs.cmu.edu

Joshua Haines*
MIT Lincoln Laboratories
Lexington, MA 02420
jhaines@sst.ll.mit.edu

Somesh Jha
Computer Sciences Department
University of Wisconsin
jha@cs.wisc.edu

Richard Lippmann*
MIT Lincoln Laboratories
Lexington, MA 02420
rpl@sst.ll.mit.edu

Jeannette M. Wing*
Computer Science Department
Carnegie Mellon University
wing@cs.cmu.edu

Abstract

An integral part of modeling the global view of network security is constructing attack graphs. *In practice, attack graphs are produced manually by Red Teams. Construction by hand, however, is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes. In this paper we present an automated technique for generating and analyzing attack graphs. We base our technique on* symbolic model checking *[4] algorithms, letting us construct attack graphs automatically and efficiently. We also describe two analyses to help decide which attacks would be most cost-effective to guard against. We implemented our technique in a tool suite and tested it on a small network example, which includes models of a firewall and an intrusion detection system.*

1. Overview

As networks of hosts continue to grow in size and complexity, evaluating their vulnerability to attack becomes increasingly more important to automate. There are several tools, such as COPS [10] and Renaud Deraison's Nessus Security Scanner [9], that report vulnerabilities of individual hosts. To evaluate the vulnerability of a network of hosts, however, we also have to analyze the effects of interactions of local vulnerabilities and find global vulnerabilities introduced by the interconnections between hosts. A typical process for vulnerability analysis of a network proceeds as follows. First, we determine vulnerabilities of individual hosts using scanning tools, such as COPS and Nessus Scanner. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, we then produce *attack graphs*. 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. We can then perform further analyses, such as risk analysis [21], reliability analysis [13], or shortest path analysis [23], on the attack graph to assess the overall vulnerability of the network.

Constructing attack graphs is a crucial part of doing vulnerability analysis of a network of hosts. Construction by hand, however, is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes. Automating the process of constructing attack graphs also ensures that the attack graphs are *exhaustive* and *succinct*. An attack graph is exhaustive if it covers all possible attacks, and succinct if it contains only those network states from which the intruder can reach his goal.

We follow these steps to produce and analyze attack graphs:

1. **Model the network.**
   We model the network as a finite state machine, where state transitions correspond to atomic attacks launched by the intruder. We also specify a desired security property (e.g., an intruder should never obtain root access to host A). The intruder's goal generally corresponds to violating this property.

2. **Produce an attack graph.**
   Using the model from Step 1, our modified version of the model checker NuSMV [16] automatically pro-

*This work was sponsored by the Defense Advanced Research Projects Agency under Air Force Contract F19628-00-C0002 and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the DOD, ARO, or the United States Government.

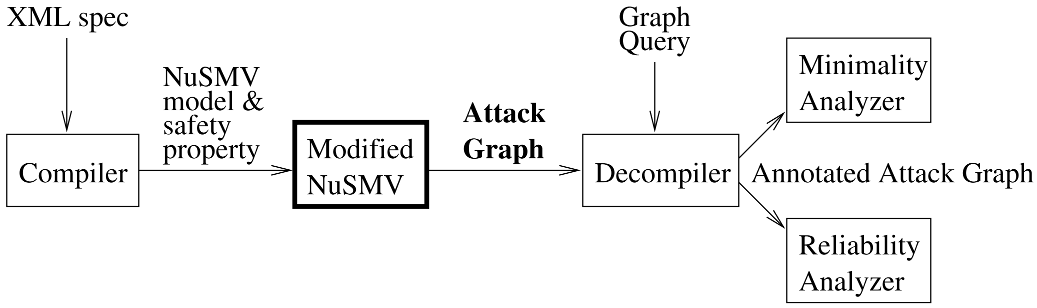

Figure 1. Tool Suite

duces the attack graph. The graphs are rendered using the GraphViz visualization package [1].

3. **Analysis of attack graphs.**
A raw attack graph is a low-level state transition diagram. To allow the domain specialist to analyze it in a meaningful way, we parse the graph and reconstruct the original meanings of the state variables as they relate to the network intrusion domain. In Section 4 we discuss two different analyses on attack graphs that quantify the likelihood of intruder success.

Figure 1 shows the architecture of our tool suite. We do not require or expect users of our tool suite to have model checking expertise. Instead of using the input language of the NuSMV model checker, a user may describe the network model and desired property in XML [5]. We built a special-purpose compiler that takes an XML description and translates it into the input language of NuSMV.

In the field of model checking, the use of fundamental data structures, such as *Binary Decision Diagrams* (BDDs) [2], enabled significant advances in the size of the systems that can be analyzed [3, 4]. More recently, model checking researchers have developed a variety of reduction and abstraction techniques to handle even larger, possibly infinite state spaces. Since our techniques build upon the underlying representation and algorithms used in model checking, we are able to leverage the recent success in that field. As model checkers handle larger state spaces, our analysis can be applied to larger networks.

Our paper reports on the following contributions to analyzing vulnerabilities in networks:

- We exhibit an algorithm for automatic generation of attack graphs. The algorithm generates exhaustive and succinct attack graphs. We provide a tool, as a part of a larger tool suite, that implements the algorithm.

- Through a small case study, we identify a level of atomicity appropriate for describing a model of the network and an intruder's arsenal of atomic attacks. The model is abstract enough to be understood by security domain experts, yet simple enough for our tool to analyze efficiently.

- Our network model includes intrusion detection components and distinguishes between stealthy and detectable attack variants. We are able to generate "stealthy" attack subgraphs (i.e. subgraphs with attacks that are not detected by the intrusion detection components). Analysis of stealthy attack subgraphs reveals the best locations for placing additional intrusion detection components.

- We describe two ways of analyzing attack graphs: an algorithm for determining a minimal set of atomic attacks whose prevention would guarantee that the intruder will fail, and a probabilistic reliability analysis that determines the likelihood that the intruder will succeed.

Paper organization. We give a detailed description of our attack graph generation algorithm in Section 2. We describe an intrusion detection case study in Section 3 and results of attack graph analysis in Section 4. We discuss related work in Section 5 and close with suggestions for future work in Section 6.

2. Attack Graphs

First, we define formally *attack graphs*, the data structure used to represent all possible attacks on a network.

Definition 1 An *attack graph* or *AG* is a tuple $G = (S, \tau, S_0, S_s)$, 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, and $S_s \subseteq S$ is a set of success states.

Intuitively, S_s denotes the set of states where the intruder has achieved his goals. Unless stated otherwise, we assume

Input:

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

Output:

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

Algorithm: $\mathit{GenerateAttackGraph}(S, R, S_0, L, p)$

(* Use model checking to find the set of states S_{unsafe} that violate the safety property $\mathbf{AG}(\neg \mathit{unsafe})$. *)
$S_{unsafe} \ = \ \mathit{modelCheck}(S, R, S_0, L, p)$.
(* Restrict the transition relation R to states in the set S_{unsafe} *)
$R^p \ = \ R \cap (S_{unsafe} \times S_{unsafe})$.
$S_0^p \ = \ S_0 \cap S_{unsafe}$.
$S_s^p \ = \ \{s | s \in S_{unsafe} \wedge \mathit{unsafe} \in L(s)\}$.
$\mathit{return}(S_{unsafe}, R^p, S_0^p, S_s^p)$.

Figure 2. Algorithm for Generating Attack Graphs

that the transition relation τ is total. We define an *execution fragment* as a finite sequence of states $s_0 s_1 ... s_n$ 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.

2.1. Constructing Attack Graphs

Model checking is a technique for checking whether a formal model M of a system satisfies a given property p. If the property is false in the model, model checkers typically output a counter-example, or a sequence of transitions ending with a violation of the property.

In the model checker NuSMV, the model M is a finite labeled transition system and p is a property written in *Computation Tree Logic* (*CTL*). In this paper, 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. In our context M is a model of the network and p is a safety property. A single counter-example shows an attack that leads to a violation of the safety property.

Attack graphs depict ways in which an intruder can force a network into an unsafe state. We can express the property that an unsafe state cannot be reached as:

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

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

AG(*network.adversary.privilege[2]* < *network.priv.root*)

We briefly describe the algorithm for constructing attack graphs for the property $\mathbf{AG}(\neg \mathit{unsafe})$. 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 [4]. 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 encapsulates the edges of the attack graph. Therefore, the attack graph is $(S_{unsafe}, R^p, S_0^p, S_s^p)$, where S_{unsafe} and R^p represent the set of nodes and set of edges of the graph, respectively; $S_0^p = S_0 \cap S_{unsafe}$ is the set of initial states; and $S_s^p = \{s | s \in S_{unsafe} \wedge \mathit{unsafe} \in L(s)\}$ is the set of success states. This algorithm is given in Figure 2.

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

2.2. Attack Graph Properties

We can show that an attack graph G generated by the algorithm in Figure 2 is exhaustive (Lemma 1a) and succinct (Lemma 1b). Whereas succinctness is a property about states in an attack graph, Lemma 1c states a similar property for transitions. Appendix A contains a proof of the lemma.

Lemma 1 (a) (Exhaustive) An execution e of the input model (S, R, S_0, L) violates the property $p = \mathbf{AG}(\neg unsafe)$ if and only if e is an attack in the attack graph $G = (S_{unsafe}, R^p, S_0^p, S_s^p)$.
(b) (Succinct states) A state s 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 contains s.
(c) (Succinct transitions) 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.

3. An 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) watches 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 (an explanation of each attack follows). 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.

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 that, the intruder needs root access on the database host ip_2.

We construct a finite state model of the 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 postconditions 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 ways the intruder can succeed.

3.1. Finite State Model

The network. We model a network as a set of facts, each represented as a relational predicate. The state of the network specifies services, host vulnerabilities, connectivity between hosts, and a remote login trust relation. Following Ritchey and Ammann [20], 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. Note that the connectivity relation incorporates firewalls and other elements that restrict the ability of one host to connect to another. Slightly abusing notation, we say $R(h_1, h_2)$ when there is a network route from h_1 to h_2. Similarly, we model trust as a binary relation $Tr \subseteq Host \times Host$, where $Tr(h_1, h_2)$ 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).

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

The intruder. The intruder has a store of knowledge about the target network and its users. This knowledge includes host addresses, known vulnerabilities, information about running services, etc. The function *plvl*$_A$*: Hosts* $\to$ {*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*. Initially, the intruder has root access on his own machine ip_a, but no access to the other hosts.

Intrusion detection system. Atomic attacks are classified as being either *detectable* or *stealthy* with respect to the Intrusion Detection System (IDS). If an attack is 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 see it.

We specify the IDS with a function *ids: Host* $\times$ *Host* $\times$ *Attack* $\to$ $\{d, s, b\}$, 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

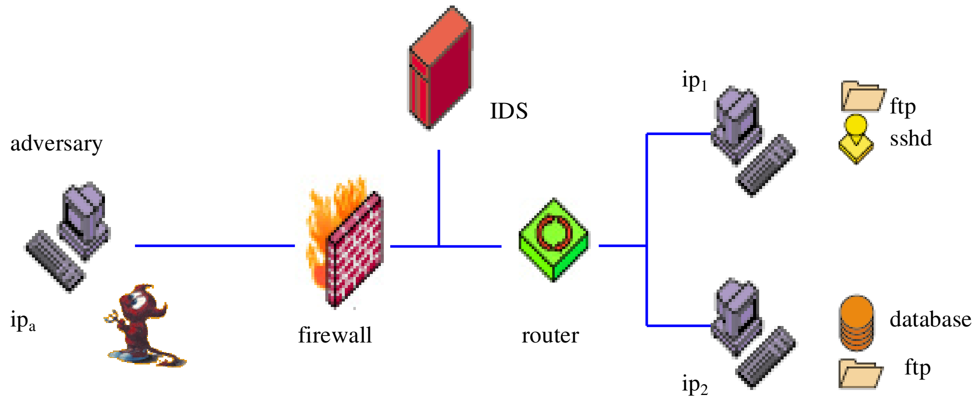

Figure 3. Example Network

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.

In our example, the paths between (ip_a, ip_1) and between (ip_a, ip_2) are monitored by a single network-based IDS. The path between (ip_1, ip_2) is not monitored. There are no host-based intrusion detection components.

Atomic Attacks. We model four atomic attacks:

1. *sshd buffer overflow*: This remote-to-root attack immediately gives a remote user a root shell on the target machine. It has detectable and stealthy variants.

2. *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. This attack is stealthy.

3. *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 point of view, it is an atomic attack. This attack is detectable.

4. *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. The intruder may transfer the necessary binary code via ftp (or scp) or create it locally using an editor such as `vi`. This attack is stealthy.

Each atomic attack is a rule that describes how the intruder can change the network or add to his knowledge about it. A specification of an atomic attack has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component lists the intruder's capabilities and knowledge required to launch the atomic attack. The *network preconditions* component lists the facts about the network that must hold before launching the atomic attack. Finally, the *intruder* and *network effects* components list the attack's effects on the intruder and on the network state, respectively. For example, the sshd buffer overflow attack is specified as follows:

attack sshd-buffer-overflow **is**
  intruder preconditions
    (User-level privileges on host S *)*
    $plvl_A(S) \geq$ user
    *No root-level privileges on host T *)*
    $plvl_A(T) <$ root
  network preconditions
    (Host T is running sshd *)*
    ssh$_T$
    (Host T is reachable from S on port* sp **)*
    $R(S, T, sp)$
  intruder effects
    (Root-level privileges on host T *)*
    $plvl_A(T) :=$ root
  network effects
    (Host T is not running sshd *)*
    $\neg$ssh$_T$
end

3.2. NuSMV Encoding

It is necessary to ensure that the model checker considers all atomic attacks in each state, so that the resulting at-

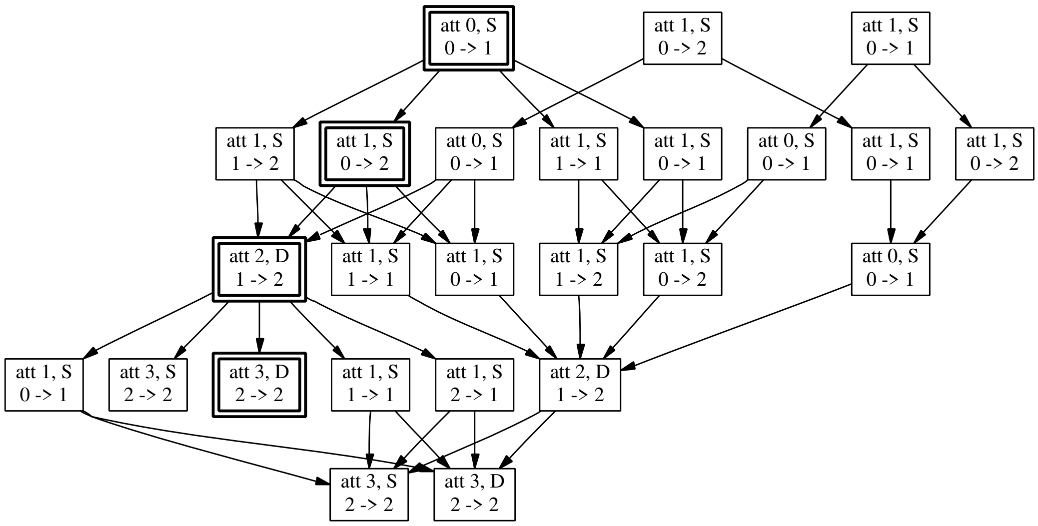

Figure 4. Attack Graph

tack graph enumerates all possible attacks. So the model checker must choose attacks nondeterministically, subject to preconditions being fulfilled. We also allow nondeterministic choices for the source host and the target host of each atomic attack. The NuSMV encoding of the model contains nondeterministically assigned state variables that specify:

- which attack (concretely, an attack number) will be tried next,
- the *source* host from which the atomic attack will be initiated,
- the *target* host of the atomic attack, and
- whether the next attack is detectable or stealthy with respect to a given intrusion detection system. This variable is set deterministically when the next attack is known to be detectable or stealthy. When the next attack has both detectable and stealthy strains, the variable is set nondeterministically.

In an effort to reduce the state space of the model, the NuSMV encoding restricts the legal states to those where the attack number, source, and target variables correspond to an enabled attack. In addition, when a variable's value is irrelevant in a particular context, we deterministically set the variable to a fixed value in that context. As an example, when the next attack is local to one host, we force the value of the variable designating the source host of the attack to be the same as the target host of the attack.

3.3. Experimental Results: Attack Graphs

Recall that the goal of our intruder is to obtain access to the database service running on host ip_2. For that, the intruder needs to get root access on ip_2 without triggering an IDS alarm. Thus, the property we want to violate (in order to get the attack graph) is that either an intruder never gets root privilege on host ip_2 or he is detected by the IDS:

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

Figure 4 shows the attack graph produced by NuSMV for this property. Each node is labeled by an attack id number (see table below), which corresponds to the atomic attack *to be attempted next*; a flag S/D indicating whether the attack is stealthy or detectable by the intrusion detection system; and the numbers of the source and target hosts. The following tables show attack and host numbers.

no.	attack
0	sshd buffer overflow
1	ftp .rhosts
2	remote login
3	local buffer overflow

no.	host
0	ip_a
1	ip_1
2	ip_2

Any path in the graph from a 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 double-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.

3.4. Performance Observations

We conducted the experiments on a Pentium III/1Ghz/1GB RAM running RedHat Linux 7.0.

The NuSMV encoding of the simple network in Figure 3 has 91 bits of state (i.e., potentially 2^{91} states), but only 101 states are reachable. The tool automatically found an appropriate BDD variable ordering under which the run time of the tool on this example is about 5 seconds.

To gauge how the run time depends on the scale of the model, we enlarged the example with two additional hosts, four additional atomic attacks, several new vulnerabilities, and flexible firewall configurations. The enlarged model has 229 bits of state and 6190 reachable states. The attack graph has 5948 nodes and 68364 edges. NuSMV took 2 hours to construct the attack graph for this model; however, the model checking part took only 5 minutes. The performance bottleneck is inside our graph generation procedure, and we are working on performance enhancements.

4. Analysis of Attack Graphs

Once we have an attack graph generated for a specific network with respect to a given safety property, the user may wish to probe it for further analysis. For example, an analyst may be faced with a choice of deploying either additional network attack detection tools or prevention techniques. Which would be more cost-effective to deploy? In doing the minimization analysis described in Sections 4.1 through 4.3, the analyst can determine a minimal set of atomic attacks that must be prevented to guarantee that the intruder cannot achieve his goal. In doing the reliability analysis described in Section 4.4, the analyst can determine the likelihood that an intruder will succeed or the likelihood that the IDS will detect his attack activity.

4.1. Minimization Analysis

Given a fixed set of atomic attacks, not all of them may be available to the intruder. Can we find a minimal set of atomic attacks that we should prevent so that the intruder fails to achieve his goal? To answer this question, we modify the model slightly, making only a subset of atomic attacks available to the intruder. 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 gets 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 the alarm. Thus, white leaf nodes are desirable for the attacker in that the 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 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 readily be 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. 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 scenario graph presented in Figure 4; it shows the ways 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).

Checking every possible subset of attacks is exponential in the number of attacks. In the next subsection, we show that finding the *minimum* set of atomic attacks which must be removed to thwart the intruder is in fact *NP*-complete. Then in the following subsection we also show how a *minimal* set can be found in polynomial-time.

4.2. Minimum and Minimal Critical Attack Sets

Assume that we have produced an attack graph corresponding to the following safety property:

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

Let $\mathcal{A}$ be the set of attacks. Let $G = (S, E, s_0, L)$ 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, and $L : E \rightarrow \mathcal{A} \cup \{\epsilon\}$

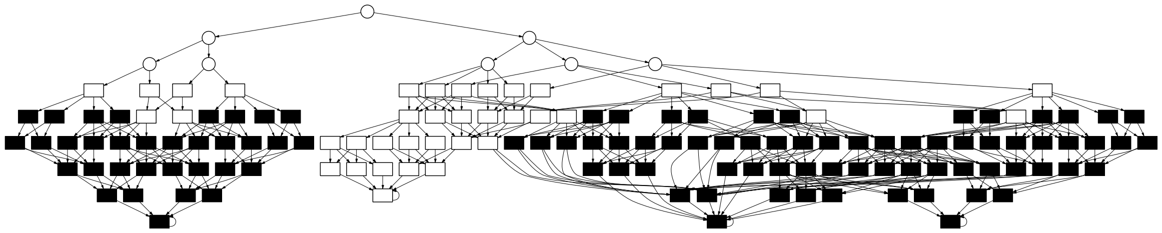

Figure 5. Attack Graph Analysis

is a labeling function where $L(e) = a$ if an edge $e = (s \rightarrow s')$ corresponds to an attack a, otherwise $L(e) = \epsilon$. 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 A. Equivalently, C is critical with respect to s if and only if every path from s to an unsafe state has at least one edge labeled with an attack $a \in C$.

A critical set corresponding to a state s is *minimal* (denoted $A(s)$) if no subset of $A(s)$ is critical with respect to s. 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 and multiple minimal critical sets corresponding to a state s. Of course, all minimum critical sets must be of the same size.

Given an attack graph $G = (S, E, s_0, L)$, consider the problem of finding a minimum critical set of attacks $M(s_0)$. We will call this problem the Minimum Critical Set of Attacks (MCSA) problem. We prove that the decision version of MCSA is *NP*-complete.

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

Proof Sketch: 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 procedure $isCritical(G, C)$ described below. Therefore, the problem is in *NP*.

To prove that the problem is *NP*-hard, we give a reduction from the *minimum cover* problem [11, Page 222]. See Appendix B for the remaining details of the proof. □

4.3. Computing Minimal Critical Sets

Consider now the problem of finding a minimal critical set $A(s_0)$ corresponding to the initial state s_0. We give an algorithm for computing $A(s_0)$ that runs in time $O(mn)$, where $m = |S| + |E|$ is the size of the attack graph G and $n = |\mathcal{A}|$ is the number of attacks. First, we describe a procedure $isCritical(G, C)$, which determines whether a set $C \subseteq \mathcal{A}$ is a critical set corresponding to the initial state s_0. This procedure runs in $O(m)$ time. We simply delete all edges from G that are labeled with an action from the set C. After that, if an unsafe state is still reachable from the initial state s_0, then C is not a critical set (because there is a path from s_0 to an unsafe state which does not use an attack from the set C). This step can be performed in $O(m)$ time using standard graph algorithms [6]. The algorithm starts with A as the empty set. At each step of the algorithm we perform the following procedure:

> if $isCritical(G, C)$ returns true, the algorithm stops and returns A. Otherwise, pick a $a \in \mathcal{A} \setminus C$ and add it to the set C.

We start with an empty set and keep adding attacks until we obtain a critical set. Notice that since $\mathcal{A}$ is a critical set, the number of steps taken by the algorithm is at most n. Each step takes $O(m)$ time, so the the worst case running time of the algorithm is $O(mn)$. If attacks have costs associated with them, then at each step we can pick an attack that has the minimum cost, i.e., pick an $a \in \mathcal{A} \setminus C$ with the minimum cost. This will bias the procedure to pick sets with lower cost.

Next, we show how the procedure described above can be carried out using model checking. Assume that the set of attacks $\mathcal{A}$ is $\{a_1, \cdots, a_n\}$. We associate a boolean variable x_i with each attack a_i. If attack a_i is activated (the intruder can use the attack), $x_i = 1$, otherwise $x_i = 0$. The variable x_i appears in the precondition corresponding to the attack a_i. Initially, all x_is are set to 0, representing that the set C is empty. Notice that if the model checker returns a counter-example, then there is a path from the initial state to an unsafe state. Recall that the specification is

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

Now in each step in the procedure, we pick an index i such that $x_i = 0$ and set $x_i = 1$. We stop the first time the model checker provides a counter-example. The set of attacks whose corresponding variables are set to 1 represents a critical set. The worst case complexity of this procedure is the same as the one given before, but in practice symbolic model checkers, such as NuSMV, will perform efficiently. Intuitively, we are using the model checker to implement the procedure $isCritical(G, C)$.

4.4. Probabilistic Reliability Analysis

When empirical information about the likelihood of certain events in the network is available, we can use well-known graph algorithms to answer quantitative questions about the attack graph. Suppose we know the probabilities of some transitions in the scenario graph. After appropriately annotating the attack graph with these probabilities, we can interpret it as a Markov Decision Process (see [12] for details).

The standard MDP *value iteration algorithm* [19] computes the optimal policy for selecting actions in an MDP that results in maximum benefit (or minimum cost) for the decision maker. Value iteration can compute the worst case probability of intruder success in an attack graph as follows. We assign all nodes where the intruder's goal has been achieved the benefit value of 1, and all other nodes the benefit value of 0. Then we run the value iteration algorithm. The algorithm finds the optimal attack selection policy for the intruder and assigns the expected benefit value resulting from that policy to each state in the scenario graph. The expected value is a fraction of 1, and it is equivalent to the probability of getting to the goal state from that node, assuming the intruder always follows the optimal policy.

We implemented the value iteration algorithm in an attack graph post-processor ("Reliability Analyzer" of Figure 1) and ran it on a slightly modified version of our example. In the modified example each attack has both detectable and stealthy variants. We assumed that for a typical network, a certain percentage of attempted intrusions is performed by sophisticated attackers who keep on top of latest IDS technology and use stealthy attack variants. We arbitrarily assigned probabilities of detecting each atomic attack as follows: 0.8 for sshd buffer overflow, 0.5 for ftp .rhosts, 0.95 for the remote login, and 0.2 for local buffer overflow. The intruder's goal is to get root access at host ip_2 while remaining undetected. Accordingly, the states where this goal has been achieved were assigned benefit value 1.

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.

The system administrator can use this technique to evaluate effectiveness of various security fixes. For instance, 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.

5. Related Work

The work by Phillips and Swiler [18] is the closest to ours. They propose the concept of attack graphs that is similar to the one described here. However, they take an "attack-centric" view of the system. Since we work with a general modeling language, we can express in our model both seemingly benign system events (such as failure of a link) and malicious events (such as attacks). Therefore, our attack graphs are more general than the one proposed by Phillips and Swiler. Recently, Swiler *et al.* describe a tool [23] for generating attack graphs based on their previous work. Their tool constructs the attack graph by forward exploration starting from the initial state. A symbolic model checker (like NuSMV) 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. Our approach can result in significant savings in space. (Swiler *et al.* refer to the advantages of the backward search in their paper [23].) More generally, the advantage of using model checking instead of forward search is that the technique can be expanded to include liveness properties, which can model service guarantees in the face of malicious activity.

Moreover, by using model checking we leverage all the advanced techniques developed in that area. For example, the *cone of influence reduction* [14] in model checking abstracts away part of the system that is not relevant to the specification. In our context, if there is a vulnerability that is not relevant to a safety property, it will not be considered during model checking. Finally, the attack graph analysis suggested by Phillips and Swiler is different from the ones presented in this paper. We plan to incorporate their analysis into our tool suite.

Templeton and Levitt [24] propose a requires/provides model for attacks. The model links atomic attacks into scenarios, with earlier atomic attacks supplying the prerequisites for the later ones. Templeton and Levitt point out that relating seemingly innocuous system behavior to known attack scenarios can help discover new atomic attacks. However, they do not consider combining their attack scenarios into attack graphs.

Dacier [8] proposes the concept of privilege graphs. Each node in the privilege graph represents a set of privileges owned by the user; edges represent vulnerabilities. Privilege graphs are then explored to construct attack state graphs, which represents different ways in which an intruder can reach a certain goal, such as root access on a host.

He also defines a metric, called the *mean effort to failure* or METF, based on the attack state graphs. Orlato *et al.* describe an experimental evaluation of a framework based on these ideas [17]. At the surface, our notion of attack graphs seems similar to the one proposed by Dacier. However, as is the case with Phillips and Swiler, Dacier takes an "attack-centric" view of the world. As pointed out above, 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. By basing our algorithm on model checking we take advantage of advances in representing large state spaces and can thus hope to represent large attack graphs. We can perform the analytical analysis proposed by Dacier on attack graphs constructed by our tool. We also plan to conduct an experimental evaluation similar to the one performed by Orlato et al.

Ritchey and Ammann [20] also use model checking for vulnerability analysis of networks. They use the (unmodified) model checker SMV [22]. They can obtain only one counter-example, i.e., only one attack corresponding to an unsafe state. In contrast, we modified the model checker NuSMV to produce attack graphs, representing all possible attacks. We also described post-facto analyses that can be performed on these attack graphs. These analysis techniques cannot be meaningfully performed on single attacks.

Graph-based data structures have also been used in network intrusion detection systems, such as *NetSTAT* [25]. There are two major components in NetSTAT, a set of probes placed at different points in the network and an analyzer. The analyzer processes events generated by the probes and generates alarms by consulting a network fact base and a scenario database. The network fact base contains information (such as connectivity) about the network being monitored. The scenario database has a directed graph representation of various atomic attacks. For example, the graph corresponding to an IP spoofing attack shows various steps that an intruder takes to mount that specific attack. The authors state that "in the analysis process the most critical operation is the generation of all possible instances of an attack scenario with respect to a given target network." Therefore, we believe that our tool can help network intrusion detection systems, such as NetSTAT, in automatically producing attack scenarios. We leave this as a future direction for research.

Cuppens and Ortalo [7] propose a declarative language (LAMBDA) for specifying attacks in terms of pre- and post-conditions. LAMBDA is a superset of the simple language we used to model attacks in our work. The language is modular and hierarchical; higher-level attacks can be described using lower-level attacks as components. LAMBDA also includes intrusion detection elements. Attack specifications includes information about the steps needed to detect the attack and the steps needed to verify that the attack has already been carried out. We are studying the possibility of converting our representation of attacks to LAMBDA.

6. Future Work

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 of properties. For example, the following liveness property states that a user will always be able to access a server whenever he wants to.

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

This property would not be true if the server can be disabled using a denial-of-service attack. We plan to explore generation of attack graphs for universally quantified fragments of Computational Tree Logic and Linear Temporal Logic.

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. Furthermore, one reason model checking has been so successful is that it discovers unknown bugs in hardware circuits and protocols[1]. Analogously, by using our tool suite based on the power of model checking techniques, we can potentially discover new, unexpected attacks, and hence identify new network vulnerabilities.

In principle, our technique is not limited to modeling attacks only—the expressive power of model checkers lets us model benign system activity as well. We believe that the ability of modern model checkers to handle more complex properties can be adapted to our tool. For example, "liveness" properties such as "a legitimate user's transaction will finish despite intruder interference" are easily specified in temporal logic and checked by a model checker. Unlike invariants, such properties cannot be handled by simple reachability analysis or other classical graph algorithms. Adapting the power of model checking to analyze such properties opens a promising research direction in automated security analysis.

References

[1] AT & T Labs. *Graphviz - open source graph drawing software. http://www.research.att.com/sw/tools/graphviz/.*

[1]The bug in the Encore Gigamax cache coherence protocol needed a minimum of 13 state transitions to lead to a state that violated the given safety property [15]; clearly this bug would be difficult for a normal human being to construct.

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

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

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

[5] W. Consortium. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/REC-xml*, February 1998.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1985.

[7] F. Cuppens and R. Ortalo. Lambda: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID)*, number 1907 in LNCS, pages 197–216. Springer-Verlag, 2000.

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

[9] R. Deraison. Nessus Scanner. *http://www.nessus.org*.

[10] D. Farmer and E. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*, 1990.

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

[12] S. Jha, O. Sheyner, and J. M. Wing. Minimization and reliability analyses of attack graphs. Technical Report CMU-CS-02-109, Carnegie Mellon University, February 2002.

[13] S. Jha and J. Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, May 2001.

[14] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[15] K. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing*. MIT Press, 1992.

[16] NuSMV. NuSMV: A New Symbolic Model Checker. `http://afrodite.itc.it:1024/~nusmv/`.

[17] 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.

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

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

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

[21] B. Schneier. Modeling security threats. *Dr. Dobb's Journal*, December 1999.

[22] SMV. SMV: A Symbolic Model Checker. `http://www.cs.cmu.edu/~modelcheck/`.

[23] L. 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.

[24] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, 2000.

[25] G. Vigna and R. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1), 1999.

A. Exhaustive and Succinct Attack Graphs

Lemma 1: (a) (Exhaustive) An execution e of the input model (S, R, S_0, L) violates the property $p = \mathbf{AG}(\neg \mathit{unsafe})$ if and only if e is an attack in the attack graph $G = (S_{unsafe}, R^p, S_0^p, S_s^p)$.
(b) (Succinct states) A state s 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 contains s.
(c) (Succinct transitions) 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) ($\Rightarrow$) 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 $0 \le k \le n$, $s_k \in S$ 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 e violates $\mathbf{AG}(\neg \mathit{unsafe})$. 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 $\mathbf{AG}(\neg \mathit{unsafe})$.

(b) ($\Rightarrow$) 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 $\mathbf{AG}(\neg \mathit{unsafe})$. 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 $\mathbf{AG}(\neg \mathit{unsafe})$. 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) ($\Rightarrow$) 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

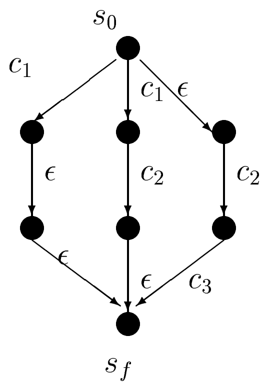

Figure 6. Attack graph corresponding to the set cover problem.

$e_2 = r_0 u_0 \ldots s_2 \ldots u_{n-1} r_n$ 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.

□

B. NP-Completeness of MCSA

Given an attack graph $G = (S, E, s_0, L)$, consider the problem of finding a minimum critical set of attacks $M(s_0)$. We will call this problem *MCSA* or the minimum critical set of attacks problem. We prove that the decision version of the problem is *NP*-complete.

Lemma 2: Assume that we are given an attack graph $G = (S, E, s_0, L)$ and an integer k. The 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 procedure $\mathit{isCritical}(G, C)$ described below. Therefore, the problem is in *NP*.

Next, we prove that the problem is *NP*-hard. The reduction is from the *minimum cover* problem [11, Page 222]. In the minimum cover problem one is given a collection C of subsets of a finite set U and a positive integer $k \leq |C|$. The problem is to determine whether C contains a cover for U of size k or less, i.e., a subset $C' \subseteq C$ with $|C'| \leq k$ such that every element of U belongs to at least one member of C'. We construct an attack graph G_C corresponding to the collection C. The set of attacks $\mathcal{A}$ is equal to C. The attack graph G_C has an initial state s_0 and a final state s_f that is unsafe. Let $U = \{u_1, \cdots u_z\}$ and $c_1, \cdots, c_m$ be an enumeration of the collection C. For each collection c_i where $i < m$ we have z new states $s_{i,1}, \cdots, s_{i,z}$. There is an edge from s_0 to all the states $s_{1,1}, \cdots, s_{1,z}$ corresponding to the collection c_1. There is an edge from $s_{i,j}$ to $s_{i+1,j}$ for all $i < m-1$ and $1 \leq j \leq z$. From each state in the set $\{s_{m-1,1}, \cdots, s_{m-1,z}\}$ there as edge to the unsafe state s_f. Label of the edge with tail $s_{i,j}$ is c_i if $u_j \in c_i$, otherwise the label is ϵ. Label of the edge with head $s_{m-1,j}$ is c_m if $u_j \in c_m$, otherwise the label is ϵ. It is easy to prove that there is a critical set of attacks C such that $|C| \leq k$ if and only if there is a cover of size less than or equal to k. ⊣

We give a short example to illustrate the reduction. Consider a set $U = \{u_1, u_2, u_3\}$. Suppose that the collection C consists of the following subsets:

$$
\begin{aligned}
c_1 &= \{u_1, u_2\} \\
c_2 &= \{u_2, u_3\} \\
c_3 &= \{u_2\}
\end{aligned}
$$

Notice that there is a cover of size 2, i.e., c_1 and c_2 form a cover. The attack graph corresponding to this problem is shown in Figure 6. The set of attacks is $\{c_1, c_2, c_3\}$. The set of attacks $\{c_1, c_2\}$ is critical because every path from s_0 to the unsafe state uses at least one edge with the label in the set $\{c_1, c_2\}$.