A Formal Approach to Sensor Placement and Configuration in a Network Intrusion Detection System

Marco Rolando, Matteo Rossi, Niccolò Sanarico, Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci, 32
Milano, Italy
{rossi,mandrioli}@elet.polimi.it

ABSTRACT

Network Intrusion Detection Systems (NIDSs) can be composed of a potentially large number of sensors, which monitor the traffic flowing in the network. Deciding *where* sensors should be placed and *what* information they need in order to detect the desired attacks can be a demanding task for network administrators, one that should be made as automatic as possible. This paper presents a logic-based model that is suitable for describing networks and intrusions. The model has been implemented in Prolog, and allows to analyze some important static properties of networks. In particular, it can be used to automatically determine, given a suitable formal definition of an attack, the location and/or the information needed by a NIDS sensor to detect the attack.

Keywords: Intrusion detection systems, formal model, network analysis, sensor configuration, sensor placement.

1. INTRODUCTION

Misuse-based Network Intrusion Detection Systems (NIDS) (e.g. Snort [9] and NetSTAT [13]) that protect networks from malicious users rely on a (possibly large) set of sensors to monitor traffic and detect attacks. Some of these attacks (which we will call *topology-dependent*) can be detected only if one takes into account the topology of the underlying network; that is, the signature of the same attack for different networks changes if the topology changes. Examples of such attacks are spoofing attacks [13], routing attacks [5][4], etc.

Detection of topology-dependent attacks is possible only if sensors are configured with the necessary (network-dependent) signatures, and also if traffic information is collected at the "right" places in the network (that is, if sensors are suitably positioned in the system). It is obvious that the "right" places to position these sensors differ from network to network, depending on their topologies. Then, determining a suitable distribution of sensors in the network and their corresponding configuration can be a daunting task (even for small networks, if the number of attacks to be detected is high), one which should be as automatic as possible.

This paper introduces a logic-based model that is suitable to describe and statically analyze computer networks. The model can be used to automatically generate incorrect configurations (that is, *signatures*) of traffic, and to determine *where* such configurations are detectable. Notice that, in our approach, network and attack modeling are dealt with in a uniform way; that is, there are no separate formalisms to describe network topology and attack conditions, but a unique, comprehensive model based on first-order logic.

The model has been implemented in Prolog [2], and at the moment includes core predicates to represent networks and few other definitions relative to incorrect packet configurations. However, it is extensible, and elements can be easily added to the core (for example to formally describe other incorrect packets) in the form of new predicates.

This paper is structured as follows: Section 2 gives an extended overview of the problem at hand, outlines the IDS architecture that was used as reference for this reserach, and presents some related works; Section 3 presents the core, logic-based model that is used to represent networks; Section 4 introduces logic predicates that describe incorrect configurations for two significant topology-dependent attacks; Section 5 shows some experimental results gathered by running the network analyzer on test network topologies; finally, Section 6 presents our conclusions, and outlines future works in this line of research.

2. PROBLEM OVERVIEW AND RELATED WORKS

Let us illustrate the problem at hand through an example. It has been shown in [13] how an IP spoofing attack, in which an attacker sends an IP packet with the source IP address of another host, can be detected by single sensors (that is, without communication with other sensors), if these are placed on suitable nodes in the network. This detection technique can recognize many different instances of IP spoofing attacks (although not all of them), provided the sensors have some notion about the topology of the network. According to this technique, detection of spoofed IP packets occurs when the following condition is satisfied:

ATTACK CONDITION 2.1 (IPSPOOFED). *Given a packet pkt with source IP address IP_{src} and source ethernet address eth_{src}, a node N (sensor) receiving pkt on one of its network interfaces nic_N recognizes pkt to be spoofed if, in the network, there is* not *a path P between the node corre-*

sponding to IP_{src} and the one of eth_{src}, such that P does not *include the link l to which eth_{src} and nic_N are attached (in other words, if* all *paths from IP_{src} to eth_{src} include l).*

This detection technique will be analyzed in greater detail in Section 4.1; notice, however, that to employ it a sensor must be able to determine if, after eliminating link l from the network, there is still a path between IP_{src} and eth_{src}.

Now, for a NIDS employing this technique to be effective, it is clear that the sensors must not be required to solve a path-existence problem every time they receive a suspect packet, since it would be too costly. If, instead, one can load onto each sensor a suitable set of offending pairs (IP_{src}, eth_{src}) (that is, malicious pairs, as defined by condition 2.1), then attack detection reduces to finding an item in this set, which can be done efficiently even at run-time, if the set is stored in a suitable data structure (for example as a red-black tree [3]).

Two problems should be apparent from condition 2.1 and the discussion above:

1. The offending (IP_{src}, eth_{src}) combinations depend on the node N (i.e. the sensor) in the network that we are taking into account. That is, different sensors can detect different attacks, and the same (IP_{src}, eth_{src}) pair can be classified as malicious by a sensor on a node N_1, but as admissible by another sensor on a different node N_2 (see also [13] for a more detailed discussion of the IP spoofing case).

2. Even for a network that is just bigger than a toy example, the number of combinations that must be loaded onto the sensors to detect the desired attacks is prohibitive, if one has to manually produce them without the aid of an automatic sensor configuration tool (one that is capable of generating the malicious combinations and send them to the sensors with the appropriate configuration primitive, which depends on the NIDS).

Notice that, although we have used IP spoofing as a reference attack to elicit and illustrate the issues above, these problems are very general, and can be found (any one of them, possibly both) in other attacks, too (see Section 4.2 as a further example).

In this paper we present a logic-based model for networks and attacks, which is suitable for carrying out automatic formal analysis of networks. In particular, thanks to our model it is possible to, among other things,

1. given some (possibly partial) information about an attack (e.g. the attacker node, the victim node, etc.), determine where in the network it is possible to put a sensor that recognizes that attack;

2. given a network node, determine which attacks can be detected by a sensor placed on it;

3. automatically generate the malicious combinations that must be loaded onto a given sensor to detect the desired attacks.

To the best of our knowledge, the current theory and practice of sensor placement for Network Intrusion Detection systems relies heavily on common sense and generic principles (which, for example, suggest sensors be placed near network firewalls, or near the server to be protected, etc. [6, 1]). However, we feel that a more systematic approach is necessary, especially (but not only) for attacks that can be detected only if sensors are deployed in certain locations in the network (as in the IP spoofing scenario described above), or only if network-specific topology information is taken into account. For these kinds of attacks, the generic guidelines currently employed in practice are not enough, and a precise analysis is in order, one that matches the attack(s) we are protecting against, with the layout of the network being protected. Such an analysis cannot be done by hand, but must be supported by software tools, which allow the security officer to gather, for each relevant attack, the necessary network-specific information for its detection (for example, on which network nodes we must place sensors to detect certain attacks, an so on). As far as we know, however, neither commercial, nor academic NIDSs offer functionalities that support this kind of analysis, yet, and users are still asked to do most of the sensor planning and configuration by hand.

With respect to existing literature and practice, the logic-based approach presented in the rest of this paper is different in that it allows systematic network analysis and supports automatic NIDS planning and configuration.

To conclude this brief overview of existing literature, let us notice that formal approaches have been applied, among other things, to the analysis of network vulnerabilities [7, 8, 11] and to automatically generate attacks that compromise a certain property of the system [12]. The work presented in this paper, however, differs from them in that it uses a formal approach to tackle another problem, that of NIDS sensor positioning and configuration.

3. THE LOGIC-BASED NETWORK MODEL

We represent the network as a tripartite undirected graph $N = \langle \mathcal{H}, \mathcal{I}, \mathcal{L}, \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{H}$, $\mathcal{I}$ and $\mathcal{L}$ are the sets of nodes of the graph, while $\mathcal{A} \subseteq \mathcal{H} \times \mathcal{I}$ and $\mathcal{C} \subseteq \mathcal{I} \times \mathcal{L}$ are the sets of edges. $\mathcal{H}$ is the set of *hosts* of the network, $\mathcal{I}$ is the set of possible IP addresses, and $\mathcal{L}$ is the set of physical *links*. $\mathcal{A}$ is a relation that associates network hosts with IP addresses, and describes which are the IP addresses assigned to the network interfaces of a host. $\mathcal{C}$ is a relation between the IP addresses and the physical links of the network and defines which are the IP addresses of the network interface cards (NICs) that are physically connected to the link. Every host of set $\mathcal{H}$ must be associated with at least one IP address of set $\mathcal{I}$ through relation $\mathcal{A}$, and every link of set $\mathcal{L}$ must be connected with at least two different IP addresses of set $\mathcal{I}$ through relation $\mathcal{C}$.

Figure 1 shows an example of network. From the graphical point of view, we represent hosts as squares (e.g. host RouterA), IP addresses as circles (e.g. ipRA1 associated with RouterA), and links as rectangles (e.g. l1 connected to ipRA1). Informally speaking, a "router" is a host that is associated with more than one IP address (e.g. RouterA). For simplicity and brevity, hosts that are not routers (that is, that have only one IP address) are graphically represented only by their lone IP address, without the attached square (e.g. the host associated with IP address ipA2).

We use logic predicates *host* and $iface$ to describe the structure of networks (basically, to define sets $\mathcal{H}$, $\mathcal{I}$, $\mathcal{L}$ and relations $\mathcal{A}$ and $\mathcal{C}$). Predicate *host* defines the hosts of the network and their associated addresses: In practice, it describes sets $\mathcal{H}$ and $\mathcal{I}$, and relation $\mathcal{A}$. *host* is defined in the

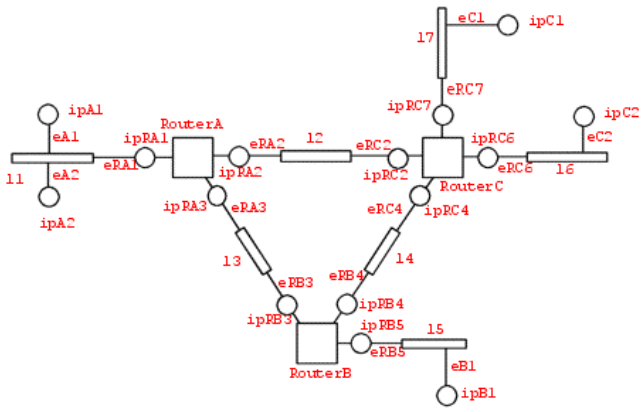

Figure 1: An example of graph representing a network

following way:

$$host(hs, ip) \triangleq (hs \in \mathcal{H}) \wedge (ip \in \mathcal{I}) \wedge ((hs, ip) \in \mathcal{A}) \quad (1)$$

For simplicity and ease of use, we define also a second version of predicate *host* (unsing an overloaded syntax), which has only one argument; this version is suitable for describing hosts that are associated with only one IP address (which can then be identified with their lone IP address) and is defined as follows (where $\exists!$ stands for "exists unique"):

$$host(ip) \triangleq (ip \in \mathcal{I}) \wedge (\exists! hs \in \mathcal{H} : (hs, ip) \in \mathcal{A}) \quad (2)$$

Predicate $iface$ has three arguments, and describes physical network interfaces. In essence, it defines relation $\mathcal{C}$ (i.e. it states which IP addresses are directly connected to a physical link), and matches ethernet addresses with IP addresses. More precisely, predicate $iface$ is defined as follows:

$$iface(ip, l, e) \triangleq ip \in \mathcal{I} \wedge l \in \mathcal{L} \wedge e \in \mathcal{E} \wedge (ip, l) \in \mathcal{C} \wedge e = P(ip, l) \quad (3)$$

where $\mathcal{E}$ is the set of all possible ethernet addresses, and $P : \mathcal{I} \times \mathcal{L} \rightarrow \mathcal{E}$ is a function associating connections (i.e. elements of relation $\mathcal{C}$) to their corresponding ethernet address.

The structure of a network can be precisely described using only predicates *host* and $iface$. For example, the structure of the subnetwork of Figure 1 composed of hosts RouterA, ipA1 and ipA2 (where ipA1 and ipA2 are associated with only one IP address each, and thus are identified with them) is captured by the predicates of Table 1.

$host(ipA1)$	$iface(ipA1, l1, eA1)$
$host(ipA2)$	$iface(ipA2, l1, eA2)$
$host(RouterA, ipRA1)$	$iface(ipRA1, l1, eRA1)$
$host(RouterA, ipRA2)$	
$host(RouterA, ipRA3)$	

Table 1: Description of subnetwork A of Figure 1

Predicates such as those of Table 1 are the known *facts* of the modeled network, and represent the *knowledge base* on which the analysis is founded.

From the basic *host* and $iface$ predicates we define other, derived predicates that describe aspects of the network that can be inferred from the initial model. For example, we define predicates $isIP$ and $isLink$, which describe sets $\mathcal{I}$ and $\mathcal{L}$, respectively, in the following way:

$$\begin{aligned} isIP(ip) &\equiv host(ip) \vee \exists hs(host(hs, ip)) \\ isLink(l) &\equiv \exists ip, e(iface(ip, l, e)) \end{aligned} \quad (4)$$

For example, from the definitions of Table 1 we can infer $isIP(ipA1)$, $isIP(RA1)$, $isLink(L1)$ (that is, ipA1 and ipRA1 are IP addresses, while l1 is a link).

We also introduce derived predicate *connected*, which describes which pairs of nodes in the network (hosts, IP addresses and links) are directly connected (that is, *adjacent*) with each other. *connected* is defined as follows:

$$\begin{aligned} connected(a, b) \equiv\ & host(a, b) \vee host(b, a) \vee \\ & \exists e(iface(a, b, e) \vee iface(b, a, e)) \end{aligned} \quad (5)$$

For example, from Table 1 we can infer, among other things, $connected(ipRA1, RouterA)$ and $connected(l1, ipA1)$.

From predicate *connected* we define predicate path, which identifies paths between any two nodes in the network. More precisely, $\mathsf{path}(a, b, p)$ is true if and only if p is a path between nodes a and b. In an intuitive manner, we define a path between nodes a and b as a sequence of adjacent nodes such that a and b are, respectively, the first and last nodes of the sequence (for example, in the network of Figure 1 [RouterA, ipRA3, l3, ipRB3, RouterB] is a path between nodes RouterA and RouterB). Formally, predicate path is defined as follows:

$$\begin{aligned} \mathsf{path}(a, b, p) \equiv\ & (a = b \wedge p = []) \vee \\ & (connected(a, b) \wedge p = [a, b]) \vee \\ & \exists c, p_c (c \neq b \wedge connected(a, c) \wedge \\ & \quad \mathsf{path}(c, b, p_c) \wedge p = a \oplus p_c) \end{aligned} \quad (6)$$

where [] represents and empty sequence, $[a, b]$ represents a sequence of two elements (a first and then b), and $a \oplus p_c$ is the sequence obtained by adding c at the beginning of sequence p_c. With this definition, however, predicate path is true also for all paths that have loops; for example, with reference to the network of Figure 1, formula (6) is true for all paths between ipA1 and ipC1 that travel n (with $n \in \mathbb{N}$) times over the loop passing through nodes RouterA, RouterB and RouterC. To ignore all paths with loops, we modify the definition of predicate path as follows:

$$\begin{aligned} &\mathsf{path}(a, b, excl, p) \equiv \\ &\quad (a = b \wedge p = []) \vee \\ &\quad (connected(a, b) \wedge b \notin excl \wedge p = [a, b]) \vee \\ &\quad \exists c, p_c (c \neq b \wedge c \notin excl \wedge connected(a, c) \wedge \\ &\qquad \mathsf{path}(c, b, \{a\} \cup excl, p_c) \wedge p = a \oplus p_c) \end{aligned} \quad (7)$$

In the new definition, an argument is added, which corresponds to the nodes of the network that must be excluded from the path for the latter to be acceptable. Then, $\mathsf{path}(a, b, excl, p)$ is true if and only if p is a loop-free path from node a to node b that does not include any nodes from the set $excl$ (that is, such that $excl \cap p = \varnothing$).

Notice that paths p satisfying predicate path defined by formula (7) are loop-free by construction. In fact, since we defined what constitutes a path in formal terms (i.e. through mathematical formulae), we can actually *prove* that paths

satisfying (7) are loop-free. The proof can be done by induction over the length of the path: In the base case, we easily determine that paths with only 0 or 2 nodes are loop-free (this follows directly from the definition of path, and more precisely from the first two subformulae of the disjunction on the right-hand side of formula (7)). In the inductive case, let us assume that, in the third subformula of the right-hand side of (7), path p_c of length $n-1$ is loop-free; then, from the definition of path we obtain that $a \notin p_c$, so path p (with $p = a \oplus p_c$), which has length n, remains loop-free.

Before concluding this section, let us remark that through predicate *host* one can model not only single hosts, but also entire subnetworks, if one is not interested in representing their detailed layout, but only their interaction with the outside world. For example, in Figure 1, nodes ipA1 and ipA2 might in fact stand for two subnetworks, which are connected to the rest of the system through a common router, RouterA. In a similar way, we might decide not to represent nodes ipC1 and ipC2, and use only node RouterC to model the entire subnetwork connected to it. The predicates presented above do not mandate any fixed modeling granularity (a *host* can be anything that is connected to the rest of the network, from single computers, to LANs, to WANs), so the user has maximum flexibility in deciding at what level of detail the network should be described.

3.1 Prolog implementation

To perform automatic analysis of networks, the logic predicates introduced above have been implemented in Prolog [2]. The implementation is mostly straightforward, and will be briefly outlined in the rest of this section.

Predicates *host* and $iface$ are the known facts of the network, so they must be postulated in the Prolog description, and are dependent on the network being analyzed. For example, the following Prolog declarations correspond to two of the logic formulae shown in Table 1:

```
host(ipA1).
host(routerA, ipRA1).
```

Predicates $isIP$ and $isLink$, instead, are defined as follows:

```
isIP(IP) :- host(IP).
isIP(IP) :- host(_,IP).
isLink(L):- host(_, L, _).
```

In the above declarations, identifiers beginning with a small letter (for example ipA1) are, in Prolog terms, *atoms*, while identifiers beginning with a capital letter are *variables* (for example IP). The underscore (_) is a special, *anonymous* variable, which, informally speaking, can be interpreted as "there is some element such that". For example, the Prolog definition of predicate *isLink* states that an element L is a link if and only if L appears as the second argument in a *host* declaration, with some other (unspecified) elements as first and third arguments.

Predicate path defined by formula (7), instead is implemented as follows (in Prolog syntax \== corresponds to $\neq$, while \+member is $\notin$):

```
path(A,B,Visited,[B|[A|Visited]]) :- connected(A,B),
                                     \+member(B, Visited).
path(A,B,Visited,Path):- connected(A,C), C \== B,
                         \+member(C,Visited),
                         path(C,B,[A|Visited],Path).
path(A,A,Visited,Visited).
```

The Prolog clauses above basically correspond to recursive algorithm FindPaths, which determines all loop-free paths between any two nodes of the network.

Algorithm 1: *FindPaths*

input : current node A, destination node B, sequence V of visited nodes (from last to first visited).

output: paths P composed by paths $\{P_{BA,i}\}$ between A and B (but with nodes in reverse order), suffixed with sequence V ($\{P_{BA,i} \oplus V\}$).

begin

1 Add node A at the beginning of sequence V ($V \longleftarrow A \oplus V$).;
$P \longleftarrow \varnothing$;
2 **if** $A = B$ **then return** V;
3 $AE \longleftarrow$ the set of edges that start from A and *do not* end in a node belonging to V;
4 **while** $AE \neq \varnothing$ **do**
Pick an edge from AE, let N be the node at its other end;
$P \longleftarrow P \cup FindPaths(N, B, V)$;
return P

end

Appendix A evaluates the theoretical computational complexity of algorithm FindPaths. In the most general case the algorithm, which finds *all* loop-free paths, has complexity $O(n!)$; however, as Appendix A shows, under certain hypotheses the complexity of FindPaths becomes polynomial.

Moreover, as shown in the next section, when modeling attacks it is often unnecessary to state *universal* properties of paths, but, rather, *existential* ones are sufficient. This turns the path finding problem into one of *reachability* (i.e. "if there exists a path ...", instead of "if for all paths ..."), which is considerably simpler, and can be solved with algorithms that are polynomial in the number of the nodes of the network [3], no matter its topology.

4. ATTACK MODELING

Armed with the basic predicates presented in Section 3, we can now formally express the conditions that describe network attacks. This section illustrates our modeling approach on two network attacks, namely IP spoofing and an attack to the Routing Information Protocol (RIP). Section 5 presents some examples of how the Prolog implementation of the model can be used to carry out network analysis for sensor placement and to generate sensor configurations in the sense outlined in Section 2.

4.1 Representing IP spoofing

Following the lead of [13], we can determine that an IP packet *pkt* is spoofed if it satisfies condition 2.1 of Section 2[1]. Condition 2.1 can be formally expressed using the following logic formula:

[1]In fact, condition 2.1 differs slightly from the one given in [13], as it requires the whole link l to be disconnected from the network, not only nic_N from l. It can be shown that the condition presented here covers more IP spoofing cases than the original formulation, but, for the sake of brevity, we will not delve into the details of this topic.

$$\begin{array}{l}\mathsf{ipspoofed}(sensor,\ srcIP,\ srcEth) \triangleq \\ \quad \exists\ srcEthIP,\ l,\ srcIPH(\\ \qquad \mathsf{monitors}(sensor,\ srcEth,\ srcEthIP,\ l) \quad \wedge \\ \qquad \mathsf{hostOf}(srcIP,\ srcIPH) \quad \wedge \\ \qquad \nexists\, p(\ \ \mathsf{path}(srcIPH,\ srcEthIP,\ [l],\ p)\ \ ))\end{array} \tag{8}$$

Formula (8) defines predicate ipspoofed using predicate path of Section 3, plus two other derived predicates monitors and hostOf. Informally speaking, $\mathsf{monitors}(p, e, ip, l)$ is true if and only if node p (with p either a host or a link) can sniff the traffic transiting on link l, on which IP ip (with physical address e) is connected, and p is not the host of ip (for example, for the network of Figure 1 predicate $\mathsf{monitors}(ipC2, eRC6, ipRC6, l6)$ holds; in fact, node ipC2 can sniff the traffic on link l6, on which IP ipRC6, which has ethernet address eRC6, is connected). Predicate $\mathsf{hostOf}(ip, hs)$, instead, is true if and only if host hs is associated with IP address ip. Then, formula (8) states that predicate $\mathsf{ipspoofed}(sensor, srcIP, srcEth)$ is true if and only if $sensor$ can sniff packets coming from physical address $srcEth$ (which is connected to link l and is associated with IP address $srcEthIP$), and there is no path p between the host $srcIPHost$ of IP address $srcIP$ and node $srcEthIP$ such that p *does not* include link l. That is, $\mathsf{ipspoofed}(sensor, srcIP, srcEth)$ is true if and only if a packet with source ethernet address $srcEth$ and source IP address $srcIP$ is spoofed, and a sensor on node $sensor$ can detect it.

Predicate ipspoofed has a suitable Prolog implementation `ipspoofed`, which is not shown here for the sake of brevity. Nonetheless, let us notice that, for the efficiency reasons outlined at the end of Section 3.1, subformula $\nexists p(\ldots)$ has been translated into a reachability problem, rather than an instance of algorithm FindPaths. The implementation of predicate ipspoofed can then be used for analysis purposes. For example, the following Prolog statement produces all malicious pairs $(sourceIP, sourceEthernet)$ that can be detected by host RouterA in the network of Figure 1:

```
ipspoofed(routerA, SRC_IP, SRC_ETH).
```

The statement below, instead, produces all sensors that can detect all spoofed packets that have ipC1 as source IP (i.e. packets in which someone is impersonating ipC1):

```
ipspoofed(S, ipC1, SRC_ETH).
```

4.2 Representing RIP attacks

We now turn our attention to a type of attack to RIP servers, that consists in a neighboring server sending a path advertisement of a target IP with an impossible distance (see [5] for detailed information about the attack). Informally speaking, we can formulate the attack condition in the following way:

ATTACK CONDITION 4.1 (RIPIMPOSSIBLEPATH). *Given a RIP advertisement adv received by a RIP server RIP_{dst} and coming from the IP address RIP_{src} of another RIP server that shares a link with RIP_{dst}, adv is illegal if it advertises a route (i.e. a path) from RIP_{src} to a target IP tgt with distance (i.e. path length) dist such that there are no paths of length dist between RIP_{src} and tgt.*

To formalize condition 4.1 we use a predicate ripadvertisement that states the opposite; that is, $\mathsf{ripadvertisement}(srcIP, dstIP, tgt, dist)$ is true if and only if $srcIP$ and $dstIP$ are on the same link, and, if we call $srcH$, $dstH$ and $tgtH$ the hosts associated with $srcIP$, $dstIP$ and tgt, respectively, $srcH$ and $dstH$ are both RIP servers, and there is a path of length $dist$ between $srcH$ and $tgtH$. Predicate ripadvertisement is defined as follows:

$$\begin{array}{l}\mathsf{ripadvertisement}(srcIP,\ dstIP,\ tgt,\ dist) \triangleq \\ \quad \mathsf{sameLink}(srcIP,\ dstIP) \quad \wedge \\ \quad \exists\ srcH,\ dstH,\ tgtH(\\ \qquad srcH \neq dstH \quad \wedge \quad \mathsf{hostOf}(srcH,\ srcIP) \quad \wedge \\ \qquad \mathsf{hostOf}(dstH,\ dstIP) \quad \wedge \quad \mathsf{hostOf}(tgt,\ tgtH) \quad \wedge \\ \qquad \mathsf{isRIPserver}(srcH) \quad \wedge \quad \mathsf{isRIPserver}(dstH) \quad \wedge \\ \qquad \exists\ p\ (\ \mathsf{path}(srcH,\ tgtH,\ [],\ p)\ \wedge\ length(p,\ dist)\ )\ )\end{array} \tag{9}$$

Similarly to predicate ipspoofed, predicate ripadvertisement has also a suitable Prolog implementation, which can be used to carry out automatic analysis. In this case, for example, one might decide to generate valid RIP advertisements and load them onto sensors that can monitor links between adjacent RIP routers (we assume here that RIP advertisements are disregarded if they come from IP addresses that are not on the same link as the RIP server). Then, sensors can verify the validity of a RIP advertisement by checking that the advertisement is in the set of valid ones. The Prolog command that generates all valid advertisements that can be received by a RIP server (with IP address $dstIP$) is the following:

```
ripadvertisement(SRC_IP, dstIP, TGT, D).
```

5. EXPERIMENTAL RESULTS

The Prolog-based analyzer implementing the logic predicates presented in Sections 3 and 4 has been tested on a variety of network topologies to assess its effectiveness.

First, we ran the analyzer to produce all triples (l, i, e) (with l a link, i an IP address and e an ethernet address) satisfying predicate ipspoofed; this is obtained through the following Prolog query:

```
isLink(L),ipspoofed(L, I, E).
```

The query above has been launched on two sets of topologies. In the first set, the backbone (i.e. the number of routers and their mutual connections) of the network remains unchanged, but the number of hosts varies. In the second set, every router is connected to exactly 4 non-routers, but the number of routers composing the backbone increases.

The backbone of the network topologies of the first set of tests is composed of 4 routers and is one link short of being completely connected (the graphs describing these topologies are not presented here for the sake of brevity; they can be found in [10]); the total number of hosts (routers included) varies from 10 to 100. Table 2 shows the results of these tests.[2] In this case, the test duration increases roughly with n^3 (with n the total number of hosts in the network), and the analyzer takes around 7 minutes to produce *all* possible attack configurations detectable on *every* link for the network with 100 hosts.

[2]All tests have been run on a PC with an Athlon XP 3000+ processor and 1GB of RAM.

Total hosts	Routers	Links	Test Duration (sec.)
10	4	10	0.11
20	4	18	1.11
50	4	40	27.30
100	4	84	427.73

Table 2: First experiment: IP spoofing, fixed backbone, variable number of hosts

Total hosts	Routers	Links	Test Duration (sec.)
25	5	30	3.94
30	6	39	10.05
35	7	49	23.45
40	8	60	50.67
45	9	72	97.75
50	10	83	180.61

Table 3: Second experiment: IP spoofing, variable completely connected backbone

In the second set of tests, the networks are composed of a backbone of completely connected routers, and every router is connected to exactly 4 non-router hosts. Table 3 shows the durations of the new tests. In this case the complexity of the problem remains tractable (it is less than exponential), a consequence of the fact that in the implementation of predicate ipspoofed the path existence subformula has been translated into a reachability condition.

In the last set of tests, we ran the analyzer to produce all valid RIP advertisements of a given network. This is achieved through the following Prolog query:

```
ripadvertisements(Src, Dst, Tgt, D).
```

In addition, for these tests the topology of the analyzed network does not change, and corresponds to the testbed presented in [5]. While the number of nodes in the network does not vary, with each test we increase the number of hosts that run a RIP server. The durations of the test runs are shown in Table 4. The increase in the test duration is clearly linear with the number r of RIP routers deployed in the network, and even in the case in which all hosts with more than one associated IP address run a RIP server (last row of Table 4) the whole computation takes less than 25 seconds.

To conclude this section, we feel that the experimental results presented above are very promising, and suggest that the modeling and analysis technique on which the Prolog analyzer is based can be effectively used on real-life networks.

6. CONCLUSIONS

This paper introduced a logic-based approach for modeling and analysis of networks, which supports automatic placement and configuration of sensors of a Network IDS. Our approach is based on a very small set of logic predicates, which are suitable for modeling the topology of networks, and a set of derived predicates that express relevant properties of the modeled networks. In particular, the derived predicates introduced in this article formally express anomalous configurations of IP packets, which correspond to possible intrusion efforts by a malicious user. Finally, the Prolog implementation of the logic model has been used to perform automatic network analysis, and to algorithmically produce lists of anomalous packets that can be loaded onto NIDS sensors to actually perform attack detection. Experimental results on some test networks have shown the effectiveness and usability of the logic approach.

Total hosts	RIP Routers	Test Duration (sec.)
22	3	3.72
22	6	7.11
22	10	15.92
22	16	24.23

Table 4: Third experiment: RIP attack, fixed network, variable RIP routers

We claim that our model is very general and extensible, and is well-suited to describe a large variety of intrusion attacks, provided the necessary derived logic predicates are defined. In fact, future works in this line of research will focus on extending the set of modeled attacks, by introducing new derived predicates in addition to those presented here. In this respect, we will explore the possibility of defining new predicates, with the intent of unifying the misuse-oriented view underlying our current model with a more anomaly-oriented approach. For example, with respect to the IP spoofing attack expressed by condition 2.1, a more anomaly-oriented approach might consider as suspect packets coming from seldom-used paths; then, a suitable predicate $suspect\_ipspoofed$ might capture this configuration in formal terms.

In addition, we plan to enrich our logic approach with temporal operators, in order to be able to express time-related attack conditions and temporal constraints on the events occurring in a network.

Moreover, the modeling technique introduced in this paper is independent of any specific NIDS that one might use in practice. Through our Prolog-based analysis tool one can produce sets describing malicious IP packets in a NIDS-independent way (in fact, as tuples of relevant data), so the next natural step is to devise algorithms to translate these sets in NIDS-specific information. Our efforts, then, will also focus on mapping the information generated by our analyses onto NIDS-specific mechanisms, such as STAT [14] scenarios or Snort [9] signatures.

Finally, the efficency of the prototype analyzer can be improved in several ways. First, the Prolog code has not been optimized, so better performance can be achieved simply by careful re-coding of the predicates. Second, we plan to realize different, non Prolog-based implementations of the analyzer, to better exploit existing tools and techniques for the solution of graph-related problems.

Acknowledgments

The authors would like to thank Giovanni Vigna and Richard Kemmerer for their support, their help, and their useful comments.

7. REFERENCES

[1] E. Carter, editor. *Cisco Secure Intrusion Detection System*. Cisco Press, 2002.

[2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog. Using the ISO Standard*. Springer-Verlag, 2003.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[4] C. Krügel, D. Mutz, W. Robertson, and F. Valeur. Topology-based detection of anomalous BGP messages. In *Proc. of RAID*, pages 17–35, 2003.

[5] V. Mittal and G. Vigna. Sensor-based intrusion detection for intra-domain distance-vector routing. In *Proc. of CCS 2002*, pages 127–137, 2002.

[6] S. Northcutt and J. Novak. *Network Intrusion Detection*. New Riders Publishing, 2002.

[7] X. Ou, S. Govindavajhala, and A. W. Appel. MulVAL: A logic-based network security analyzer. In *Proc. of 14th USENIX Security Symposium*, pages 113–128, 2005.

[8] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Symp. on Security and Privacy*, pages 156–165, 2000.

[9] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 13th Conf. on Systems Administration*, pages 229–238, November 1999.

[10] Matteo Rossi. On probe placement and configuration in an intrusion detection system. Technical report, DEI, Politecnio di Milano, 2002.

[11] G. Rothmaier and H. Krumm. A framework based approach for formal modeling and analysis of multi-level attacks in computer networks. In *Proc. of FORTE*, pages 247–260, 2005.

[12] O. Sheyner, J. W. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symp. on Security and Privacy*, pages 273–284, 2002.

[13] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[14] G. Vigna, F. Valeur, and R. A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *Proc. of FSE/ESEC*, pages 88–97, 2003.

APPENDIX

A. COMPLEXITY OF PATH FINDING

In this section, let us analyze the theoretical complexity of algorithm FindPaths shown in Section 3.1.

Blocks 3-4 are the core of algorithm FindPaths; they state that in each recursive instance of FindPaths all the unexplored branches of the current path, which starts from the current node A, are examined one by one. Now, in a completely connected graph of n nodes, the number of edges ending in a node is $n-1$. As a consequence, running FindPaths on a completely connected network would generate $(n-1)*(n-2)*\ldots*2*1=(n-1)!$ recursive calls, thus resulting in a complexity of $O(n!)$. However, real-life computer networks are never completely connected, so the real complexity of the algorithm is much smaller than $O(n!)$. In the rest of this section we will show how, under certain hypotheses, the worst-case complexity of algorithm FindPaths in much more favorable than $O(n!)$ (in fact, we will show it to be polynomial).

Let us assume, for simplicity, the following:

HYPOTHESIS A.1. *Every node of the network is connected to* exactly *three edges.*

Under hypothesis A.1, set AE cannot contain more than two edges, since the node that was visited right before A is already marked, and the corresponding edge does not satisfy the condition of line 3. As a consequence, every call of FindPaths applies itself recursively twice on a subnetwork with one node less than the original one; this leads us to conclude that a first upper bound for the complexity of the algorithm is $O(2^n)$ (which is already an improvement over $O(n!)$). A closer analysis yields even better results. In fact, the complexity of algorithm FindPaths is $O(2^n)$ if *in every instance* we can choose between two edges. In statement 1, however, current node A is added to the set V of visited nodes; this implies that, from hypothesis A.1, the subnetwork on which FindPaths is recursively applied contains one node and two edges (those ending in A) less than the original one. Then, at the i^{th} instance of the algorithm we would have pruned from the original network 2^i edges. Now, from hypothesis A.1, the number of edges of the network is linear ($O(n)$) with respect to the number of nodes. Then, after roughly $\lfloor \log_2(l) \rfloor$ instantiations of algorithm FindPaths, AE does not contain two edges, but one at most, since most of the nodes (and relative edges) have already been visited. As a consequence, the complexity of FindPaths on a network of n nodes that satisfies hypothesis A.1 is roughly $O((l-2^{\lfloor \log_2(l) \rfloor})2^{\lfloor \log_2(l) \rfloor})$, which is $O(n^2)$ (l is $O(n)$).

This result can be generalized to the following property:

PROPERTY A.2 (COMPLEXITY OF ALGORITHM FindPaths). *Given a network N such that the number of edges connected to a node is at most k, the complexity of algorithm* FindPaths *applied to N is $O(n^{k-1})$.*

Notice that in the case of a completely connected graph, $k=n$ and property A.2 gives an overestimation of the real complexity of the algorithm.[3]

In addition, if the network is made of a core of strongly connected routers, plus other hosts loosely connected with these (that is, with only one or, at the very maximum, two paths to the routers, as in Figure 1), the complexity of algorithm FindPaths is better still. In fact, in this case what is relevant is how the routers are connected with each other, and the complexity of FindPaths becomes $O(r^k)$, with r the number of routers of the network.

To conclude this section, notice that algorithm FindPaths finds *all loop-free* paths between any two nodes in the given network. If, on the other hand, we limit the goal to simply finding *one* (any) of the many paths between two nodes in the network (that is, we switch to the problem of path *existence* between two nodes, as is the case for predicate ripadvertisement of Section 4.2), algorithm FindPaths can stop after the first path is found (that is, after the first time line 2 is hit). For this simplified version of the algorithm, the complexity is $O(n^2)$ no matter the topology of the network.

[3]This is theoretically acceptable, since an algorithm with complexity $O(n!)$ is also $O(n^n)$.