

SRI International

SDL Technical Report SRI-SDL-04-04 • September 7, 2004

Automatic Analysis of Firewall and Network Intrusion Detection System Configurations

Tomás E. Uribe
Steven Cheung



This research is sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency.

DISTRIBUTION STATEMENT "A": Approved for public release; distribution is unlimited.

Abstract

Given a network that deploys multiple firewalls and network intrusion detection systems (NIDSs), ensuring that these security components are correctly configured is a challenging problem. Although models have been developed to reason independently about the effectiveness of firewalls and NIDSs, there is no common framework to analyze their interaction. This paper presents an integrated, constraint-based approach for modeling and reasoning about these configurations. Our approach considers the dependencies among the two types of components, and can reason automatically about their combined behavior. We have developed a tool for the specification and verification of networks that include multiple firewalls and NIDSs, based on this approach. This tool can also be used to automatically generate NIDS configurations that are optimal relative to a given cost function.

Keywords: Formal specification and analysis, network intrusion detection, firewalls, network configuration and security

Contents

1	Introduction	1
2	Preliminaries	2
3	Implementation	7
4	Examples: Corporate Scenario	10
5	Applications: Configuring NIDS and Firewalls	11
5.1	NIDS Ruleset Reduction	11
5.2	Checking Firewall Configurations at Runtime	12
5.3	False Alarm Reduction	12
5.4	Generating NIDS Configurations from Event Specifications	13
5.5	Generating Optimal NIDS Configurations	14
5.6	Detecting Multistep Attacks	14
6	Extensions	17
6.1	Packet Transformations	17
6.2	Encrypted Traffic	18
7	Related Work	18
8	Conclusion	19

List of Figures

1	Corporate network: router-area graph	3
---	--	---

1 Introduction

Correctly configuring a network that includes multiple servers, routers, firewalls, and other security mechanisms is a challenging task, particularly when the intended properties are never defined, or only vaguely specified. To address these problems, a number of methods for formalizing network configurations and policies have been proposed, together with tools for ensuring that a particular configuration satisfies a given policy specification (e.g., [12, 13]). We build on this work, extending it by including specifications and requirements for the network intrusion detection systems (NIDSs). We can then reason about network topology, filtering, and NIDS placement and configuration, within a single framework. To our knowledge, this is the first attempt to reason about joint NIDS and firewall configurations using a formal framework.

An example of the questions we want to answer is: For a given topology, NIDS, and firewall configurations, is a particular class of “bad” events always detected? Can false alarms be raised for a given set of “good” events? How should the NIDSs be configured to minimize false alarms, maximize the number of detected bad events, and minimize a given deployment cost?

The effectiveness of the proposed method, and the cost of applying it, must be considered. One of our goals is to produce a “lightweight” formal tool, whose cost ranges from minimal (a “one-click” sanity check of the network that will report inconsistencies, redundancies, and incomplete configurations) to moderate (automatically checking security policies specified by a designer). As with any formal method, the guarantees that it provides are relative to the accuracy of the models and specifications, and the level of abstraction used.

For example, we note that our IDS model does not characterize low-level NIDS behavior such as packet/stream reassembly and payload encoding normalization, used to counter NIDS evasion and denial of service attacks (e.g., [19]). Instead, we assume that the NIDS’s used are robust against these attacks and can keep up with the traffic load. However, even an idealized model can help uncover design flaws, or find improvements.

Formalizing network behavior can help make systems more secure, by identifying potential weaknesses in the configuration. It can also be used to minimize the allowed services and connections, while still maintaining needed functionality, making the system less vulnerable to novel exploits and attacks [21]. Redundancy can increase the robustness of a network, but also presents difficulties for network intrusion detection. For instance, if a server has multiple network interfaces for failover or load balancing, attack packets may reach a target through multiple paths. By considering connectivity, firewall configuration, and NIDS placement, we can ensure that these packets are detected.

Outline: Section 2 presents the basic framework and definitions. Section 3 briefly describes the implementation, and a simple example of checking a network that includes NIDSs and filtering. Section 4 gives some examples of policy specifications, and NIDS configurations for enforcing them. Section 5 presents our main contribution, describing applications to different aspects of network configuration, including ruleset and false alarm reduction, checking firewalls, and automatically generating NIDS configurations. Section 6 discusses extensions to the basic formalism to capture more complex network behavior, such as IP address translation and port mappings. Sections 7 and 8 present related work and conclusions.

2 Preliminaries

Network Graph: Our formal network description language is based on that of Guttman [12]. It is expressive enough to describe interesting properties of network configurations, but simple enough to support efficient and automatic reasoning. We model a network as a directed graph, where packets travel from node to node. A useful special case is to model the network as a bipartite graph, where nodes are either routers or areas, and each edge connects a router and an area; but our tool can also handle the general case. Firewalls are modelled at the router nodes, which can filter the traffic that goes through them.

Figure 1 shows an example corporate network graph, based on that of [12, 13]. (In the figure, each edge represents two directed edges, one in each direction.) It contains five routers and six areas, and a designated host protected by a host-based firewall, such as an Autonomic Distributed Firewall (ADF) [1]. Each router has a different network interface for each area to which it is connected. Areas contain disjoint sets of hosts, which can be the source or destination of individual packets. In this model, individual hosts are not themselves nodes in the graph, but their addresses are used to constrain the packets that can flow from one area to another. If a particular host includes its own firewall, as in the case of ADFs, then it can be given its own area, as is the case for the host `home` in Figure 1.

Host Sets: Our tool uses either numeric IP addresses or symbolic host names. Sets of addresses are expressed using constraints. For instance, the hosts in `engineering` can be represented as `[129,42,2,*]`, or, if the more abstract representation is chosen, as an abstract set of three hosts, `[eng_mail_server,db_server,eng_untrusted_host]`.

Given the hosts in each area, new host sets can be symbolically defined with the usual operations on these finite sets. For this example, we have

```
defhostset(internal,union(engineering,finance)).
```

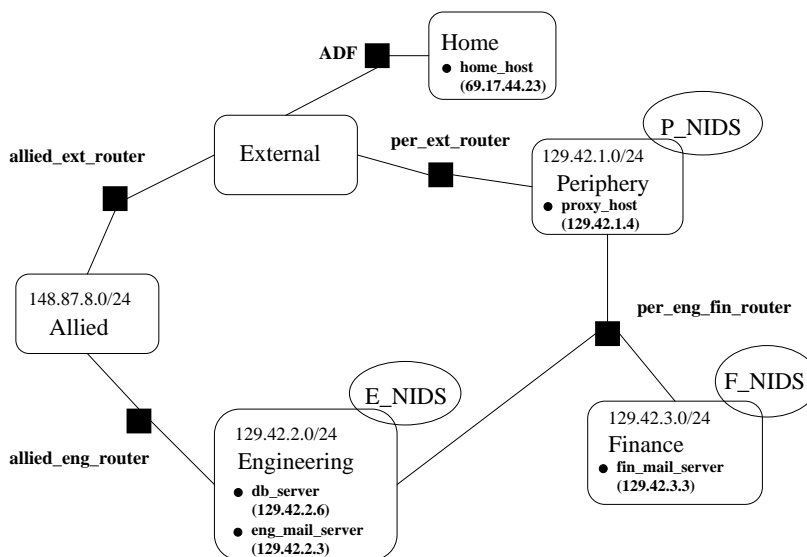


Figure 1: Corporate network: router-area graph

```

defhostset(corporate, union(periphery, internal)).
defhostset(non_corporate, complement(corporate)).
defhostset(mail_hosts, [eng_mail_server, fin_mail_server]).
defhostset(trusted_hosts,
            union(periphery, union(mail_hosts, [db_server]))).
defhostset(eng_untrusted, difference(engineering, trusted_hosts)).

```

If **finance** has addresses $[129, 42, 3, *]$, and **periphery** has $[129, 42, 1, *]$, then the derived **corporate** area is identified with the address constraint $[129, 42, [1, 2, 3], *]$.

Packets: A *packet* p is a structure with a fixed set of fields, including source and destination ports and IP addresses, type, orientation, and payload. If nonsymbolic IPv4 addresses are used, each address field is split into four segments, each constrained to the range 0..255. Following [12], an *oriented service* is one of **from_server** or **to_server** together with a service type S , one of a finite set of services, in our case, $\{\text{ftp, telnet, http, smtp, db_service}\}$. Oriented services capture the ability of firewalls

and NIDSs to distinguish between client and server for certain traffic, depending on who initiated the connection.

Our tool assumes that the payload field is abstracted to match the IDS capabilities. In the simplest case, which we use in our examples, this is a finite-domain datatype where, for example, a particular `smtp` attack recognized by the IDS has payload `smtp_attack_payload`. Supporting more expressive payload descriptions, such as the regular expressions used by the Snort IDS [20], is left for future work.

A *packet language* L is a set of packets. It is important to note that packet languages describe the sources that packets *purport* to have, regardless of whether they have been spoofed. We will often represent languages as a constrained packet structure, of the form

```
R = packet with [from_ip:F,to_ip:T,service:S,payload:P,...]
```

where each field is associated with a finite-domain constraint. Thus, R represents the set

$$\{p \mid from_ip(p) \in F, to_ip(p) \in T, service(p) \in S, payload(p) \in P, \dots\}$$

The set of packet languages that can be expressed in this way is closed under intersection:

$$R_1 \cap R_2 = \{p \mid from_ip(p) \in F_1 \cap F_2, to_ip(p) \in T_1 \cap T_2, service(p) \in S_1 \cap S_2, \dots\}$$

We use lists of constrained packets for closure under union and complementation.

Filters and Postures: Firewall configurations are described by associating a *filter* with each (directed) edge in the network graph. For an edge e , $filter(e)$ is the set of packets allowed to traverse that edge. Each router can have a different inbound and outbound filter for each area to which it is connected, corresponding to its network interfaces.

A *filtering posture* is an assignment of inbound and outbound filtering rules to each router interface. For example, the symbolic declaration

```
declare_filter(allied_eng_router,inbound,engineering,
[policy(allied, any, []),
 policy(union(engineering,finance),allied,all_tcp_services)])
```

specifies that the router between the engineering and allied areas refuses any incoming packet from `engineering` whose source is in `allied`, and allows all TCP packets with source in `engineering` or `finance`, and destination in `allied`. (All other packets are refused, by default.) A corresponding set of Cisco PIX firewall declarations [7] would be:

```

access-list AFE deny ip 148.87.8.0 255.255.255.0 any
access-list AFE permit tcp 129.42.2.0 255.255.255.0
    148.87.8.0 255.255.255.0
access-list AFE permit tcp 129.42.3.0 255.255.255.0
    148.87.8.0 255.255.255.0

```

Our tool translates the PIX declarations to the constraints:

```

Policy AFE allows: packet with [type:tcp,
    from_ip : [129, 42, [2, 3], *],
    to_ip : [148, 87, 8, *], to_server]
Policy AFE denies: packet with [from_ip : [148, 87, 8, *]]

```

(If no other rules are present, we use firewall semantics that deny any packet that is not explicitly allowed.) In the nonsymbolic formulation, the `type` field ranges over `{tcp,udp,icmp}`, and service types, such as `http`, are identified by their port numbers. Services with variable ports, such as `ftp`, can be given their own abstract type field.

Path Filtering, Trajectories, and Events: For a path $P : \langle e_1, \dots, e_n \rangle$, the set of packets that can traverse P is $\text{packets}(P) \stackrel{\text{def}}{=} \bigcap_{e \in P} \text{filter}(e)$. A *trajectory* is a pair $\langle L, P \rangle$ for a nonempty packet language L and a path P . The trajectory is *feasible* if $L \subseteq \text{packets}(P)$. For area sets S_1 and S_2 , we write $\text{paths}(S_1, S_2)$ for the set of noncyclic paths that start at a node in S_1 and end in S_2 . For areas a_1, a_2 , we write $\text{paths}(a_1, a_2)$ for $\text{paths}(\{a_1\}, \{a_2\})$. For host sets H_1 and H_2 ,

$$\text{paths}(H_1, H_2) \stackrel{\text{def}}{=} \bigcup \{ \text{paths}(a_1, a_2) \mid (\text{hosts}(a_1) \cap H_1) \neq \emptyset \text{ and } (\text{hosts}(a_2) \cap H_2) \neq \emptyset \} .$$

For a path P and node n , we write $n \in P$ if n appears in P .

An *event* is a triple (H_1, H_2, L) , for host sets H_1 and H_2 , and packet language L . Informally, it describes a packet in L that travels from a host in H_1 to one in H_2 . For an event $E : (H_1, H_2, L)$, the set of *feasible trajectories* is

$$\text{trajectories}(E) \stackrel{\text{def}}{=} \{ \langle L', P \rangle \mid P \in \text{paths}(H_1, H_2) \text{ and } L' = (\text{packets}(P) \cap L) \neq \emptyset \} .$$

NIDS Configuration and Detection Policies

We use a simple NIDS model, where an NIDS configuration is given by its location in the network, and the set of packets that cause it to raise an alert. This is sufficient to describe most behavior of signature-based network IDSs [20, 18]. However, the utility of our framework is not limited to such NIDSs. If the traffic that a monitor

needs to observe can be specified as events in our framework, then our tool can help correctly place and configure the monitor.

To model correlation, we use a finite set of *correlation engines* E . When a packet raises an alert, we specify the subset of E to which the alert is sent, defining a mapping $correlate : (Ids, Location, Packet) \rightarrow 2^E$. Thus, we use the declaration

```
define_ids_config(Ids_name, [location:Area, packets:L, ce:C])
```

where **Area** is a node in the graph, **L** is a packet language, and **C** is a set of correlation engines. If more than one NIDS monitors an area, we can use a single name to represent the combined NIDS detection capability at that area. Since an NIDS may be used to monitor multiple subnets, and different rules used for each, we allow more than one configuration statement for an NIDS. This also lets us specify NIDSs that send different alerts to different correlation engines, depending on the packets involved. We write $idsconfig(I, n, L, C)$ if NIDS I raises alerts at node n for packet language L and correlation set C , which can be empty. Then,

$$\begin{aligned} alerts(I, n, C) &\stackrel{\text{def}}{=} \bigcup \{L \mid idsconfig(I, n, L, C)\} \\ alerts(I, n) &\stackrel{\text{def}}{=} \bigcup \{L \mid idsconfig(I, n, L, C) \text{ for some } C\} \\ alerts(n) &\stackrel{\text{def}}{=} \bigcup \{L \mid idsconfig(I, n, L, C) \text{ for some } I, C\} \end{aligned}$$

Often, there is at most one NIDS on each node, and we write $alerts(n)$ for $alerts(I_n, n)$. An NIDS *detection policy* is given by an *event description*, which describes conditions that might indicate an attack and should result in NIDS alerts. In the simplest case, we consider single events, where a packet in a language L leaves area S and arrives at area D :

```
event:[leaves:S, arrives:D, packets:L]
```

In our IDS configuration and policy specifications, the packet language L can specify a particular class of attacks, by constraining the payload, in addition to the source, destination and services. Note that S and D constrain the actual beginning and end of the path that the policy is concerned about; the packets in L may have IP addresses for other sources and destinations. (This can be the case if they are spoofed at S , or if they are in transit from other areas.)

For a path P , we define $guaranteedalert(P) \stackrel{\text{def}}{=} \bigcup_{N \in P} \{alerts(N)\}$. This is the set of packets guaranteed to raise an alert if they traverse P . A network *enforces* an NIDS detection policy iff the event specified by the policy always raises an alert. For an event $E : (S, D, L)$, an alert should be raised if any packet in L leaves a host in S and arrives at a host in D . That is, if $L \subseteq guaranteedalert(P)$ for all $\langle L, P \rangle \in trajectories(E)$.

3 Implementation

We have developed a tool that uses the above framework to reason about network filters and NIDSs. For ease of representation, manipulation and propagation of packet constraints, as well as searching through the network graph, we implemented the tool in the ECLIPSE Constraint Logic Programming language [5], whose syntax we have already used in Section 2.

Given the network graph and the filtering rules, it is straightforward to compute the packet language $packets(P)$ for any path P . However, it is sometimes useful to adopt a dynamic programming approach, and precompute all the (maximal) trajectories in the graph, of the form $\langle packets(P), P \rangle$ for a path P . To do this, we start out with the empty path, which we define as allowing any packet:

$$packets(\langle \rangle) := \text{any} \quad (\text{base case})$$

Given the constraints for the paths of length n , we compute the constraints on all paths that extend them by one edge, intersecting their packet language with the filter for the new edge:

$$\begin{aligned} packets(\langle e_1, \dots, e_n, e_{n+1} \rangle) := \\ packets(\langle e_1, \dots, e_n \rangle) \cap filter(e_{n+1}) \quad (\text{extend}) \end{aligned}$$

The set of packets $packets(a_1, a_2)$ that can flow from area a_1 to area a_2 is the union of $packets(P)$ for all paths $P \in paths(a_1, a_2)$. Note that filters can express routing behavior: particular packets may be allowed to continue only along particular edges.

If we are interested in the possible paths that a particular packet language L can take through the network, we can invoke the same procedure, using L as the initial value of $packets(\langle \rangle)$, rather than any , in the base case above. In this way, we can compute the trajectories for which IDS I will generate an alarm at node N : starting with $alerts(I, N)$, compute all feasible paths, and then choose the ones that contain N .

From the point of view of individual nodes n in the graph, we are computing the set of packets $reach(n)$ that can reach it, as a fixpoint:

$$reach(n) = \bigcup_{\langle i, n \rangle \in Edges} (reach(i) \cap filter(\langle i, n \rangle)) .$$

The notion of policy enforcement can be adapted if particular areas or hosts can be trusted to do no spoofing [12], or, in general, if particular areas can be trusted to generate only particular types of packets. (One way to achieve this is to equip them with nonbypassable ADFs.) The set of packets $\mathcal{O}(a)$ that can originate at area a

is defined as **any** if a is untrusted, and $\{P \mid source(P) \in a\}$ otherwise. The packets that can reach a are now

$$reach(a) = \bigcup_{a' \neq a} (\mathcal{O}(a') \cap packets(a', a)) .$$

The set $contents(a)$ of packets that can be found at area a is then $contents(a) \stackrel{\text{def}}{=} \mathcal{O}(a) \cup reach(a)$.

Checking Event Detection: We check that $E : (H_1, H_2, L)$ raises an alert by (1) computing $trajectories(E)$, which represents all the paths that a packet in L might take from a host in H_1 to a host in H_2 , and (2) checking that $L' \subseteq guaranteedalert(P)$ for each $\langle L', P \rangle \in trajectories(E)$. The trajectories can be obtained by computing $L \cap packets(P)$ for each $P \in paths(H_1, H_2)$, or by the dynamic programming algorithm above, with L as the initial language in the base case.

Reading Configuration Files: Our tool constructs a representation of the packet language allowed by a sequence of Cisco PIX firewall rules by processing them in order. When processing rule r_i , the packets that match rules r_1, \dots, r_{i-1} are excluded. Our implementation can handle medium-size real-world PIX configuration files, with up to 130 rules, in under one minute. A more compact packet representation, such as ordered binary decision diagrams (OBDDs) [3], may be needed to read larger ones, as done in [13].

Our prototype implementation is highly efficient for small networks. For our running example of Figure 1, the filtering posture and policies are defined, evaluated, and all feasible paths computed, in less than 0.1 second on a standard desktop machine, when symbolic addresses are used. When actual IP addresses and firewall rules are used, the time is less than 20 seconds, with most of the time spent processing the input rules. Algorithms that compute feasible paths only on an on-demand basis will be needed for larger, highly connected networks, where the number of feasible paths through the network can be exponential in the number of nodes.

We are working on automatically processing Snort [20] NIDS specifications as well, using similar techniques. In this case, the packet payload matches should be abstracted to a finite datatype that captures the main classes of packets that the NIDS must monitor.

Example: Analyzing Network Flow: Consider a potential **http** attack from **external** to a Web server in **finance**. Given a standard firewall configuration, our tool can determine that this event cannot occur: the two possible paths, through **periphery** and **engineering**, are blocked for **http** packets from **external** to **finance**.

On the other hand, the tool can be used to check that events expressing desired functionality can indeed occur, by verifying that these “good” events have a

nonempty set of trajectories. The number of trajectories gives a measure of redundancy as well.

Example: Checking NIDS Configuration

To illustrate the basic concepts of NIDS specification and configuration in our framework, consider a detection policy for attacks that exploit `smtp` vulnerabilities, from outside `corporate` to `engineering`. As Figure 1 shows, we assume that the three corporate areas, namely, `engineering`, `finance`, and `periphery`, correspond to three subnets that can each be monitored by an NIDS. (If these areas consist of multiple network segments, one can decompose them into subareas so that each is monitored by a separate NIDS.)

First, we define a packet language for a general version of the attack:

```
define_packets(smtp_attack, [source:any, dest:any, service:smtp,
                             orientation:any, payload:smtp_attack_payload]).
```

This definition is evaluated to a constrained packet structure that describes all `smtp_attack` packets. The NIDS detection policy states that if such a packet departs a noncorporate host, and arrives at `engineering`, an alarm should be raised:

```
define_detection_policy(detect_smtp,
                        [leaves:non_corporate, arrives:engineering,
                         packets: smtp_attack]).
```

The NIDS configuration specifies the location and the packets for which it raises an alert:

```
define_ids_config(candidate_ids, [location: periphery,
                                  packets: smtp_attack]).
```

In this case, the NIDS is deployed on the `periphery`, and is meant to detect `smtp` attacks against `engineering` hosts. However, this configuration does not cover the attack path from `allied` to `engineering`. Indeed, when we ask our tool to check configuration `candidate_ids` against policy `detect_smtp`, the following output is produced:

```
Path: allied->allied_eng_router->engineering.
Undetected: packet with [service:smtp, payload:smtp_attack_payload,
                          from_ip:allied_host,
                          to_ip:[eng_mail_server, fin_mail_server]].
```

When the location of the NIDS is moved from `periphery` to `engineering`, our tool reports that there is no violation of policy `detect_smtp`.

4 Examples: Corporate Scenario

In this section, we further elaborate the corporate scenario of Figure 1, and specify network events that should be detected and reported by the NIDS subsystem. The detection policy is violated if a network event it describes can happen without an alarm being raised by the NIDS. The following are some events we would like to detect:

AC: Violations of the access control policy specified for the corporate scenario in [12] (e.g., a host in **engineering** accesses a Web server in **finance**)

EC-FHT-C: Attacks from **external** to **corporate** exploiting known vulnerabilities of **ftp**, **http**, or **telnet** clients

EC-S-SC: Attacks from **external** to **corporate** exploiting known vulnerabilities of **smtp** servers or clients

AC-S-SC: Attacks from **allied** to **corporate** exploiting known vulnerabilities of **smtp** servers or clients

SC: Transfer of source code from **engineering** to **external**

To check the access control policy that concerns the proxy host, we can use a rule-based NIDS to detect the complement of the access control policy. In particular, we want to detect the following packets:

```
define_packets(AC1, [source:external,destination:[proxy_host],
                    orientation: to_server, service: [ftp,http,telnet])
define_packets(AC2, [source:external,destination:[proxy_host],
                    orientation: any, service:complement([ftp,http,telnet]))
define_packets(AC3, [source:[proxy_host],destination:internal,
                    orientation: to_server, service:[ftp,http,telnet])
define_packets(AC4, [source:[proxy_host],destination:internal,
                    orientation: any, service:complement([ftp,http,telnet]))
```

We deploy an NIDS configured to detect these packets to monitor the periphery network:

```
define_ids_config(periphery_ids, [location: periphery,
                                packets: disjoint([AC1,AC2,AC3,AC4]), ce: []]).
```

As another example, we construct the following packet language to enforce Policy EC-FHT-C:

```
define_packets(EC_FHT_C1, [source:external, destination:corporate,
    orientation: from_server, service:[ftp,http,telnet],
    payload:known_attack])
```

Here, we use `known_attack` to abstract the class of known attacks.

For Policy SC, the detection policy regarding source code transfer directly from `engineering` to `external` can be specified as follows:

```
define_detection_policy(detect_direct_sourcecode_leak,
    [leaves:engineering, arrives:external,
    packets: [payload: source-code]]).
```

To enforce this detection policy, one may deploy an NIDS to monitor the engineering area to detect source code transfer to `external` via one of `ftp`, `http`, `smtp`, `telnet`. The packet language `SC1` considers `smtp` connections between the external area and the engineering area, and `SC2` considers `ftp`, `http`, and `telnet` connections initiated within the engineering area to a server in the external area. Note that source code transfer via other services is not considered in the IDS configuration, since it should be blocked by the firewalls.

```
define_packets(SC1, [source:engineering, destination:external,
    orientation: any, service:[smtp], payload:source-code])
define_packets(SC2, [source:engineering, destination:proxy_host,
    orientation: to_server, service:[ftp,http,telnet],
    payload:source-code])
define_ids_config(eng_nids, [location: engineering,
    packets: disjoint([SC1,SC2]), ce:[])
```

5 Applications: Configuring NIDS and Firewalls

We already saw how our tool can help with NIDS placement in Section 3. We now describe how it can be used to analyze and improve various other aspects of filtering and NIDS configuration.

5.1 NIDS Ruleset Reduction

If an NIDS cannot keep up with the network traffic, it may miss attacks. The resources required usually depend on the nature and number of rules activated, so one way to improve NIDS throughput is to reduce the set of rules. The challenge is to do this without missing events that violate the network security policy.

If we assume that the firewalls will work as specified, we can first use our tool to compute $contents(A)$ for the area A where an NIDS I is deployed, and then remove from its configuration at A any rule whose domain is disjoint from this set.

5.2 Checking Firewall Configurations at Runtime

Section 5.1 shows that we can minimize NIDS rulesets if we assume that the firewalls work as advertised. On the other hand, if some firewalls are not trusted, we can generate an NIDS specification that checks for any violation of the firewall specification. To check a set of firewalls S in a network \mathcal{N} , first create a network description \mathcal{N}' that does not include the filters from S . Then compute $reach_{\mathcal{N}'}(n)$ for the node n where the NIDS will be placed. The new IDS is configured to raise an alert if any packet in $reach_{\mathcal{N}'}(n) - reach_{\mathcal{N}}(n)$ is observed at n .

This process can be reversed: given an NIDS configuration, we can generate a firewall configuration that replaces it, that is, find filters that block all the packets that generate alerts. This, provided the firewalls can inspect the same packet fields, including payload, that the NIDSs do.

5.3 False Alarm Reduction

NIDS false alarms have two main sources: (1) inaccurate attack models, and (2) network misconfiguration. An example of the first is an attack signature with a small footprint of a buffer overflow that also matches normal system activities. An example of network misconfiguration is an incorrectly located NIDS that generates alerts for traffic that it is not designed to monitor. For example, a company may deploy a heuristic looking for the string “Unpublished source code of Company X” in packet payloads, to detect unauthorized transfer of proprietary source code. If an NIDS with this detection heuristic monitors the network used by a software engineer to access the source code repository, it will generate a large number of false alarms.

Given the network configuration and the normal activities that the network must support, we can automatically detect misconfiguration. For example, we can identify false alarms by considering all possible network paths used by authorized employees to access the repository, and the NIDSs that monitor a node in these paths.

Algorithm: We can identify the NIDSs that generate an alert for a given event E by checking, for each $T : \langle L, P \rangle \in trajectories(E)$, whether $L \cap alerts(N) = \emptyset$ for each $N \in P$. To identify possible false alarms, we do this for *normal* events, which describe desired network functionality.

5.4 Generating NIDS Configurations from Event Specifications

We can go further and generate an NIDS specification from a description of attacks and normal events. Assume that as input we are given a fixed filtering posture, a set of attack events \mathcal{E}_{bad} , and a set of normal events, \mathcal{E}_{good} . The goal is to find an NIDS configuration that generates alerts for all the events in \mathcal{E}_{bad} , but does not raise alerts for events in \mathcal{E}_{good} , since these would be false alarms. We first generate the corresponding sets of good and bad trajectories:

$$T_{good} \stackrel{\text{def}}{=} \bigcup_{e \in \mathcal{E}_{good}} \text{trajectories}(e) \quad \text{and} \quad T_{bad} \stackrel{\text{def}}{=} \bigcup_{e \in \mathcal{E}_{bad}} \text{trajectories}(e)$$

For a node n we define $L_{good}(n)$ (resp. $L_{bad}(n)$) as the set of packets that pass through n while traversing a trajectory for a good (resp. bad) event:

$$L_{good}(n) \stackrel{\text{def}}{=} \bigcup \{L \mid (L, Path) \in T_{good}(L) \text{ and } n \in Path\}$$

$$L_{bad}(n) \stackrel{\text{def}}{=} \bigcup \{L \mid (L, Path) \in T_{bad}(L) \text{ and } n \in Path\}$$

Let \mathcal{O} be the set of observable nodes, where we can place an NIDS. Consider the configuration that places an NIDS I_n at every observable node n , such that

$$\text{alerts}(I_n, n) \stackrel{\text{def}}{=} L_{bad}(n) - L_{good}(n) .$$

(If this set is empty, no NIDS is placed at n .) It is clear that this configuration will not raise false alerts for the good events. Furthermore, among the set of such configurations, it is the one that will raise the most alerts for the bad events. Therefore, a solution exists iff this configuration works, that is, if for all $T : \langle L, P \rangle \in T_{bad}$, $L \subseteq \text{guaranteedalert}(P)$.

While the above construction finds a solution if one exists, it may be suboptimal: the NIDSs are likely to be redundant, checking for the same packets multiple times along the same path. As with the filtering posture generation in [12], the NIDS configuration problem becomes more open-ended when different notions of optimality are considered. We can take a more incremental approach, tightening different NIDS nodes until we succeed or no further tightening is possible. In general, a redundancy-free configuration is not always possible. But a second procedure that results in fewer redundancies follows:

Initially, let $\text{alerts}(I_n, n) = \emptyset$ for all nodes n .

We say that a node n is *saturated* if $\text{alerts}(I_n, n) = L_{bad}(n) - L_{good}(n)$.

Let the initial set \mathcal{Q} of pending trajectories be T_{bad} .

LOOP: **If** \mathcal{Q} is empty, **stop**; **else**

Remove from \mathcal{Q} all trajectories $T : \langle L_{bad}, P \rangle$ such that
 $L_{bad} \subseteq \text{guaranteedalert}(P)$.
Find a $T : \langle L_{bad}, P \rangle \in \mathcal{Q}$ such that
 P contains an unsaturated observable node n ;
If there is no such T , **stop**; **else**
Add $L_{bad} - L_{good}(n)$ to $\text{alerts}(I_n, n)$, goto LOOP.

If \mathcal{Q} is empty at the end of this procedure, we have found a suitable NIDS configuration. Otherwise, no such configuration exists, and the trajectories in \mathcal{Q} describe overlaps between T_{good} and T_{bad} that cannot be resolved. For each remaining trajectory $\langle L_{bad}, P \rangle$, the problematic set will be $L_{bad} - \text{guaranteedalert}(P)$. A human designer might be able to inspect these and decide whether the filtering posture should be adjusted, or the topology changed, to meet the desired specification.

5.5 Generating Optimal NIDS Configurations

Many different NIDSs configurations can enforce a given detection goal. Given a cost function, our tool can generate an optimal NIDS configuration, searching through the possible correct configurations to find one with a minimal cost.

This can have various applications, depending on how the cost function is defined. An NIDS must be able to keep up with the traffic at the node where it is placed. Given a choice, we may prefer placing NIDSs at nodes with lower bandwidth, which will reduce the chances of dropped packets. For similar reasons, we may want to keep the number of rules at each NIDS to a minimum. The cost of placing an NIDS I at a node n can then be defined as a function of the number of rules in I and the bandwidth at node n .

For many cost functions, the cost of a partial configuration is a lower bound for the total cost. The Constraint Logic Programming approach makes it easy to implement a branch-and-bound strategy to search for an optimal configuration. Here, the algorithm of Section 5.4 is used to generate candidate configurations; partial configurations are discarded as soon as their cost exceeds that of a known solution. We have implemented this within our tool.

The case where both NIDS and firewall rules are to be generated simultaneously is left for future work (see Section 8).

5.6 Detecting Multistep Attacks

A complex attack may involve multiple steps. In attack scenario recognition for multistep attacks, e.g. [6], the detection process is divided into two parts: First, the individual attack steps are detected by NIDSs, which send alerts to a correlation

engine. The correlation engine then performs some inference on these alerts to detect multistep attacks.

For example, assume that an adversary in the external area plans to obtain proprietary source code stored in `engineering`. To circumvent the network access control policy enforced by the firewalls and the likely detection policy for source code transfers to an external host, the adversary uses a three-step attack: (1) exploit a vulnerability in an email server in the corporate area; (2) use the compromised email server to obtain the source code from the repository in `engineering`; and (3) encrypt the source code and email it to a host in `external`.

Attack steps such as (3) may not be observable by an NIDS. Still, we want the critical observable steps to be detected by one or more NIDSs that report their alerts to the same correlation engine, which may recognize the scenario from these alerts.

To specify multistep attacks, our detection policies can specify (unordered) event sets, in addition to single events. For example, the following policy states that an alarm should be raised if an email attack against a corporate mail host, and a source code transfer to a mail host, both occur:

```
define_packets(code_xfer_to_mailhost, [services:any,
    source:corporate, dest:mail-hosts, payload:source_code]).
define_detection_policy(detect_sourcecode_exfiltration,
    [event: sequence([leaves:non-corporate, arrives:mail-hosts,
        packets:smtp_attack], [leaves:corporate, arrives:mail-hosts,
        packets:code_xfer_to_mailhost]))].
```

To enforce this detection policy, we may configure NIDSs at `finance` and `engineering` to detect attacks against corporate mail hosts and source code transfers to them. Moreover, these NIDSs will send alerts to the same correlation engine, `corpCor`:

```
define_ids_config(e_nids, [location: engineering,
    packets: [source: non_corporate, destination: mail-hosts,
        payload: smtp_attack_payload], ce:[corpCor]])
define_ids_config(f_nids, [location: finance,
    packets: [source: non_corporate, destination: mail-hosts,
        payload: smtp_attack_payload], ce:[corpCor]])
define_ids_config(e_nids, [location: engineering,
    packets: code_xfer_to_mailhost, ce:[corpCor]])
define_ids_config(f_nids, [location: finance,
    packets: code_xfer_to_mailhost, ce:[corpCor]])
```

Correlation Check Algorithm: We do not model the correlation specifics, limiting ourselves to checking that a correlation engine will always receive a sufficient

set of alerts. To do this for a multistep attack E consisting of events $\{E_1, \dots, E_n\}$, we check if there is a correlation engine that will receive an alert for any feasible trajectory of each event. Let \mathbf{Cset} be the set of correlation engines in the system. Define:

$$\begin{aligned} \mathit{guaranteedalert}(P, C) &\stackrel{\text{def}}{=} \\ &\bigcup \{L \mid \mathit{idconfig}(I, N, L, CS) \text{ and } N \in P \text{ and } C \in CS\} \ . \end{aligned}$$

Given a network configuration, the following algorithm quickly checks that some correlation engine C_i will receive the alerts necessary to detect a multistep attack $E : \{E_1, \dots, E_n\}$:

```

foreach  $C_i \in \mathbf{Cset}$ 
  foreach  $E_j \in \{E_1, \dots, E_n\}$ 
    foreach trajectory  $T_k : \langle L_k, P_k \rangle \in \mathit{trajectories}(E_j)$ 
      if  $L_k \subseteq \mathit{guaranteedalert}(P_k, C_i)$  then next  $T_k$ 
      else next  $C_i$ 
    next  $E_j$ 
  print  $C_i$  “detects  $E$ ”

```

This is a sound, but incomplete test: If the above code terminates without reporting that E is detected by some C_i , then there may be an assignment of trajectories to events E_1, \dots, E_n such that no correlation engine detects all of them.

This, however, is not always the case: Consider a multi-step attack with events $E : \{E_0, E_1\}$, where E_0 has a single trajectory T_0 , and E_1 has two possible trajectories $\{T_1, T_2\}$. Consider correlation engines C_1, C_2 , such that C_1 detects $\{T_0, T_1\}$ and C_2 detects $\{T_0, T_2\}$. Then the multi-step attack E is always detected by either C_1 or C_2 , though neither correlation engine is guaranteed to always do so.

Finding Attacks that Evade Correlation: If the above algorithm does not find a correlation engine that detects a given multistep attack E , we can run a more expensive test to find an instance of E that goes undetected.

Since each NIDS declaration along a trajectory can generate a different set of alerts, each sent to different correlation engines, we cannot take the union of the alert packets along each path P , as $\mathit{guaranteedalert}(P)$ does for the single-event case (see Section 2). For a packet $p \in \mathit{packets}(P)$, the set of correlation engines that will receive alerts when packet p traverses path P is

$$\mathit{c_alerts}(p) \stackrel{\text{def}}{=} \bigcup \{C \mid \mathit{idconfig}(I, n, L, C) \text{ for some } n \in P \text{ and } p \in L\} \ .$$

A trajectory $\langle L, P \rangle$ partitions L into (disjoint) equivalence classes L_1, \dots, L_n , defined by a common value of $\mathit{c_alerts}(p)$. These classes can be computed by enumerating the subsets S of NIDS declarations for nodes in P , such that

$$\bigcap \{L \mid \mathit{idconfig}(I, n, L, C) \in S\}$$

is nonempty.

This maps each trajectory to a set of “subtrajectories” $\{\langle L_i, P \rangle\}$. For each assignment of subtrajectories $\{T_1, \dots, T_n\}$ to events $\{E_1, \dots, E_n\}$, we check that the corresponding alert sets have a common correlation engine, that is, $c_alerts(T_1) \cap \dots \cap c_alerts(T_n) \neq \emptyset$. This is the set of correlation engines that will raise an alert for that choice of subtrajectories. If empty, the chosen subtrajectories describe an instance of the multistep attack that can go undetected.

6 Extensions

6.1 Packet Transformations

Our framework so far assumes that packets are not modified as they traverse the network. This assumption no longer holds when some modern networking techniques are used. Techniques such as network address translation, port mapping, and IPsec mean that the source, destination, ports, and payloads of packets can change as they move from one node to the next.

To accommodate this, we can extend our framework to include the *history* of the packets as part of each trajectory. Instead of being a pair $T : \langle L, P \rangle$, a trajectory T is now of the form

$$T : \langle \langle L_1, n_1 \rangle, \langle L_2, n_2 \rangle, \dots, \langle L_k, n_k \rangle \rangle,$$

where each L_i describes the packets at node n_i .

The notion of event can now distinguish the packet at the source from the packet that arrived at the destination. In the case of attacks and confidentiality violations, we can let the initial and ending packet be arbitrarily different from each other. For example, we can then specify security policies that are violated if a confidential packet P originating in a protected location ends up as a packet P' in an unprotected one; conversely, if a packet P that originates at an untrusted location, causes a derived packet P' to arrive at a sensitive one.

The algorithms we have presented can be adapted in a straightforward way. In practice, the main new task that must be performed by the tool is to compute the *post-image* of a packet language L as it passes a node n : Consider a node n that performs NAT, port mapping, or encryption; let f_n describe this transformation, where a packet p is transformed to $f_n(p)$. We must then compute

$$post(n, L) \stackrel{\text{def}}{=} \{p' \mid p' = f_n(p) \text{ for some } p \in L\} .$$

If the mappings are relatively uniform, as they often are (e.g. translating a block of private addresses to a public address and port), this can be efficiently computed as a transformation on constraints.

6.2 Encrypted Traffic

An important special case of packet transformations is encryption. To protect data transmitted over an insecure network, routers can use protocols such as IPsec [10, 14] to encrypt and authenticate packets. This presents additional challenges for NIDSs. If the packet payload is encrypted, an NIDS will not detect certain attacks unless it knows the decryption key. Because of the overhead of sharing the cryptographic keys with NIDSs, additional risks of key exposure, and performance impact for NIDSs to perform decryption, this is usually not done. To ensure that NIDSs can function in an environment where some firewalls may perform encryption, they must reside at locations where they can examine the packets in their unencrypted state.

We can reason about NIDS placement in the presence of encrypted links if we can identify the paths that encrypted traffic may traverse. We can then make sure that the relevant NIDS is placed where traffic is in the clear. If no such location exists, we can conclude that NIDSs cannot securely monitor this communication path, and perhaps a host-based IDS should be deployed instead.

The framework of [13] extends the one we use by including IPsec (see also [14]). We are considering similar extensions, so we can automate this reasoning within our tool. We can model packet encryption at a coarse level by introducing a new packet field, `crypt`, and two new operations on packet languages. For a language L , `encrypt`(L, K) produces the packet language where the `crypt` field X in L is replaced by `encrypt`(K, X). Similarly, `decrypt`(L, K) produces a packet language that replaces `encrypt`(K, X) by X . Initially, the value of the `crypt` field is the constant `clear`. This extension does not capture all the security properties of IPsec, but is sufficient to express the fact that encrypted packet content is invisible to NIDSs, if they do not have the right key. IDS specifications will normally not match packets whose `crypt` field is not `clear`. Observability is now a function of the trajectory. For a path P , the function $packets_C(P)$ describes the packet language at the end of P , including the encryption state. To check if an IDS at node N raises an alarm for a path P , we compute $packets_C(P')$ for the subpath that ends at N .

We compute $packets_C(\langle e_1, \dots, e_{n+1} \rangle)$ by intersecting $filter(e_{n+1})$ and $packets_C(\langle e_1, \dots, e_n \rangle)$, as in Section 3, but we now apply the `decrypt` or `encrypt` operations if they have been specified for edge e_{n+1} and the packet language so far. To restrict the number of trajectories considered, we assume that encrypted packets originate only at trusted host sets that have the corresponding key.

7 Related Work

Our work differs from [12] in that we represent and reason about NIDS specifications. Our implementation makes it easier to extend the constraints for packet

languages, so they can include features other than source, destination, and services, such as payload. Also, although our running example is a bipartite graph, our implementation is not limited to this case. The work in [12] is extended in [13], to include IPsec authentication and confidentiality, which we address in more limited ways. *Directed interfaces* split router nodes and use directed edges to capture more detail about router processing.

A number of firewall management tools, such as Firmato [2], Fang [17], and Lumeta [22], can specify an abstract network access control policy, and check or generate firewall rules that satisfy the policy. A distributed firewall architecture based on Autonomic Distributed Firewalls (ADFs) is presented in [16]. ADFs (e.g. [1]) have several advantages over conventional firewalls relevant to enforcing network security policies, particularly nonbypassability, and can prevent spoofing. Section 3 shows how our framework can model the spoofing prevention feature of ADFs, and perform a more accurate analysis.

IPsec policies are automatically generated in [10] by identifying areas and paths that require encryption. In [15], the KeyNote trust management system is used for distributed firewall configuration, including IPsec as well. Policy validation is used in [4] as the basis for adaptive network reconfiguration, to enforce a security policy in a dynamic environment.

An expert system to analyze firewall configurations that is also implemented in the ECLIPSE Constraint Logic Programming language is presented in [9]. This tool can also read Cisco router access tables, and answer basic questions about connectivity, spoofing, and access permissions, which helps find problems in network configurations. Our tool performs similar reasoning; in addition, we reason about NIDS specifications, and generate NIDS configurations.

8 Conclusion

This paper presents an integrated, constraint-based approach for modeling and reasoning about the two main types of network security components, namely NIDS and firewalls. Based on this approach, we have developed a tool that can process Cisco PIX firewall rules, and analyze abstract NIDS configurations to determine whether a detection policy is enforced. As discussed in Section 5, our approach provides a rigorous yet lightweight tool to help a security administrator address a variety of network configuration problems.

The work presented in this paper can be extended in several directions. The algorithm for deriving NIDS configurations can be combined with the firewall policy generation procedures of [12] if unresolvable good and bad trajectories remain (see Section 5.4). Our model does not consider the order of events in a multistage

attack. A more advanced model could. This is particularly relevant, if the network can be reconfigured to respond to an ongoing attack, as done, for example, by the self-securing network interfaces of [11].

If the NIDSs or the filtering rules are reconfigured at runtime, our tools can always be used to automatically check each new configuration. (Note that this is not always necessary: for example, possible changes in routing behavior may be modeled statically simply by underspecifying the set of edges that certain packets can traverse when they leave a node.) However, this does not guarantee that the network will always be correctly configured. To do this, it may be possible to model the evolution of the network as a transition system, and formulate its correctness as a temporal verification problem, to which model checking [8] can be applied.

Acknowledgements

We thank Mike Hogsett, Joshua Levy, Al Valdes, and Neil Yorke-Smith for their suggestions and comments.

Bibliography

- [1] 3Com. *3Com Embedded Firewall. Software for the 3CR990 Network Interface Card (NIC) Family*, Dec. 2001.
- [2] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.
- [3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
- [4] J. Burns, A. Cheng, P. Gurung, S. Rajagopalan, P. Rao, D. Rosenbluth, A. Surendran, and J. D.M. Martin. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II) Volume 2*, pages 1012–1026, Anaheim, California, June 12–14, 2001.
- [5] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECLiPSe: An Introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, 2003.
- [6] S. Cheung, U. Lindqvist, and M. W. Fong. Modeling multistep cyber attacks for scenario recognition. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*, pages 284–292, Washington, D.C., Apr. 22–24, 2003.
- [7] Cisco Systems, Inc. *Cisco PIX Firewall and VPN Configuration Guide: Version 6.3*, 2003. <http://www.cisco.com>.
- [8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [9] P. Eronen and J. Zitting. An expert system for analyzing firewall rules. In *Proc. 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, pages 100–107, Nov. 2001.
- [10] Z. J. Fu and S. F. Wu. Automatic generation of IPSec/VPN security policies in an intra-domain environment. In *12th International Workshop on Distributed Systems: Operations and Management (DSOM'2001)*, France, Oct. 15–17, 2001.

- [11] G. R. Ganger, G. Economou, and S. M. Bielski. Finding and containing enemies within the walls with self-securing network interfaces. Technical Report CMU-CS-03-109, School of Computer Science, Carnegie-Mellon University, Jan. 2003.
- [12] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, pages 120–129, Oakland, California, May 1997. Extended version available as MITRE technical report, 1997.
- [13] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. Technical report, MITRE Corp., Aug. 2003. Preliminary version appeared in *Proc. VERIFY 2002*.
- [14] J. D. Guttman, A. L. Herzog, and F. J. Thayer. Authentication and confidentiality via IPsec. In *ESORICS*, LNCS. Springer-Verlag, June 2000.
- [15] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conf. on Computer and Communications Security*, pages 190–199, 2000.
- [16] T. Markham and C. Payne. Security at the network edge: A distributed firewall architecture. In *DARPA Information Survivability Conference and Exposition (DISCEX II) Volume 1*, pages 279–286, Anaheim, California, June 12–14, 2001.
- [17] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, Oakland, California, May 2000.
- [18] P. Porras and P. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, MD, Oct. 1997.
- [19] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Jan. 1998.
- [20] M. Roesch. Snort: Lightweight intrusion detection for networks. In *USENIX LISA '99*, Nov. 1999. www.snort.org.
- [21] T. E. Uribe, S. Cheung, J. Levy, and A. Valdes. Intrusion tolerance and worm spread. In *Fast Abstracts, Dependable Systems and Networks*. IEEE, June 2003.
- [22] A. Wool. Architecting the Lumeta firewall analyzer. In *Proc. of the 10th USENIX Security Symposium*, Aug. 2001.