An expert system for analyzing firewall rules

Pasi Eronen and Jukka Zitting
Helsinki University of Technology
{pasi.eronen, jukka.zitting}@hut.fi

Abstract

When deploying firewalls in an organization, it is essential to verify that the firewalls are configured properly. The problem of finding out what a given firewall configuration does occurs, for instance, when a new network administrator takes over, or a third party performs a technical security audit for the organization. While the problem can be approached via testing, non-intrusive techniques are often preferred.

Existing tools for analyzing firewall configurations usually rely on hard-coded algorithms for analyzing access lists. In this paper we present a tool based on constraint logic programming (CLP) which allows the user to write higher level operations for, e.g., detecting common configuration mistakes. Our tool understands Cisco router access lists, and it is implemented using Eclipse, a constraint logic programming language.

The problem of analyzing firewall configurations lends itself quite naturally to be solved by an expert system. We found it surprisingly easy to use logic statements to express knowledge on networking, firewalls, and common configuration mistakes, for instance. Using an existing generic inference engine allowed us to focus on defining the core concepts and relationships in the knowledge base.

1 Introduction

Firewalls are essential for organizations that are connected to the Internet. However, it is not enough to simply have a firewall—it must also be configured properly. Firewall configurations are often written in a low-level language which is very hard to understand. For instance, the order of the rules is often very important. Thus, it is often quite difficult to find out which connections and services are actually allowed by the configuration.

This brings up two related problems: how to express the organization's security policy in a language understood by the firewall; and finding out what a given firewall configuration actually does. This second problem often occurs when a new network administrator takes over, for instance, or when a third party is performing a technical security audit for the organization.

In this paper, we present a tool which helps administrators in analyzing firewall rules. The tool is designed to be interactive: the administrator can ask questions about the network traffic permitted, and the tool answers, for instance, by listing which ports are allowed on a given host. No network traffic is generated; the analysis is based solely on the configuration files and topology information given by the user.

The tool is implemented using Eclipse, a constraint logic programming system, which presents an elegant framework for adding new rules and making complex queries. In addition to performing relatively simple operations on the list—for instance, finding rules which are never matched—the tool also includes expert knowledge about the field. This knowledge includes details of different network protocols (and their security implications) and common configuration mistakes seen in real life. The flexibility offered by logic programming makes adding new operations and checks easy.

The rest of this paper is organized as follows. In Section 2 we describe the background in firewalls, expert systems, and logic programming. We present the basic structure and concepts of our software, especially of the knowledge base, in Section 3. The different functions are demonstrated using a concrete example in Section 4. Related work is briefly discussed in Section 5, and Section 6 evaluates our approach in the light of the alternative solutions. We also give some ideas for future work. Finally, Section 7 contains our conclusions from this research.

2 Background

Due to space limitations, we assume the reader has some background in TCP/IP networking.

2.1 Firewalls

Firewalls usually function as routers which connect different network segments together. Based on their configuration, they restrict the traffic flowing between the different networks. Depending on the protocol layer they operate at, firewalls can be classified into packet filters, circuit proxies, and application level proxies [23]. Often these techniques are employed together. Since any organization connected to the Internet already has some kind of router, and

```
1  permit udp any host 10.0.0.1 eq 53
2  deny udp any host 10.0.0.2
3  permit udp any 10.0.0.0 0.0.0.255 eq 123
4  permit udp any host 10.0.0.2 eq 177
5  deny ip any any
```

Figure 1: An example of a Cisco router access list. Note that the fourth rule is never matched because of the second rule.

most routers have at least simple packet filtering capabilities, routers are often used in addition or instead of more complex firewall products. However, routers (and many other simple packet filters) lack good user interfaces for specifying the desired security policy.

Simple packet filters usually use simple ordered lists of rules. An example of a Cisco router access list is shown in Figure 1. When a packet is received, the list is scanned from the start to the end, and the action (either "permit" or "deny") associated with the first match is taken. If a packet doesn't match any of the rules, the default action is "deny". Often a "deny all" rule is included at the end of the list to make it easier to verify that a list has not been truncated. Separate lists can be specified for each network interface.

The rules can use the following fields from the IP protocol header: next level protocol (e.g., TCP or UDP), source and destination IP addresses, type-of-service, and precedence. In addition, some fields for upper level protocols, such as TCP and UDP port numbers can be used. For a more complete discussion of the syntax of the rules used by Cisco routers, see [4] or [17].

Since the first matching rule is always used, it is very easy to make mistakes when writing access lists, especially when the lists are long (several hundred rules is not uncommon). For instance, the fourth rule in Figure 1 is never matched because the packets are stopped at the second rule.

2.2 Expert systems

Expert systems are computer programs that are used to solve problems and answer questions in a problem domain that ordinarily requires human expertise. This goal is usually achieved by combining a logical inference engine with a knowledge base. The information in the knowledge base contains a set of known facts and a set of production rules that allow if-then inferences on the facts and other acquired information. [18]

Expert systems have been applied in a wide range of industrial and commercial problems. Typical applications include diagnosis, planning, scheduling, decision support, and process monitoring and control. [8]

Perhaps the most prominent application of expert systems in security has been in the field of intrusion detection. Axelsson's survey [1] gives a good overview of the field.

2.3 Constraint logic programming

Logic programming is a programming paradigm that uses logical inference to solve problems. Instead of giving the computational steps required to solve a problem, a logic program gives the logical facts and dependencies that describe a solution and uses an inference engine to solve the problem.

Constraint logic programming (CLP) is logic programming extended with constraint satisfaction capabilities. Constraint satisfaction is a more general operation than unification used in normal logic programming. The unification operation is able to resolve the structure and finally the exact value of a variable, and constraint satisfaction adds the possibility to constrain the possible values of a variable. [6]

The ability of setting truly generic constraints is normally not feasible, and therefore real CLP systems only allow certain types of constraints, such as linear equalities and inequalities over real numbers.

A complete explanation of how a constraint logic programming system works is beyond the scope of this paper. Interested readers should check [6] or a textbook.

Applications of logic programming and other computational logic techniques in computer security range from generating security test cases for the AIX operating system [7] to analyzing cryptographic protocols (e.g., [21]).

3 Implementation

Our tool has three major components: a knowledge base, an inference engine, and a user interface. This structure is typical of expert systems. The knowledge base is a collection of facts and if-then production rules that represent stored knowledge about the problem domain. The inference engine is the processing unit that solves presented problems by making logical inferences on the given facts and rules stored in the knowledge base. The user interface controls the inference engine and manages system input and output.

The knowledge base, the core of the system, is described below in detail. The inference engine and the user interface are briefly discussed at the end of this section.

3.1 Knowledge base

The knowledge base of our system is a collection of logical rules and facts expressed in the Prolog-based programming language Eclipse [24]. The declarative language makes it easy to express knowledge without mixing the information with computational details.

The knowledge base contains both static and dynamic information. Knowledge about different network protocols and common configuration mistakes, for instance, is static; the access lists themselves and information about the network topology is given by the user when starting the system.

In this section, we use a combination of Prolog and standard logical notation to show examples of the rules of our knowledge base. This should make it easier for readers without Prolog background to understand the expressions. However, to keep with the Prolog style we have omitted existential qualifiers and expect that all variables (words starting with a capital letter) are existentially qualified.

3.1.1 Representing packets

The simplest and most fundamental concept of our system is a packet. IP packets are represented as 6-tuples (*protocol, source IP, destination IP, source port, destination port, flags*), where all entries are numbers. The *flags* field is used only for TCP connections: 0 represents a packet which starts a new connection (i.e., the SYN bit it set and ACK bit is cleared), and 1 represents packets belonging to existing connections.

The following predicate defines this basic concept in the knowledge base:

$$\begin{aligned}&\text{packet}(\ (\mathit{Proto},\mathit{Src},\mathit{Dst},\mathit{SrcPort},\mathit{DstPort},\mathit{Flags})\ ) \leftarrow \\ &\quad 0 \leq \mathit{Proto} \leq 255 \wedge \\ &\quad 0 \leq \mathit{Src} \leq 4294967295 \wedge \\ &\quad 0 \leq \mathit{Dst} \leq 4294967295 \wedge \\ &\quad 0 \leq \mathit{SrcPort} \leq 65535 \wedge \\ &\quad 0 \leq \mathit{DstPort} \leq 65535 \wedge \\ &\quad 0 \leq \mathit{Flags} \leq 1\end{aligned}$$

In effect, the declaration spans a 6-dimensional finite, discrete space that we call the "packet space". Individual packets can be thought of as points in the packet space.

Parts of the packet space can be specified by giving more constraints. For example, the following declaration specifies the set of TCP (protocol 6) packets whose destination is the HTTP port (80) of host 10.0.0.1 (167772161):

$$\begin{aligned}&\text{packet}(\ (\mathit{Proto},\mathit{Src},\mathit{Dst},\mathit{SrcPort},\mathit{DstPort},\mathit{Flags})\ ) \wedge \\ &\quad \mathit{Proto} = 6 \wedge \mathit{Dst} = 167772161 \wedge \mathit{DstPort} = 80\end{aligned}$$

3.1.2 Representing access lists

Access lists are collections of packet filtering rules. A rule consists of a 6-tuple of ranges for matching packets and an action token ("permit" or "deny") for specifying how the packet filter should treat packets matched by the rule. The access list rules are processed in the order they are given until a match is found. The first matching rule specifies the action taken by the packet filter.

Our system represents access list rules as constraints on the packet space defined above. Each rule is associated with a part of the packet space (a 6-dimensional hypercube defined by the 6-tuple of ranges) and the specified action token. For instance, the rule "permit udp any 10.0.0.0 0.0.0.255 eq 53" would be associated with the range "(17, 0..255.255.255.255, 10.0.0.1, 0..65535, 53, 0..1)" and the token "permit". The system also remembers the line number and contents of the rule declaration in the access list file.

At first we tried representing the entire access list just as an ordered collection of rules. We defined the packet matching and list traversal algorithms as rules in the knowledge base. This approach turned out to be counterintuitive and inefficient. To overcome these difficulties and to get a more elegant and simple representation of access lists we decided to "decorrelate" the rules; that is, split them into non-overlapping rules [22].

When an access list file is loaded, the rules are first decorrelated and the resulting rule set is added to the knowledge base as *match_list* predicates like the one shown below:

$$\begin{aligned}&\text{match_list}(100, \text{permit}, (\mathit{Proto},\mathit{Src},\mathit{Dst},\mathit{SrcPort},\mathit{DstPort},\mathit{Flags})) \leftarrow \\ &\quad \mathit{Proto} = 6 \wedge \\ &\quad 3232235776 \leq \mathit{Src} \leq 3232236031 \wedge \\ &\quad \mathit{Dst} = 167772161 \wedge \\ &\quad 0 \leq \mathit{SrcPort} \leq 65535 \wedge \\ &\quad \mathit{DstPort} = 23 \wedge \\ &\quad 0 \leq \mathit{Flags} \leq 1\end{aligned}$$

This predicate corresponds to the access list rule "access-list 100 permit tcp 192.168.1.0 0.0.0.255 host 10.0.0.1 eq 23". We use standard logic notation here to emphasize the fact that the knowledge base is a collection of logic statements. The actual syntax used in defining the knowledge base is a straightforward mapping of the corresponding logical notation.

It is important to notice that this rule has a very different meaning in CLP and ordinary Prolog—indeed, many actual CLP systems also use a different syntax (such as "#<=" instead of ordinary "<="). While the ordinary Prolog rule could be used to *test* if a given packet matches the rule, this CLP rule can also answer the question "which kinds of packets the rule matches".

Based on these predicates the inference engine is able to find answers to questions like "What is the action for this packet?", "What packets are permitted by this access list?", or "From which sources are packets to this destination permitted?". Using the auxiliary information stored for each rule, the inference engine can even answer questions like "Why is this packet denied?", or "What rules permit packets from this network?".

3.1.3 Representing topology and connections

Firewalls often have many network interfaces, and to effectively analyze the access lists, it is necessary to know which networks or address ranges are located behind which interface.

This information is represented in the knowledge base by two simple predicates, *network(interface, network)* and *network_internet(interface)*, the latter one indicating a connection to the Internet (where addresses not explicitly mentioned are located).

The access lists combined with the topology information can be used to construct higher level constructs.

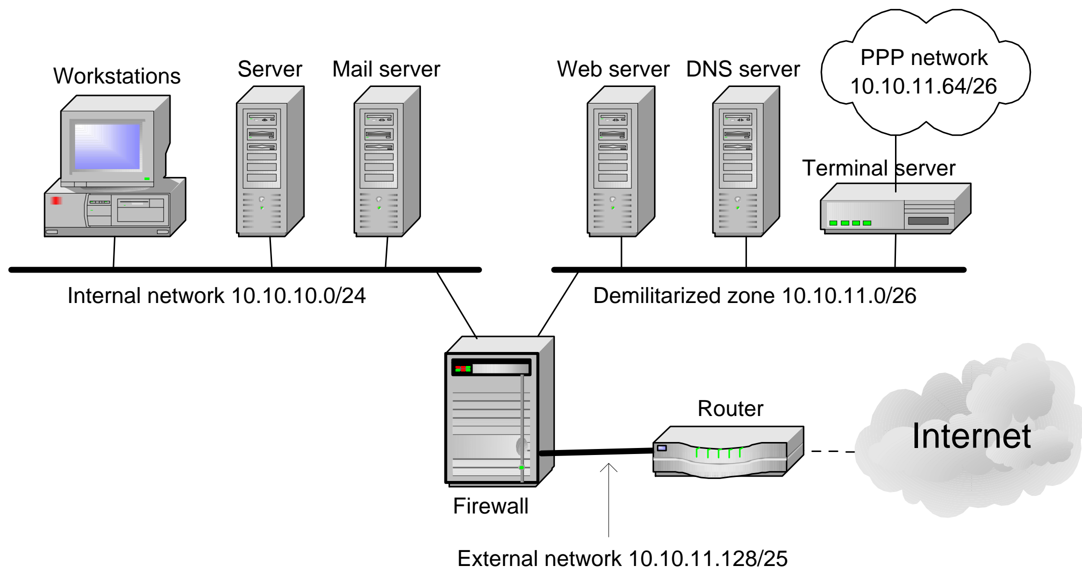

Figure 2: The network of a hypothetical small company.

For example, a predicate indicating that a TCP connection from a certain source address to some host and port is possible, can be implemented as follows. Here *address_in_interface(address, interface)* is a simple predicate which simply uses the information given by the *network* predicates.

tcp_connection(*Src, Host, Port*) ←
  address_in_interface(*Host, HostInterface*) ∧
  interface(*SrcInterface, AccessList*) ∧
  address_in_interface(*Src, SrcInterface*) ∧
  SrcInterface ≠ *HostInterface* ∧
  match_list(*AccessList*, (6, *Src*, *Host*, _, *Port*, 0), permit)

Note that this predicate only checks the action on the first SYN packet and expects that the firewall will permit the other packets of the connection if the first one is permitted. This is a common configuration, and it can be checked using another predicate. Requiring the "permit" action on the other packets as well would make this predicate less useful in detecting erroneous configurations.

This predicate can be used to answer questions such as "Which hosts can connect to the Telnet port on our web server?" or "What connections are allowed from host X to our intranet?".

3.1.4 Extending the knowledge base

The knowledge base of our system can be extended by adding new higher level rules and facts. The existing concepts and knowledge can be used as a basis when designing new functionality. For example, the connection concept defined in the previous section is a very good starting point for new rules.

The following predicate tests that a DNS server can be accessed correctly using both UDP and TCP connections:

dns_server(*Host, Src*) ←
  udp_connection(*Src, Host*, 53) ∧
  tcp_connection(*Src, Host*, 53)

Another useful predicate is the following one, that can be used to query all permitted incoming TCP connections from the Internet:

incoming_tcp(*Host, Port*) ←
  internet_address(*Src*) ∧
  tcp_connection(*Src, Host, Port*)

A slightly more complex example, with corresponding Eclipse source code, is given in Section 4.4.

3.2 Inference engine

The inference engine of our tool is Eclipse, a constraint logic programming language based on Prolog [24]. By using an existing general purpose tool we were able to concentrate to the core problems and avoid laborious implementation work. We also gained a standard and extensible platform with proven performance and quality. Furthermore, porting our system to some other Prolog-based constraint logic programming language should be relatively easy.

3.3 User interface

The user interface of our tool contains a preprocessor for parsing access list files, a set of input and output handling routines, and a simple command prompt interface for managing the system. Together the user interface components allow an administrator to interactively inspect the access lists.

The operations implemented in our tool can be classified in three categories: 1) queries and operations on access lists themselves, 2) queries about the network data flows allowed by access lists, and 3) expert rules for recognizing and solving common configuration problems. The operations in these categories are described using a concrete example in Section 4.

Most of the user interface is written in Prolog using the I/O libraries of Eclipse. The preprocessor for parsing of text files containing Cisco access lists is written in Perl, since Perl has nicer facilities for string processing, and we had much more experience in Perl programming.

4 User interface functions

The operations are best described using a concrete example. Figure 2 illustrates the network of a hypothetical small company. The network has three segments: internal network (containing employee workstations and servers), a demilitarized zone (e.g., web server) and external network (connection to the Internet service provider).

When the system is started, the administrator first loads the access lists, and then gives system a description of the network's topology (commands typed by the user are shown in boldface). This information can also be stored in a file for further use.

```
$  eclipse -b all.pl -e shell

?  read_list int_l cisco "acl_int.txt"
Read 11 rules, decorrelated into 31 clauses

?  read_list dmz_l cisco "acl_dmz.txt"
Read 15 rules, decorrelated into 70 clauses

?  read_list ext_l cisco "acl_ext.txt"
Read 16 rules, decorrelated into 123 clauses

?  interface dmz dmz_l
?  interface int int_l
?  interface ext ext_l

?  network int 10.10.10.0/24
?  network dmz 10.10.11.0/26
?  network dmz 10.10.11.64/26
?  network ext 10.10.11.128/25
?  network ext internet
```

4.1 Network properties

The basic problem in analyzing an access list is to recognize the network data flows allowed by the list. The following are examples of common questions:

- Which services are accessible on a given host?
- Is a given host/network accessible from another given host/network?
- What kinds of traffic is allowed between two networks?
- From which networks is a given host accessible?

These questions can easily be answered by using the *match_list* predicate itself. The inference engine will query the knowledge base to find all possible substitutions for the free variables in a given goal. The user interface components collects the results and presents them in an easily readable format.

For instance, the following command can be used to show all services on host 10.10.11.8 (the web server in DMZ).

```
?  show_services 10.10.11.8

UDP services on 10.10.11.8:

port          from
any           10.10.10.0/24

TCP services on 10.10.11.8:

port          from
<= 79         10.10.10.0/24
80            0.0.0.0 - 10.10.10.255
              10.10.11.128 - 255.255.255.255
>= 81         10.10.10.0/24
```

The report shows not only which services are allowed, but also from which source addresses they can be used. Thus, the port 80 is correctly accessible everywhere (as it should be), but also all other ports can be accessed from the internal network. Whether this is a good idea depends on the circumstances; in many cases it is probably OK, but in some environments, such as schools, end users are usually not trusted very much.

4.2 Configuration problems

Operations mentioned in the previous section can be implemented directly with a specification of the behavior of access lists. They do not require or contain "expert knowledge". In addition to these simple operations, the expert system can also be used to recognize common configuration problems and mistakes. Simple such operations are for example:

- Firewall doesn't block directed broadcasts (that is, packets destined for the broadcast address of some connected network).
- Insufficient prevention of address spoofing, both for incoming and outgoing packets (ingress filtering).
- Domain Name System (DNS) server is reachable only by UDP, not TCP (often undetected, since it works just fine 99% of the time).

Predicates for checking these kinds of configuration problems can easily be built on top of the previously defined predicates about network properties. For example, the proper blocking of directed broadcasts can be checked as shown below.

```
?  check_broadcasts

10.10.10.0/24       OK
10.10.11.0/26       Allowed from [ext]
10.10.11.64/26      OK
10.10.11.128/25     OK
```

The report shows that broadcast packets to 10.10.11.0/26 are allowed from the external interface, but other broadcasts are properly blocked.

Section 4.4 describes how these operations can be easily built from existing rules.

4.3 Access list properties

The properties of the access list itself can also be analyzed. The most useful operation on the access list itself is recognizing rules which are never matched. This usually indicates a configuration mistake (for instance, the fourth rule in Figure 1).

In the following example, the network administrator has probably intended to allow SMTP and IMAP protocols (ports 25 and 143) to host 10.10.10.5, but some earlier rule blocks these packets.

```
?  show_lists

int:    11 rules
dmz:    15 rules
ext:    16 rules

?  show_never_matched dmz

Rules which are never matched:
12: permit tcp 10.10.11.64 0.0.0.63 host 10.10.10.5 eq 25
13: permit tcp 10.10.11.64 0.0.0.63 host 10.10.10.5 eq 143
```

4.4 Defining new predicates

While the operations described in the previous sections are clearly useful to administrators, the real power of our tool lies in the ability to easily add higher level rules to the knowledge base. We demonstrate this by describing a rule to check for *address spoofing*.

Informally, hosts located behind an interface can spoof (or forge) addresses from a certain network, if some packet with that network source address is accepted, but that network is actually located behind some other interface. In Eclipse, this can be expressed as follows.

```
can_spoof(Interface, NetworkString) :-
      network(RealInterface, NetworkString),
      network_string(Network, NetworkString),
      interface(Interface, List),
      match_list(List, [_, Network, _, _, _, _], permit),
      Interface \= RealInterface.
```

For those unfamiliar with Prolog, ":-" is the same as "←", the commas at the end of each line represent conjunction ("and"), and the underscore is a wildcard which matches any value. This predicate can be used in Eclipse directly to find out if spoofing is allowed:

```
[eclipse]:  can_spoof(If, Net).
If = ext
Net = "10.10.11.0/26"
```

Eclipse replies with a substitution which makes the query true. The result shows that the external interface is missing a "deny ip 10.10.11.0 0.0.0.63 any" rule. Add a couple of lines of output formatting, and we have a ready check_address_spoofing command!

```
?  check_address_spoofing

10.10.10.0/24       OK
10.10.11.0/26       Can be spoofed from [ext]
10.10.11.64/26      OK
10.10.11.128/25     OK
```

5 Related work

Much of the research in firewalls has focused either on performance or the problem of expressing an organization's security policy in a language understood by a firewall, i.e., tools for creating access lists [2, 3, 11]. There are also some commercial products available, such as Cisco's Access Control List Manager [5] and Secure Policy Manager [16]. Modern firewall products usually allow the specification of rules using a graphical user interface.

The work most similar to ours has been done by Mayer, Wool, and Ziskind [19]. Their firewall analysis engine is based on graph algorithms, and thus representing any expert knowledge or rules is harder than in the logic programming. On the other hand, their tool has at least some support for network address translation (NAT), and is being turned into a commercial product by Lumeta corporation [25].

Similar work based on a logic background has been done by Hazelhurst et al. [13, 14, 15]. They have used ordered binary decision diagrams to analyze access lists. This representation allows efficient manipulation of the lists, and finding redundant rules is easy, for instance. However, the system does not allow expressing custom rules using a logic programming syntax.

Several researches have also implemented tools for describing the contents of an access list based on other approaches. Guttman describes an approach for generating filters based on a security policy and verifying that a packet filter implements some security policy [11]. Molitor describes a tool which prints a more human readable description of an access list [20]. Bartal et al. have written a "rule illustrator", a tool for drawing an access list in a graphical form [2].

The low-level implementation of packet filters, called the *packet classification problem* has also received quite a lot

of attention. However, most of the work has focused on performance issues and hardware implementations; Feldman and Muthukrishnan [10] give a recent summary and a good bibliography of the topic. Although performance issues are not directly related to security, the problem of detecting conflicts in packet filters is. Hari et al. [12] have applied these techniques to analyzing access lists from a security viewpoint, and Eppstein et al. [9] present a fast algorithm for detecting conflicts.

6 Evaluation and future work

The main benefits of our system are the use of logic programming and a generic inference engine. Logic programming makes it easy to express and extend the knowledge base of the system. The knowledge is expressed in a standard declarative language that closely matches the constructs of logic. A generic inference engine, in this case Eclipse, greatly reduced the implementation effort of our system. We were able to prototype and implement the system in just a couple of weeks of time, fully focusing on expressing knowledge and building the user interface. Our tool shows that solving these kinds of analysis problems does not require custom algorithms or data structures.

Our tool is still in the "research prototype" stage, demonstrating that using constraint logic programming for this problem is feasible. There are several possibilities for enhancements. A better user interface, which would hide the Prolog syntax for at least most operations, would make the tool easier to use. Also, supporting more firewalls and more complex rules would make the tool more useful in real-world environments.

We are also considering porting the system to some open source CLP implementation, because using Eclipse in commercial environments requires buying a license.

A very interesting development possibility would be to reverse the problem and use the CLP capabilities to plan, and possibly optimize, an access list that implements the higher level security policies expressed as logical constraints.

7 Conclusions

The ability to clearly analyze what a certain firewall configuration does is very important in many circumstances. While existing tools based on hard-coded algorithms can be valuable, this approach makes adding new functionality harder. For instance, it can be difficult to embed knowledge about common configuration mistakes.

We have taken a different approach, and used a general purpose constraint logic programming system. Logic programming makes it easy to write useful functions for analyzing rules, and expert knowledge can be represented in a very compact form. Our implementation, which analyzes Cisco router access lists, shows that this approach works also in practice.

Acknowledgements

We would like to thank Jonna Särs from Nixu Ltd. for her comments on earlier versions of this paper, and for providing a real-world access list example for testing.

References

[1] Stefan Axelsson. Intrusion detection systems: A taxonomy and survey. Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, Sweden, March 2000.

[2] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 17–31, Oakland, California, May 1999.

[3] Christopher J. Calabrese. A tool for building firewall-router configurations. *Computing Systems*, 9(3):239–253, Summer 1996. USENIX Association.

[4] Cisco Systems, Inc. Cisco IOS release 10.3 router products command reference. http://www.cisco.com/univercd/cc/td/doc/product/software/ios103/rpcr/, 1998.

[5] Cisco Systems, Inc. CiscoWorks2000 Access Control List Manager 1.2 overview. http://www.cisco.com/warp/public/cc/pd/wr2k/caclm/prodlit/aclm_ov.htm, November 2000.

[6] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[7] Janet A. Cugini, Shau-Ping Lo, Matthew S. Hecht, Chii-Ren Tsai, Virgil D. Gligor, Radhakrishna Aditham, and T. John Wei. Security testing of AIX system calls using Prolog. In *Proceedings of the Summer 1989 USENIX Conference*, pages 223–237, Baltimore, Maryland, June 1989.

[8] Robert S. Engelmore and Edward Feigenbaum. Expert systems and artificial intelligence. In *Knowledge-based systems in Japan*. Japanese Technology Evaluation Center, May 1993.

[9] David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 827–835, Washington, D.C., January 2001.

[10] Anja Feldman and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings of IEEE INFOCOM 2000*, pages 1193–1202, Tel Aviv, Israel, March 2000.

[11] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, California, May 1997.

[12] Adiseshu Hari, Subhash Suri, and Guru Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE INFOCOM 2000*, pages 1203–1212, Tel Aviv, Israel, March 2000.

[13] Scott Hazelhurst. Algorithms for analysing firewall and router access lists. Technical Report TR-Wits-CS-1999-5, Department of Computer Science, University of the Witwatersrand, South Africa, July 1999.

[14] Scott Hazelhurst, Adi Attar, and Raymond Sinnappan. Algorithms for improving the dependability of firewall and filter rule lists. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 576–585, New York, June 2000. IEEE Computer Society Press.

[15] Scott Hazelhurst, Anton Fatti, and Andrew Henwood. Binary decision diagram representations of firewall and router access lists. Technical Report TR-Wits-CS-1998-3, Department of Computer Science, University of the Witwatersrand, South Africa, October 1998.

[16] Susan Hinrichs. Policy-based management: Bridging the gap. In *Proceedings of the 15th Annual Computer Security Applications Conference*, Phoenix, Arizona, December 1999. IEEE Computer Society Press.

[17] Kent Hundley and Gil Held. *Cisco Access Lists Field Guide*. McGraw-Hill, March 2000.

[18] L. C. Jain. Introduction to knowledge-based systems. In *Proceedings of the Electronic Technology Directions to the Year 2000*, pages 18–27, Adelaide, Australia, May 1995. IEEE Computer Society Press.

[19] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 177–187, Oakland, California, May 2000.

[20] Andrew Molitor. An architecture for advanced packet filtering. In *Proceedings of the 5th USENIX UNIX Security Symposium*, Salt Lake City, Utah, June 1995.

[21] Pekka Nikander. Modelling of cryptographic protocols. Licenciate's thesis, Helsinki University of Technology, December 1997.

[22] Luis A. Sanchez and Matthew N. Condell. Security policy protocol. Work in progress, Internet draft ietf-ipsp-spp-00, http://www.ietf.org/proceedings/00jul/I-D/ipsp-spp-00.txt, July 2000.

[23] Christoph L. Schuba. *On the Modeling, Design, and Implementation of Firewall Technology*. Doctoral dissertation, Purdue University, December 1997.

[24] Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, August 1997.

[25] Avishai Wool. Architecting the Lumeta firewall analyzer. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.