2008

Formal analysis of firewall policies

Robert Mathias Marmorstein

College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
https://dx.doi.org/doi:10.21220/s2-w2z8-5904

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.
Formal Analysis of Firewall Policies

Robert Mathias Marmorstein
Aberdeen, South Dakota

Master of Science, College of William and Mary, 2003
Bachelor of Arts, Washington and Lee University, 2000

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
May 2008
This Dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Robert Mathias Marmorstein

Approved by the Committee, April 2008

Committee Chair
Dr. Phil Kearns, Computer Science
College of William and Mary

Dr. Weizhen Mao, Computer Science
College of William and Mary

Dr. Haining Wang, Computer Science
College of William and Mary

Dr. David Coppit, Computer Science
College of William and Mary

Dr. Jean Mayo, Computer Science
Michigan Technological University
This dissertation describes a technique for formally analyzing a firewall security policy using a quasi-reduced multiway decision diagram model. The analysis allows a system administrator to detect and repair errors in the configuration of the firewall without a tedious manual inspection of the firewall rules.

We present four major contributions. First, we describe a set of algorithms for representing a firewall rule set as a multi-way decision diagram and for solving logical queries against that model. We demonstrate the application of these techniques in a tool for analyzing iptables firewalls. Second, we present an extension of our work that enables analysis of systems of connected firewalls and firewalls that use network address translation and other packet mangling rules. Third, we demonstrate a technique for decomposing a network into classes of equivalent hosts. These classes can be used to detect errors in a firewall policy without apriori knowledge of potential vulnerabilities. They can also be used with other firewall testing techniques to ensure comprehensive coverage of the test space. Fourth, we discuss a strategy for partially automating repair of the firewall policy through the use of counterexamples and rule history.

Using these techniques, a system administrator can detect and repair common firewall errors, such as typos, out-of-order rules, and shadowed rules. She can also develop a specification of the behaviors of the firewall and validate the firewall policy against that specification.
This dissertation is dedicated to Mrs. Sharon Lust, my junior high reading teacher. Mrs. Lust is an exemplary teacher. In the classroom she is a brilliant and engaging instructor.

Her reading class made a lasting impact on me. She demands excellence from her students, but helps them to develop the skills they need to meet her expectations. Outside the classroom she is her students' strongest advocate. For years after I finished her class she opened doors for me and other students by recommending us for contests, journals, and other academic opportunities. She is a model of everything I hope to be as a teacher and scholar.
# Table of Contents

Acknowledgments vii

1 Introduction 3

1.1 iptables 5

1.1.1 Creating an iptables Policy 6

1.1.2 iptables Operation 6

1.1.3 Firewall Errors 7

2 Representations of a Firewall Policy 11

2.1 Other Models 12

2.1.1 Binary Decision Diagrams 12

2.1.2 Interval Decision Diagrams 13

2.1.3 Firewall Decision Diagrams 13

2.1.4 Quasi-Reduced Multi-way Decision Diagrams 14

2.2 Notation 14

2.3 Multi-way Decision Diagrams 16

2.4 Implementation 19

2.5 Building an MDD for a Filter Rule 19

2.6 Correctness of MDD generation for simple terminal rules 21

2.7 Inserting a Rule into the Chain MDD 23

2.8 Correctness of Assign operator 24
## 2.9 Intersection

2.9.1 Correctness of the Special Intersection Algorithm

- Base Case
- Induction Hypothesis
- Induction Step

## 2.10 Replace Algorithm

2.11 Correctness of Replace Algorithm

- Base Case
- Induction Hypothesis
- Induction Step

## 2.12 Chain Building Algorithm

2.13 Correctness of Chain Building Algorithm

- Base Case
- Induction Hypothesis
- Induction Step

### 3 Linux Firewall Analysis using Queries

3.1 ITVal, An Open Source Tool

3.2 Query Language

3.3 Query Statements

3.4 Simple Conditions

3.5 Complex Queries

3.6 Group and Service Definitions

3.7 Implementation

3.7.1 ACCEPTED and DROPPED primitives

3.7.2 Correctness of the Projection Operation

- Base Case
- Induction Hypothesis
3.7.3 Other Primitives ............................................. 57
3.7.4 Correctness of the Query MDD Generation Algorithm .......... 59
    Base Step ................................................. 59
    Induction Hypothesis ..................................... 59
    Induction Step ............................................. 59
Base Step ..................................................... 61
    Induction Hypothesis ..................................... 61
    Induction Step ............................................. 61
3.8 Combining Queries ........................................... 61
3.9 Correctness of the Intersection Operation ......................... 64
    Base Step ................................................. 65
    Induction Hypothesis ..................................... 65
    Induction Step ............................................. 65
    Base Step ................................................. 66
    Induction Hypothesis ..................................... 66
    Induction Step ............................................. 66
3.10 Performance ................................................ 67
3.11 Application of ITVal Queries ................................ 67
3.12 Advantages of the Query Language ................................ 69

4 Composition of Firewalls ........................................ 71
    4.1 Analyzing Firewall Systems using Existing Tools ................ 74
    4.2 Composing Nested Firewalls ............................. 75
    4.2.1 Correctness of the Composition Operation ............... 78
4.2.2 Network Address Translation ........................................ 80
4.2.3 Correctness of the NAT algorithm ........................................ 85
   Base Case .................................................. 86
   Induction Hypothesis ........................................ 86
   Induction Step ........................................ 86
   Base Case .................................................. 88
   Induction Hypothesis ........................................ 88
   Induction Step ........................................ 88
4.3 Nested Composition with Network Address Translation ............. 89
4.3.1 Correctness of NAT with Composition Operations ................... 90
4.4 Analyzing Firewall Systems with ITVal .................................. 91

5 An Equivalence Class Approach to Policy Testing ......................... 92
   5.1 Host Classification ........................................ 96
   5.2 Policy-Based Host Classification .................................... 97
     5.2.1 Calculating Host Classes .................................... 98
     5.2.2 Structure-Based Classification ............................. 100
     5.2.3 Implementation .......................................... 101
   5.3 Correctness of Equivalence Class Generation ....................... 104
     Base Case ................................................ 105
     Induction Hypothesis ........................................ 105
     Induction Step ........................................ 106
   5.4 Error Detection ............................................. 107
     5.4.1 Detecting Remotely Accessible Services ..................... 107
     5.4.2 Detecting Shadowed Rules .................................. 109
     5.4.3 Detecting Outdated Services ................................ 110
   5.5 Using Equivalence Classes with Other Tools ....................... 110
   5.6 Advantages of Policy-based Classification .......................... 113
6 Guided Repair of Firewall Policies

6.1 Existing Techniques ........................................... 114
6.2 Partially Automated Firewall Repair ......................... 117
6.3 Directed Repair .................................................. 117
6.4 Relevant Counterexamples ................................. 121
6.5 Rule History .................................................... 123
6.6 Implementing Rule History .................................. 124
6.7 Correctness of the History MDD representation .......... 127
   Base Case ...................................................... 128
   Induction Hypothesis ....................................... 128
   Induction Step ............................................... 128
6.8 Directed Repair and Equivalence Classes ............... 130

7 Conclusion and Future Work .......................... 134

A Query Selection for Effective Analysis ............. 138

A.1 Using ITval .................................................. 138
A.2 Constructing Queries .................................... 139
A.3 Avoiding pitfalls .......................................... 140
   A.3.1 Accounting for State ............................... 141
   A.3.2 Accounting for Spoofing .......................... 142
A.4 Putting it All Together .................................. 143
A.5 Conclusion .................................................. 147

Bibliography .................................................... 148
ACKNOWLEDGMENTS

My wife Beth has very patiently endured the uncertainty and inconveniences of being married to a graduate student. I am very grateful for her love and her faithful support in finishing this dissertation. It is amazing to have in-laws who understand LaTeX, equivalence classes, and induction proofs. Beth's parents, Marty and Ginny, have been wonderfully generous and loving with their time and advice. I am also grateful to my parents, whose faithful prayers for this dissertation are undoubtedly responsible for its completion.
FORMAL ANALYSIS OF FIREWALL POLICIES
Chapter 1

Introduction

System administrators rely very heavily on firewalls for protection against external and internal threats to the network. This reliance has led to the development of sophisticated and powerful filtering software for enforcing a security policy on the packets that enter a network. Features such as stateful inspection and network address translation (NAT) greatly enhance the power and flexibility of these filtering tools. Unfortunately, a packet filter only provides adequate protection if the policy that it implements is correct. If the policy is not sufficiently restrictive, attackers can compromise the network by exploiting errors in the policy. On the other hand, if the policy is too restrictive, the firewall may interfere with legitimate traffic.

A policy that contains errors exposes the network to many kinds of threats from both sides of the network perimeter. External threats, such as denial-of-service attacks or SSH brute force attacks can take advantage of these weaknesses to compromise important servers and workstations. Internal threats posed by compromised systems or malicious users with access to internal resources can amplify existing problems and extend them throughout the network.

The importance of ensuring that the firewall policy is correct has led to the development of formal firewall testing procedures [57, 50] that employ many different kinds of tools to verify that the firewall policy meets the security requirements of the network. Unfortunately, detecting and repairing errors is a very difficult process, which requires a significant
expenditure of time and resources. Furthermore, it is possible to introduce new errors into
the policy while attempting to repair old ones.

Inspecting the firewall policy manually is especially time-consuming for large networks
with many hosts and multiple firewalls. The difficulty of repairing a firewall often leads to
poor security practices. In order to avoid errors, some experts advocate using a sophisticated
firewall design process \([35, 39]\) in which the firewall design passes through several testing
phases before deployment on production systems, but this approach requires a significant
allocation of resources to firewall policy development. In fact, some experts \([46, 47, 20]\)
suggest that security can be better protected by reducing or eliminating the use of firewalls
for security and instead relying on other techniques, such as patching, to defend the network.

Some system administrators work around these problems by deploying generic firewall
policies obtained from the Internet \([14, 38, 55]\) or using graphical firewall policy wizards \([29,\n54, 31, 21]\). These policies protect a system against common threats, but are not tailored to a
network's particular needs. This means that they seldom implement restrictive policies and
leave important services vulnerable to attack. Adapting these policies to the requirements
of a particular environment can sometimes be as challenging as developing a correct policy
from scratch.

Configuration of the firewall can sometimes be made simpler using policy visualization
tools. Visualization tools such as PolicyVis \([53]\) and FireVis \([45]\) allow a system administra-
tor to better understand the behavior of a firewall, which can make it easier to detect errors
in the policy. Unfortunately, many policies are so complicated that even these visualization
tools can be difficult to use.

There are several reasons firewall configuration is so difficult. One reason is that firewall
policies are written in a complex language which the administrator must master to be able
to maintain the policy. Another reason is that policies can be extremely long, containing
hundreds or even thousands of rules.

Maintaining a restrictive firewall policy is especially difficult on networks which change
rapidly as users demand new services and systems are brought online or removed from
service. Securing these networks often requires the use of multiple interacting firewalls, each of which has a long and complicated policy. Every the policy is modified, an opportunity for error is created in which the system administrator may inadvertently introduce new and potentially devastating problems into the policy.

Errors introduced into the policy can open the doors for a malicious intruder to compromise a server or launch a denial of service attack. Even a simple typo can expose a network to a barrage of hostile traffic. In some firewall systems, merely reversing the order of two rules can completely invalidate the policy.

1.1 iptables

The Linux kernel implements an interface called netfilter [58, 43, 3], which provides the internal hooks for the iptables firewall. The iptables packet filter supports many advanced features such as packet mangling and stateful inspection and is freely available as part of any recent Linux distribution. This makes it an extremely cost-effective solution for organizations that need effective security, but cannot afford expensive commercial products, such as Cisco's PIX firewall or a Checkpoint firewall.

The rising popularity of Linux as a desktop environment has made iptables the firewall of choice for many home users as well as business users. Unfortunately, because a restrictive firewall policy can be difficult to construct and maintain, many of these firewalls implement generic, minimally-restrictive, and infrequently tested policies. As a result, these policies provide very little protection for the hosts they are deployed to secure.

In this dissertation, we will explore ways to simplify the testing, analysis, and repair of iptables firewalls. In order to follow the algorithms and examples we will use, it is helpful to have some understanding of the configuration and operation of an iptables firewall.
1.1.1 Creating an iptables Policy

To create an iptables firewall policy, the system administrator constructs chains of filtering rules. Each rule in a chain identifies a filtering action and a set of packets to which the action should be applied. The action ACCEPT indicates that the firewall should allow the packet to pass through the firewall and enter the network. The action DROP indicates that the firewall should discard the packet. The user can also specify that the packet should be passed to some other firewall chain for processing.

iptables provides three built-in chains: the INPUT chain, the FORWARD chain, and the OUTPUT chain. Packets that are intended for the firewall itself are processed by the INPUT chain. Packets intended to pass through the firewall on their way to some other host are processed by the FORWARD chain. The OUTPUT chain processes packets generated by the firewall host. The rules in each of these chains can ACCEPT a packet, DROP a packet, or pass it to a user-defined chain for further processing. The three built-in chains also have a default policy that determines what action is taken on packets that do not match any rule of the chain.

1.1.2 iptables Operation

When determining what action should be taken on a particular packet, the rules in a chain are considered in first-to-last order. The first rule that matches will cause processing of that packet to cease or pass to some other chain. This means that inserting rules in the wrong order can seriously impact the behavior of the firewall.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>TCP</td>
<td>192.168.1.0/24</td>
<td>192.168.2.0/24</td>
<td>TCP dpt:ssh</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
<td>TCP dpt:http</td>
</tr>
<tr>
<td>DROP</td>
<td>all</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Example FORWARD chain

An example FORWARD chain is shown in figure 1.1. This example policy secures a network 192.168.2.0/24 against threats from the outside world. In the example, the default
policy is ACCEPT. This means that a packet will be allowed to enter the network unless show rule in the policy specifically drops it. Each of the rules in the chain has a target and a set of match conditions. The target specifies an action that should be applied to packets that match all of the conditions. The conditions specify criteria for determining which packets match the rule. For instance, the first rule has the target ACCEPT. It also has a protocol match, a source match, a destination match, and a flag match.

The protocol match specifies that only TCP packets should be considered. The source match ensures that the rule is only applied to packets from the trusted 192.168.1.0/24 subnet. The destination match indicates that only packets sent to hosts on the 192.168.2.0/24 subnet will be processed. Finally, the additional match “TCP dpt:ssh” ensures that only SSH packets will be matched. In other words, the first rule of the chain specifies that ssh traffic should be allowed to the protected network from hosts on trusted subnet 192.168.1.0/24.

Similarly, the second rule specifies that http traffic should be allowed from any host. The third rule blocks all other traffic to the protected network.

1.1.3 Firewall Errors

Figure 1.2 shows the policy of a firewall that secures an internal network 192.168.2.0/24 from intrusions by hosts on an unsecured wireless network 192.168.1.0/24. All traffic, including HTTP traffic, should be dropped from that insecure network. Rule 1 drops any incoming ICMP packets. Rule 2 drops traffic from the insecure network. The remaining rules secure various services and allow access to the web server. All other traffic is dropped, unless it comes from a trusted subnet 113.192.10.0/24.

Suppose the administrator decides to modify this configuration to allow trusted machines to send IPP printing traffic (on port 631) to the secure network. If she inserts an accept rule in the wrong place, she can produce the incorrect configuration in figure 1.3. This configuration allows printing service from the insecure network, because the new rule has been inserted before the rule which restricts the insecure subnet. Switching rules 2 and 3 yields a correct configuration. This sort of error becomes harder to detect as the number
Chain FORWARD (policy DROP):

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DROP</td>
<td>ICMP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>2</td>
<td>DROP</td>
<td>all</td>
<td>192.168.1.0/24</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>3</td>
<td>DROP</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>4</td>
<td>ACCEPT</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>5</td>
<td>DROP</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>6</td>
<td>ACCEPT</td>
<td>all</td>
<td>113.192.10.0/24</td>
<td>192.168.2.0/24</td>
</tr>
</tbody>
</table>

**Figure 1.2:** A sample firewall that secures subnet 192.168.2.0/24 against intrusions from untrusted network 192.168.1.0/24

of rules grows and the complexity of their structure increases.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DROP</td>
<td>ICMP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.2.0/24</td>
<td>TCP dpt:631</td>
</tr>
<tr>
<td>3</td>
<td>DROP</td>
<td>all</td>
<td>192.168.1.0/24</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>4</td>
<td>DROP</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>5</td>
<td>ACCEPT</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>6</td>
<td>DROP</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>7</td>
<td>ACCEPT</td>
<td>all</td>
<td>113.192.10.0/24</td>
<td>192.168.2.0/24</td>
</tr>
</tbody>
</table>

**Figure 1.3:** A misconfigured firewall that allows the untrusted network to access printing services

Consider also the firewall rule set described in figure 1.4, which protects an internal subnet 192.168.2.0/24 from the outside world. Can hosts on the protected network send SMTP traffic through the firewall? At first glance, it appears that hosts from 192.168.2.0/24 can access SMTP (they are granted access in rule 5). That rule, however, only grants access if the connection is in an ESTABLISHED state. In order for a host to transmit, SMTP traffic to an outside host, it must first establish the connection. But this cannot be done, because NEW connections will be dropped by the default policy of the firewall. If the system administrator desires to allow SMTP traffic from protected hosts, the policy must be changed to allow creation of new connections from hosts on the 192.168.2.0/24 subnet.

While the problems in these example policies could easily be avoided by a careful system administrator, far more complex errors can exist in a real-world firewall policy. As the
**Figure 1.4:** A stateful rule set which allows SMTP access only for established connections.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCEPT</td>
<td>ICMP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
<td>TCP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TCP dpt:ssh</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>flags:SYN,ACK/SYN</td>
</tr>
<tr>
<td>3</td>
<td>ACCEPT</td>
<td>UDP</td>
<td>anywhere</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>4</td>
<td>ACCEPT</td>
<td>all</td>
<td>113.117.1.4</td>
<td>192.168.2.0/24</td>
</tr>
<tr>
<td>5</td>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.2.0/24</td>
<td>anywhere</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TCP dpt:smtp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>state ESTABLISHED</td>
</tr>
</tbody>
</table>

A firewall policy becomes longer and more complicated, it becomes more difficult to implement and make changes to the policy without introducing mistakes. In a survey of 37 corporate firewalls, Wool [60] discovered an average of 7 configuration errors per system. While his study examined only Checkpoint and PIX firewalls, it is not unreasonable to assume that other firewall systems have comparable error rates.

In this dissertation, we explore techniques for performing a detailed formal analysis of an iptables firewall policy and for detecting errors in the firewall policy using an equivalence class decomposition of the hosts on a network. These analytical techniques have many practical applications to the problem of detecting and correcting firewall errors. In examining this area, we also address several theoretical issues such as how to efficiently model a firewall policy in software, how to derive useful queries for testing the firewall policy, and the tradeoff between producing insufficiently detailed output and producing more information than a user can easily process.

To this end, we explore four major areas. First, we describe a multi-way decision diagram representation of the firewall policy which allows us to answer logical queries about the behavior of the firewall. Second, we describe the application of this MDD model to a tool for testing Linux firewalls. Third, we present an extension of this technique that enables the analysis of systems of connected firewalls and firewalls that use advanced packet mangling techniques, such as network address translation. Fourth, we discuss a technique for generating an equivalence class representation of the firewall policy, which can be used
to detect certain classes of errors without constructing a series of queries or test cases. Last, we consider ways to partially automate repair of a policy.
Chapter 2

Representations of a Firewall Policy

A firewall is a facility (which can be either hardware or software) that implements a filtering policy for one or more network hosts. Usually, the policy is specified as a set of rules in which each rule consists of an action and one or more conditions. The conditions identify a set of packets to which the action should be applied. In this work, we consider matches against the attributes source address, destination address, protocol, source port, destination port, connection state, incoming network interface, outgoing network interface, and the six TCP flags (SYN, ACK, URG, PSH, RST, and FIN). It is possible to extend our work to consider other attributes, but we will focus on these attributes in order to simplify the discussion of our techniques.

In order to perform formal analysis of the policy, we must address the issue of constructing a model of the firewall which can be used to accurately represent the policy. There have been many approaches to this problem. We will consider first the efforts of others to produce an analytic model and then describe our own technique.
2.1 Other Models

One approach is to use linked lists of rules to describe the firewall policy. This is the approach used by many firewall implementations. For instance, in the Linux iptables system, rules are grouped into sequential lists called chains, which are further organized into four tables. The filter table contains chains directly related to filtering. The nat table contains chains related to network address translation. Chains in the mangle table modify packets in more exotic ways (such as increasing the TTL of a packet). There is also a raw table which allows processing of packets before connection tracking analysis is applied.

Internally, each rule consists of a data structure with pointers to a linked list of matches and a linked list of targets (a linked list is necessary for implementation of user-defined target values which perform processing and then pass control on to other targets). The chains of the firewall are represented using doubly linked lists. Caches are used to speed up operations on these lists. The tables are simple structures which contain pointers to the various chains along with some additional management information [42].

The linked list approach suffers from several disadvantages. First, performance can be poor, especially for policies with many rules, since lookups may require searching the entire list. Caching can reduce the impact of this problem, but does not entirely eliminate it. Second, the linked list model is not very amenable to analysis. One major drawback is the fact that a policy can have two different linked list representations with identical behavior. This makes it difficult to compare two policies for equivalence or perform more complex operations on them.

2.1.1 Binary Decision Diagrams

A more sophisticated approach to modeling the firewall is to use Binary Decision Diagrams (BDDs). Hazelhurst [24, 25] demonstrated a BDD-based technique for representing firewall policies in which each node of the BDD corresponds to exactly one bit of a match condition. Every path through the BDD corresponds to one packet seen by the firewall.
Using reduction and merging operations, duplicate and redundant nodes are removed from the BDD to produce a compact and canonical representation of the firewall policy.

2.1.2 Interval Decision Diagrams

Fleury and Christensen [9, 11, 10] extended this work to implement a Decision Diagram based packet filter for Linux that outperforms Netfilter for policies with more than 100 rules. Their work uses Interval Decision Diagrams (IDDs), a generalization of BDDs in which each node represents a range of values rather than a single boolean variable. This is a more natural representation of the policy, since most of the matches in a firewall rule correspond to integer ranges rather than boolean values. The IDD representation is a reduced decision diagram in which duplicate nodes are not allowed and redundant nodes with all arcs pointing to the same descendant are removed. A major drawback to the IDD approach is that application of the reduction rules causes generation of the IDD to require polynomial time — a significant cost. Furthermore, a path through the IDD can contain multiple nodes that reference the same attribute (for instance, there may be two nodes which impose conditions on the source address). This affects the performance of algebraic operations on the IDD representation slightly and makes analysis drastically more difficult.

2.1.3 Firewall Decision Diagrams

Gouda, Liu, et. al [22] constructed a decision diagram representation of the policy using Firewall Decision Diagrams (FDDs). An FDD is similar to an IDD, but does not allow any path to contain multiple nodes corresponding to the same attribute. Using this model, they presented a technique for detecting structural errors (such as duplicate rules) in the firewall policy. In other work, they described a system for evaluating SQL-like queries [34] against a firewall policy modeled using FDDs.

In a Firewall Decision Diagram, duplicate nodes and redundant nodes are removed by the repeated application of reduction rules. While applying these reduction rules can lower the memory requirement for storing the final decision diagram and slightly reduces the cost
of performing lookups, other operations may incur some overhead whenever a removed node needs to be "put back" into the MDD in the middle of an operation.

2.1.4 Quasi-Reduced Multi-way Decision Diagrams

In our work, we use quasi-reduced MDDs [12] in which redundant nodes are allowed, but duplicate nodes are removed using a hashing algorithm. Through the efficient use of caches, we can obtain very efficient manipulation of the firewall policy model which enables fast and accurate analysis of the firewall policy.

2.2 Notation

In the remainder of this chapter, we will carefully define the terms, models, and algorithms we employ in describing and constructing a representation of the firewall policy. We will then show that the MDD model is an accurate representation of the firewall policy. To this end, we first provide a formal characterization of an iptables firewall policy by carefully defining each of its constituent parts: match conditions, firewall rules, and chains of rules. We will then describe Quasi-Reduced Multiway Decision Diagrams and give several algorithms which can be used to construct a (QR)MDD model of a firewall policy. We also demonstrate the correctness of these algorithms by proving that the MDD model accepts (or drops) exactly those packets accepted (or dropped) by the firewall policy.

To formally describe an iptables firewall, we use the following definitions.

**Definition 1** The domain of an attribute is the set of possible values that can hold. A match condition, \( m \), over an attribute \( a \), is a boolean function which maps a subset of the domain of \( a \) to the values \( TRUE \) and \( FALSE \).

Each match condition identifies a set of packets to which a filtering rule should be applied. Because the match condition corresponds to exactly one attribute of the firewall rule, the user must combine several match conditions to obtain fine-grained definition of the match set.
For each match condition $m$, we let $D(m)$ represent the domain of values which satisfy the condition. For instance, if $m$ is a match condition on the source port, $D(m)$ is a set of ports that satisfy the condition.

There are several different ways in which we can use match conditions to describe the behavior of the firewall. For instance, to represent source address, we could construct a single match condition over the source address attribute or we could split the source address into octets and create separate match conditions for each octet of the source address. We will take the latter approach in our implementation, but our theoretical results generalize to any definition of the match conditions. To simplify discussion, we define a constant, $K$, which represents the total number of attributes to be considered in modeling the policy.

This definition of a match condition allows us to formally describe both filtering rules and filtering chains. Because these concepts are inextricably linked, we first define “filtering rule” and use that definition to define the concept of a filtering chain.

**Definition 2** A filtering rule $r$ is a tuple $(t, m_1, \ldots, m_K)$ in which each $m_i$ is a match condition, and $t$, the target, is either the action ACCEPT, the action DROP, or a filtering chain. The match set of $r$, written $M(r)$, is the set of packets which match each of the match conditions in $r$.

We sometimes use the notation $r[k]$ to represent the $k$th match of rule $r$ using the convention that $r[0] = t$ and $r[i] = m_i$. When discussing a named attribute, we may also use the notation $r.X$ to specify attribute $X$ of rule $r$.

**Definition 3** A firewall chain is an ordered sequence of firewall rules.

For iptables firewalls, chains are processed in first-match order so that when a packet matches a rule of the chain processing halts and the packet is either accepted, dropped, or passed to some other chain for handling.

We are interested in which packets a firewall chain will accept and which it will reject. In order to describe this concept fully, we define two terms: the accept set of a firewall
chain and the accept set of a firewall rule. It is necessary to employ a circular reference in defining these concepts in order to account for the possibility that a rule targets a firewall chain. Therefore, we first define “accept set of a rule” using the idea of “accept set of a chain” and then define “accept set of a chain” using “accept set of a rule”. The fact that iptables does not allow a rule to reference any chain which is its own ancestor ensures that, despite the circular reference, these definitions are well-formed.

**Definition 4** The accept set of a rule $r$ is defined as follows:

$$A(r) = \begin{cases} \emptyset & \text{if } r[0] \text{ is "DROP"} \\ \cap_{i \in [1,K]} D(r[i]) & \text{if } r[0] \text{ is "ACCEPT"} \\ \cap_{i \in [1,K]} D(r[i]) \cap A(r[0]) & \text{if } r[0] \text{ is a chain.} \end{cases}$$

Less formally, the accept set describes the set of packets which are accepted by rule $r$. The reject set of a rule, $R(r)$ is defined similarly by replacing “ACCEPT” with “DROP” and $A(r[0])$ with $R(r[0])$.

**Definition 5** The accept set of a chain $c = (r_0, r_1, \ldots, r_n)$ of size $n+1$ is defined as follows:

$$A(c) = A(r_0) \cup \cup_{i \in [1,n]} (A(r_i) \cap \overline{R(r_{i-1})} \cap \ldots \cap \overline{R(r_0)})$$

Or (more concisely):

$$A(c) = A(r_0) \cup \cup_{i \in [1,n]} (A(r_i) \cap \cap_{s \in [0,i-1]} \overline{R(s)}).$$

In other words, the accept set of a chain is the set of packets accepted by a rule in the chain that are not dropped by any previous rule of the chain. The reject set of a chain, $R(c)$ is defined similarly.

### 2.3 Multi-way Decision Diagrams

A multi-way decision diagram (MDD) is a directed acyclic graph, $M = (V, E, L)$, where $V$ is a set of nodes, $E \subseteq V \times V$ is a set of directed edges, and $L : E \rightarrow Z$ is a labeling function which maps each edge to a distinct integer value. The nodes of the MDD are
organized into $K + 1$ levels and all edges from a non-terminal level $k > 0$ point to nodes at level $k - 1$. We assign a unique index $p$ to each node at level $k$. This allows us to describe node $<k:p>$ where $k$ is the level and $p$ identifies the node. We use the notation $<k:p>[i]$ to describe the child of node $<k:p>$ which can be reached by following the edge with label $i$.

We will frequently use lowercase subscripts to refer to the index of a node. For instance, we sometimes use $n_p$ to refer to the index of node $<k:n_p>$. Since the root node of an MDD often requires special consideration, we will use the notation $<K:N_p>$ for the node at level $K$ which has index $N_p$. When referring to the entire decision diagram, we will use the capital letter $M$, for instance, we might say that the MDD $M_r$ has root node $<K:N_r>$.

![Figure 2.1: A rule set MDD for the chain in figure 1.2](image)

In this application, every path through the MDD represents a packet potentially received by the firewall. Each of the non-terminal levels of the MDD correspond to a specific attribute
of the packet. For instance, in figure 2.1, the MDD representation of the rule set given in figure 1.2, the top four levels represent the source address and the next four levels represent the destination address. The next level represents the protocol. Below these levels are levels for the source port and the destination port, the six TCP flags (URG, PSH, SYN, ACK, RST, FIN), and the connection state. Level 0 is a special _terminal_ level which represents the target of the firewall rule (ACCEPT, DROP, LOG, or a user-defined chain) as a unique integer index. We reserve terminal index 0 for the special meaning "not yet specified".

For readability, we represent the edge-labeling function, $L$, by drawing labels above each arc. Although these arcs appear inside the box representing a node, they should be interpreted as labels for the edges leading from that node to levels below.

A non-terminal node at level $k$ represents a subset of packets that share some attributes. An arc from a node at level $k$ to a node at level $k - 1$ represents a choice of value for the attribute represented at level $k$. When many arcs from a node point to the same child, we use ellipses in the figure to save space. In the actual MDD there would be arcs for each value we have hidden in this manner.

To see that an HTTP packet from 68.10.1.3 to 192.168.2.10 is accepted by the firewall, start with the node at level 20 of the MDD. Since the first source octet of the packet is 68, which falls between 0 and 113, follow the first arc to the highlighted node at level 19. Now there is only one arc to follow, since all values between 0 and 255 have been grouped together using ellipses. Since 10 falls between 0 and 255, we follow the highlighted arc to a node at level 18. Again, 1 falls between 0 and 255 so follow the arc to level 17. The last octet of the source address is 3, which falls between 0 and 255, so follow the highlighted arc to the node at level 16.

Level 16 represents the first octet of the destination address, which for our example is 192. Since there is an arc for 192, proceed to level 15. If the destination address had been 193.1.1.1, you would know that the packet is dropped by the firewall, since there is no arc for 193 and DROP is the default policy. Instead, at level 15, examine the second octet of the destination address. Since there is an arc for 168, proceed to level 14. Continue in this
manner to level 12.

At level 12, there is an arc for TCP and an arc for ICMP. Since HTTP is a TCP protocol, follow the arc for TCP to the highlighted node at level 11. Continue in this manner until you reach the node at terminal level 0. Since it is the ACCEPT node, the packet will be accepted by the firewall.

2.4 Implementation

Nodes at each level are stored in a dynamic array and are referenced by a unique integer index. At every level, we reserve index 0 for a special node, node zero, which represents the empty set. This can be thought of as a node with all its arcs pointing to node zero at the level below. To save a small amount of memory, we do not explicitly store node zero, but instead handle it as a special case in our algorithms. Using the notation presented above, we will sometimes use \( <k:0> \) to mean node zero at level \( k \).

Like nodes, edges are stored in a per-level dynamic array. Each element of the array holds the index of a child node. The index of each element corresponds to the label of the edge, offset so that the edges for each node can be stored separately. We keep track of the offset for each node's edges in the node structure. More details on the MDD implementation are available in [36].

2.5 Building an MDD for a Filter Rule

In order to construct an MDD for a rule, we first parse the rule into target, source address, destination address, source port, destination port, protocol, state, incoming interface, outgoing interface, and flag components. From these components, we create a parsed rule, which represents each component at level \( k \) as an integer range. We store these ranges in an array of size \( K + 1 \). We use the notation \( pr[k].low \) and \( pr[k].high \) to reference the lower and upper bounds of the range. We also define an operation \( \text{MakeMDDFromRule} \), which
takes the parsed firewall rule and returns the root node of an MDD representing that rule. Pseudocode for MakeMDDFromRule is given in figure 2.2.

The algorithm starts at level 0 and builds upward toward the root node. At each level, it creates new nodes that represent the criteria of the parsed rule. In line 1, node <0:n> is calculated by finding the integer index which represents the rule target. For ACCEPT, DROP, and LOG targets this is a predefined constant less than 4. For user-defined rules, the index comes from a pre-generated table that maps the user-defined chains, in the order of their discovery during parsing, to integers greater than 3.

Lines 2–7 construct nodes at levels 1 through K. The call to NewNode in line 3 creates a new node and initializes all its arcs to point at node zero. Lines 4–7 examine each potential value i of filter rule attribute k. If i falls within the range specified by the parsed rule, arc <k:n>[i] is connected to node <k − 1:old>. Otherwise, the arc is left at its default value, which points to node zero.

When we reach line 7, we have considered all the potential values of attribute k, so we now call CheckForDuplicates, which uses hashing to identify any nodes that exactly duplicate node <k:n>. If such a node exists, <k:n> is freed and CheckForDuplicates returns the index of the duplicate node. Otherwise, it returns <k:n>.
2.6 Correctness of MDD generation for simple terminal rules

To demonstrate that the rule generation algorithm is correct, we first define the idea of an accept set of an MDD. We then show that the accept set of the MDD representation of a rule generated by MakeMDDFromRule is equivalent to the accept set of the rule.

**Definition 6** The accept set of an MDD node \(<k:p>\) is the set of packets \(s\) such that there exists a path \((e_0,e_1,...,e_k)\) from \(<k:p>\) to the terminal node ACCEPT such that for all \(0 \leq i \leq k, L(e_i) = s[i]\). The accept set of a rule set MDD \(M\) with root node \(<K:N_p>\) is given by the formula \(A(M) = A(<K:N_p>)\).

In other words, the accept set of an MDD node is the set of packets for which there is a path from that node to the node ACCEPT, such that every edge in the path is labeled with an attribute that corresponds to the packet. The accept set of an MDD is the accept set of its root node.

We define \(R(<k:p>)\), the reject set of an MDD node \(<k:p>\) and \(R(M)\), the reject set of MDD \(M\), similarly, by replacing ACCEPT with DROP.

In order to demonstrate that the rule generation algorithm is correct, we show that the accept set of the MDD representation is identical to the accept set of the original rule.

**Lemma 1** Given a firewall rule \(r\), let \(M_r\) be the MDD generated by MakeMDDFromRule\((r)\). Then, \(A(M_r) = A(r)\).

**Proof:**

Step 1: \(A(r) \subseteq A(M_r)\).

We must show for any packet \(s \in A(r)\), that \(s \in A(M_r)\).

If \(A(r)\) is the empty set (i.e. \(r\) is not an accept rule), then \(A(M_r)\) is likewise empty, since the lookup in line 1 will never return ACCEPT. This means that the first iteration of the loop will not create an arc to the ACCEPT node. Since no other arcs
are created to the terminal nodes, there can be no path to the accept node and, by
the definition of the accept set of an MDD, $A(M_r)$ will be the empty set.

Suppose that $A(r)$ is not the empty set and let $s \in A(r)$. We must show that $s \in A(M_r)$.

The loop in line 2 considers each level of the MDD in turn. For each level, the inner
loop in line 4 creates arcs from node $<k:n>$ to node $<k-1:old>$ for every value
$i$ in the range of the $k^{th}$ attribute of $r$. Let $e_k$ be the arc labeled $s[k]$. The path
$S = (e_0, e_1, \ldots, e_K)$ is a path in which every edge $e_i$ is labeled $s[i]$. Because $s \in A(r)$,
we know that $S$ leads to the ACCEPT node. Therefore, by definition, $s \in A(M_r)$.

Step 2: $A(M_r) \subseteq A(r)$.

If $A(M_r)$ is empty, then $A(r)$ must also be empty, since $A(M_r)$ can be empty only when
the rule is not an accept rule. If $A(M_r)$ is not empty, let $s \in A(M_r)$ be a packet in the
accept set of $M_r$. By the definition of $A(M_r)$, there exists a path $S = (e_0, e_1, \ldots, e_K)$
from the root node of the MDD to the node ACCEPT such that each edge $e_k$ is
labeled with value $s[k]$.

Now, since arcs are only created in line 6, each edge of $S$ is labeled with value $i_k$. The
if statement in line 5 ensures that this label must be between $pr[k].low$ and $pr[k].high$.
Since this holds true for all $k$, we know that $s$ matches $r$. Furthermore, $s$ is accepted
by $r$, since the only way a path can point to ACCEPT is when the lookup in line 1
returns ACCEPT and an arc is created to the terminal node by line 6 during the first
iteration of the loop.

Therefore, since $A(M_r) \subseteq A(r)$ and $A(r) \subseteq A(M_r)$, $A(M_r) = A(r)$. ■

The proof that $R(M_r) = R(r)$ is similar — simply replace “ACCEPT” with “DROP”.
Furthermore, if $T(r)$ represents the set of packets that match some target $t$ (perhaps another
chain), we can adapt this proof to show that $T(M_r) = T(r)$ by replacing “ACCEPT” with
$t$, a fact we will make use of later.
2.7 Inserting a Rule into the Chain MDD

```plaintext
node_index Assign(level k, node_index nc, ParsedRule pr)
1 if k = 0:
2 return LookUpTarget(pr.target).
3 nt=NewNode(k).
4 for i = 0 to MaxValue(k):
5 if i E pr[k):
6 nt[i]=Assign(k-1,nc[i],pr).
7 else:
8 nt[i]=nc[k][i].
9 return CheckForDuplicates(nt).
```

*Figure 2.3: Inserting a rule*

Algorithm for inserting a rule into an MDD

The assignment operator in figure 2.3 generates an MDD representing the insertion of a new rule \( r \) into an existing chain \( c \), where the MDD representation of \( c \) is an MDD, \( M_c \). The insertion is performed in such a way that the new rule overrides the rules already inserted.

The algorithm is recursive, starting from node \( <k: nc> \) and descending the graph until it reaches a terminal node at level 0. Initially, we set node \( nc \) to the index of the root node of \( M_c \). Lines 1 and 2 of the algorithm handle the terminating case. If \( k = 0 \), we return a node representing the target of the rule. If \( k > 0 \), the algorithm constructs a new node to represent the result of insertion (line 3).

This result is constructed by the loop in lines 4 – 8. In each iteration of the loop we consider a value, \( i \), of attribute \( k \). If \( i \) does not match the condition \( r[k] \), we create an arc to the corresponding child of \( <k: nc> \). If \( i \) does match the rule, we use recursion to descend to the next level of the graph.

Because values that do not match the range of the new rule are linked to nodes from the old MDD, packets that do not match the rule are not affected by the insertion. However, the algorithm will create a path to the appropriate terminal node for packets which do match the new rule.
2.8 Correctness of Assign operator

Lemma 2 Let \( M_n = \text{Assign}(K, M_c, r) \) for some MDD \( M_c \) and rule \( r \). \( A(M_n) = A(r) \cup (A(M_c) \cap \overline{R(r)}) \).

Proof: We must show that the accept set of the new MDD is all packets accepted by the new rule, plus all those accepted by the old rules that are not dropped by the new rule.

Step 1: \( A(M_n) \subseteq A(r) \cup (A(M_c) \cap \overline{R(r)}) \).

We must show that for any packet \( s \in A(M_n) \), that \( s \in A(r) \cup (A(M_c) \cap \overline{R(r)}) \). To do this we show that \( s \) is either in \( A(r) \) or in \( A(M_c) \cap \overline{R(r)} \).

Since \( s \in A(M_n) \), there exists a path \( e = (e_0, \ldots, e_K) \) from the root of \( M_n \) to the terminal node ACCEPT such that each edge, \( e_k \), of the path is labeled with the value \( s[k] \). Now, arcs are only created in lines 6 and 8. We have two cases.

Case 1: Every edge in the path is labeled with a value that satisfies the match condition of the rule.

If this is the case, then \( s \in A(r) \) by the definition of \( A(r) \) and the fact that since \( s \in A(M_n) \), \( e \) is a path to ACCEPT. Therefore \( s \in A(r) \cup (A(M_c) \cap \overline{R(r)}) \), which is what we wish to prove.

Case 2: There is an edge \( e_k \in e \) which is labeled with a value outside of the range \( pr[k].\text{low} \) to \( pr[k].\text{high} \).

If this is the case, we choose the first such arc (that is, the arc at the lowest level) and note that the if statement in line 5 evaluates to false for this value. This means that arc \( e_k \) points to node \( <k:n_c>[i] \). But this means that \( s \in A(M_c) \), by the definition of \( A(M_c) \) and the fact that \( s \in A(M_n) \). Because \( e_k \) does not match \( pr[k] \), we know that \( s \) is not in \( R(r) \). Thus, \( s \in A(M_c) \cap \overline{R(r)} \). Therefore, \( s \in A(r) \cup (A(M_c) \cap \overline{R(r)}) \).

Step 2: \( A(r) \cup (A(M_c) \cap \overline{R(r)}) \subseteq A(M_n) \).
We must show for any packet \( s \in A(r) \cup (A(M_c) \cap \overline{R(r)}) \), that \( s \in A(M_n) \). By properties of set union, we know that either \( s \in A(r) \) or \( s \in (A(M_c) \cap \overline{R(r)}) \).

If \( s \in A(r) \), then the path labeled \((ACCEPT, s[1], \ldots, s[K])\) is in \( M_n \). To see this, note that each call of "Assign" will create a new node and set the arc labeled \( s[k] \) in that node to the result of calling the algorithm at the level below. When the algorithm reaches level 0, the lookup in line 2 will evaluate to ACCEPT, since the target of \( r \) is ACCEPT. Therefore, \( s \in A(M_n) \).

If \( s \) is not in \( A(r) \), we know that \( s \in A(M_c) \cap \overline{R(r)} \). This means that \( s \in A(M_c) \) and \( s \in \overline{R(r)} \). Let the path \( e = (e_0, e_1, \ldots, e_K) \) be the path in \( M_n \) labeled \((s[0], s[1], \ldots, s[K])\).

Since \( s \in A(M_c) \), there exists a corresponding path \( c = (c_0, \ldots, c_K) \) in \( M_c \) for which each edge is labeled with the values \((s[0], s[1], \ldots, s[K])\).

Since \( s \in \overline{R(r)} \) and \( s \) is not in \( A(r) \), there is at least one level \( k \) for which \( s[k] \) is not in the range \( pr[k].low \) to \( pr[k].high \). Consider the highest such level, \( k_{\text{max}} \).

Because \( s[k_{\text{max}}] \) is not in the range \( pr[k_{\text{max}}].low \) to \( pr[k_{\text{max}}].high \), the if statement in line 5 will fail and the algorithm will create an arc to node \(<k_{\text{max}} - 1:n_c>\). Now, let \((f_0, f_1, \ldots, f_{k_{\text{max}} - 1})\) be a path from \(<k_{\text{max}} - 1:n_c>\) to \( ACCEPT \). Then, the path \( e = (f_0, \ldots, f_{k_{\text{max}} - 1}, e_{k_{\text{max}}}, \ldots, e_K) \) in \( M_n \) is a path to the ACCEPT node from the root of \( M_n \). Since this path has labels \((s[0], s[1], \ldots, s[K])\), \( s \in A(M_n) \).

Therefore, since \( A(r) \cup (A(M_c) \cap \overline{R(r)}) \subseteq A(M_n) \) and \( A(M_n) \subseteq A(r) \cup (A(M_c) \cap \overline{R(r)}) \), we have that \( A(M_n) = A(r) \cup (A(M_c) \cap \overline{R(r)}) \).

\[ \blacksquare \]

2.9 Intersection

The algorithms above can be used to construct an MDD representation of a chain which only has rules with the targets ACCEPT or DROP. An iptables firewall also allows rules in which the target is a user-defined chain. Figure 2.4 shows two chains of a policy. The chain
myChain is a user-defined chain which allows traffic to enter the 192.168.3.0/24 subnet and blocks all ftp traffic that is not sent to that subnet. This chain, however, will only be applied to certain packets.

The FORWARD chain drops any SMTP packets and then passes any packets from untrusted network 192.168.1.0/24 to the user-defined chain. Any packets that are not accepted or dropped by myChain will be dropped unless they match rule 3, which allows any packets bound for subnet 192.168.2.0 to pass the firewall.

<table>
<thead>
<tr>
<th>Chain myChain:</th>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.3.0/24</td>
<td>anywhere</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DROP</td>
<td>all</td>
<td>anywhere</td>
<td>anywhere</td>
<td>dpt tcp:ftp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chain FORWARD (policy DROP):</th>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 DROP all anywhere anywhere dpt tcp:smtp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 myChain all 192.168.1.0/24 anywhere</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 ACCEPT all anywhere 192.168.2.0/24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.4:** Example policy with a user-defined chain

To represent chains with rules that link to some other chain, we implement a special...
intersection operator that combines the MDD representation of the target chain with an
MDD representing the rule which targets the chain. This special intersection operation
is illustrated in figure 2.5. An MDD representation of myChain is given in the left-most
column of the figure. The next column illustrates the MDD representing rule 2 of the
FORWARD chain, which passes packets from the 192.168.1.0/24 subnet to the user-defined
chain. The right-most column shows the result of intersecting these two chains. The
intersection restricts the chain MDD to only those packets that match the targeting rule,
in this case, rule 2. By examining the resulting MDD, you can see that the result does
not cause packets from 192.168.3.0/24 to be accepted, even though myChain accepts such
packets, because the targeting rule only applies to packets from the 192.168.1.0/24 subnet.

<table>
<thead>
<tr>
<th>node_index SpecialInt(level k, mdd n_t, n_s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1   if n_s = 0:</td>
</tr>
<tr>
<td>2   return 0</td>
</tr>
<tr>
<td>3   if n_t = 0:</td>
</tr>
<tr>
<td>4   return 0</td>
</tr>
<tr>
<td>5   if k = 0:</td>
</tr>
<tr>
<td>6   return n_t</td>
</tr>
<tr>
<td>7   n_n = NewNode(k)</td>
</tr>
<tr>
<td>8   for i = 0 to MaxValue(k):</td>
</tr>
<tr>
<td>9   if n_t[i] ≠ 0 and n_s[i] ≠ 0:</td>
</tr>
<tr>
<td>10  n_n[i] = SpecialInt(k - 1, n_t[i], n_s[i])</td>
</tr>
<tr>
<td>11  return n_n[i].</td>
</tr>
</tbody>
</table>

Figure 2.6: Intersection Operation

Pseudocode for the special intersection algorithm is given in figure 2.6. The arguments
to SpecialInt are a level and the indices of two MDD nodes. To intersect two MDDs, M_s
and M_t, we pass the root nodes of each of the MDDs in the initial call to SpecialInt.

In lines 1–6, we check for the base cases. If n_s is the zero node, then no corresponding
path exists in the match set of the targeting rule. Therefore, we return the empty node. If
n_t is the zero node, then the target chain does not match any packets along the path we
have followed and we return 0. If neither node is the zero node and we have reached the
terminal level, then the value of M_t (the targeted chain) overrides the existing value, so we
return the value of \( n_t \).

Lines 7 - 10 handle nodes at the non-terminal levels. In each case, a new node is created and populated using recursive calls. To simplify the description of the algorithm, we assume that \( n_x[i] \) is the index of the node reached by following the arc with label \( i \) from node \( <k:n_x> \) and that \( n_x[i] = 0 \) if there is no arc labeled \( i \) in \( <k:n_x> \).

### 2.9.1 Correctness of the Special Intersection Algorithm

**Lemma 3** Given a rule \( r \), such that \( r\text{.target} = c \), where \( c \) is a chain of the firewall, let \( C(r) \) be the set of packets that \( r \) maps to \( c \). If \( <K:N_c> \) and \( <K:N_r> \) are the root nodes of MDDs \( M_c \) and \( M_r \), respectively, and \( N_n = \text{SpecialInt}(K,N_c,N_r) \), then \( A(M_n) = C(r) \cap A(M_c) \), where \( M_n \) is the MDD rooted at \( <K:N_n> \).

By lemma 1, we know that \( C(r) = C(M_r) \). Therefore, we must show that \( A(M_n) = C(M_r) \cap A(M_c) \). It is sufficient to show that at every level \( k \), \( \text{SpecialInt}(k,n_t,n_s) \) returns a node \( <K:n_n> \) such that \( A(<k:n_n>) = A(<k:n_t>) \cap C(<k:n_s>) \), since then the node returned by \( \text{SpecialInt}(K,N_c,N_r) \), where \( <K:N_c> \) and \( <K:N_r> \) are the root nodes of \( M_c \) and \( M_r \), respectively, will prove the lemma.

**Proof:**

We proceed by induction on \( k \), the level of the MDD at which we are performing the intersection operation.

**Base Case** Consider the case that \( k = 0 \). There are three possibilities:

Case 1: \( n_s = 0 \)

If \( n_s = 0 \), then \( C(n_s) = \emptyset \) by the definition of \( C(X) \). Lines 1 and 2 of the algorithm return 0, so \( n_n = 0 \). Therefore, \( A(<k:n_n>) = \emptyset = A(<k:n_t>) \cap \emptyset = A(<k:n_t>) \cap C(<k:n_s>) \).

Case 2: \( n_s \neq 0 \), but \( n_t = 0 \).
If \( n_t = 0 \), then \( A(<k:n_t>) = \emptyset \) by definition of accept set of an MDD. Since \( ns \) is not 0, the if statement in line 1 will evaluate to false and we will skip line 2. Lines 3 and 4, however, will cause the algorithm to return 0, so \( n_n = 0 \). Therefore, \( A(<k:n_n>) = \emptyset = \emptyset \cap C(<k:n_s>) = A(<k:n_t>) \cap C(<k:n_s>) \).

Case 3: \( ns \) and \( nt \) are both non-zero.

Since \( ns \) and \( nt \) are both non-zero, the first two if statements will evaluate to false. Since we are at level \( k = 0 \), the if statement in line 5 will evaluate to true and the algorithm will return \( nt \), so \( n_n = nt \). Now, \( ns \) can be one of two terminals. It can be the terminal 0 or it can be the terminal representing chain \( c \). We know that \( ns \) is not 0, so it must be the terminal representing chain \( c \). Now, by definition of \( C(X) \), a packet \( p \) is an element of \( C(X) \) if there exists a path \( P \) from node \( X \) to the terminal node \( c \) which has the property that every edge \( e_i \) in the path, that \( L(e_i) = p[i] \). When \( X \) is the terminal node for \( c \), this definition is trivially satisfied for every packet. Therefore, \( <k:n_s> \) is a node representing the set of all packets. This means that \( A(<k:n_t>) \cap C(<k:n_s>) = A(<k:n_t>) \). Therefore, \( A(<k:n_n>) = A(<k:n_t>) = A(<k:n_t>) \cap C(<k:n_s>) \).

**Induction Hypothesis**  Assume for some integer \( K \), that if at each level \( 0 < k < K \),

\[
n_n = SpecialInt(k, n_t, n_s)
\]

then

\[
A(<k:n_n>) = A(<k:n_t>) \cap C(<k:n_s>).
\]

**Induction Step**  We will show that if at level \( k = K > 0 \),

\[
n_n = SpecialInt(k, n_t, n_s)
\]

then

\[
A(<k:n_n>) = C(<k:n_s>) \cap A(<k:n_t>).
\]
We have three cases.

Case 1: \( n_s = 0 \)

If \( n_s = 0 \), then line 2 will return 0. Thus \( n_n = 0 \) and

\[
A(<K:n_n>) = \emptyset = \emptyset \cap A(<K:n_t>) = C(<K:n_s>) \cap A(<K:n_t>).
\]

Case 2: \( n_t = 0 \)

If \( n_t = 0 \), then line 4 will return 0. Thus \( n_n = 0 \) and

\[
A(<K:n_n>) = \emptyset = C(<K:n_s>) \cap \emptyset = C(<K:n_s>) \cap A(<K:n_t>).
\]

Case 3: \( n_s \) and \( n_t \) are both non-zero.

Step 1: \( A(<K:n_n>) \subseteq A(<K:n_t>) \cap C(<K:n_s>). \)

Let \( p \in A(<K:n_n>) \). By definition of accept set of an MDD, there is a path \( P_n \)
from \( <K:n_n> \) to the terminal node ACCEPT such that for every edge \( e_k \in P_n \),
\( L(e_k) = p[k] \).

Since \( n_s \) and \( n_t \) are non-zero, the if statements in lines 1 and 3 will evaluate to
false, so the algorithm will continue past them. Since \( K > 0 \), the if statement in
line 5 will evaluate to false and the algorithm will proceed to lines 7 through 10.

Now, the node \( <K:n_n> \) created in lines 7 through 9 is formed by calling SpecialInt recursively to obtain a node at level \( K - 1 \). The algorithm then attached
an arc with label \( i \) to this node from \( <K:n_n>[i] \). We refer to the child node as
\( <K:n_n>[i] \). Now consider the child node produced when \( i = P[K] \). We know
that such a node exists, because \( p \in A(M_n) \). But this means that there is an arc
with label \( P[K] \) from \( <K:n_t> \) to \( <K:n_t>[i] \). Call this arc \( f_K \). There is also an
arc with label \( P[K] \) from \( <K:n_s> \) to \( <K:n_s>[i] \). Call this arc \( g_K \).

By the induction hypothesis, we know that \( A(<K:n_n>[i]) = A(<K:n_t>[i]) \cap
C(<K:n_s>[i]) \). Therefore there is a path \( P_l = (f_0, f_1, \ldots, f_{K-1}) \) from \( <K:n_t>[i] \)
to the terminal node for \textit{ACCEPT} such that \( L(f_k) = p[k] \) for each edge \( f_k \).

There is also a path \( P_s = (g_0, g_1, \ldots, g_{K-1}) \) from \( M_s \) to the terminal node for chain \( c \) such that \( L(g_k) = p[k] \) for each edge \( g_k \).

Therefore, the path \( (f_0, f_1, \ldots, f_K) \) is a path from \( <K:n_t> \) to \textit{ACCEPT} such that \( L(f_k) = p[k] \) for each edge \( f_k \) and the path \( (g_0, g_1, \ldots, g_K) \) is a path from \( <K:n_s> \) to the terminal node for chain \( c \) such that \( L(g_k) = p[k] \) for each edge \( g_k \). Thus, \( p \in A(M_t) \) and \( p \in C(M_s) \). Therefore, \( A(M_n) \subseteq A(M_t) \cap C(M_s) \).

\textbf{Step 2:} \( A(M_t) \cap C(M_s) \subseteq A(M_n) \).

Let \( p \in A(M_t) \cap C(M_s) \). We will show that \( p \in A(M_n) \).

Since \( p \in A(M_t) \cap C(M_s) \), \( p \in A(M_t) \). Therefore, there is a path \( P_t = (e_0, e_1, \ldots, e_K) \) from node \( <K:n_t> \) to the terminal node \textit{ACCEPT} such that for each edge \( e_k \), \( L(e_k) = p[k] \). Similarly, \( p \in C(M_s) \), so there is a path \( P_s = (f_0, f_1, \ldots, f_K) \) from \( <K:n_s> \) to the terminal node for chain \( c \) such that \( L(f_k) = p[k] \). This means that \( p \in A(<K:n_t>[i]) \) and \( p \in C(<K:n_s>[i]) \) by definition of accept set of an MDD and \( C(X) \).

Consider what happens when \( i = p[K] \) in the for loop in line 8. In line 9, we create a new arc, \( G_K \), from \( <K:n_t> \) to a new node \( <K:n_n>[i] \) which has label \( i = p[K] \). By the induction hypothesis, we know that \( A(<K:n_n>[i]) = A(<K:n_t>[i]) \cap C(<K:n_s>[i]) \). Therefore, there is a path \( P_n = (g_0, g_1, \ldots, g_{K-1}) \) from \( <K:n_n>[i] \) to the terminal node \textit{ACCEPT} such that for each edge \( g_k \), \( L(g_k) = p[k] \). Since \( G_K \) has label \( i = p[K] \), the path \( P_n^+ = (g_0, g_1, \ldots, g_K) \) is a path from \( <K:n_n> \) to the terminal node \textit{ACCEPT} such that for each edge \( g_k \), \( L(g_k) = p[k] \). Therefore, \( p \in A(M_n) \). Thus, \( A(M_t) \cap C(M_s) \subseteq A(M_n) \).

Therefore, by induction, \( A(M_n) = A(M_t) \cap C(M_s) \).
2.10 Replace Algorithm

While the intersection operation restricts the chain MDD to only those packets that match the rule, we also need to be able to insert the result into the chain containing that rule. We want to do this in such a way that packets that match the new rule are filtered appropriately, but packets that don’t match the rule are handled exactly as before.

This process is illustrated by the graphics in figure 2.7. The MDD in the left column represents the FORWARD chain of figure 2.4 before insertion of rule 2. The middle column gives the MDD representation of rule 2 after the intersection operation has been applied. The right column shows the effect of the replace operation. Notice that FTP packets from the 192.168.1.0/24 subnet are dropped, but that packets from any other network are accepted as long as they are sent to the 192.168.2.0/24 subnet.

Pseudocode for the replace algorithm is given in figure 2.8. The algorithm takes two MDDs. The first MDD represents the chain before the rule is inserted. The second MDD represents the result of intersecting the rule with the targeted chain. The algorithm returns a new MDD which applies filtering to only those packets that match the new rule.

The algorithm is recursive. In lines 1 and 2, we check for a base case in which there is
Figure 2.8: Algorithm which inserts a filtered rule into a chain

no node representing the existing chain. If this is the case, we return a node representing
the rule MDD. In lines 3 and 4, we check for the base case in which the rule MDD is 0. If
this is the case, we return a node representing the existing chain. In lines 5 and 6, we check
to see whether we have reached the bottom level of the MDD. If so, the MDD for the new
rule takes precedence over the existing chain and we return a node representing it. If none
of the base cases intercepts control, we create a new node at level $k$ and use recursion to
set its arcs to the result of calculating the replacement at the level below.

2.11 Correctness of Replace Algorithm

**Lemma 4** Let $<K: N_c>$ be the root node of MDD $M_c$, $<K: N_r>$ be the root node of MDD $M_r$ and $N_v$ be the root node of the MDD, $M_v$, created by $N_v = \text{Replace}(K, N_c, N_r)$. Then

$$A(M_v) = A(M_r) \cup (A(M_c) \cap \overline{R(M_r)}).$$

To show this, it is sufficient to prove that at every level $k$, that $\text{Replace}(k, n_c, n_r)$ returns
a node $n_n$ that satisfies the property that $A(<k: n_n>) = A(<k: n_r>) \cup (A(<k: n_c>) \cap
\overline{R(<k: n_r>)}).

**Proof:**

We proceed by induction on $k$, the level of the MDD.
**Base Case**  Let \( k = 0 \). Then we have the following three possibilities:

Case 1: \( n_c = 0 \).

If \( n_c = 0 \), then \( A(n_c) = \emptyset \). The if statement in line 1 will evaluate to true, so the algorithm will return \( n_r \). Thus \( A(<k:n_n>) = A(<k:n_r>) = A(<k:n_r>) \cup \emptyset = A(<k:n_r>) \cup (\emptyset \cap \overline{R(<k:n_r>))} = A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>))}.

Case 2: \( n_c \) is non-zero, but \( n_r = 0 \).

If \( n_r = 0 \), then \( A(<k:n_r>) = \emptyset \) and \( R(<k:n_r>) = \emptyset \). The if statement in line 3 will evaluate to true, so the algorithm will return \( n_c \). Therefore, \( A(<k:n_r>) = A(<k:n_c>) = \emptyset = A(<k:n_c>) \cup (A(<k:n_c>) \cap \emptyset = A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>))}.

Case 3: \( n_r \) and \( n_c \) are both non-zero nodes.

If \( n_r \) and \( n_c \) are both non-zero, then the algorithm will proceed past the first two if statements. Since \( k = 0 \), the if statement in line 5 will evaluate to true and the algorithm will return \( n_r \).

Now, since \( n_r \) is non-zero and we are at the terminal level, \( n_r \) is either the ACCEPT node or the DROP node. If \( n_r \) is the ACCEPT terminal, then \( A(<k:n_r>) \) is the set of all packets and \( R(<k:n_r>) \) is the empty set. So

\[
A(<k:n_n>) = A(<k:n_r>) = A(<k:n_r>) \cup \emptyset
\]

\[
= A(<k:n_r>) \cup (A(<k:n_c>) \cap \emptyset) A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>))}.
\]

If \( n_r \) is the DROP terminal, then \( A(<k:n_n>) = \emptyset \) and \( R(<k:n_r>) \) is the set of all packets. Therefore,

\[
A(<k:n_n>) = A(<k:n_r>) = \emptyset = \emptyset \cup \emptyset
\]
Thus,

\[ A(M_n) = \emptyset \cup (A(M_c) \cap \overline{R(M_r)}) = A(M_r) \cup (A(M_c) \cap \overline{R(M_r)}). \]

**Induction Hypothesis**  Assume that for all levels \(0 < k < K\), if \(n_n = \text{Replace}(k, n_c, n_r)\), then \(A(<k:n_n>) = A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>)})\).

**Induction Step**  There are three cases:

Case 1: \(n_c = 0\).

If \(n_c = 0\), then \(A(<k:n_c>) = \emptyset\) and the if statement in line 1 evaluates to true. Therefore, the algorithm returns \(n_r\) and \(A(<k:n_r>) = A(<k:n_r>) \cup (A(<k:n_c>) \cap \emptyset) = A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>)})\).

Case 2: \(n_c\) is non-zero, but \(n_r = 0\).

If \(n_r = 0\), then \(A(<k:n_r>) = \emptyset\) and \(\overline{R(<k:n_r>)}) = \emptyset\). The if statement in line 1 will evaluate to false, but the if statement in line 3 will evaluate to true. So the algorithm will return \(n_c\). Thus, \(A(<k:n_n>) = A(<k:n_c>) = \emptyset \cup (A(<k:n_c>) \cap \emptyset) = A(<k:n_r>) \cup (A(<k:n_c>) \cap \overline{R(<k:n_r>)})\).

Case 3: Both \(n_r\) and \(n_c\) are non-zero.

If both \(n_r\) and \(n_c\) are non-zero, then since \(k > 0\), none of the if statements will evaluate to true and the algorithm will proceed to the for loop in lines 7 through 10.

We will show that if \(k = K\) and \(N_n = \text{Replace}(K, n_c, n_r)\), then \(A(M_n) = A(M_r) \cup (A(M_c) \cap \overline{R(M_r)})\).

**Step 1:** \(A(M_n) \subseteq A(M_r) \cup (A(M_c) \cap \overline{R(M_r)})\).
Let $p \in A(M_n)$. From the definition of accept set of an MDD, we know that there is a path $P_n = (e_0, e_1, \ldots, e_K)$ such that for each edge $e_k$ in $P_n$, $L(e_k) = p[k]$.

Now, the node $<k:n_t>$ created in lines 7 through 9 is formed by calling $\text{Replace}$ recursively to obtain nodes at level $k - 1$. In each iteration of the loop, we attach an arc with label $i$ to node $<k:n_t>[i]$ created by the recursive call. Now consider the child node produced when $i = p[k]$. We know that such a node exists, because $p \in A(M_n)$. But this means that there is an arc with label $p[k]$ from $M_c$ to $<K:n_c>[i]$. Call this arc $f_K$. There is also an arc with label $p[K]$ from $<K:n_r>$ to $<K:n_r>[i]$. Call this arc $g_K$.

By the induction hypothesis we know that

$$A(<K:n_n>[i]) = A(<K:n_r>[i]) \cup (A(<K:n_c>[i]) \cap \overline{R(<K:n_r>[i])}).$$

Therefore, there is either a path

$$P_r = (f_0, f_1, \ldots, f_{K-1})$$

from $<K:n_r>$ to $\text{ACCEPT}$ such that $L(f_0) = p[k]$ or both of the following conditions are true:

- There is a path $P_c = (g_0, g_1, \ldots, g_{K-1})$ from $<K:n_c>$ to $\text{ACCEPT}$ such that $L(g_k) = p[k]$.
- There is no path $(h_0, h_1, \ldots, h_{K-1})$ from $<K:n_r>$ to $\text{REJECT}$ such that $L(h_k) = p[k]$ for each edge $h_k$ on the path.

If there is a path from $<K:n_r>[i]$ to $\text{ACCEPT}$, then the path $(f_0, f_1, \ldots, f_K)$ is a path from $<K:n_r>$ to $\text{ACCEPT}$ such that $L(f_k) = p[k]$ for each edge $f_k$, so $p \in A(M_r)$. Thus, $p \in A(M_r) \cup (A(M_c) \cap R(M_r))$.

If there is no path from $<K:n_r>[i]$ to $\text{ACCEPT}$, then the path $(g_0, g_1, \ldots, g_K)$ is a path from $<K:n_c>$ to $\text{ACCEPT}$ such that $L(g_k) = p[k]$ for each edge $g_k$.

There can be no path $(h_0, h_1, \ldots, h_K)$ from $<K:n_r>$ to $\text{REJECT}$ for which
each edge $h_k = p[k]$, because the only arc labeled $p[k]$ points to $<K:n_r>[i]$ and the induction hypothesis guarantees that there are no paths from $<K:n_r>[i]$ to $REJECT$. Thus, $p \in A(M_r) \cup (A(M_c) \cap \overline{R(M_r)})$.

Therefore, $A(M_n) \subseteq A(M_r) \cup (A(M_c) \cap \overline{R(M_r)})$.

Step 2: $A(M_r) \cup (A(M_c) \cap \overline{R(M_r)}) \subseteq A(M_n)$.

Let $p \in A(M_r) \cup (A(M_c) \cap \overline{R(M_r)})$.

There are two cases:

Case 1: $p \in A(M_r)$.

If $p \in A(M_r)$, then there is a path $P_r = (e_0, e_1, \ldots, e_K)$ such that $L(e_k) = p[k]$ for each edge $e_k$ in the path. Let $f_K$ be the edge created when the for loop reaches $i = p[k]$ and we call Replace($k - 1$, $<k:n_c>[i]$, $<k:n_r>[i]$) to create a node $<k:n_n>[i]$ at level $k - 1$. Since $p \in A(M_r)$, we know that $p \in A(<k:n_r>[i])$. This means that

$$p \in A(<k:n_r>[i]) \cup (A(<k:n_c>[i]) \cap \overline{R(<k:n_r>[i])}).$$

By the induction hypothesis,

$$A(<k:n_n>[i]) = A(<k:n_r>[i]) \cup (A(<k:n_c>[i]) \cap \overline{R(<k:n_r>[i])})$$

so $p \in A(<k:n_n>[i])$. This means that there is a path

$$P_n = (f_0, f_1, \ldots, f_{K-1})$$

from $<k:n_n>[i]$ to $ACCEPT$ such that $L(f_k) = p[k]$ for each edge $f_k$ of $P_n$.

But this means that the path $(f_0, f_1, \ldots, f_K)$ is a path from $<K:N_n>$ to $ACCEPT$ such that for every edge $f_k$, $L(f_k) = p[k]$.

Therefore, $p \in A(M_n)$.

Case 2: $p \notin A(M_r)$ and $p \in A(M_c) \cap \overline{R(M_r)}$. 
Since $p \in A(M_c) \cap \overline{R}(M_r)$, there is a path $P_c = (e_0, e_1, \ldots, e_K)$ such that $L(e_k) = p[k]$ for each edge $e_k$ in the path. Let $f_K$ be the edge created when the for loop reaches $i = p[k]$ and we call $Replace(k-1, <k:n_c>[i], <k:n_r>[i])$ to create a node $<k:n_n>[i]$ at level $k-1$. Since $p \notin R(M_c)$, we know that there is no path $P_c = (g_0, g_1, \ldots, g_K)$ from $<K:N_r>$ to DROP such that $L(g_k) = p[k]$ for every edge of the path. Therefore, on any path from $<K:N_r>$ to DROP, there is some edge $g_i$ such that $L(g_i) \neq p[k]$. We know that $L(g_K) = p[k]$, because $i = p[k]$. Therefore, there is no path from $<K:n_r>[i]$ to DROP for which every edge $g_k$ has the label $p[k]$. Thus, $p \in \overline{R}(<K:n_r>[i])$. Since $p \in A(M_c)$, we know that $p \in A(<K:N_c>[i])$.

This means that

$$p \in A(<K:N_r>[i]) \cup (A(<K:n_c>[i]) \cap \overline{R}(<K:n_r>[i]))$$

By the induction hypothesis,

$$A(<K:N_n>[i]) = A(<K:M_r>[i]) \cup (A(<K:N_c>[i]) \cap \overline{R}(<K:N_r>[i]))$$

so $p \in A(<K:N_n>[i])$. This means that there is a path

$$P_n = (f_0, f_1, \ldots, f_{K-1})$$

from $<K:N_n>[i]$ to ACCEPT such that $L(f_k) = p[k]$ for each edge $f_k$ of $P_n$. But this means that the path

$$(f_0, f_1, \ldots, f_K)$$

is a path from $<K:N_n>$ to ACCEPT such that for every edge $f_k$, $L(f_k) = p[k]$.

We have that $p \in A(M_n)$. 
In either case, \( p \in A(M_n) \). Therefore, \( A(M_r) \cup (A(M_c) \cap \overline{R(M_r)}) \subseteq A(M_n) \).

Therefore, by induction, we know that \( A(M_n) = A(M_r) \cup (A(M_c) \cap \overline{R(M_r)}) \).

\[ \Box \]

### 2.12 Chain Building Algorithm

<table>
<thead>
<tr>
<th>node_index</th>
<th>MakeMDDFromChain(Chain c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( M_c = \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>For each rule ( r \in c ) in reverse order:</td>
</tr>
<tr>
<td>3</td>
<td>if ( r ).target = ACCEPT or ( r ).target = DROP:</td>
</tr>
<tr>
<td>4</td>
<td>( M_c = \text{Assign}(M_c, r) )</td>
</tr>
<tr>
<td>5</td>
<td>else:</td>
</tr>
<tr>
<td>6</td>
<td>( M_r = \text{MakeMDDFromRule}(r) )</td>
</tr>
<tr>
<td>7</td>
<td>( M_n = \text{MakeMDDFromChain}(r \text{.target}) )</td>
</tr>
<tr>
<td>8</td>
<td>( M_t = \text{SpecialInt}(M_n, M_r) )</td>
</tr>
<tr>
<td>9</td>
<td>( M_c = \text{Replace}(M_c, M_t) )</td>
</tr>
<tr>
<td>10</td>
<td>return ( M_c ).</td>
</tr>
</tbody>
</table>

Figure 2.9: Algorithm for building an MDD from a chain

Figure 2.9 gives pseudocode for generating an MDD from a firewall chain. The algorithm makes calls to the SpecialInt and Replace algorithms discussed previously to generate an MDD in which rules at the beginning of the chain mask later rules.

### 2.13 Correctness of Chain Building Algorithm

**Theorem 1** Let \( c \) be any chain of the firewall and let \( M_c \) be the MDD representation of \( c \) produced by MakeMDDFromChain. Then, \( A(M_c) = A(c) \).

In order to prove that the chain building algorithm constructs an MDD which is equivalent to the original chain, we consider the call-graph of the firewall defined as follows.

**Definition 7** Let \( F \) be a firewall with chains \( c_0, c_1, \ldots, c_n \) and let \( G_F = (V, E) \) where \( V \) is the set \( \{c_0, c_1, \ldots, c_n\} \) and \( E : V \times V \) is the set \( \{(c_0, c_1) \mid \exists r \in c_0 \text{ such that } r \text{.target} = c_1\} \).
Then $G_F$ is the call-graph of $F$. For each node $v \in V$, we say the set $C(v) = \{ u \in V \mid \exists e \in E \text{ such that } e_0 = v \text{ and } e_1 = u \}$ represents the children of node $v$.

In other words, $G_F$ is the graph formed over the chains of the firewall for which there is an edge between two chains if and only if the first chain contains a rule which targets the second chain. Since iptables does not allow cyclic references in chains, $G$ is a directed, acyclic graph.

To each node $v$ of $G_F$, we assign a height, $H(v)$. If $v$ has no outgoing edges (i.e. the chain contains only ACCEPT and DROP statements, but does not target another chain), then $H(v) = 0$. Otherwise, $H(v) = \max_{u \in C(v)} H(u) + 1$.

**Theorem 2** Let $c = (r_0, \ldots, r_n)$ be a chain in firewall $F$. Then, $A(M_c) = A(C)$.

**Proof:**

We proceed by induction on the height of $c$ in the call-graph.

**Base Case** Consider the case that $H(c) = 0$.

Since $H(c) = 0$, $c$ contains only ACCEPT and DROP rules. Therefore, in each iteration of the loop in line 2, the algorithm will use the Assign operator to add one rule at a time to the MDD.

After the first iteration of the loop, we know that $A(M_c) = A(r_n)$ by lemma 2 since $A(M_c) = \emptyset$. Applying the same lemma to the next iteration yields that $A(M_c) = A(r_{n-1}) \cup (A(R_n) \cap \overline{R(r_{n-1})})$. Let $M_{ci}$ be the value of $M_c$ after iteration $i$. By continued application of lemma 2, we obtain the recurrence relation $A(M_{ci}) = A(r_{(n+1)-i}) \cup [A(M_{ci-1}) \cap \overline{R(r_{(n+1)-i})}]$.

Expanding this recurrence, we find that in iteration $i$, $A(M_{ci}) = A(r_{n-i}) \cup_{j \in [(n+1)-i,n]} (A_j \cap_{s \in [n-i,j-1]} \overline{R(r_s)})$.

After the last iteration (iteration $n$), we have that $A(M_c) = A(M_{cn}) = A(r_0) \cup_{j \in [1,n]} (A_j \cap_{s \in [0,j-1]} \overline{R(r_s)})$ (by substitution of $n$ for $i$).

But by the definition of $A(c)$, we can see that this is the same as $A(c)$. 

**Induction Hypothesis**  Assume that $A(M_x) = A(x)$ for any chain $x$ such that the height of $x$ in the call-graph is less than or equal to some value $h_x$.

**Induction Step**  We will show that for any chain $c$ with a call graph of height $h_x + 1$, that $A(M_c) = A(c)$. To see this, let $c = (r_0, \ldots, r_n)$. Let $M_{c_i}$ represent the value of $M_c$ after iteration $i$ of the loop in line 2. In each iteration, we have two cases:

Case 1: The target of $r_i$ is a terminal node.

In this case, we have that $A(M_{c_i}) = A(r_{(n+1)-i}) \cup [A(M_{c_i-1}) \cap R(r_{(n+1)-i})]$ by application of lemma 2.

Case 2: The target of $r_i$ is a non-terminal node of height less than $h_x$.

We have that $C(r_{n+1-i})$ is the set of all packets that match $r_{n+1-i}$. By lemma 3, we know that after line 6, $C(M_r) = C(r_{n+1-i})$.

After line 7, we have that

$$A(M_n) = A(r_{n+1-i}.target)$$

and that

$$R(M_n) = R(r_{n+1-i}.target)$$

from the induction hypothesis.

After line 8, we have that

$$A(M_t) = C(r) \cap A(M_n)$$

and that

$$R(M_t) = C(r) \cap R(M_n)$$

by lemma 3.

After line 9, we have that

$$A(M_{c_i}) = A(M_t) \cup [A(M_{c_i-1}) \cap R(M_t)].$$
This gives us $A(M_c) = \[M(r_{n+1-i}) \cap A(M_n)\] \cup [A(M_{c_{i-1}}) \cap \overline{M(r_{n+1-i})} \cup R(M_n)]$ by the above mentioned properties of $A(M_t)$ and an application of DeMorgan's Law.

Using the induction hypothesis, we have that

$$A(M_{c_t}) = [M(r_{n+1-i} \cap A(r_{n+1-i} \cdot target)) \cup [A(M_{c_{i-1}}) \cap \overline{M(r_{n+1-i})} \cup R(r_{n+1-i} \cdot target)].$$

But by the definition of accept set of a rule, this means that $A(M_{c_t}) = A(r_{n+1-i}) \cup A(M_{c_{i-1}}) \cap (\overline{R(r_{n+1-i})}).$

Therefore, in either case, we have that

$$A(M_c) = A(r_{n+1-i}) \cup [A(M_{c_{i-1}}) \cap \overline{R(r_{n+i-1})}] = A(c).$$

Similarly, we can see that $R(M_c) = R(c).$  

$\blacksquare$
Chapter 3

Linux Firewall Analysis using Queries

Despite the increasing popularity of iptables firewalls, there are very few tools for testing and debugging a Linux firewall policy. Some work has been done on testing iptables itself for software bugs [26], but these tools do not provide assurance in the configuration of the firewall.

Existing tools for testing the firewall policy fall into two categories: active testing solutions, which rely on transmitting traffic across the wire, and passive testing solutions, which analyze a model of the firewall off-line using sophisticated data structures.

Active testing tools such as SATAN [18], nessus [32], and Ftester [4] subject a firewall to a sequence of carefully crafted packets and see which ones get through. Active testing can also be accomplished using port scanners such as nmap [19] and hping [6]. Since it is impossible to test every possible packet, active tools test only a portion of the firewall configuration. This makes them well-suited for detecting specific vulnerabilities and for detecting implementation bugs in the firewall software, but not for generating trust in the overall security of a firewall configuration. It also means that testing can interfere with normal network activity.

Passive testing tools can test the entire packet space, but require the use of an analysis
### Figure 3.1: An iptables firewall as printed by the `iptables -L -n` command

<table>
<thead>
<tr>
<th>Chain</th>
<th>Policy</th>
<th>Actions</th>
<th>Target</th>
<th>Source</th>
<th>Protocol</th>
<th>Port</th>
<th>Options</th>
<th>Target IP/Port</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>Deny</td>
<td>DROP</td>
<td>eth0</td>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>tcp</td>
<td>22</td>
<td>192.168.0.22</td>
<td>DROP 0.0.0.0/0</td>
</tr>
<tr>
<td></td>
<td>Accept</td>
<td>ACCEPT</td>
<td>eth0</td>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>tcp</td>
<td>22</td>
<td>192.168.0.22</td>
<td>ACCEPT 0.0.0.0/0</td>
</tr>
<tr>
<td>FORWARD</td>
<td>Accept</td>
<td>ACCEPT</td>
<td>eth0</td>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>tcp</td>
<td>22</td>
<td>192.168.0.22</td>
<td>ACCEPT 0.0.0.0/0</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Drop</td>
<td>DROP</td>
<td>eth0</td>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>tcp</td>
<td>22</td>
<td>192.168.0.22</td>
<td>DROP 0.0.0.0/0</td>
</tr>
<tr>
<td></td>
<td>Accept</td>
<td>ACCEPT</td>
<td>eth0</td>
<td>0.0.0.0/0</td>
<td>0.0.0.0/0</td>
<td>tcp</td>
<td>22</td>
<td>192.168.0.22</td>
<td>ACCEPT 0.0.0.0/0</td>
</tr>
</tbody>
</table>

Note: The output includes details such as packet counts, bytes, and protocol types for various traffic flows. The firewall policy blocks certain types of traffic (e.g., DROP) and allows others (e.g., ACCEPT). The specific rules include criteria for traffic to or from specific network addresses and ports.
engine. The current state of the art in passive analysis is a commercial tool produced by Algorithmic Security called “Algosec Firewall Analyzer”, which is available for Cisco’s PIX and Checkpoint’s FW-1 firewalls. It is a closed-source commercial project based on Wool’s Fang [37] and Lumeta [59] engines. Fang allowed the user to perform simple queries such as “what types of packets can reach the mail server?” In Lumeta, the developers replaced Fang’s query functionality with a graphical tool that checks for specific configuration errors. Algosec is a more capable commercial version of Lumeta. Each of their systems is capable of analyzing multiple firewalls in a specified network topology. Redseal [40] produces a quantitative analysis tool which is similar to the Algosec product. Given a Cisco or Checkpoint firewall policy and a description of the network topology, it identifies each system on the network with a risk factor that can be used to determine which hosts are most in danger of compromise by an intruder.

Another branch of research has focused on simplifying a firewall configuration by removing redundant and conflicting rules. Gouda and Liu [22] present an algorithm for constructing a firewall decision diagram and applying reduction techniques to derive a complete, compact, and consistent firewall. Their technique can reduce the complexity of a poorly configured firewall and uncover some configuration errors, but has a different purpose than such engines as Algosec and SATAN. Gouda and Liu’s work focuses on errors in the structure of a firewall rule set rather than in design flaws such as typos and incorrect rule order.

Other passive analysis engines use expert systems and constraint solvers [16] or computational geometry [15] to analyze a firewall policy.

Unfortunately, none of these passive techniques is widely available in an open source tool which can be used with iptables. The commercial tools are available only for PIX and FW-1 firewalls and no implementations of the academic work have been made available. This means that system administrators must use active testing techniques to evaluate Linux firewalls. Using the quasi-reduced MDD representation of the firewall policy presented in the previous chapter, we have implemented a passive testing tool, ITVal, which allows a system administrator to analyze the behavior of the firewall using logical queries. This
enables a system administrator to detect errors in the policy. For instance, the tool can determine to which hosts the firewall permits SSH access. Using this information, the system administrator can determine whether a host that should be protected provides unwanted connections to the outside world. In this chapter, we will describe the basic query solving functionality provided by the tool. More advanced features of the tool will be described in the remaining chapters.

3.1 ITVal, An Open Source Tool

ITVal is implemented using FDDL [36], a Multi-way Decision Diagram (MDD) library. We chose to use quasi-reduced MDDs [13] over Binary Decision Diagrams (BDDs) [7] because they are better suited for representing integral values such as ports and IP address.

The analysis engine generates an MDD model of the firewall from the textual description of the firewall rule set generated by the “iptables -L -n” command. An example of this input is given in figure 3.1. The example is taken from the filtering table of a firewall used to isolate research machines from workstations and servers. As a security precaution, we have modified the addresses and ports, but we have kept the structure of the firewall in place. From the figure, you can see that working with a textual representation of the firewall is very difficult. The policy is moderately long (about 75 lines of text containing roughly 70 rules) and uses two user-defined chains to process packets sent out over three different ethernet interfaces and the loop back device. Debugging this policy by hand is very difficult and time-consuming. The analysis engine allows the administrator to avoid the arduous task of processing all 75 lines of policy manually.

To use the analysis engine, the system administrator creates a file containing queries written in a simple specification language. In the query file, the user can ask questions such as “What services can be reached on host X?” or “Which machines can be reached with SSH?” The analysis engine can handle many of the features of iptables, including stateful inspection. The tool parses this query file and generates intermediate MDDs for
the elements of each query. These intermediate MDDs are combined to produce a result MDD using intersection and union operations. From the result MDD, the tool obtains a list of packets which match the query from which it can generate the query results.

Using the output produced by the query engine, a system administrator can verify the important security invariants of the network. By running an analysis of the firewall before and after making a change to the policy, she can ensure that the change has not violated any important security constraints. If the query engine returns an unexpected result, she can repair the policy and re-apply the security check to ensure that everything is working correctly.

3.2 Query Language

The analysis tool provides a straightforward query language which allows complex queries to be built from simple primitives.

```
GROUP internalnet 68.10.120.* 68.10.121.*;
GROUP wlan 68.10.122.*;

SERVICE mail TCP 25 TCP 110;
SERVICE ftp TCP 21 TCP 20;

QUERY DADDY FROM wlan AND (FOR mail OR FOR TCP 80)
   AND ACCEPTED forward;
QUERY SPORT TO internalnet AND FOR ftp AND IN NEW
   AND ACCEPTED output;

QUERY SADDY TO internalnet AND FOR 68.11.230.45 AND
   (NOT IN NEW AND NOT IN RELATED)
   AND ACCEPTED forward;

QUERY DPORT FROM internalnet AND TO wlan AND
   (IN NEW OR IN ESTABLISHED)
   AND ACCEPTED forward;
```

Figure 3.2: An example ITVal query file

An example query file is given in figure 3.2. The query file is made up of group and
service definitions followed by one or more query statements. In this example, the first four
lines are definitions. The next four lines are query statements.

3.3 Query Statements

Query statements begin with the word QUERY followed by a subject, a condition, and
a semicolon. The subject of the query specifies what information should be printed about
packets that match the query. For instance, the query in line 5 uses the subject “DADDY”
to indicate that the destination address should be printed. The valid subjects are:

• SADDY : Source Address
• DADDY : Destination Address
• SPORT : Source Port
• DPORT : Destination Port
• STATE : Connection State

The rest of the query statement consists of a condition which specifies the packets to
consider.

3.4 Simple Conditions

The query engine allows the user to build complex conditions out of a few simple con­
ditions. Conditions are built from nine primitives:

• FROM <address group> : Specifies one or more source addresses to match.
• TO <address group> : Specifies one or more destination addresses to match.
• ON <service> : Specifies one or more source ports to match.
• FOR <service> : Specifies one or more destination ports to match.
• WITH <flag> : Specifies a TCP flag (URG, SYN, ACK, PSH, RST, or FIN) to match.

• IN <state> : Specifies a connection state to match.

• LOGGED : indicates that there is a rule which potentially logs the arrival of the packet.

• ACCEPTED <chain> : Specifies the set of packets accepted by the firewall in chain chain.

• DROPPED <chain> : Specifies the set of packets dropped by the firewall in chain chain.

Each of the primitives selects those packets that are accepted and that match the specified criteria. For instance "FROM 127.0.0.1" specifies those packets accepted by the firewall which are outbound from localhost.

In the FROM and TO queries, the address group can either be the name of a predefined address group or the numeric IP address of a host. Asterisks or CIDR notation may be used in numerical addresses to describe an entire subnet.

In the ON and FOR queries, the service can be either the name of a user-defined service or the name of a protocol type (TCP, UDP, BOTH, or ICMP) followed by either the numeric port number of the service, a range of ports expressed using the syntax “[low—high]” (where low and high define an inclusive interval of port numbers), or an asterisk. The asterisk signifies that all packets of the given protocol type should match the query primitive. If the protocol ICMP is chosen, the ICMP packet type number should be specified instead of a numerical port. If BOTH is specified, the query primitive will match TCP and UDP packets, but not ICMP packets.

The WITH primitive allows queries against any of the significant TCP flags.

The IN primitive allows the connection state to be described as any of the following: INVALID, NEW, ESTABLISHED, or RELATED.
The LOGGED primitive stands on its own without any parameters. It indicates that a packet may have been logged by the firewall. Since iptables LOG rules can specify time-related and other external criteria for logging, there is no guarantee that every matching packet will actually be logged.

These conditions can be combined with the ACCEPTED and Dropped primitives to analyze the behavior of the firewall. The ACCEPTED condition specifies all packets that are accepted by some chain of the firewall. The DROPPED condition specifies the packets that are dropped by a chain. The chain must be one of the three built-in chains: INPUT, FORWARD, or OUTPUT. If no chain is explicitly given, the analysis engine assumes that the FORWARD chain should be considered.

### 3.5 Complex Queries

The boolean connectives NOT, AND, and OR allow the user to posit queries of arbitrary complexity. These operators work as one would expect. The expression "NOT FROM TCP 21" matches all packets which are not TCP packets on port 21. The combination "FOR mail OR FROM 127.0.0.1" selects both mail packets and packets outbound from localhost. The AND and OR operators are left associative, while the NOT operator is right associative. Parentheses may be used to disambiguate subexpressions containing multiple operators.

### 3.6 Group and Service Definitions

If the user had to explicitly mention every host address in every query, creating a query file would be a tedious and error prone process. To address this issue, we allow named groups of addresses to be defined and used throughout the query file. The syntax for specifying a group is the word GROUP followed by a name and a space separated list of addresses. As with the FROM and TO primitives, subnets can be specified using asterisks or CIDR notation. Group names must consist entirely of letters and may not match any keyword of the query language.
Similarly, named groups of services may be defined. The syntax for defining a service is the word SERVICE followed by a name and a space separated list of protocols and ports.

Here are some examples of queries that can be used in ITVal:

- **QUERY SADDY TO 192.168.* AND ACCEPTED forward;**
  List all hosts with access to subnet 192.168.0.0/16.

- **QUERY DPORT FROM 113.137.10.* AND NOT FOR TCP 993 AND ACCEPTED forward;**
  List all destination ports, except the secure IMAP port(993), that can be accessed by hosts in the 113.137.10.0/24 subnet.

- **QUERY SPORT NOT FROM 192.168.1.101 AND FOR 137.113.6.2 AND ACCEPTED forward;**
  List all source ports open on host 137.113.6.2 to machines other than host 192.168.1.101.

- **QUERY DADDY FOR TCP 25 AND (IN NEW OR IN ESTABLISHED) AND ACCEPTED forward;**
  List all hosts that can receive packets on port 25 on a connection in the NEW or ESTABLISHED state.

- **QUERY DADDY FROM 192.168.1.* AND (FOR TCP 25 OR FOR TCP 80 OR FOR TCP 110) AND ACCEPTED forward;**
  List all hosts that can receive SSH, SMTP, or HTTP traffic from hosts on the 192.168.1.0/24 subnet.

### 3.7 Implementation

The MDD representation of a query is very similar to the MDD representation of a firewall chain and we can reuse most of our notation from the previous chapter. As with the MDD representation of a firewall chain, nodes in the query MDD represent sets of
packets. Each arc represents a choice of value for a particular attribute. Instead of terminal nodes for ACCEPT and DROP, however, level 0 of a query MDD consists of two special terminal nodes. The node MATCHES represents the set of packets that match the query. The terminal node 0 represents those packets which do not match the query. As with the rule set MDD, we reserve index 0 at each level to store a special node symbolizing the empty set, which is not explicitly stored, but treated as a special case by the MDD algorithms. We will continue to use the notation $<k:p>$ to describe the node with index $p$ at level $k$ and the function $L(e_i)$ to represent the label on edge $e_i$.

In order to create the MDD representation of a query, we associate each of the query primitives with a set of attributes which correspond to levels of the MDD. For instance, the primitive "FROM" corresponds to the four levels which represent the source address. The primitive "TO" corresponds to the set of levels which encode the destination port.

### 3.7.1 ACCEPTED and DROPPED primitives

<table>
<thead>
<tr>
<th>Project(node_index $Target$, level $k$, node_index $n_r$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $k = 0$:</td>
</tr>
<tr>
<td>2 if $n_r = Target$:</td>
</tr>
<tr>
<td>3 return MATCHES.</td>
</tr>
<tr>
<td>4 else:</td>
</tr>
<tr>
<td>5 return 0.</td>
</tr>
<tr>
<td>6 if $n_r = 0$:</td>
</tr>
<tr>
<td>7 return 0.</td>
</tr>
<tr>
<td>8 $n_t = $NewNode().</td>
</tr>
<tr>
<td>9 for $i$ from 0 to maxVal[$k$]:</td>
</tr>
<tr>
<td>10 $&lt;k:n_t&gt;[i] = Project(Target, k - 1, &lt;k:n_r&gt;[i])$.</td>
</tr>
<tr>
<td>11 return $n_t$.</td>
</tr>
</tbody>
</table>

**Figure 3.3:** MDD Projection Algorithm

MDDs for the ACCEPTED and DROPPED primitives are created using the projection operation given in figure 3.3. The projection operation isolates only the accepted (or rejected) packets from the rule set MDD. For instance, if $n_r$ is index of the root node of MDD $M_r$, where $M_r$ is the representation of the a chain of the firewall, then
Project(ACCEPT, K, nr) returns an MDD which matches the packets accepted by Mr.

Lines 1 through 5 of the algorithm handle the case in which the algorithm has reached the terminal level. In this case, if nr is the correct terminal node, we return the index of a terminal node which represents “matches the query”. Otherwise, we return terminal node 0.

Lines 6 and 7 handle the possibility that nr is node 0. If this is the case, <k: nr> represents the empty set, which means that none of the packets represented by that node are accepted (or dropped) by the chain. Therefore, we return index 0, which represents the empty set.

In lines 8 through 10, we create a new node to represent the result of projecting the sub-MDD rooted at <k: nr> onto the target. We look at each possible value of the attributes at level k and create an arc <k: nr>[i] to the result of calling Project on each child of <k: nr> at level k – 1.

Figure 3.4 demonstrates the projection algorithm. The left panel contains a copy of the rule MDD for the firewall described in the introduction. The right panel shows the result of the projection operation on that MDD for the primitive “ACCEPTED forward”. You can determine visually that every path in the rule MDD from the root to the terminal node ACCEPT has a corresponding path to the terminal node MATCHES in the result MDD. All of the paths to DROP in the rule MDD have been removed by the projection operation.

3.7.2 Correctness of the Projection Operation

To show that the Project algorithm is correct, we must demonstrate that given an MDD Mr, the algorithm returns an MDD which matches exactly those packets accepted by Mr.

**Lemma 5** Let Mr be the MDD for a firewall chain, let <K: Nr> be the root node of Mr, and let

\[ n_t = \text{Project}(\text{ACCEPT}, K, Nr). \]
Then, $A(M_r) = M(M_t)$, where $M_t$ is the MDD with root node $<K:N_t>$ and $M(M_t)$ represents the set of packets that map to the MATCHES terminal in $M_t$. That is, that for every packet $p$, $p \in A(M_r)$ if and only if $p \in M(M_t)$.

**Proof:**

Step 1: If $p \in A(M_r)$, then $p \in M(M_t)$.

Since $p \in A(M_r)$, there is a path $P = (e_0, e_1, \ldots, e_K)$ from $<K:N_r>$ to the terminal node ACCEPT such that for each edge $e_i \in P$, $L(e_i) = p[i]$. We must show that there is a path $Q = (f_0, f_1, \ldots, f_K)$ from $<K:N_t>$ to the terminal node MATCHES such that for each edge $f_i \in Q$, $L(f_i) = p[i]$. We proceed by induction on $k$, the level of the MDD.
**Base Case** Let \( k = 0 \). To see that \( A(<k:n_r>) = M(<k:n_t>) \), observe that since \( k = 0 \), the if statement in line 1 will always evaluate to true. Since \( p \in A(M_r) \), \(<k:n_r>\) is the terminal node \( ACCEPT \). Therefore, the if statement in line 2 will also evaluate to true and the \( Project \) statement will return the node \( MATCHES \). Therefore, \(<k:n_t>\) is the terminal node \( MATCHES \) and it is obvious that \( p \in M(M_t) \) since the path from \( MATCHES \) to \( MATCHES \) has no edges and therefore trivially satisfies the requirement that each edge have an appropriate label.

**Induction Hypothesis** Assume that when \( k < K \), that there is a path \( Q = (f_0, f_1, \ldots, f_K) \) from \(<K:n_t>\) to the terminal node \( MATCHES \) such that for each edge \( f_i \in Q \), \( L(f_i) = p[i] \).

**Induction Step** Let \( k = K \). Since \( p \in A(M_r) \), there is a path \( P = (e_0, e_1, \ldots, e_K) \) from \(<K:n_r>\) to the terminal node \( ACCEPT \) such that for each edge \( e_i \in P \), \( L(e_i) = p[i] \). If \( k = 0 \), by the base case we know that \( p \in M(M_t) \). If \( k \neq 0 \), then the if statement in line 1 will be false in the first call to \( Project \). Since \( p \in A(M_r) \), \( n_r \) is not node 0 and the if statement in line 6 will also be false. Therefore, the algorithm will proceed to the \( for \) loop in lines 9 and 10.

When the \( for \) loop reaches iteration \( p[k] \), the algorithm will create an arc \( f_n \) with label \( p[k] \) from \(<k:n_t>\) to the result of calling \( Project(ACCEPT, k-1, <k:n_r>[i]) \). Since \( k-1 < K \), by the induction hypothesis we know that there is a path \( (f_0, f_1, \ldots, f_{K-1}) \) from the returned node to the terminal node \( MATCHES \) such that for each edge \( f_i \) in the path, \( L(f_i) = p[i] \). Thus, the path \( Q = (f_0, f_1, \ldots, f_{K-1}, f_K) \) is a path from \(<K:n_t>\) to \( MATCHES \) which satisfies the condition that for each edge \( f_i \in Q \), \( L(f_i) = p[i] \).

**Step 2:** If \( p \in M(M_t) \), then \( p \in A(M_r) \).

Since \( p \in M(M_t) \), there is a path \( P = (e_0, e_1, \ldots, e_K) \) from \(<K:n_t>\) to the terminal node \( MATCHES \) such that for each edge \( e_i \in P \), \( L(e_i) = p[i] \).
We will show, by induction on \( k \), that for each level there is a path \( Q = (f_0, f_1, \ldots, f_k) \) from \(<k:n_r>\) to the terminal node \( ACCEPT\) such that for each edge \( f_i \in Q, L(f_i) = p[i] \).

**Base Case**  Let \( k = 0 \). Since \( p \in M(M_t) \), \( Project \) returned the node \( MATCHES \). This can only happen in line 3, which means that the if statement in line 2 evaluates to true. Therefore, \(<k:n_r>\) was the node \( ACCEPT\). Thus, it is trivially true that \( p \in A(<k:n_r>) \).

**Induction Hypothesis**  Assume that when \( k < K \), that there is a path \( Q = (f_0, f_1, \ldots, f_k) \) from \(<k:n_r>\) to the terminal node \( ACCEPT\) such that for each edge \( f_i \in Q, L(f_i) = p[i] \).

**Induction Step**  Consider \( Project(ACCEPT, K, n_r) \) when \( k = K > 0 \). Since \( k > 0 \), we know that \(<k:n_t>\) was not created by the return statement in line 2. We also know that \(<k:n_t>\) was not created in line 7, because \( n_t \neq 0 \). Therefore, \( n_t \) was created in lines 8 through 11.

We know that there is a path from \(<k:n_t>\) to \( MATCHES \) for which each edge of the path at level \( i \) is labeled with \( p[i] \). Therefore, in iteration \( p[k] \) of the loop, the algorithm created an edge from \(<k:m_t>[p[k]]\) to a node at level \( k - 1 \) which forms a path from \(<k:n_t>\) to \( MATCHES \).

This edge could only be created by following an arc \( f_K \) with label \( p[K] \) from \(<K:N_r>\). Since this arc points to a node at level \( K - 1 \), by the induction hypothesis, there is a path \((f_0, f_1, \ldots, f_{K-1})\) from this child node to \( ACCEPT\) such that for each edge \( f_i, L(f_i) = p[i] \). Therefore, the path \((f_0, f_1, \ldots, f_{K-1}, f_K)\) is a path from \(<K:N_r>\) to \( ACCEPT\) such that for each edge \( f_i, L(f_i) = p[i] \).
The proof of correctness for the DROPPED keyword is similar. Simply replace
:\textit{ACCEPT} with \textit{DROP} and \( A(M_r) \) with \( R(M_r) \).

### 3.7.3 Other Primitives

Primitives such as FROM and TO specify sets of packets whose attributes match the argument of the query. For instance, the query primitive FROM 192.168.1.2 specifies the set of all packets with source address 192.168.1.2. Each of the attributes matched by a particular query primitive is associated with a set of levels in the MDD representation of the query. We call this set the \textit{level set} of the query.

![MDD for the query primitive “FROM 192.168.1.*”](image)

**Figure 3.5:** MDD for the query primitive “FROM 192.168.1.*”

To construct an MDD for a query primitive other than the ACCEPTED or DROPPED primitives, we identify each level in the level set with a range of values that match the argument of the query. For instance, the query primitive “FROM 192.168.1.*”, can be represented by the list of ranges ([192–192], [168–168], [1–1], [0–255]). The query primitive “FOR TCP 22” can be represented by the list of ranges([TCP–TCP], [0–0], [22–22]).

To generate an MDD representing the primitive, we begin at the bottom level with node MATCHES and work our way up creating a single node at each level. The node created
at a level which is not in the level set has arcs for every value to the node created at the level below. A node created at a level in the level set has arcs only for those values in the corresponding range.

An example MDD for the primitive “FROM 192.168.1.*” is shown in figure 3.5. The top four levels of the MDD correspond to the source address attributes of the query and make up the level set of the FROM operator and have a single node with arcs only for values in the corresponding range. The remaining levels consist of “wildcard nodes” which have arcs for every possible value.

```
node_indexQuery2MDD(query q, level k):
  1  if k=0:
  2    return MATCHES.
  3  nr = NewNode(k).
  4  nj = Query2MDD(q,k-1).
  5  if k $\notin$ LS(q):
  6    for i from 0 to maxVal[k]:
  7      <k: nr>[i] = j.
  8  else:
  9    for i from q[k].low to q[k].high:
 10      <k: nj>[i] = j.
 11  return nr.
```

**Figure 3.6**: Pseudocode for constructing an MDD from a query primitive

We use the algorithm given in figure 3.6 to construct a decision diagram from these ranges. The algorithm is recursive and takes two parameters: an object representing the elements of the query and an integer representing the level of the MDD to construct.

Lines 1 and 2 handle the base case in which the algorithm has reached level 0. In this case, we simply return the terminal node MATCHES. Line 3 creates a new node to store the result of the operation. In line 4, we use recursion to obtain the index of the node created at level $k-1$. The if statement in line 5 determines whether the current level is in the level set of the query. If not, lines 6 and 7 create arcs to node $<k: nj>$ for all possible values of the attribute at level $k$. If so, lines 9 and 10 create arcs only for those values in the appropriate range. Finally, line 11 returns the new node.
3.7.4 Correctness of the Query MDD Generation Algorithm

To demonstrate the correctness of the query MDD produced by Query2MDD, we first define a few terms.

**Definition 8** A packet p matches a query q if for every level k in the level set of q, $q[k].low \leq p[k] \leq q[k].high$. We call the set of packets that match a query its match set.

We will use the notation $M(q)$ to represent the match set of query q.

**Lemma 6** Given a query q, let $N_t = \text{Query2MDD}(q, K)$ and let $M_t$ be the MDD with root node $<K:N_t>$. Then, $M(M_t) = M(q)$.

**Proof:**

Step 1: $M(q) \subseteq M(M_t)$.

Let $p \in M(q)$. We must show that $p \in M(M_t)$. That is, that there is a path $P = (e_0, e_1, \ldots, e_K)$ from $<K:N_t>$ to the terminal node MATCHES such that for each edge $e_i \in P$, $L(e_i) = p[i]$. We proceed by induction on k.

**Base Step** Let $k = 0$. The if statement in line 1 will always evaluate to true, so the algorithm will return the node MATCHES. Thus, $<k:n_t>$ is the terminal node MATCHES. Since $M(MATCHES)$ is the set of all packets, $p \in M(<k:n_t>)$.

**Induction Hypothesis** Assume that when $k < K$, for any integer K, that if $p \in M(q)$ and $n_t = \text{Query2MDD}(k, q)$, then $p \in M(<k:n_t>)$. That is, that there is a path $P = (e_0, e_1, \ldots, e_k)$ such that for each edge $e_i \in P$, $L(e_i) = p[i]$.

**Induction Step** We must show that when $k = K$, the algorithm produces a node $<k:n_t>$ such that if $p \in M(q)$, then $p \in M(<k:n_t>)$. Let $N_t = \text{Query2MDD}(K, q)$. Since $k > 0$ (otherwise, the base case takes care of everything), the if statement in
line 1 will evaluate to false. Therefore, we create a new node in line 3. There are two cases:

Case 1: $k \notin LS(q)$

If $k \notin LS(q)$, the if statement in line 4 will evaluate to true and in lines 5 through 7 we set the arcs of the new node to point to the result of calling Query2MDD at the level below. Let $e_k$ be the arc with label $p[i]$. This arc points to a node formed by calling Query2MDD at level $k - 1$. By the induction hypothesis, we know that $P = (e_0, e_1, \ldots, e_{k-1})$ is a path to the terminal node MATCHES such that for each edge $e_i \in P$, $L(e_i) = p[i]$. Therefore, the path $Q = (e_0, e_1, \ldots, e_{n-1}, e_k)$ is a path from $<k:n_t>$ to the terminal node MATCHES that satisfies the condition that each edge $e_i \in Q$ has label $p[i]$.

Case 2: $k \in LS(q)$

If $k \in LS(q)$, the if statement in line 4 will evaluate to false, so the algorithm proceeds to lines 9 and 10. Now, $q[k].low \leq p[k] \leq q[k].high$, since $p \in M(q)$. Therefore, the for loop will eventually create an arc $e_k$ with label $p[k]$ to a node at level $k - 1$. By the induction hypothesis, there is a path $P = (e_0, e_1, \ldots, e_{k-1})$ from this node to MATCHES such that for each edge $e_i \in P$, $L(e_i) = p[i]$. Therefore, the path $Q = (e_0, e_1, \ldots, e_{k-1}, e_k)$ is a path from $<k:n_t>$ to the terminal node MATCHES such that for each edge $e_i \in Q$, $L(e_i) = p[i]$.

In either case, we have that $p \in M(<K:N_t>)$, so $p \in M(M_t)$. Therefore, by induction, $M(q) \subseteq M(M_t)$.

Step 2: $M(M_t) \subseteq M(q)$.

Let $p \in M(M_t)$. By definition, there exists a path $P = (e_0, e_1, \ldots, e_K)$ from $<K:N_t>$ to the terminal node MATCHES such that for each edge $e_i \in P$, $L(e_i) = p[i]$. We must show that $p \in M(q)$. That is, that for each level $k$, if $k \in LS(q)$, then $q[k].low \leq p[k] \leq q[k].high$. We proceed by induction on $k$. 
**Base Step** If \( k = 0 \), it is trivially true that \( p \in M(q) \), since level 0 cannot be in the level set of a query.

**Induction Hypothesis** Assume that when \( k < K \), for some \( K \), that if \( k \in LS(q) \), \( q[k].\text{low} \leq p[k] \leq q[k].\text{high} \).

**Induction Step** Consider the case that \( k = K \). Then, since \( k > 0 \), \( n_t \) could not have been returned by line 2. Therefore, \( n_t \) was returned by line 11. Let \( e_K \) be the arc from \(<k:n_t>\) which has label \( p[K] \). There are two cases:

Case 1: \( K \notin LS(q) \).

By the induction hypothesis, we know that for every level \( k < K \), that if \( k \in LS(q) \), \( q[k].\text{low} \leq p[k] \leq q[k].\text{high} \). Since \( K \notin LS(q) \), all the levels that are in the level set are at levels less than \( K \). Therefore, we can say that for every level \( k \leq K \), if \( k \in LS(q) \), \( q[k].\text{low} \leq p[k] \leq q[k].\text{high} \).

Case 2: \( K \in LS(q) \).

Since \( K \in LS(q) \), \( e_K \) must have been created in lines 9 and 10. But this can only happen if the label on arc \( e_K \) is between \( q[K].\text{low} \) and \( q[K].\text{high} \), since these are the boundaries of the for loop. Therefore, since \( e_K \) has label \( p[K] \), we know that \( q[K].\text{low} \leq p[K] \leq q[K].\text{high} \). By the induction hypothesis, we know that at every level \( k < K \) in the level set of \( q \), \( q[k].\text{low} \leq p[k] \leq q[k].\text{high} \). Therefore, for all \( k \leq K \) such that \( k \in LS(q) \), we know that \( q[k].\text{low} \leq p[k] \leq q[k].\text{high} \).

In either case, \( p \in M(q) \), so \( M(M_t) \subseteq M(q) \).

### 3.8 Combining Queries

The MDDs created for each query primitive can be combined using MDD intersection and union operations to construct representations of more complex queries. Pseudocode
for the MDD intersection operator is given in figure 3.7. An MDD union algorithm can be derived directly from the intersection operator by modifying the base cases.

```
node_index IntersectMDD(level k, node_index p, node_index q)
1  if k = 0:
2     if <k:p>=MATCHES and <k:q>=MATCHES then return MATCHES.
3     else return 0.
4  if p = 0 or q = 0 then return 0.
5  result = NewMDDNode().
6  for value from 0 to MaxValue(k):
7      Arckresultvalue =
8             IntersectMDD(k -1, <k:p>[value], <k:q>[value]).
9  result = CheckForDuplicates(<k: result>).
10 return result.
```

**Figure 3.7: MDD Intersection Algorithm**

Lines 1 – 4 of the algorithm handle the base cases. The if statement at line 1 checks to see if the algorithm has reached the terminal level. If so, then we return MATCHES if and only if both of the input nodes are MATCHES. If we are not at the terminal level, we return node <0:0>. In line 4, we handle the special case that one of the argument nodes is node <k:0>, which represents the empty set. If so, we indicate that the result is the empty set by returning index 0.

Lines 5 – 9 create a new MDD node representing the intersection of the arguments. The for loop in lines 6 and 7 examines each possible value for the attribute at level k and recursively computes the intersection of the corresponding children of the argument nodes. Line 8 ensures that the result node does not duplicate any existing node. Line 9 returns the result.

The function MaxValue called in line 6 is a helper function that returns the maximum value for the field associated with level k. For instance, the maximum value of level K is 255, since level K represents the first octet of the source IP address.

A copy of the rule set MDD from chapter 2 is given on the left panel of figure 3.8. The middle panel shows the MDD representation of the query “FROM 68.10.122.* AND (FOR TCP 25 OR FOR TCP 80 OR FOR TCP 110)”. The result of intersecting the query MDD
Figure 3.8: Intersecting a query MDD and a rule MDD

with the MDD produced from the rule MDD by the primitive “ACCEPTED forward” is given in the right panel.

To see which packets are represented by the result MDD, we start with the top node of the graph. This node has a single arc representing the value 68, so all packets in the result have a source address that begins with 68.

Following the arc, we reach another node with a single arc. This node represents all packets with second source octet equal to 10. Continuing in this manner, we see that the source address of all packets in the result must be in the group 68.10.122.* and the destination address must be in the group 192.168.2.*.

The protocol must be TCP and the source port can have any value, but the destination port must be port 80, the HTTP port. When we continue down the graph, we find that the result contains packets with any TCP flag condition and in any connection state. In other words, the result of our query is exactly: all HTTP packets from hosts on the 68.10.122.0/24 subnet to hosts on the 192.168.2.0/24 subnet.
In the context of ITVal, the output of the sample query applied to the sample rule set is shown in figure 3.9. Note that the human-readable output corresponds directly to the result MDD of figure 3.8.

3.9 Correctness of the Intersection Operation

To demonstrate the correctness of the intersection algorithm in figure 3.7, we must demonstrate that the match set of the result of intersecting two MDDs is the same as the intersection of the match sets of those two MDDs.

Lemma 7 Let $M_a$ be an MDD with root node $<K:N_a>$, let $M_b$ be an MDD with root node $<K:N_b>$, and let $N_t = \text{IntersectMDD}(K, N_a, N_b)$. If $M_t$ is the MDD with root node $<K:N_t>$, then $M(M_t) = M(M_a) \cap M(M_b)$.

Proof:

Step 1: $M(M_a) \cap M(M_b) \subseteq M(M_t)$.

Let $p \in A(M_a) \cap A(M_b)$. Then, there is a path $P = (e_0, e_1, \ldots, e_K)$ from $<K:N_a>$ to $MATCHES$ such that for each edge $e_i \in P$, $L(e_i) = p[i]$ and there is a path $Q = (f_0, f_1, \ldots, f_K)$ such that for each edge $f_i \in Q$, $L(f_i) = p[i]$.

We show by induction on $k$, that at each level, there is a path from $<k:n_t>$ to $MATCHES$ with appropriate labels on each edge.
Base Step \ Let \( k = 0 \). Then, since \( P \) and \( Q \) are both paths to MATCHES, \( p \) and \( q \) are both the terminal node MATCHES. Therefore, the algorithm will return MATCHES in line 2. Thus, the trivial path consisting of the node MATCHES is a path from \(<k:n_t>\) to MATCHES with appropriate labels and \( p \in M(M_t) \).

Induction Hypothesis \ Assume that for every level \( k < K \), that there is a path \( R = (e_0, e_1, \ldots, e_k) \) from \(<k:n_t>\) to MATCHES such that for each edge \( e_i \in R \), \( L(e_i) = p[i] \).

Induction Step \ Let \( k = K \). We can assume that \( k > 0 \), since otherwise the base case demonstrates that \( p \in M(M_t) \). Since \( k > 0 \), the if statement in line 1 evaluates to false. The if statement in line 4 also evaluates to false, because \( p \in M(M_a) \) and \( p \in M(M_b) \). Therefore, a new node is created in line 5. When the for loop in line 6 reaches the value \( i = p[K] \), it will create a new arc \( g_K \) with label \( p[K] \) to the node created by intersecting the nodes obtained by following arcs \( e_K \) and \( f_K \). By the induction hypothesis, there is a path \( R = (g_0, g_1, \ldots, g_{K-1}) \) from this new node to MATCHES such that for each edge \( g_i \in R \), \( L(g_i) = p[i] \). The arc \( g_K \) has label \( p[K] \), so the path \( (g_0, g_1, \ldots, g_{K-1}, g_K) \) is a path from \(<K:N_t>\) to MATCHES for which every edge \( g_i \) satisfies the property that \( L(g_i) = p[i] \). Therefore, \( p \in M(M_t) \).

Thus, by induction, \( M(M_a) \cap M(M_b) \subseteq M(M_t) \).

Step 2: \( M(M_t) \subseteq M(M_a) \cap M(M_b) \).

Let \( p \in M(M_t) \). We know that there is a path \( P = (e_0, e_1, \ldots, e_K) \) from \(<K:N_t>\) to the terminal node MATCHES such that for each edge \( e_i \in P \), \( L(e_i) = p[i] \).

We must show that there are paths \( Q = (f_0, f_1, \ldots, f_K) \) from \( M_a \) to MATCHES and \( R = (g_0, g_1, \ldots, g_K) \) from \( M_b \) to MATCHES such that for each edge \( f_i \in Q \), \( L(f_i) = p[i] \) and for each edge \( g_i \in R \), \( L(g_i) = p[i] \).

We proceed by induction on \( k \).
**Base Step**  Let $k = 0$. Then, $<k:n_t>$ is the node terminal node $MATCHES$. Since $k = 0$, the if statement in line 1 will always be true. Since $<k:n_t>$ is not node 0, we must have returned $n_t$ from line 2. Therefore, $<k:n_a>$ and $<k:n_b>$ must also be the node $MATCHES$.

This means that $p \in M(<k:n_a>)$ and $p \in M(<k:n_b>)$, since the path consisting solely of the node $MATCHES$ is a path from $<k:n_a>$ to $MATCHES$ and also a path from $<k:n_b>$ to $MATCHES$, which trivially satisfies the requirement that each edge be labeled with an appropriate attribute of $p$. Thus, $p \in M(M_a) \cap M(M_b)$.

**Induction Hypothesis**  Assume that when $k < K$, for some integer $K$, that if $p \in M(<k:n_t>)$, then $p \in M(<k:n_a>) \cap M(<k:n_b>)$. That is, if there is a path $(e_0, e_1, \ldots, e_k)$ from $<k:n_t>$ to $MATCHES$ such that $L(e_i) = p[i]$ for all $0 \leq i \leq k$, then there is a path $P = (f_0, f_1, \ldots, f_k)$ from $<k:n_a>$ to $MATCHES$ and a path $Q = (g_0, g_1, \ldots, g_k)$ from $<k:n_b>$ to $MATCHES$ such that $f_i = g_i = p[i]$ for all $0 \leq i \leq k$.

**Induction Step**  Let $k = K$. We can assume that $k > 0$, since if $k = 0$, the base case proves that $p \in M(M_a) \cap M(M_b)$. Therefore, $<k:n_t>$ was not returned by lines 2 or 3. Since $n_t \neq 0$, it was not returned by line 4. Therefore, $<k:n_t>$ was constructed by lines 5 through 8. Now the loop in line 6 considers each possible value of attribute $k$ and creates an arc for each value. Consider the iteration which creates an arc $e_k$ with label $p[k]$. We know that $e_k$ does not have label 0, since $p \in M(M_t)$. Therefore, neither $<k:p>[i]$ nor $<k:q>[i]$ points to the node $<k-1:0>$. From the induction hypothesis, we know that they must instead point to nodes from which there are paths $(f_0, f_1, \ldots, f_{k-1})$ and $(g_0, g_1, \ldots, g_{k-1})$ to the terminal node $MATCHES$ which have appropriate labels.

Let $f_k = <k:p>[i]$ and let $g_k = <k:q>[i]$. The paths $(f_0, f_1, \ldots, f_{k-1}, f_k)$ from $<k:N_a>$ to $MATCHES$ and $(g_0, g_1, \ldots, g_k)$ from $<k:N_b>$ to $MATCHES$ satisfy
the condition that each edge at level \( i \) is labeled \( p[i] \).

Therefore, by induction, \( p \in M(M_a) \cap M(M_b) \).

Thus, \( M(M_t) \subseteq M(M_a) \cap M(M_b) \), so \( M(M_t) = M(M_a) \cap M(M_b) \).

3.10 Performance

MDD operations can be performed very efficiently through the judicious use of operation caches. Caching the MDD node created by applying an MDD operation to one or more argument ensures that each we consider each combination of arguments exactly one and gives us a guarantee that the algorithm is linear in the product of nodes in the argument MDDs. For clarity of the correctness proofs, we have omitted the caching mechanism from the description of the algorithms above. Adding caching to the operations as outlined is straightforward and does not affect correctness.

3.11 Application of ITVal Queries

```
GROUP wlan 192.168.1.*;
SERVICE special ICMP * TCP 53 TCP 80 TCP 22;
QUERY DPORT FROM wlan AND ACCEPTED forward;
QUERY DADDY FOR ICMP * AND ACCEPTED forward;
QUERY SADDY FOR TCP 53 AND ACCEPTED forward;
QUERY SADDY FOR TCP 80 AND ACCEPTED forward;
QUERY SADDY FOR TCP 22 AND ACCEPTED forward;
QUERY SADDY NOT (FOR special OR FROM wlan) AND ACCEPTED forward;
```

Figure 3.10: Example queries

To illustrate how a hypothetical system administrator might use ITVal to detect and correct configuration errors, we return to the rule set described in the introduction as figure 1.2 (the MDD for this policy is reproduced in this chapter on the left side figure 3.4). The query file shown in figure 3.10 can be used to verify the most important assertions
about this network. For instance, the query on line 3 lists all services which can be accessed by the wireless network. The result of this query should be the empty set, since we want to restrict all access from that network. Similarly, the last query lists all hosts not on the wireless network that can access a service other than those explicitly permitted or denied. Only hosts from the trusted 113.192.10.0/24 network should appear in the answer to this query.

```
GROUP wlan 192.168.1.*;
SERVICE special ICMP * TCP 53 TCP 80 TCP 22;
QUERY DPORT FROM wlan AND ACCEPTED forward;
# Ports:
# 0 results.
QUERY DADDY FOR ICMP * AND ACCEPTED forward;
# Addresses:
# 0 results.
QUERY SADDY FOR TCP 53 AND ACCEPTED forward;
# Addresses:
# 0 results.
QUERY SADDY FOR TCP 80 AND ACCEPTED forward;
# Addresses: [0-191].*.*.*
  # 192.[0-167].*.*
  # 192.168.0.*
  # 192.168.[2-255].*
  # 192.[169-255].*.*
  # [193-255].*.*.*
# 4278190080 results.
QUERY SADDY FOR TCP 22 AND ACCEPTED forward;
# Addresses:
# 0 results.
QUERY SADDY NOT (FOR special OR FROM wlan) AND ACCEPTED forward;
# Addresses: 113.192.10.*
# 256 results.
```

**Figure 3.11:** Output of ITVal when run on a rule set without errors

Running ITVal on the initial configuration gives the output shown in figure 3.11. It is easy to verify that all the requirements are satisfied. Suppose that, as described in the introduction (figure 1.3), the administrator incorrectly inserts a rule allowing IPP printing traffic to the beginning of the policy. This change allows undesirable behavior in that hosts
from untrusted network 192.168.1.0/24 are allowed to print from systems on the trusted network.

GROUP wlan 192.168.1.*;
SERVICE special ICMP * TCP 53 TCP 80 TCP 22;
QUERY DPORT FROM wlan AND ACCEPTED forward;
#Ports: 631
#1 result.
QUERY DADDY FOR ICMP * AND ACCEPTED forward;
#Addresses: #0 results.
QUERY SADDY FOR TCP 53 AND ACCEPTED forward;
#Addresses: #0 results.
QUERY SADDY FOR TCP 80 AND ACCEPTED forward;
#Addresses: [0-191].**.**
  # 192.[0-167].**
  # 192.168.0.*
  # 192.168.[2-255].*
  # 192.[169-255].**
  # [193-255].**.**
# 4278190080 results.
QUERY SADDY FOR TCP 22;
# Addresses: #0 results.
QUERY SADDY NOT (FOR special OR FROM wlan) AND ACCEPTED forward;
# Addresses: **.*.*
# 4294967296 results.

Figure 3.12: Output of ITVal after an error has been introduced into the rule set

Running ITVal on this new rule set produces figure 3.12. It is evident from the fact that the first query no longer produces an empty result that this rule set is incorrect. Realizing her mistake, the system administrator can then move the rule to its correct location in the rule set. Now the output of the first query will once again be the empty set.

3.12 Advantages of the Query Language

One advantage of using a query language is that generic queries generated for one firewall system can be employed on another firewall system with only a few modifications.
This means that even without a complete understanding of the query language syntax, a system administrator can use ITVal to check fundamental security properties. Since manual inspection of the firewall policy requires detailed knowledge of the policy language, this is a significant improvement in usability.

Furthermore, queries are easier to generate correctly than the firewall rule set, because they are more general, not order dependent, and don’t involve a complex interaction between independent chains. Therefore, it is much easier for a system administrator to test the firewall using queries than it is to design a correct rule set initially or make manual changes to the policy.
Chapter 4

Composition of Firewalls

Networks with a large number of hosts must defend against both external and internal intruders. While a perimeter firewall will block many external threats, it is useless against attacks from inside the network. With Trojan horses and viruses extremely prevalent, the problem of intrusions from internal hosts is growing rapidly [33]. To solve this problem, many system administrators complement the perimeter firewall with local firewalls on important internal hosts [28, 20]. On moderately complex networks, the system administrator may place additional firewalls between the perimeter firewall and groups of related hosts to further regulate access between important subnets.

![Image of firewall architecture](image.png)

**Figure 4.1:** Common firewall architecture for defeating insider threats

The resulting architecture looks something like figure 4.1, which depicts a network with
a perimeter firewall, one unprotected host (host 113.137.10.2), two protected servers (hosts 113.137.10.3 and 113.137.10.4), and an internal network (subnet 192.168.1.0/24), which requires additional restrictions. The protected systems could be a mail server and a web server, while the protected subnet might include clients in an accounting department with financial information that must be secured.

The perimeter firewall can mitigate denial of service and other external threats, while the firewall on each workstation secures services that must be protected from an inside intruder. The firewall separating the protected subnet from the rest of the network provides additional protection against internal threats such as zombie hosts and rogue users. Usually, most of the workstations have very similar filtering policies, which makes the distribution of changes easier since the policy can be edited on a single system and then distributed across the network. One or more of the firewalls may also use network address translation (NAT) to further protect critical hosts or to work around the IPv4 address space problem.

The multiplicity of firewalls, in networks like the one described, greatly increases the difficulty of avoiding configuration errors. Removing a rule at the perimeter often means exposing hosts that are not sufficiently protected by their local firewalls. Incorrectly adding rules to a local or intermediate firewall can unintentionally block important network services. Firewalls with NAT are even more complex because a set of translation rules must be considered in addition to the filtering rules.

For example, consider the filtering chains given in figure 4.2 that could be used to secure the network shown in figure 4.1. The first chain is the forwarding chain of a perimeter firewall that protects the 113.137.10.0/24 network against intrusions from an insecure network 113.137.9.0/24. Rule 1 of the chain blocks traffic from the insecure network. Rule 2 protects the mail server by blocking all traffic from the outside world. The remaining rules secure various services that should be allowed to pass through the firewall.

The second chain is the INPUT chain of the internal mail server, 113.137.10.3. It permits SMTP, secure IMAP, and SSH traffic, but blocks anything else.

There are several invariant properties that the administrator wishes to preserve on this
network. First, web traffic from anywhere but the insecure network should always be allowed to host 113.137.10.4, the web server. Second, hosts from the outside world should never be able to SSH into the mail server. Third, no traffic should ever be permitted from the insecure network.

Let's assume that the administrator decides to allow SSH traffic through the perimeter firewall for hosts on subnet 113.137.8.0/24 by adding a rule to the forwarding chain of the perimeter firewall as shown in figure 4.3. The first rule and the last three rules are the same as those in figure 4.2, but the second rule is new. This change preserves the first and third invariant, but violates the second, because SSH traffic from the outside world can now reach the mail server. To correct the violation, the system administrator can either add
restrictions to the filter on the mail server or switch the order of rules two and three in the perimeter filter.

<table>
<thead>
<tr>
<th></th>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DROP</td>
<td>tcp</td>
<td>113.137.9.0/24</td>
<td>anywhere</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DROP</td>
<td>tcp</td>
<td>anywhere</td>
<td>113.137.10.3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ACCEPT</td>
<td>tcp</td>
<td>113.137.8.0/24</td>
<td>anywhere</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ACCEPT</td>
<td>tcp</td>
<td>anywhere</td>
<td>anywhere</td>
<td>TCP dpt:80</td>
</tr>
<tr>
<td>5</td>
<td>ACCEPT</td>
<td>tcp</td>
<td>anywhere</td>
<td>anywhere</td>
<td>TCP dpt:53</td>
</tr>
</tbody>
</table>

**Figure 4.4**: A Correctly Modified Rule Set

A correct rule set for the perimeter firewall is shown in figure 4.4. The new rule set allows HTTP and DNS traffic from 113.137.8.0/24, but preserves all three invariants.

### 4.1 Analyzing Firewall Systems using Existing Tools

Active testing tools such as SATAN [18], Nessus [32, 2], ISS [27], and SARA [51] can be used to test multiple firewalls at once and can test firewalls that use NAT. In fact, with these tools it is often more difficult to segregate firewalls on the network than it is to process them together. Since these tools analyze traffic on the network with little or no information about the firewall topology, it is more difficult for them to determine which firewall is blocking a particular set of packets.

Active tools also have the disadvantage of consuming bandwidth and interfering with normal traffic. Sending a test packet to the wrong host might crash an important server or cause significant delays on the network link. Furthermore, active tools are very inflexible. Rather than providing general functionality for investigating the firewall configuration, they are designed to test specific vulnerabilities. Since they usually simulate packets originating from a single host or small group of hosts, they will miss errors that allow packets from an untested host. Some tools [4] use address spoofing to mitigate this problem, but because of bandwidth constraints, no active tool can test every possible address that might originate a packet to the firewall.
When using active tools with multiple firewalls and address translation, it can be very challenging to obtain adequate coverage of the policy. Because packets dropped by one firewall are never seen by the second, it is often difficult for an active tool to generate a spoofed packet that will exploit configuration errors in both firewalls. Also, replies to packets with NAT'd source addresses may never be seen by the active analysis tool.

Because active tools have these drawbacks, passive tools, which perform an offline analysis of the firewall can be more practical. Passive tools, such as Redseal [40] and Lumeta [59], provide general query capability for offline testing of firewalls and provide support for testing multiple firewalls. Unfortunately, these tools are closed source tools that are not designed to work with iptables firewalls. Existing structure analysis tools [22, 34] also do not support NAT or analysis of multiple firewalls.

Some work has been done [23, 30] on modeling NAT using passive model checking tools, but this work is targeted at demonstrating the correctness of a few specific firewall configurations and is not easily generalizable to arbitrary firewall policies. Other research [56] performs a passive analysis of a distributed firewall system, but handles only a limited number of queries regarding address spoofing.

In this chapter, we show how the quasi-reduced MDD model can be extended to analysis of groups of connected firewalls, network address translation, and more advanced packet mangling techniques such as masquerading and redirection. We illustrate the implementation of these features in ITVal.

### 4.2 Composing Nested Firewalls

In order to extend our technique to work with multiple firewalls, we introduce the concept of a meta-firewall. A meta-firewall is an imaginary firewall that represents a composition of the rule sets of two or more serially connected firewalls.

The meta-firewall has three filter chains analogous to the built-in chains of a normal firewall. The FORWARD chain of the meta-firewall regulates traffic passing through all
the firewalls in either direction. The INPUT chain of the meta-firewall regulates traffic inbound to the innermost firewall through all of the outer firewalls. The OUTPUT chain represents traffic generated by the innermost firewall that successfully passes through the outer firewalls to the outside world.

Queries are performed against the meta-firewall as if it were a single `iptables` firewall. For instance, the query

```plaintext
QUERY DPORT FOR 192.168.* AND IN NEW AND ACCEPTED forward;
```

will list the destination ports of packets bound for the 192.168.0.0/16 subnet that pass through all the firewalls in the set.

<table>
<thead>
<tr>
<th>Firewall* ConstructFirewall(int n, Firewall* fws)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1     newFW = NewFirewall()</td>
</tr>
<tr>
<td>2     newFW.forward = fws[0].forward</td>
</tr>
<tr>
<td>3     newFW.input = fws[0].input</td>
</tr>
<tr>
<td>4     newFW.output = fws[0].output</td>
</tr>
<tr>
<td>5     for i in 1 to n - 1:</td>
</tr>
<tr>
<td>6     newFW.forward = IntersectMDD(K, newFW.forward, fws[i].forward).</td>
</tr>
<tr>
<td>7     newFW.input = IntersectMDD(K, newFW.input, fws[i].forward).</td>
</tr>
<tr>
<td>8     newFW.output = IntersectMDD(K, newFW.output, fws[i].forward).</td>
</tr>
<tr>
<td>9     return newFW.</td>
</tr>
</tbody>
</table>

*Figure 4.5: Algorithm for Constructing a Meta-Firewall*

To construct the meta-firewall, `ITVal` joins the MDD for each chain of the component filters using the MDD intersection operator described in chapter 2.

Pseudocode for generating the meta-firewall is shown in figure 4.5. The INPUT chain of the meta-firewall is constructed by intersecting the FORWARD chains of the outer \( n - 1 \) firewalls and the INPUT chain of the innermost firewall. The OUTPUT chain is created by intersecting the OUTPUT chain of the innermost firewall with the FORWARD chains of the outer \( n - 1 \) firewalls. The FORWARD chain is the intersection of all \( n \) FORWARD chains.

An MDD depicting the meta-firewall for the rule sets in figure 4.2 is shown in figure 4.6. In order to save space, only the levels for source address, destination address, and destination
Figure 4.6: Combining two rule sets into a meta-firewall

port are shown. We also display only paths to the node ACCEPT.

Figure 4.7 illustrates how ITVal might be used to detect the errors described in section 4. We depict the results of three queries before and after the incorrect change. The original, valid, results are shown in Roman font, while the query results for the incorrect policy are shown in bold.

Each query corresponds to one invariant that the administrator wishes to preserve. The first query asks which hosts, other than those on the insecure net, can access the web server. In the original results, we see that, as expected, any host not on the insecure network can access the web server. The new results show that this important invariant still holds in the modified policy.

The second query asks whether the SSH port on the mail server can be accessed from outside the firewall. The original results show that no external machine can reach the mail server. After the modification, however, the results show that the SSH port can be accessed from outside the firewall. The change allows SSH traffic from 113.137.8.0/24 to reach the
# First invariant: Web traffic not from insecure net
# can always reach the web server
GROUP insecure 113.137.9.*;
QUERY SADDY FOR TCP 80 AND NOT FROM insecure AND FOR 113.137.10.3 AND ACCEPTED input;

# Addresses: [0--112].*.*.* [114--255].*.*.* 113.[0--136].*.
# 113.[138--255].*.* 113.137.[0--8].* 113.137.[10--255].*.
# 4294967040 results.

# Second invariant: External hosts should never be able to SSH to the mail
# server.
GROUP internal 113.137.10.*;
QUERY DPORT NOT FROM internal AND FOR TCP 22 AND TO 113.137.10.3 AND ACCEPTED input;

# Ports: 22
# 1 results.

# Third invariant: No traffic from the insecure network can reach the
# mail server.
QUERY DPORT FROM 113.137.9.* AND TO 113.137.10.3 AND ACCEPTED input;

# Ports: 0 results.

Figure 4.7: Query results before and after the change

By comparing these results, the administrator will realize that she has made a mistake and take steps to correct it. The last query tests whether services on the mail server are available to the insecure network. For both policies, the answer is no.

4.2.1 Correctness of the Composition Operation

To show that the composition algorithm is correct, we must demonstrate that the traffic accepted by the FORWARD chain of the meta-firewall is exactly that accepted by the FORWARD chain of every intermediate firewall, that the traffic accepted by the INPUT chain of the meta-firewall is exactly that accepted by the INPUT chain of the innermost firewall and by the FORWARD chains of the other n – 1 firewalls, and that the OUTPUT
chain of the meta-firewall is exactly that accepted by the OUTPUT chain of the innermost firewall and by the FORWARD chains of the other \( n - 1 \) firewalls.

**Theorem 4.1** Given an ordered list \( FW = (F_0, F_1, \ldots, F_{n-1}) \), of \( n \) firewalls, let \( F_m \) be the meta-firewall created by \( \text{ConstructFirewall}(n, FW) \). Then, for every packet \( s \) such that \( s \in A(F_i.\text{forward}) \) for all \( 0 \leq i \leq n - 1 \), \( s \in A(F_m.\text{forward}) \). Also, for each packet \( t \in A(F_m.\text{forward}) \), \( t \in A(F_i.\text{forward}) \) for all \( 0 \leq i \leq n - 1 \).

**Proof:**

Let \( s \in A(F_i.\text{forward}) \) for all \( 0 \leq i \leq n - 1 \). Line 2 copies the FORWARD chain of \( F_0 \) into \( F_m \). Thus, since \( s \in A(F_0.\text{forward}) \), \( s \in A(F_m.\text{forward}) \) immediately after line 2 has executed. The for loop in lines 5 and 6 intersects \( F_m.\text{forward} \) with each firewall \( F_i.\text{forward} \) and stores the result in \( F_m.\text{forward} \). However, since \( s \in A(F_i.\text{forward}) \) for all \( i \), these intersections cannot remove \( s \) from \( A(F_m.\text{forward}) \). Therefore, when we return in line 9, \( s \in A(F_m.\text{forward}) \).

Now, let \( t \in A(F_m.\text{forward}) \). Since \( F_m.\text{forward} \) is created by the intersection of the FORWARD chains of each firewall of \( FW \) in lines 5 through 8, we know that

\[
A(F_m.\text{forward}) = \bigcup_{0 \leq i \leq n-1} A(F_i.\text{forward})
\]

But the correctness proof of the intersection operator from chapter 2 implies that \( A(F_m.\text{forward}) \subseteq A(F_i.\text{forward}) \) for \( 0 \leq i \leq n - 1 \). Thus, \( t \in A(F_i.\text{forward}) \) for each firewall \( F_i \in FW \).

This means that the FORWARD chain of the meta-firewall is exactly the intersection of the FORWARD chains of the constituent firewalls. By trivial modification of the proof, we can also see that the algorithm correctly sets the OUTPUT chain of the meta-firewall to be the intersection of the OUTPUT chain of the innermost firewall with the FORWARD chains of the other firewalls and the INPUT chain of the meta-firewall to be the intersection of the INPUT chain of the innermost firewall with the FORWARD chains of the other firewalls.
4.2.2 Network Address Translation

In addition to filtering packets that pass through the firewall, **iptables** provides a mechanism for modifying a packet's destination address and destination port before filtering or source address and source port after filtering. Properly handling network address translation (NAT) in the query engine is important, because the modified packet may be treated differently by the filtering rules than the original packet. In order for our queries to take NAT into account, we must modify the rule set MDD to reflect each of the NAT rules.

A NAT rule is a function which maps certain packets seen by the firewall to new, translated, packets. These new packets may differ from the original packets in source address, source port, destination address, or any other attribute. The most common types of NAT rules are "source NAT" rules, which modify the source address and/or source port of a packet and "destination NAT" rules, which modify the destination address and/or destination port of the packet.

In practice, the translation rule is described as a set of match conditions that identify a set of packets which should be modified (just as every filtering rule has a set of match conditions which specify which packets should be filtered) and a set of target values which specify the attributes of the new packet produced by translation. Often, most of the target values will be the same as the original packet and only a few of the attributes will change.

The match conditions describe a "match set" of packets which satisfy all of the conditions. The translation rule maps packets which are not in the match set to themselves. That is, it leaves them unchanged. Packets which are in the match set are mapped to new packets. These new packets are usually identical to the original packet, except that some of the attributes have been set to new target values.

Like filtering rules, translation rules are stored using chains. Destination NAT (DNAT) rules for incoming packets are specified in the PREROUTING chain, which is processed before filtering. In addition to the PREROUTING chain, **iptables** provides a POSTROUTING chain, processed after filtering, which is the appropriate place for source NAT (SNAT) rules, and an OUTPUT chain for performing DNAT on locally generated packets.
An example rule set for a firewall which uses NAT, is shown in figure 4.8. This rule set might represent the policy of the intermediate firewall, depicted at the beginning of the chapter in figure 4.1, which protects internal network 192.168.1.0/24 from the outside world by allowing internal hosts to use non-routable IP addresses. To access a host, an external system must connect directly to the NAT'ing firewall on a designated port. The firewall will then forward the connection to the appropriate machine.

The PREROUTING chain given in figure 4.8 contains several NAT rules. The first three rules map packets arriving on ports 2002–2004 of the firewall host to the SSH ports of various internal hosts. The last rule of the chain maps incoming connections on port 3000 to port 9999 of machine 192.168.1.2 (a proprietary financial database is running on that port). The FORWARD chain prevents internal hosts from accessing the web server. The match set of the first rule in the PREROUTING chain is “all TCP packets from 113.137.10.101 on port 2002”. The output of the rule is a new packet which is identical to the original packet except that the destination address is now 192.168.1.2 and the destination port has been changed to TCP port 22 (the SSH port).

To access host 192.168.1.2, a user must connect to port 2002 of the firewall host (host 113.137.10.101). The firewall will then replace the destination address of the packet with
“192.168.1.2” and the destination port with “22” before applying the filter rules and handing the packet over to the router.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DROP</td>
<td>tcp 113.137.10.4</td>
<td>192.168.1.0/24</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DROP</td>
<td>tcp 113.137.10.3</td>
<td>113.137.10.101</td>
<td>TCP</td>
</tr>
</tbody>
</table>

**Figure 4.9**: Incorrectly Configured Filter on the NAT’ing firewall

NAT adds another layer of complexity to the configuration of a firewall. One common mistake is to add filtering rules for the original address rather than the NAT’d address. For instance, if the system administrator decides to further restrict access to the internal hosts, she might add rule 2 of figure 4.9 to the FORWARD chain. This change should prevent the mail server from accessing the proprietary database. The new rule fails to block forwarded connections to the internal hosts, however, because filtering rules are not applied until processing of the translation rules has been performed. Such a mistake is difficult to catch, since it involves the complex interplay of rules in two different tables of the firewall policy.

**Figure 4.10**: Applying NAT to the Example Rule Set
To model network address translation using MDDs, we create an operator which takes as inputs a NAT rule and the MDD representation of a set of packets. It produces as output an MDD which has been modified in such a manner that rules which applied to the original, unmodified packets apply instead to the translated packets. This enables us to perform analysis of the policy which takes into account the effects of address translation.

Figure 4.10 shows the MDD for a filtering policy before and after application of the NAT algorithm. Tracing a path through the MDD along the edges representing "a packet from address 113.137.10.3 to port 3000 of host 113.137.10.101" shows that such a packet will be accepted by the firewall.

```
node_index NAT(NAT_RULE tr, level k, node_index p, node_index q)
1  if k = 0:
2      return q.
3  r = newNode(k).
4  let j = tr[k].target.
5  for each arc i ∈ <k:p>:
6      if i.tf.(tr[k].low, tr[k].high):
7          <k:r>[i] = <k:p>[i],
8      otherwise:
9          <k:r>[i] = NAT(tr, k-1, <k:p>[i], <k:q>[j]).
10     return r.
```

Figure 4.11: Algorithm for NAT of MDDs

Pseudocode for applying a NAT transformation on the MDD representation of the rule set of a single firewall is shown in figure 4.11. The algorithm takes as parameters a NAT rule and two MDD nodes. The first node, <k:p>, represents the set of rules which apply to untranslated packets. The second node, <k:q>, represents the rules which apply to translated packets. To apply a NAT rule, tr to an MDD Mp which represents a filtering chain, we call NAT(tr, K, p, p) where <K:p> is the root node of Mp.

The NAT rule is represented as an array of structures of size K. Each structure consists of two parts: a match interval and a target value. The match interval at position k of the array identifies a range (tr[k].low, tr[k].high) of values for the attribute at level k. These ranges are chosen in such a manner that every packet s matching the NAT rule will satisfy
the condition that at every level $k$, $\text{low}[k] \leq s[k] \leq \text{high}[k]$.

In general, we are not guaranteed that it is possible to decompose the NAT rule into independent ranges for which this condition will hold. However, for the special case of iptables firewalls, the syntax of the policy language ensures that it is always possible to do so. We can extend this technique to other types of firewalls by splitting each NAT rule up into a list of arrays, rather than a single array of structures (in fact, ITval implements NAT translation in this matter), but since this process significantly complicates discussion of the algorithm, we will treat the NAT rule as a single array.

The target value at level $k$, $\text{tr}[k].\text{target}$, specifies a new value for the attribute at that level which should be applied to packets that match the NAT rule.

Using this representation of a NAT rule, $\text{tr}$, we have the property that for any packet $s$ such that $\text{tr}[k].\text{low} \leq s[k] \leq \text{tr}[k].\text{high}$, $\text{tr}(s)[k] = \text{tr}[k].\text{target}$. For all other packets, $\text{tr}(s)[k] = s[k]$.

As we descend the graph recursively, we use index $p$ to track the rules currently applied to a set of packets and index $q$ to track the rules which should be applied to those packets after NAT.

Lines 1 and 2 (see figure 4.11) check for the base case condition in which we have reached level 0. If this has happened, we don’t need to do any more work as all the NAT related information is contained in the preceding levels. We return $q$, the index of the terminal node representing the action applied to the NAT’d packets.

The NewNode operation in line 3 creates a new MDD node $<k:r>$ and returns its index. This new node will represent the result of the NAT operation. Line 4 sets a temporary variable $j$ to the target value of the translation rule for level $k$.

In lines 5 through 10, we consider each possible value $i$ of the attribute represented by level $k$. If the value is not in the match interval of the NAT rule, the node reached by arc $i$ represents rules applied to packets which do not match the NAT rule. Line 7 handles this case by copying the arc with label $i$ from $<k:p>$ into $<k:r>$.

If $i$ is in the match set of the NAT rule, line 9 creates a new node at level $k-1$. This node
is constructed by a recursive call of the NAT operation on nodes $<k:p>[i]$ and $<k:q>[j]$. It represents the set of translated filtering rules which apply to packets with value $i$ for attribute $k$. We create an arc with label $i$ to this node from $<k:r>$. Finally, in line 10 we return $r$.

```
GROUP insecure 113.137.10.3;
QUERY SADDY FROM insecure
AND FOR TCP 3000 AND
ACCEPTED forward;

# Addresses:
# 0 results.
```

**Figure 4.12:** Query for detecting errors in the NAT’ing firewall

```
GROUP insecure 113.137.10.3 113.137.10.4;
QUERY SADDY FROM insecure
AND FOR TCP 3000
AND ACCEPTED forward;

# Addresses: 113.137.10.4
# 1 result.
```

**Figure 4.13:** Results for an Incorrectly Configured Firewall

Figure 4.12 provides a query that might be used to detect the error in figure 4.9. The group “insecure” is a list of hosts that should be prevented from accessing the secure server. A correctly configured firewall should always return an empty result, as in the figure. After making the incorrect change to the filtering rules, the system administrator adds 113.137.10.3 to the group. If she now runs the query a second time, she will see the results in figure 4.13 and detect the error.

### 4.2.3 Correctness of the NAT algorithm

To prove the NAT algorithm is correct, we will show that every packet that is accepted by firewall after translation is accepted by the MDD for the translated rule set and vice-versa.
**Theorem 4.2** Let $M_p$ be a rule set MDD with root node $<K:p>$ and let $M_r$ be the MDD produced by $\text{NAT}(tr,K,p,p)$. For every packet $s$ such that $s \in A(M_r)$, $tr(s) \in A(M_p)$. Also, for every packet $t$ such that $tr(t) \in A(M_s)$, $t \in A(M_r)$.

**Proof:**

**Step 1:** If $s \in A(M_r)$, then $tr(s) \in A(M_p)$.

Since $s \in A(M_r)$, there exists a path $S = (e_0, e_1, \ldots, e_K)$ from the root node of $M_r$ to the terminal $\text{ACCEPT}$ such that $L(e_i) = s[i]$ for each edge $e_i$ in $S$. We must show that there is a path $T = (f_0, f_1, \ldots, f_K)$ from the root node of $M_p$ to $\text{ACCEPT}$ such that $L(f_a) = tr(s)[a]$ for each edge $f_a$ in $T$.

To show this, we use induction on $k$ to show that at any level, for some node $<k:p>$ of $M_p$, there is a path $T = (f_0, \ldots, f_k)$ from $<k:p>$ to the terminal $\text{ACCEPT}$ such that $L(f_a) = tr(s)[a]$ for each edge $f_a$ in $T$.

**Base Case** Let $k = 0$. Then since $s \in A(M_r)$, $M_r$ consists of the terminal node $\text{ACCEPT}$. This can only happen if $q$ was returned in line 2. Thus, $<k:q>$ was also the terminal node $\text{ACCEPT}$. But since, initially, $<K:q> = <K:p>$, this means that the trivial path containing only the node $\text{ACCEPT}$ is a path from $<k:p>$ to $\text{ACCEPT}$ such that for every edge $f_a$ in the path, $L(f_a) = tr(s)[a]$.

**Induction Hypothesis** Assume that at every level $0 < k < n$, there is a path $T = (f_0, f_1, \ldots, f_k)$ from $<k:p>$ to $\text{ACCEPT}$ such that $L(f_a) = tr(s)[a]$ for each edge $f_a \in T$.

**Induction Step** Let $k = n > 0$. Since $k > 0$, the algorithm will skip lines 1 and 2. Therefore, $<k:r>$ was created in line 3 and returned by line 10. Let $i$ be the arc from $<k:r>$ which has label $s[k]$. We know that $i$ was created in the for loop of lines 5 through 9.
There are two cases.

Case 1: Arc $i$ was created in line 7.

In this case, the if statement in line 6 must have been true. This means that $s[k] > tr[k].high$ or $s[k] < tr[k].low$. In either case, we know that $s$ did not match the NAT rule, so $s[a] = tr(s)[a]$ for all $0 \leq a \leq k$.

Now, line 7 copied an arc from node $<k:p>$, so there is an arc $f_k$ with label $tr(s)[k]$ from node $<k:p>$ that points to the same child as arc $i$ of node $<k:r>$. Thus, the path $T = (e_0, e_1, \ldots, e_{k-1}, f_k)$ is a path from $<k:p>$ to ACCEPT which satisfies the condition that $L(f_a) = tr(s)[a]$ for each edge $f_a$ in the path.

Case 2: Arc $i$ was created in line 9.

In this case, the if statement in line 6 was false, so $tr[k].low \leq s[k] \leq tr[k].high$. Line 9 created an arc with label $s[k]$ to the result of performing NAT at level $k - 1$ by following the arc from $<k:q>$ with label $j = tr(s)[k]$ to a node at level $k - 1$.

By the induction hypothesis, we know that there is a path $T = (f_0, f_1, \ldots, f_{k-1})$ from this node to ACCEPT such that $L(f_a) = tr(s)[a]$ for each edge $f_a$ in the path. Thus, the path $(f_0, f_1, \ldots, f_k)$ is a path from $<k:q>$ to ACCEPT such that $L(f_a) = tr(s)[a]$ for each edge $f_a$ in the path.

Since in the initial call to NAT, $<K:p> = <K:q>$, this path is also a path from $<k:p>$ to ACCEPT.

In either case, we have that $tr(s) \in A(M_p)$.

Step 2: If $tr(s) \in A(M_p)$, then $s \in A(M_r)$.

If $tr(s) \in A(M_p)$, then there exists a path $S = (e_0, e_1, \ldots, e_K)$ from $<k:p>$ to the terminal node ACCEPT such that $L(e_a) = tr(s)[a]$ for every edge $e_a \in P$. We must show that there is a path $T = (f_0, f_1, \ldots, f_K)$ from $<k:r>$ to ACCEPT such that $L(f_a) = s[a]$ for every edge $f_a \in T$. 


We proceed by induction.

**Base Case** Let $k = 0$. Then, the if statement in line 1 will be true and we will return the terminal node $ACCEPT$ in line 2. This means that $<k:r> = ACCEPT$. Thus, the trivial path containing only the terminal node $ACCEPT$ is a path from $<k:r>$ to $ACCEPT$ such that for each edge $f_a$ in the path, $L(f_a) = s[a]$.

**Induction Hypothesis** Assume that for $0 < k < n$, there is a path

$$T = (f_0, f_1, \ldots, f_k)$$

from $<k:r>$ to $ACCEPT$ such that $L(f_a) = s[a]$ for each edge $f_a \in T$.

**Induction Step** Let $K = n > 0$. Since $k > 0$, we skip lines 1 and 2 and proceed immediately to line 3. This line creates a new node $<k:r>$. We then proceed to the for loop in lines 5 through 9, which examines each possible value, $i$, of the attribute at level $k$. When the for loop reaches $i = s[k]$, we have two possibilities:

**Case 1: $tr[k].low \leq s[k] \leq tr[k].high$**

If this is the case, line 9 creates an arc $f_k$ with label $s[k]$ from node $<k:r>$ to the node created by performing NAT at level $k-1$. By the induction hypothesis there is a path $(f_0, f_1, \ldots, f_{k-1})$ from this node to $ACCEPT$ such that for each edge $f_a$ in the path, $L(f_a) = s[a]$.

Thus, the path $T = (f_0, f_1, \ldots, f_{k-1}, f_k)$ is a path from $<k:r>$ to $ACCEPT$ such that $L(f_a) = s[a]$ for each edge $f_a \in T$.

**Case 2: $s[k] < tr[k].low$ or $s[k] > tr[k].high$**

In this case, $s$ does not match the NAT rule, so we know that $s[a] = tr(s)[a]$ for all $0 \leq a \leq k$. Line 7 creates an arc $f_k$ with label $s[k]$ from $<k:r>$ to a child of $<k:p>$. But since $(e_0, e_1, \ldots, e_{k-1})$ is a path from this child to $ACCEPT$ such
that $L(e_a) = tr(s)[a] = s[a]$, we know that the path $T = (e_0, e_1, \ldots, e_{k-1}, f_k)$ is a path from $<k:t>$ to $ACCEPT$ such that for each edge $f_a \in T$, $L(f_a) = s[a]$.

Thus, in either case, $s \in A(M_r)$. 

4.3 Nested Composition with Network Address Translation

<table>
<thead>
<tr>
<th></th>
<th>Firewall* NAT(int n, Firewall** FW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>newFW = NewFirewall()</td>
</tr>
<tr>
<td>2</td>
<td>newFW.forward = DNAT_ALL(fws[0].dnat, fws[0].forward).</td>
</tr>
<tr>
<td>3</td>
<td>newFW.input = DNAT_ALL(fws[0].dnat, fws[0].input).</td>
</tr>
<tr>
<td>4</td>
<td>newFW.output = DNAT_ALL(fws[n - 1].nat, fws[n - 1].output).</td>
</tr>
<tr>
<td>5</td>
<td>for i in 1 to n - 1:</td>
</tr>
<tr>
<td>6</td>
<td>newFW.forward =</td>
</tr>
<tr>
<td></td>
<td>IntersectMDD(K, newFW.forward, fws[i].forward).</td>
</tr>
<tr>
<td>7</td>
<td>newFW.forward = SNAT_ALL(fws[i - 1].snat, newFw.forward).</td>
</tr>
<tr>
<td>8</td>
<td>newFW.forward = DNAT_ALL(fws[i].dnat, newFW.forward).</td>
</tr>
<tr>
<td>9</td>
<td>newFW.input =</td>
</tr>
<tr>
<td></td>
<td>IntersectMDD(K, newFW.input, fws[i].forward).</td>
</tr>
<tr>
<td>10</td>
<td>newFW.input = SNAT_ALL(fws[i - 1].snat, newFw.input).</td>
</tr>
<tr>
<td>11</td>
<td>newFW.input = DNAT_ALL(fws[i].dnat, newFW.input).</td>
</tr>
<tr>
<td>12</td>
<td>newFW.output =</td>
</tr>
<tr>
<td></td>
<td>IntersectMDD(K, newFW.output, fws[(n - i) - 1].forward).</td>
</tr>
<tr>
<td>13</td>
<td>newFW.output = SNAT_ALL(fws[(n - i)].snat, newFw.output).</td>
</tr>
<tr>
<td>14</td>
<td>newFW.output = DNAT_ALL(fws[(n - i) - 1].output, newFW.output).</td>
</tr>
<tr>
<td>15</td>
<td>newFW.forward = SNAT_ALL(fws[n - 1].snat, newFW.forward).</td>
</tr>
<tr>
<td>16</td>
<td>newFW.input = SNAT_ALL(fws[n - 1].snat, newFw.input).</td>
</tr>
<tr>
<td>17</td>
<td>newFW.output = SNAT_ALL(fws[0].snat, newFw.output).</td>
</tr>
<tr>
<td>18</td>
<td>return newFW.</td>
</tr>
</tbody>
</table>

Figure 4.14: NAT with multiple firewalls

The pseudocode in figure 4.14 combines NAT with analysis of multiple firewalls. The procedure DNAT_ALL applies the chain of DNAT rules pointed to by its first parameter to the rule set MDD specified by the second parameter. The procedure SNAT_ALL works similarly for SNAT.

In order to correctly derive the output chain of the meta-firewall, we work from the outermost firewall toward the innermost firewall combining pairs of firewalls. We DNAT
the outermost firewall, then enter a loop in which we intersect the result with the unNAT'd filter rules of the next firewall to be considered. In each iteration of the loop, we perform SNAT on the result of the intersection using the SNAT rules of the first firewall. We then DNAT using the DNAT rules of the second firewall. This alternating behavior simulates the traversal of a packet first through the PREROUTING chain, then through the filtering rules, and finally through the POSTROUTING chain.

To derive the input and forward chains, we perform the same operations in reverse order, working from the innermost firewall to the outermost firewall.

4.3.1 Correctness of NAT with Composition Operations

Correctness of the algorithm in figure 4.14 follows directly from the proofs presented for the correctness of the NAT algorithm and the MDD intersection.

From the correctness of the DNAT algorithm, we know that after line 2, $newFW.forward$ contains an MDD which accepts exactly those packets accepted by firewall 0.

To see that the for loop in lines 5 through 14 correctly merges the remaining firewalls into $newFW.forward$, notice that since the intersection algorithm is correct, we know that after line 6, $newFW.forward$ is an MDD which accepts exactly those packets which have passed through firewalls 0 through $i - 1$ and through the forward chain of firewall $i$, but have not yet been processed by the POSTROUTING chain. From the correctness of the SNAT algorithm, we know that after line 7, $newFW.forward$ accepts those packets which have passed through all chains of firewalls 0 through $i$. Therefore, at the end of the algorithm, the $newFW.forward$ is an MDD which accepts exactly those packets which can pass through the filtering and translation chains of all of the firewalls.

Similarly, we know that the MDD for $newFW.input$ accepts exactly those packets which pass through the $n - 1$ outermost firewalls and the input chain of the innermost firewall and the MDD for $newFW.output$ accepts exactly those packets which pass from the innermost firewall through the remaining $n - 1$ firewalls.
4.4 Analyzing Firewall Systems with ITVal

The techniques described in this chapter have been implemented by extending our tool, ITVal, to allow analysis of multiple firewalls and firewalls that use address translation.

To analyze a meta-firewall, the user passes the names of several rule set description files on the command line. The order of the filenames must reflect the topology of the firewalls, with the innermost filter first on the command line and the outermost filter last. The user may optionally specify a topology file that identifies the IP addresses of each interface on any firewall host.

In addition to supporting destination and source NAT, iptables provides two special case NAT targets. The REDIRECT target rewrites the destination address of a packet so that it will be routed to the firewall itself. The MASQUERADE target rewrites the source address of a packet so that it appears to have been originated by the firewall. The REDIRECT and MASQUERADE targets are extremely useful for environments in which addresses are assigned dynamically, since the address of the original host need not be known apriori when designing the rule set. In order to represent REDIRECT and MASQUERADE rules, ITVal looks up the IP address of the host in the topology table provided by the user and performs SNAT or DNAT using the correct address for that interface.
Chapter 5

An Equivalence Class Approach to Policy Testing

Passive tools such as ITVal can make the process of testing and debugging the firewall much easier. Writing queries is simpler than constructing the policy, because the queries only need to provide a partial specification of the firewall policy. Furthermore, the query file is less complex than the rule set, because queries are order independent, while rules in a rule set often have very complicated dependencies.

In theory, passive tools can test every possible behavior of the firewall. In practice, however, such a test produces too much unstructured output to be useful. Testing all eventualities would produce output for every possible packet seen by the network. Since there are $255^4$ possible source addresses and the same number of destination addresses, there are billions of packets to consider — an overwhelming amount of output. Since the decision of which behaviors are desirable and which are undesirable must be made by the user, the tool is unable to structure these outputs in a way that makes it easy to identify incorrect behavior.

To avoid this problem, the user must carefully construct a set of queries that test for specific vulnerabilities. While it is often easier to construct these tests than to inspect the rule set manually, it can be difficult to create queries that test enough interesting behaviors.
to provide confidence in the policy and also produce useful output.

There is no way to guarantee that all important behaviors have been tested. To obtain an ideal set of queries in which exactly those behaviors are tested that could lead to a security violation, the administrator would need to be omniscient.

This means that query-based tools may miss important vulnerabilities. If the system administrator fails to provide a test for an important threat, the testing software cannot detect that the firewall is vulnerable. Since mistakes that are difficult to catch by manually inspecting the rule set are also likely to be overlooked when writing queries, it is likely that the query tool will fail to detect a significant number of errors.

Furthermore, constructing a set of comprehensive and effective queries can be an extremely challenging task, which requires a significant investment of time and resources. Designing good queries requires apriori knowledge of potential firewall problems and familiarity with the subtleties of firewall design.

This problem is not unique to passive testing tools that use query engines. Active tools require the user to decide which behaviors to test. For instance, to use a port scanner, the system administrator must provide a list of hosts to analyze. Scanning all ports on a few important servers will often catch the most critical vulnerabilities, but it is often helpful to also scan individual workstations for less obvious errors. To check for as many vulnerabilities as possible, the user must craft a testing pattern that balances running time against the number of hosts to scan, the number of ports to check, and the number of spoofed source addresses to employ.

Vulnerability scanners such as Nessus [32] also require a significant amount of user input. These tools make use of a database of pre-designed tests. While well-known vulnerabilities can usually be caught using the scripts provided with the scanner, creating new tests requires learning a sophisticated scripting language.

One reason firewalls are so difficult to manage is that slight differences in the rule set can cause dramatic changes in the behavior of the firewall. For instance, on iptables firewalls [17], the filtering policy is specified using ordered chains of rules. In each chain, the
first rule that matches a packet is used to determine the fate of the packet. Reversing two rules can introduce an error that is difficult to detect, but significantly modifies the behavior of the firewall. Other firewall systems have similarly deceptive semantics. For instance, the firewall policy on an ipfilters system uses a “last-match” policy to determine the fate of the packet [41]. Features such as network address translation and stateful filtering can also create opportunities for introducing difficult-to-detect errors.

Another problem is that subtleties in the syntax of the query language can cause the query engine to generate unexpected results. This means that in addition to understanding the structure of a firewall policy, the user must learn the intricacies of the query language in order to employ the tool effectively.

```
QUERY DADDY
   FOR TCP 80 AND
   NOT FROM 192.168.1.*;

# Addresses: *.*.*.*
```

**Figure 5.1:** Query for detecting web hits from outside an internal network

The ITVal query given in figure 5.1 might be used to discover which servers provide web access to hosts outside the network. The “DADDY” subject tells ITVal to list the destination addresses of these machines. The query condition “FOR TCP 80” specifies a match against all HTTP packets while the condition “NOT FROM 192.168.1.*” excludes internal hosts from consideration.

```
CHAIN FORWARD (policy DROP):
  target prot source destination flags
  1 ACCEPT all 192.168.1.0/24 Anywhere TCP dpt:80
  2 ACCEPT all Anywhere Anywhere state ESTABLISHED
```

**Figure 5.2:** Forwarding chain of a stateful firewall

For many firewalls, the query in figure 5.1 will work as expected. However, for a stateful firewall, such as the `iptables` rule set of figure 5.2, it is likely that this query will generate many false positives.
When ITVal processes the query against the stateful firewall of figure 5.2, it will report that any host can send web traffic through the firewall. This surprising result is correct because of a technicality in how stateful filtering works. The rule on line 2 allows arbitrary access on established connections. A careful examination of the rule set, however, reveals that only machines on the internal network can initiate new connections to the web server.

A more precise query that examines only new connections is given in figure 5.3. The new query correctly reports that only internal hosts can initiate new HTTP connections.

Although query-based testing tools can be a significant help to the system administrator, they are limited by the user's ability to construct a comprehensive set of useful queries. It is difficult to tell whether a set of queries tests every important behavior of the firewall. Furthermore, testing techniques often generate too much output for the user to easily distinguish dangerous vulnerabilities from desired behavior.

Another approach to firewall analysis is to look for errors in the structure of the policy specification. Structure analysis tools [1, 22] detect problems such as duplicate or conflicting rules. Although these tools do not directly identify vulnerabilities, they often uncover fundamental weaknesses in the policy that can produce more significant errors. Some of these tools also generate a simpler version of the policy that removes these structural weaknesses. The generated policy is often easier to inspect manually than the original policy.

One significant advantage of structure analysis tools is that they can be fully automatic. The only input the user must provide is the firewall policy itself. The tool builds a list of anomalies and outputs a report or a restructured version of the policy. Unfortunately, there are many types of vulnerabilities that cannot be detected using these tools. For instance,
allowing mail traffic from the outside world to certain workstations could be undesirable behavior on some networks. A structure analysis tool would not detect a problem of that nature unless the rule that permitted the flow of such traffic also conflicted with another rule or violated the structural criteria in some other way.

5.1 Host Classification

The “Lumeta Firewall Analyzer” [59], a commercially available tool derived from FANG [37], combines some of the advantages of a structure analysis tool with the flexibility of a passive analysis tool. Lumeta automatically generates a comprehensive set of queries by using routing information to classify hosts into groups [59]. This reduces the amount of output since results can be provided on a per-subnet or per-zone basis rather than a per-host basis. It also automates the process of designing good queries by providing a set of hard-coded default tests that cover most of the interesting behaviors on the network.

The idea of classifying hosts into groups allows a query engine to provide much simpler output and addresses the problem of creating good queries. Using the topology of the network to classify hosts, however, has the drawback that hosts with very different properties, but that have similar addresses, are grouped together.

<table>
<thead>
<tr>
<th>Chain FORWARD (policy DROP):</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.4: Controlling mail with a packet filter

Consider, for instance, the filtering policy shown in figure 5.4. This simple policy restricts outgoing mail from an internal network 10.239.202.0/24. Outgoing traffic is only allowed from the mail server, host 10.239.202.38. Other hosts on the subnet are allowed to send mail to the mail server, but cannot send mail to each other or to the outside world. Incoming SMTP mail traffic from the outside world is also dropped unless it is destined for
the mail server.

A classification based on the network topology would break the network into two groups: the set of hosts on 10.239.202.0/24 and the set of all other hosts. However, the mail server is a different type of host from the other machines on the network. As a result, queries about mail traffic will return imprecise results. For instance, the answer to the query “Can a host in 10.239.202.0/24 send mail to the outside world?” will be “yes” since the mail server is allowed to forward mail through the firewall. The query “Can all hosts in 10.239.202.0/24 send mail to the outside world?” will be “no” since the client machines are not allowed to send mail to anywhere but the mail server.

Neither of these queries accurately describes the fundamental organization of the network: a special mail server which can send mail to the outside world and a set of clients which cannot. To improve the precision of these queries, we must use a different classification scheme that allows us to group hosts by their function as well as by their placement in the network topology.

5.2 Policy-Based Host Classification

Hosts on a network play a variety of roles. Some hosts are workstations. Some are database servers. Some are web servers. Some provide multiple services. The firewall policy usually treats these various types of hosts very differently from each other, but treats hosts of the same type similarly. This means that the firewall implicitly classifies hosts into various groups based on their function. Sometimes the implicit classification of the firewall policy is not quite as straightforward as simply sorting hosts by the services they provide. For instance, the network may have web servers that provide service exclusively to hosts inside the network and a separate block of general purpose web servers that anyone can access. The filtering policy for these two kinds of hosts could be drastically different even though all of the hosts are web servers.

The rule set in figure 5.5 prevents hosts on an untrusted network 192.168.2.0/24 from
accessing systems on a protected network 192.168.1.0/24. Rule 1 divides the set of hosts into three groups. One group consists of hosts in the untrusted network. The second group contains hosts from the protected network. The third group contains all other hosts on the Internet. Rule 2 refines this classification by further restricting which services are available to the web server. This defines a fourth group by distinguishing the web server from other hosts on the protected network. The fourth group contains only the web server, while the third group contains all other protected hosts.

This classification scheme has many advantages over a topological classification. An error in the firewall policy will often cause the firewall to treat similar hosts differently or to treat different hosts alike. This means that a classification scheme based on the structure of the firewall policy can be used to directly detect many kinds of errors. Furthermore, classifying hosts according to their treatment by the firewall produces groups of hosts that can be used to increase the precision of query-based testing techniques.

5.2.1 Calculating Host Classes

There are several possible ways in which we might use the structure of the firewall policy to categorize the hosts on a network. One approach is to search through the firewall policy and record every address or group of addresses mentioned in a rule as a separate host class. Unfortunately, this naive approach has some serious drawbacks.

The core difficulty is that the algorithm may generate overlapping classes. For instance, the host 192.168.1.1 in figure 5.5 would be represented twice: once in its own class and once in the class containing all hosts from the 192.168.1.0/24 subnet. This is undesirable because it decreases the precision we can obtain in our analysis.
A host that appears in two classes is fundamentally different from the other hosts in those classes. To preserve this information, it is preferable to separate these hosts into their own classes. This will enable us to obtain more accurate and useful results.

```plaintext
set CalculateClasses(Policy P):
  1 set C = 0.0.0.0/0.
  2 for each rule r in P:
  3    for each addr_range S in r:
  4    C = InsertAddr(C, S).
  5 return C.

set InsertAddr(set C, addr_range S):
  1 for each element T of C:
  2    I = IntersectAddress(S, T).
  3    if I is empty:
  4    C = SetAdd(C, S).
  5    return C.
  6    C = SetDelete(C, T).
  7    C = InsertAddr(C, S-I).
  8    C = InsertAddr(C, I).
  9    C = InsertAddr(C, T-I).
 10 return C.
```

Figure 5.6: Naive algorithm for computing host classes

The algorithms in figure 5.6 reduce the amount of overlap by splitting overlapping classes into smaller pieces using set operations. The algorithm examines every host and set of addresses mentioned explicitly in the rule set. Each new range of addresses is added to a set of potential classes, C.

If a new set of addresses overlaps with an existing class, we break both classes into three non-overlapping pieces and replace both original classes with the result. When we have considered every address of every rule, the elements of C describe a set of classes that can be used to analyze the behavior of the firewall.

This approach yields an approximation of the firewall designer's view of the network. Addresses that are explicitly mentioned usually correspond to important components that the designer intended to control. Unfortunately, the technique does not give a perfect picture of the actual behavior of the firewall. For instance, the firewall rule set in figure 5.7
seems at first glance to have three groups. The algorithm will create a group for subnet 192.168.2.0/24 and for subnet 192.168.3.0/24. It will also create a group representing "all other addresses".

In reality, hosts on the 192.168.3.0/24 subnet are treated exactly the same as hosts in the "all other addresses" group, because rule 3 of the firewall policy is an unreachable rule. Since all packets from 192.168.2.0/24 will be dropped in rule 1, no packet can ever match rule 3. This is probably an error in the firewall configuration, but the naive algorithm will happily report that 192.168.3.0/24 is a separate class. Since this is what the user expects, the error will go undetected.

To correct this problem we need to more carefully define the concept of a host class. We do this by constructing an equivalence relation over the set of all network hosts. The equivalence classes determined by this relation will give us a precise and complete characterization of the policy that we can use for performing vulnerability analysis.

5.2.2 Structure-Based Classification

Every firewall policy can be described as a function, $F$, that maps the set of all network packets to the set \( \{ACCEPT, DROP\} \) of filtering decisions. For a specific packet $s$, we say $F(s) = ACCEPT$ if the packet would be accepted by the firewall and $F(s) = DROP$ if the packet would be dropped by the firewall.

We define an equivalence relation, $\equiv_{SD}$, as follows: let $x$ and $y$ be any two hosts. We say that $x \equiv_{SD} y$ (pronounced "$x$ and $y$ are source equivalent") if and only if for any two packets $s$ from $x$ and $t$ from $y$ that differ only by source address, $F(s) = F(t)$. Similarly,
$x \equiv_D y$ (pronounced "x and y are destination equivalent") if and only if $F(s) = F(t)$ for any two packets $s$ from $x$ and $t$ from $y$ that differ only by destination address. If $x \equiv_S y$ and $x \equiv_D y$, then we say that $x \equiv_{SD} y$ ("x and y are source and destination equivalent").

Informally, two hosts are source equivalent if replacing the source address of a packet from one host with the source address of the other does not affect the filtering decision of the firewall. They are destination equivalent if replacing the destination address does not affect the filtering decision. If they are both source and destination equivalent, we say that they are equivalent under the relation $\equiv_{SD}$. The relation $\equiv_{SD}$ is derived directly from the function $F$, which describes the filtering policy of the firewall and can be computed without any other input from the user.

It can be shown that $\equiv_{SD}$ is an equivalence relation, since it is reflexive, transitive, and symmetric. This means that $\equiv_{SD}$ partitions the set of network hosts into equivalence classes. In other words, a packet from a host in a particular equivalence class will only be accepted if identical packets from other hosts in the class would also be accepted.

This means that if one host in the class has a vulnerability, all hosts in the class are vulnerable. On the other hand, if that host is adequately protected by the firewall, then all the others are too. This guarantee makes the equivalence class paradigm much more useful than the naive classification algorithm or a classification based on topology.

5.2.3 Implementation

We can use the reduction properties of MDDs to compute the equivalence classes of the firewall. Since we are using quasi-reduced MDDs, duplicate nodes are not allowed. This means that each node in the MDD represents a distinct class of packets from the other nodes at its level.

In ITVal, we use quasi-reduced MDDs in which duplicate nodes, with all arcs the same, are not allowed. This requirement means that each node at level $k$ represents an equivalence class over the set of attributes $K$ through $k+1$, where level $K$ is the top level of the MDD. For instance, node $<2:1>$ of figure 5.8 represents the source equivalence class containing
addresses 0 and 1. Node <2:2> represents the class containing source address 2. By reordering the levels of the MDD, we can calculate equivalence classes over first the source address and then the destination address. We can use these intermediate classes to construct classes of hosts that are equivalent under the \( \equiv_{\text{SD}} \) relation.

An extremely simplified example MDD is given in figure 5.8. The top level of the MDD corresponds to the source address of a packet, while the second level corresponds to the destination address of the packet. The bottom level is a special terminal level representing the action that the firewall should take on a packet. The integer value 0 means to drop the packet. The integer value 1 means to accept the packet.

An outline of the class generation procedure is given in figure 5.9. In step 1, we generate an MDD representation for each of the three built-in chains. The MDD representation

![Figure 5.8: A simplified rule set MDD](image)

<table>
<thead>
<tr>
<th>1</th>
<th>Construct the MDD representation of each firewall chain.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>For each chain:</td>
</tr>
<tr>
<td>3</td>
<td>Reorder the levels of the chain MDD so that source address is on top.</td>
</tr>
<tr>
<td>4</td>
<td>Record the source equivalence classes.</td>
</tr>
<tr>
<td>5</td>
<td>Reorder the levels of the chain MDD so that destination address is on top.</td>
</tr>
<tr>
<td>6</td>
<td>Record the destination equivalence classes.</td>
</tr>
<tr>
<td>7</td>
<td>Merge the source and destination classes of all three chains together.</td>
</tr>
</tbody>
</table>

**Figure 5.9: Outline of the equivalence class computation algorithm**
takes into consideration network address translation and other packet mangling rules. We then consider each chain in turn. In steps 3 and 4, we compute a list of source equivalent addresses. To do this, we first use a level swapping algorithm to bring the levels encoding source address to the top of the graph. The reduction properties of the MDD now guarantee that each node at the level immediately below the source address levels represents an equivalence class with respect to source address. Each path from the root node to a node at that level represents one element of the equivalence class associated with that node. Step 4 extracts these equivalence classes and stores them in a new MDD.

```
<table>
<thead>
<tr>
<th>Source Class 0</th>
<th>0.0, 0.1, 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Class 1</td>
<td>1.1, 2.0, 2.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Destination Class 0</th>
<th>0.0, 0.1, 1.0, 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Class 1</td>
<td>1.1</td>
</tr>
<tr>
<td>Destination Class 2</td>
<td>2.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Host Class 0</th>
<th>0.0, 0.1, 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Class 1</td>
<td>1.1</td>
</tr>
<tr>
<td>Host Class 2</td>
<td>2.0</td>
</tr>
<tr>
<td>Host Class 3</td>
<td>2.1</td>
</tr>
</tbody>
</table>
```

In steps 5 and 6 we perform an identical operation to collect a list of equivalence classes.
with respect to destination address. When we have considered source and destination address in every chain, we now merge the various classes together using MDD union, intersection and difference operators. Finally, we print the result.

A more detailed illustration of the algorithm is given in figure 5.10.

5.3 Correctness of Equivalence Class Generation

To show that the equivalence class generation algorithm is correct, we first establish that if the top four levels of the MDD represent the source address attributes, each node at level \( K - 4 \) represents an equivalence class under the \( \equiv_s \) relation.

**Theorem 5.1** Let \( g \) be a host with address \( g_0.g_1.g_2.g_3 \) and let \( h \) be a host with address \( h_0.h_1.h_2.h_3 \). Then \( g \equiv_s h \) if and only if, there is a node \( <(K - 4):s> \) such that there exist two paths \( P_s = (e_0, e_1, e_2, e_3) \) and \( Q_s = (f_0, f_1, f_2, f_3) \) from the root node \( <K:r> \) to node \( <(K - 4):s> \) for which \( L(e_i) = g_i \) and \( L(f_i) = h_i \) for all \( 0 \leq i \leq 3 \).

We will first show that each node at level \( K - 4 \) defines an equivalence class over the attributes at levels \( K \) through \( K - 3 \). We will then show that each equivalence class corresponds to exactly one node at level \( K - 4 \).

**Proof:**

Step 1: Let

\[
P_s = (e_0, e_1, e_2, e_3)
\]

and

\[
Q_s = (f_0, f_1, f_2, f_3)
\]

be paths from \( <K:r> \) to \( <(K - 4):s> \). Further, let \( L(e_i) = g_i \) and \( L(f_i) = h_i \) for all \( 0 \leq i \leq 3 \). Then \( g \equiv_s h \).

Let \( p \) be and \( q \) be any two packets such that \( p \) is from host \( g \), \( q \) is from host \( h \), and \( p \) and \( q \) differ only by source address. To demonstrate that \( g \equiv_s h \), we must show that \( F(p) = F(q) \).
Let $P_p = (e_{K-4}, e_{K-3}, \ldots, e_0)$ be a path from $<(K-4):s>$ to the terminal node representing $F(p)$ such that $L(e_i) = p[i]$ for all $0 \leq i \leq K - 4$. Since $p$ and $q$ differ only by source address, we also know that $L(e_i) = q[i]$ for all $0 \leq i \leq K - 3$. Since $P_p$ is a path from $<(K-4):s>$ to the terminal node for $F(p)$, and $Q_s$ is a path from the root node to $<(K-4):s>$, the path $P = (f_0, f_1, f_2, f_{K-4}, c_{K-3}, \ldots, c_0)$ is a path from the root node to the terminal node for $F(p)$. But this means that $F(p) = F(q)$.

Since this is true for any packets $p$ from $g$ and $q$ from $h$ that differ only by source address, $g \equiv_S h$.

Step 2: Let $g \equiv_S h$. Then there is a node $<(K-4):p>$ such that for every packet $s$ from $g$ and every packet $t$ from $h$ that differ only by source address, node $<(K-4):p>$ lies on both the path $P_s = (e_0, e_1, \ldots, e_K)$ and $P_t = (f_0, f_1, \ldots, f_K)$ where $L(e_i) = s[i]$ and $L(f_i) = t[i]$ for all $0 \leq i \leq K$.

In order to prove that each equivalence class corresponds to a node at level $K - 4$, we prove the more general proposition that at every level $k$, and for every pair of packets $s$ from $g$ and $t$ from $h$ that differ only by source address, the paths $P_s$ and $P_t$ pass through some node $<k:p>$, where $P_s = (e_0, e_1, \ldots, e_K)$, $P_t = (f_0, f_1, \ldots, f_K)$, $L(e_i) = s[i]$ for all $0 \leq i \leq K$ and $L(f_i) = t[i]$ for all $0 \leq i \leq K$.

We proceed by induction $k$.

**Base Case** Let $k = 0$. Then, for all packets $s$ from $g$ and $t$ from $h$ that differ only by source address, the path $P_s = (e_0, e_1, \ldots, e_K)$ such that $L(e_i) = s[i]$ for all $0 \leq i \leq K$ passes through terminal node $<0:s>$ and the path $P_t = (f_0, f_1, \ldots, f_K)$ passes through terminal node $<0:t>$. Since $g \equiv_S h$, we know that $F(s) = F(t)$. Therefore, $<0:s> = <0:t>$, so both paths pass through the same node.

**Induction Hypothesis** Let $k = n \leq K - 4$. Then, for all packets $s$ from $g$ and $t$ from $h$, the path $P_s = (e_0, e_1, \ldots, e_K)$ such that $L(e_i) = s[i]$ for all $0 \leq i \leq K$ passes
through node $<k:s>$, and the path $P_t = (f_0, f_1, \ldots, e_K)$ such that $L(f_i) = t[i]$ for all $0 \leq i \leq K$ passes through the same node, node $<k:p>$.

**Induction Step**  Let $k = n + 1$. Then, for all packets $s$ from $g$ and $t$ from $h$, the path $P_s = (e_0, e_1, \ldots, e_K)$ such that $L(e_i) = s[i]$ for all $0 \leq i \leq K$ passes through node $<(n + 1):s>$, and the path $P_t = (f_0, f_1, \ldots, f_K)$ such that $L(f_i) = t[i]$ for all $0 \leq i \leq K$ passes through node $<(n + 1):t>$. Suppose, in order to obtain a contradiction, that $<(n + 1):s> \neq <(n + 1):t>$. Then, since we do not allow duplicate nodes in a quasi-reduced MDD, there exists some value $j$ such that $<(n + 1):s>[j] \neq <(n + 1):t>[j]$.

Now consider the packet $\hat{s}$ obtained from $s$ by changing attribute $k$ to value $j$ and the packet $\hat{t}$ obtained from $t$ by changing attribute $k$ to value $j$. Since $s$ differs from $t$ only by source address, $\hat{s}$ differs from $\hat{t}$ only by source address.

By the induction hypothesis, we have that the path $P_{\hat{s}} = (a_0, a_1, \ldots, a_K)$ such that $L(a_i) = \hat{s}[i]$ for all $0 \leq i \leq K$ and the path $P_{\hat{t}} = (b_0, b_1, \ldots, b_K)$ for all $0 \leq i \leq K$ both pass through node $<n:p>$. But this contradicts the conclusion that $<(n + 1):s>[j] \neq <(n + 1):t>[j]$, since $<(n + 1):s>[j] = <n:\hat{s}> = <n:p>$ and $<(n + 1):t>[j] = <n:\hat{t}> = <n:p>$.

Therefore, for all packets $s$ from $g$ and $t$ from $h$ that differ only by source address, we have that $P_s = (e_0, e_1, \ldots, e_K)$ and $P_t = (f_0, f_1, \ldots, f_K)$, where $L(e_i) = s[i]$ and $L(f_i) = t[i]$ for all $0 \leq i \leq K$, both pass through some node $<K-4:s>$.

Thus, each class of the relation $\equiv_S$ corresponds to a unique node at level $K - 4$ and each node at level $K - 4$ represents exactly one equivalence class of the $\equiv_S$ relation.

Using the same strategy as in the proof above, we can show that if the top four levels represent the destination address, each class of the relation $\equiv_D$ corresponds to a node at level $K - 4$. This means that we can obtain both the $\equiv_S$ relation and the relation $\equiv_D$ by
enumerating nodes of the quasi-reduced MDD. By combining these results, we can compute all classes of the $\equiv_{SD}$ relation.

5.4 Error Detection

The information provided by the host classification algorithm can be extremely useful for detecting errors in the firewall policy. The list of classes is usually much shorter and simpler than the rule set, so it is easier for a system administrator to examine. Also, since the hosts tend to be categorized according to their intended functionality, the class list reinforces intuition and makes discovery of the error a much more straightforward process for the administrator.

5.4.1 Detecting Remotely Accessible Services

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCEPT</td>
<td>all</td>
<td>Anywhere</td>
<td>192.168.3.0/24</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.2.0/24</td>
<td>Anywhere</td>
</tr>
<tr>
<td>3</td>
<td>DROP</td>
<td>all</td>
<td>168.192.1.0/24</td>
<td>Anywhere</td>
</tr>
<tr>
<td>4</td>
<td>ACCEPT</td>
<td>all</td>
<td>Anywhere</td>
<td>192.168.2.20</td>
</tr>
<tr>
<td>5</td>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.1.0/24</td>
<td>Anywhere</td>
</tr>
</tbody>
</table>

Figure 5.11: Rule set with errors

Simple errors such as typos and rule transpositions can often be detected by the presence of a strange and unexpected class of hosts. The policy in figure 5.11 is intended to protect networks 192.168.1.0/24, 192.168.2.0/24, and 192.168.3.0/24 by restricting access from the outside world. Because 192.168.1.0/24 contains several hosts with important financial information, outgoing traffic from that subnet should also be restricted. Mail traffic from the other subnets is allowed only to the mail server (host 192.168.2.20) to prevent compromised machines from becoming spam relays. The rule set also allows arbitrary web access to a group of web application servers located on the 192.168.3.0/24 subnet. The policy contains several errors, including a typo in rule 3 that allows remote access to a protected service.
The equivalence classes of this example network are listed in figure 5.12. There are five classes of hosts identified by the algorithm. Class 0 represents the group of web servers. Class 1 represents a strange class of hosts that is created by the typo in rule 3. The strange, unexpected class makes the effect of the typo immediately obvious to the administrator. While this may not directly allow him to diagnose and repair the problem, it does provide a significant amount of information about the error. In this case, the user can look for rules which refer to the 168.192.1.0/24 network, rather than pouring over the entire policy to discover the cause of the error.

Class 2 combines the protected financial network and the unprotected 192.168.2.0 network, minus the mail server. This should also arouse the analyst's suspicion since the financial network is supposed to have much stricter protection than the unprotected subnet. The fact that they are treated the same by the firewall indicates that a serious vulnerability exists. Class 3 contains the mail server. It is in a class by itself since it requires special privileges in order to accept and relay mail. Everything else belongs to class 4.

Using the equivalence classes to detect these errors is much easier than using query based tools. The presence of a class of hosts consisting entirely of strange addresses is a clear indication of an error in the policy. Since the tool requires no input but the policy, all the user has to do to discover the error is “fire and forget”.

A small amount of work is required to interpret the results of the classification system, but compared to the effort of constructing precise queries or compiling a list of hosts for
active testing, using the equivalence classes is fairly simple. For large installations, the gain is even greater due to the number of rules required to administer a large number of hosts and the greater difficulty of specifying a comprehensive set of queries that covers all the services provided by the network.

5.4.2 Detecting Shadowed Rules

If a packet matches more than one rule in the policy, the firewall will use the first rule that matches. This can mean that the policy contains useless or unreachable rules. The presence of these rules usually indicates an error in the policy.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCEPT</td>
<td>192.168.2.0/24</td>
<td>Anywhere</td>
<td>TCP dpt:80</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
<td>192.168.2.0/24</td>
<td>192.168.3.0/24</td>
<td>TCP dpt:80</td>
</tr>
<tr>
<td>3</td>
<td>DROP</td>
<td>192.168.2.0/24</td>
<td>192.168.4.0/24</td>
<td>TCP dpt:80</td>
</tr>
</tbody>
</table>

Figure 5.13: Rule set with shadowed rules

When one rule shadows another, the class list will often contain fewer classes than expected. For instance, the rule set in figure 5.13 contains two rules that are shadowed by rule 1. Rule 2 is a useless rule. Web packets from 192.168.2.0/24 to 192.168.3.0/24 are already accepted by rule 1. Rule 3 is also unreachable. The class list for the example network is given in figure 5.14.

| Class 0: 192.168.2.*
| Class 1: [0.0.0.0] - [192.168.1.255] [192.168.3.0] - [255.255.255.255] |

Figure 5.14: Equivalence classes for figure 5.13

Notice that there are no classes for the networks 192.168.3.0/24 and 192.168.4.0/24 mentioned in rules 2 and 3. These networks are included in class 1, the “all other hosts” class. When the system administrator discovers that the policy produces fewer classes than expected, she will examine the policy more closely and find the error or errors. Shadowed
rules often indicate that a rule contains an incorrect address. For instance, one way in which rules 2 and 3 may have become shadowed is if the source address in rule 1 was supposed to be 192.168.3.0/24, but was typed incorrectly.

### 5.4.3 Detecting Outdated Services

| Chain FORWARD (policy DROP): |
|---|---|---|---|---|
| target | prot | source | destination | flags |
| 1 ACCEPT | all | 192.168.2.0/24 | Anywhere | TCP dpt:22 |
| 2 ACCEPT | all | 192.168.4.0/24 | Anywhere | TCP dpt:8080 |
| 3 DROP | all | 192.168.4.0/24 | 192.168.2.0/24 | TCP dpt:25 |

**Figure 5.15:** Rule set with outdated rules

Host classification can solve real world problems. One of our firewalls originally supported a wireless network on subnet 192.168.4.0/24. When wireless service was transferred to another network, we neglected to update the firewall rules. A portion of our rule set looked something like figure 5.15. A quick analysis using host classification immediately identified subnet 192.168.4.0 as a host group, enabling us to correct the problem. This error would have been very difficult to detect using query-based analysis tools. Without apriori knowledge of the error, we had no reason to create a query testing for service on that subnet. Active analysis tools like Nessus would have detected no vulnerabilities, since no hosts were available on that subnet. Using host classification, however, we were able to immediately identify a serious weakness in our policy.

### 5.5 Using Equivalence Classes with Other Tools

While a system administrator can detect many important vulnerabilities simply by studying the host equivalence classes of a firewall policy, even greater gains can be achieved by combining the equivalence class analysis with active and passive testing techniques. To combine the analysis with other testing paradigms, we can use the equivalence classes to determine which groups of systems to test. We can then perform active or passive testing
on a small selection of systems from each class, rather than on each host individually. By taking one or two systems from each equivalence class, rather than testing a random selection of hosts, we decrease the number of tests that we must perform, while increasing the probability that we have tested all the important behaviors of the firewall.

<table>
<thead>
<tr>
<th>Chain FORWARD (policy DROP):</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>prot</td>
</tr>
<tr>
<td>1</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>3</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>4</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>5</td>
<td>DROP</td>
</tr>
</tbody>
</table>

**Figure 5.16**: Rule set for preventing spam relays

The filtering policy in figure 5.16 secures the mail service on an internal network 192.168.2.0/24. Mail from the internal network can only be sent to the mail server, host 10.239.202.13. The mail server is allowed to distribute mail to both internal and external hosts. All other mail traffic should be dropped. Unfortunately, a copy and paste error created rule 3 of the policy, which allows mail traffic from a workstation, host 192.168.2.3 to escape the network. If that workstation is compromised, an intruder can set up a spam relay on that host and transmit thousands of unauthorized messages through the firewall.

| Class 0: | 10.239.202.13 |
| Class 1: | 192.168.2.3 |
| Class 2: | 192.168.2.[0–2] |
| | 192.168.2.[4–255] |
| Class 3: | [0.0.0.0]–[10.239.202.13] |
| | [192.168.3.0–255.255.255.255] |

**Figure 5.17**: Equivalence classes for figure 5.16

The system administrator can easily detect this problem by combining host classification with a passive testing tool. The host classes for the example network are listed in figure 5.17. Class 0 contains the mail server. Class 1 contains the workstation which can circumvent
the security policy. Class 2 contains the remainder of the 192.168.2.0/24 subnet. Class 3 contains all other hosts on the Internet.

By taking a source address from each of these groups and matching it with a destination address from each of the groups, we can construct the sixteen ITVal queries described in figure 5.18. While the increased amount of output makes it slightly more difficult to interpret these results, combining equivalence class analysis with the query analysis does help us find the problem. The query results show that the SMTP port on host 192.168.3.0 can be accessed by host 192.168.2.3. Since 192.168.3.0 is an external host, the system administrator should recognize this as a legitimate security concern.

Using the query tools by themselves would either have produced an enormous amount of data or required a large time investment in writing queries. However, by combining classification with passive testing, we are able to limit the scope of the query to the important distinctions between hosts. This combination also requires very little work by the user.
5.6 Advantages of Policy-based Classification

Policy-based host classification has several significant advantages over existing firewall analysis techniques. Examining the classes implicitly defined by the firewall policy allows a system administrator to detect many kinds of firewall errors and anomalies. When combined with active or passive testing tools, the technique can be even more powerful. Using the equivalence classes significantly decreases the amount of the work required to verify the policy and is a step toward a fully automatic firewall analysis solution. The equivalence classes are also easy to extend. A recently published paper [52] adapts our technique to "packet classification automata", a formalism similar to fully-reduced MDDs. This allows them to produce classes over attributes other than the source and destination address.
Chapter 6

Guided Repair of Firewall Policies

Queries and equivalence class analysis address the problem of detecting errors in a firewall policy. Using this information, the system administrator can examine the firewall rule set and attempt to repair the firewall. To accomplish this, however, the system administrator must not only discover the existence of an error but determine which rules in the policy are incorrect. For policies with hundreds of rules, or policies distributed across multiple firewalls, tracing a problem to its source can be tedious and expensive, even when the existence of an error is obvious. This is a significant burden. Tracing through dozens or perhaps hundreds of correct rules to find the two or three critical inconsistencies can take hours or even days.

To address this issue, we present two novel techniques for performing a “directed repair” of the firewall policy. Using these techniques, a system administrator can trace an error to its root causes without an expensive manual inspection of the rule set.

6.1 Existing Techniques

Both active and passive tools can be used for error detection in a policy, but most of these tools provide only a limited amount of information about each error. For instance, if the system administrator uses passive tools to analyze the query “Which hosts can connect to the mail server?”, the analysis engine will list those hosts that have unwanted access to
the server, but will not provide any additional information that can be used to understand
why the firewall failed to deny them access. It may be that the error only occurs when
connections are made on a particular network interface or for a particular type of network
traffic. While access to this information could greatly assist the system administrator in
repairing the policy, traditional tools do not provide these helpful clues.

The system administrator can sometimes obtain helpful information by refining a query
to provide more information or by using multiple queries to obtain additional data. Unfor­
tunately, the process of developing a sufficiently detailed set of queries requires almost as
much effort as manual repair of the policy. Furthermore, effective refinement of the query
set requires apriori knowledge both of significant threats to the network and potential weak­
nesses in the firewall. If the system administrator does not have enough information to be
able to pose a useful query, he is out of luck.

This means that query tools are usually limited to detecting whether an error exists and
have only limited utility for guiding repair of the policy. To repair the policy by hand, the
system administrator must carefully consider each filtering rule to determine whether it is
relevant to the error and, if so, whether it is correct. Since most of the rules will usually
be irrelevant or already correct, manual repair is a very inefficient and time consuming
process. When an error has many potential causes, debugging the policy can be especially
difficult, since it may be challenging to distinguish the real cause of the problem from other
possibilities.

Active tools also provide only a limited amount of information about an error. For
instance, suppose the system administrator uses a port scanner to detect that SSH traffic
is blocked to a critical host. There are several reasons the port may be blocked: the SSH
daemon may have crashed, the hosts.allow file may contain errors, or the firewall may be
blocking the port. The port scanner does not provide any information that will enable the
administrator to distinguish between these scenarios. When the problem is caused by an
error in the policy, the port scanner can provide no information about which chain of the
policy is responsible for the error. Therefore, the system administrator must investigate all
of these potential problems manually. This process is almost as tedious and error prone as manual inspection of the firewall policy.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>input interface</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP</td>
<td>all</td>
<td>anywhere</td>
<td>192.168.1.0/24</td>
<td>eth1</td>
<td></td>
</tr>
<tr>
<td>DROP</td>
<td>all</td>
<td>192.168.3.0/22</td>
<td>anywhere</td>
<td>any</td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>all</td>
<td>anywhere</td>
<td>192.168.1.4</td>
<td>any</td>
<td>dpt:tcp 80</td>
</tr>
<tr>
<td>DROP</td>
<td>all</td>
<td>anywhere</td>
<td>192.168.1.10/24</td>
<td>any</td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>all</td>
<td>192.168.1.0/24</td>
<td>anywhere</td>
<td>any</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: An example rule set

Figure 6.1 shows how difficult it can be to trace an error in the policy to its source. This rule set protects subnet 192.168.1.0/24 against attacks from the outside world. The system administrator wants to allow access to the web server, host 192.168.1.4, from any system in the outside world except those on an unsecured wireless network 192.168.3.0/24.

An attempt to access the web server from a host on the 192.168.1.0/24 subnet will very quickly demonstrate that the rule set given in the figure fails to enforce the desired policy. Instead, hosts on the trusted subnet are prevented from accessing the web server.

Determining the cause of the error is far more difficult. Almost any rule of the policy could be at fault. An error in rule 4, which drops traffic to the protected subnet, could be the source of the error. An error in rule 3, which overrides rule 4 to allow web traffic to enter the network could also cause the problem. Rule 1, an anti-spoofing rule which blocks traffic from the "wrong" interface, could also be to blame.

As it turns out, none of these rules causes the error. The error is created by an incorrect subnet mask in rule 2, which causes the firewall to block traffic from the protected network as well as the untrusted net. A manual analysis of the policy would require a careful and tedious inspection of every rule in the policy to identify this problem. While this process might not take long for the five rule policy shown here, a policy with more than a few dozen rules would be much more difficult to inspect. Partially automating the repair process in a way that narrows down the potential sources of the error to just one or two rules could
save the administrator a significant amount of effort.

6.2 Partially Automated Firewall Repair

Unfortunately, it is impossible to fully automate repair of a generic firewall policy because incorrect behavior on one network may be expected behavior on another. For instance, on one network it may be desirable to allow SMTP traffic to reach certain hosts, such as the mail servers. On another network, however, a policy that permits SMTP traffic may enable infected machines to send spam to systems outside the network. Without input from the user, a repair algorithm cannot distinguish between these two cases.

While a fully automatic strategy for firewall repair is impossible, partial automation is possible. Gouda, Liu, et al. [22] have done significant work on repair of structural errors in the firewall policy. Their technique uses transformation of decision diagrams to produce an improved rule set in which problems such as shadowed or duplicate rules have been eliminated without any input from the user. Unfortunately, these techniques do not address repair of logical errors such as typos or out-of-order rules.

Another approach is to allow the user to make the final decision about how to repair the policy, but automate the process of determining the root causes of the error. By providing information about the possible causes of the problem, we can guide her toward a limited set of solutions from which she can choose the one best suited to her network and policy goals. This “directed repair” of the policy alleviates much of the tedious work required to fix the policy.

6.3 Directed Repair

In previous chapters, we explored ways to detect errors in a firewall configuration using logical queries and an equivalence class decomposition of the network. In this chapter, we describe two novel techniques that enable directed repair of the firewall policy. One technique generates relevant counterexamples from which the system administrator can
obtain detailed information about security failures in the policy. The second technique provides an extensive "history analysis" that identifies potential sources of the error and lists rules which should be considered for modification.

To use these techniques, the user specifies the desired behavior of the firewall using logical assertions. The syntax for assertions is derived from the query language explained in previous chapters. The right and left conditions of the assertion are built from the same primitives as those in chapter 3.

For example, we can match all accepted SSH packets from subnet 192.168.1.0/24 on interface eth0 with the condition "FOR TCP 22 AND FROM 192.168.1.* AND INFACE eth0 AND (ACCEPTED forward OR ACCEPTED input)".

Using these conditions, the user can construct two types of assertions to describe the expected behavior of the packet filter. These assertions allow the user to describe important high-level security invariants which the policy should always satisfy. Equality assertions have the form: ASSERT \(<A> IS <B>\) where \(A\) and \(B\) are conditions. Containment assertions have the form ASSERT \(<A> SUBSET OF <B>\).

Equality assertions specify that those packets which match condition \(A\) are exactly those that match condition \(B\). Containment assertions specify that the match set of \(A\) is (non-strictly) contained in match set \(B\).

For instance, the containment assertion "ASSERT FROM 192.168.2.* SUBSET OF DROPPED FORWARD;" specifies that any packet from subnet 192.168.2.0/24 is dropped. The equality assertion "ASSERT FROM 192.168.1.* IS (FOR TCP 80 AND ACCEPTED forward)" can be used to check that only HTTP packets are allowed to enter the network from the 192.168.1.0/24 subnet and that no other web connections are allowed by the firewall.

We call the set of packets that match a condition its "match set" and the set of packets that cause an assertion to fail the assertion's "fail set". We can easily represent each match set as an MDD using the technique given in chapter 3 for creating an MDD representation of a query condition.
We can then combine the match set MDDs together to generate an MDD representation of the fail set. By examining the fail set MDD, we can determine whether or not the assertion evaluates to true or false.

```c
bool testContainmentAssertion(condition A, condition B):
1 mddA = condition_to_MDD(A)
2 mddB = condition_to_MDD(B)
3 notB = MDD_complement(mddB)
4 resultMDD = MDD_intersect(mddA, notB)
5 if notEmpty(resultMDD) then:
6 return ASSERTION_FAILED
7 else:
8 return ASSERTION_HELD
```

Figure 6.2: Checking a containment assertion

Figure 6.2 gives pseudocode for determining whether a containment assertion holds. Lines 1 and 2 of the algorithm generate MDD representations of each condition in the assertion. Line 3 uses an MDD complement operation in to find the set of packets which do not match condition \( B \), the right-hand side of the assertion. Line 4 intersects the MDD returned by the complement operation with the MDD representing condition \( A \), the left-hand side of the assertion, to find the fail set of the assertion. If the assertion fails, this resulting set will be non-empty, as illustrated by the left hand side of figure 6.3. If the assertion holds, we obtain the situation on the right hand side, in which the fail set is empty.

Figure 6.3: Fail sets for the SUBSET OF operator

Pseudocode for testing an equality assertion is given in figure 6.4. As in the algorithm for
evaluating a containment assertion, we use MDD complement and intersection operations, as above, to find the set of packets which match condition $A$, but not condition $B$, as above. However, in lines 3 and 6, we repeat the process, switching $A$ and $B$, to find the set of packets which match condition $B$ but not condition $A$. Line 7 combines the resulting MDDs together using the MDD union operation to obtain the fail set for the assertion.

```python
bool TestISAssertion(condition A, condition B):
1 mddA = condition_to_MDD(A)
2 mddB = condition_to_MDD(B)
3 notA = MDD_complement(mddA)
4 notB = MDD_complement(mddB)
5 resultA = MDD_intersect(mddA, notB)
6 resultB = MDD_intersect(notA, mddB)
7 resultMDD = MDD_union(resultA, resultB)
8 if notEmpty(resultMDD) then:
9    return ASSERTION_FAILED
10 else:
11    return ASSERTION_HELD
```

Figure 6.4: Checking an equality assertion

If the fail set is non-empty, we have the situation illustrated by the left hand side of figure 6.5 and the assertion fails. If the fail set is empty, we have the situation given on the right hand side and the assertion holds true.

![Assertion fails. Assertion holds.](image)

Figure 6.5: Fail sets for the IS operator

Using these techniques, we can determine whether or not a firewall policy satisfies a set of assertions. We can also extend these techniques to provide detailed information about the firewall policy and enable directed repair.
6.4 Relevant Counterexamples

One useful extension of our MDD techniques is the generation of relevant counterexamples that illustrate the failure of an assertion. These counterexamples provide a context for the error which can often help the administrator discover why a failure has occurred.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>input interface</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCEPT</td>
<td>all</td>
<td>anywhere</td>
<td>192.168.1.0/24</td>
<td>eth0</td>
</tr>
<tr>
<td>2</td>
<td>ACCEPT</td>
<td>all</td>
<td>anywhere</td>
<td>131.106.3.253</td>
<td>eth1</td>
</tr>
<tr>
<td>3</td>
<td>DROP</td>
<td>all</td>
<td>63.118.7.16</td>
<td>anywhere</td>
<td>eth0</td>
</tr>
<tr>
<td>4</td>
<td>DROP</td>
<td>all</td>
<td>192.168.2.0/24</td>
<td>anywhere</td>
<td>any</td>
</tr>
<tr>
<td>5</td>
<td>ACCEPT</td>
<td>all</td>
<td>anywhere</td>
<td>anywhere</td>
<td>any</td>
</tr>
</tbody>
</table>

**Figure 6.6**: An incorrect forwarding chain

The example policy in figure 6.6 isolates an untrusted research network 192.168.2.0/24 from the outside world. SSH traffic from the untrusted network to hosts on subnet 192.168.1.0/24 is accepted, but all other traffic from that network is denied. The 192.168.1.0/24 subnet contains several world-accessible web servers to which the policy grants access. The rule set blocks connections from 63.118.7.16, a malicious host. Trusted hosts are allowed to make connections to the web servers and an external server, host 131.106.3.253, but cannot make any other connections.

To test whether the untrusted hosts are sufficiently restricted by the firewall, the administrator uses the assertion “ASSERT (FROM 192.168.2.* AND NOT FOR TCP 22) SUBSET OF DROPPED FORWARD” which specifies that only SSH traffic is accepted from hosts on the untrusted network. Due to an error in the ordering of rules 2 and 4, the assertion will fail. This subtle error could be very difficult to detect manually in a lengthier policy in which the rules were much further apart. Using ITVal, however, the administrator can easily discover that the assertion fails.

Knowing that the assertion does not hold is an important first step, but does not give much information about the cause of the error. To give the user more information about the source of the error, we generate a counterexample — a packet that demonstrates the
falsity of the assertion. Figure 6.7 shows the generation of one possible counterexample.

Examination of the counterexample gives the system administrator important information about the assertion failure. One significant clue is that the example packet arrived on interface eth1. Since only rule 2 mentions eth1, this fact draws the administrator's immediate attention to rule ordering error. The destination address is also a helpful clue, since only rule 2 allows traffic specifically to host 131.106.3.253. Using either of these clues, the user can now correct the error by moving rule 2 to the correct location in the policy.

```
packet testContainmentAssertion(condition A, condition B):
1 mddA = condition_to_MDD(A).
2 mddB = condition_to_MDD(B).
3 notB = MDD_complement(mddB).
4 resultMDD = MDD_intersect(mddA, notB).
5 if notEmpty(resultMDD) then:
6     return choose_element(resultMDD).
7 else:
8     return choose_element(mddA).
```

Figure 6.8: Generating an example for a subset assertion

To generate the counterexample for an assertion, we change the algorithms in figure 6.2 and figure 6.4 to return an arbitrary element from the fail set by replacing the last four lines of each algorithm as shown in figure 6.8.

The function choose_element(X) picks an arbitrary element from the set represented by MDD X. If the assertion fails, we choose an element from the fail set as the counterexample. If the assertion succeeds, we choose an element from the match set of the left-hand condition as a witness, since the elements of that set must match both conditions. To select an element,
the choose_element function walks the MDD from the root node to the bottom of the graph, arbitrarily selecting arcs at each level (in practice, we select the first non-zero arc of each node) and storing each selected attribute in a "packet" structure which can be printed at the end of the traversal.

6.5 Rule History

Counterexamples provide the system administrator with a great deal of information about the causes of an assertion failure. Nevertheless, tracing an error to the rules that cause the problem can be difficult even when a good counterexample is available. Fortunately, we can extend the example generation technique to provide the user with even more useful information about the potential causes of the firewall error. We do this by using MDDs to create a "history map" that allows us to remember which packets match each rule of the policy.

Using the history map, we associate packets in an assertion's fail set with a small number of filtering rules, which the administrator should examine for errors. This permits the administrator to narrow his inspection of the policy to just a few critical areas. Since the set of rules to examine includes every rule that may match a packet in the assertion's critical set, it is possible that we may list some correct rules as well as the incorrect ones. However, constructing the history map allows the system administrator to ignore many rules that are completely unrelated to the problem.

Given a packet $p$, we define the history set of $p$, $H(p)$, to be the set of rules in the firewall that match $p$. Formally, we say $H(p) = \{r | p \in M(r)\}$. Given a set of packets, $P$, the history set of $P$ is given by $H(P) = \bigcup_{p \in P} H(p)$, that is, the history set of $P$ is the union of the history sets of each member of $p$. Another way to say this is that the history set contains all the rules that match any packet of $P$. 
6.6 Implementing Rule History

In order to build the history map, assign a unique identifier to each of the firewall policies provided by the user. For each firewall, we also assign a unique identifier to each chain of the policy and give each rule of the chain an integer index.

During construction of the MDD for each chain of the firewall, we construct a "history MDD" representing the rule set of the policy. The history MDDs are constructed similarly to the rule set MDDs, but have three extra levels at the bottom of the graph. The extra levels store the firewall identifier, chain identifier, and index for each rule. We assign index 0 to the default policy and index the remaining rules in each chain sequentially starting from 1. Pseudocode for constructing a history MDD for a rule is given in figure 6.9.

```
node_index MakeHistMDD(ParsedRule pr, int rule_index, int chain_index, int fw_index)
1  old = MATCHES.
2  node n = NewNode(1).
3  <1:n>[rule_index] = old.
4  old = CheckForDuplicates(n).
5  n = NewNode(2).
6  <2:n>[chain_index] = old.
7  old = CheckForDuplicates(n).
8  n = NewNode(3).
9  <3:n>[fw_index] = old.
10 old = CheckForDuplicates(n).
11 for k = 4 to K + 3:
12  n = NewNode(k).
13  for i = 0 to MaxValue(k - 3):
14     if i ≥ pr[k - 3].low and i ≤ pr[k - 3].high:
15        <k:n>[i] = old.
16    old = CheckForDuplicates(n).
17  return n.
```

**Figure 6.9:** Algorithm for building a history MDD for a rule

Lines 1 through 10 of the algorithm create nodes to represent the rule index, chain index, and firewall index. The for loop in lines 11 through 16 creates one new node at each level of the MDD which identifies which packets match the rule.

We construct the history MDD for a chain using an MDD union operation to combine
the history MDDs of each rule in the chain. The history MDD for each rule is created on-the-fly during generation of the rule set MDD for a chain. If we encounter a rule which matches packets already matched by some other rule, the union operation ensures that the history MDD for the chain maps those packets to both rules.

<table>
<thead>
<tr>
<th>Chain FORWARD (policy DROP):</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1 Accept</td>
</tr>
<tr>
<td>2 Drop</td>
</tr>
<tr>
<td>3 Accept</td>
</tr>
</tbody>
</table>

Figure 6.10: Example Rule Set

An example history MDD for the rule set of figure 6.10 is given in figure 6.11. To save space, only the levels for source address, destination address, protocol, destination port, firewall index, chain index, and rule index are represented in the figure. The source address and destination address are each represented by four levels of the MDD. These levels correspond to each octet of the address. The destination port is represented using two levels. The top level corresponds to the most-significant byte of the port and the level below it corresponds to the least-significant byte. By following the right-most path of the MDD, you can see that a packet from 192.168.2.4 to 192.168.4.9 on TCP port 25 will match rules 0, 1, 2, and 3 of chain 2 (in this example chain 2 is the FORWARD chain of the firewall).

In addition to constructing history MDDs for the built-in chains of the firewall, we create history MDDs for each assertion by simply appending "wildcard nodes" to the bottom of the MDDs for the fail set. These wildcard nodes match every rule in every chain.

By intersecting the fail set history MDD with the history MDD for a rule set, we obtain a MDD which represents a mapping from the critical packets that cause an assertion to fail to the rules that they match. This history map can be used to generate a list of rules which the system administrator should consider as important to the failure of the assertion.

An example MDD for the assertion "NOT FROM 192.168.2 AND TO 192.168.4.* AND
Figure 6.11: History MDD for a firewall chain

FOR TCP 25 SUBSET OF FORWARD ACCEPTED" is given on the left side of figure 6.12. When applied to the rule set given in the example, the assertion fails. The source of this error is non-obvious. A typo in the subnet mask of rule 2 causes the firewall to drop packets from the 192.168.4.0/24 subnet. Intersecting the extended fail set MDD for this assertion with the rule set MDD gives us the history map MDD on the right side of figure 6.12.

An examination of the result MDD shows that rules 2 and 3 match the packets which cause the assertion to fail (the default policy, rule 0 also matches, but since that is always true, it is not very significant). Therefore, the system administrator can ignore rule 1, which
is not relevant to the failure of the policy. Instead, he will focus on rule 2, which contains the error, and rule 3, the rule which should have accepted the incorrectly discarded packet.

6.7 Correctness of the History MDD representation

To show that the MDD representation of the rule history accurately lists those rules which match a packet, we formally define the term "history set of an MDD" and then prove that the history set of an MDD for a rule is equivalent to the history set of the rule.

We use the notation $r_{rid}, r_{cid}$ and $r_{fid}$ to denote the rule index, chain index, and firewall index of rule $r$ in the MDD.

**Definition 9** Given an MDD $M_r$ and a packet $p$, we define the history set of $M_r$ on $p$, $H(M_r,p)$, to be the set of rules $r$ in the firewall such that there exists a path $P = (e_0, e_1, \ldots, e_{K+3})$ from the root of $M_r$ to the terminal node $MATCHES$ such that $L(e_0) = r_{rid}$, $L(e_1) = r_{cid}$, $L(e_2) = r_{fid}$, and for each edge $j > 2$, $L(e_j) = p[j-3]$.

We can now show the equivalence of $H(p)$ and $H(M_r,p)$ for a given rule $r$.

**Theorem 6.1** For each rule $r$ such that $r \in H(p)$, $r \in H(M_r,p)$. Similarly, for each rule $r$ such that $r \in H(M_r,p)$, $r \in H(p)$.

**Proof:**

Step 1: Given an arbitrary packet $p$, let $r \in H(p)$ be any rule of the firewall policy, with rule index $rid$, chain index $cid$, and firewall index $fid$, and let

$$M_r = MakeHistMDD(r, rid, cid, fid).$$

Then, $r \in H(M_r,p)$.

From the definition of $H(M_r,p)$, we see that it is sufficient to show that there is a path $P = (e_0, e_1, e_2, \ldots, e_k)$ from the root node of $M_r$ to the terminal node $MATCHES$
such that \( L(e_0) = rid, L(e_1) = cid, L(e_2) = fid \), and for each edge \( e_j \) such that \( 2 < j \leq K + 3 \), \( L(e_j) = p[j-3] \).

We proceed by induction on \( K \), the number of attributes in a packet.

**Base Case** Let \( K = 0 \). To see that \( \text{MakeHistMdd} \) creates a path \((e_0, e_1, e_2)\) from node \(<3:n>\) to \( MATCHES \) such that \( L(e_0) = rid, L(e_1) = cid, \) and \( L(e_2) = fid \), consider that line 3 creates an arc from node \(<1:n>\) to \( MATCHES \) with label \( rid \). Similarly, line 6 creates an arc from node \(<2:n>\) to \(<1:n>\) with label \( cid \) and line 9 creates an arc with label \( fid \) from \(<3:n>\) to \(<2:n>\). Therefore, when \( K = 0 \), there is a path from \(<3:n>\), the root node of \( M_r \), to \( MATCHES \) such that which has the appropriate labels on each edge.

**Induction Hypothesis** Assume that for all \( 0 < k < K \), that is a path \( Q = (e_0, e_1, \ldots, e_{k+3}) \) in \( M_r \) from node \(<k+3:n>\) to \( MATCHES \) such that \( L(e_0) = rid, L(e_1) = cid, \) and \( L(e_2) = fid \), and \( L(e_j) = p[j-3] \) for every edge \( e_j \) such that \( 2 < j \leq k+3 \).

**Induction Step** Let \( k = K \). Consider node \(<K+3:n>\) created at line 12 in the last iteration of the for loop in line 11. To see that there is a path from this node to \( MATCHES \) with appropriate labels, note when the inner for loop of line 13 reaches value \( p[K] \), the if statement in line 14 will evaluate to true, since \( p \in M(r) \). Therefore, line 15 will create an arc to node \( old = <((K + 3) - 1):n> \) with label \( p[K] \). Call this arc \( e_{new} \).

By the induction hypothesis, there is a path

\[
Q = (e_0, e_1, \ldots, e_{k+3-1})
\]

from \(<((k + 3) - 1):n>\) to \( MATCHES \) such that \( L(e_0) = rid, L(e_1) = cid, \) \( L(e_2) = fid \), and \( L(e_j) = p[j-3] \) for every edge \( e_j \) such that \( 2 < j \leq (k + 3) - 1 \).
Therefore, when \( k = K \), there is a path

\[
P = (e_0, e_1, \ldots, e_{(k+3)-1}, e_{\text{new}})
\]

from root node \( <K+3:n> \) to \( MATCHES \) such that \( L(e_0) = rid \), \( L(e_1) = cid \), \( L(e_2) = fid \), and \( L(e_j) = p[j-3] \) for each edge \( e_j \) such that \( 2 < j < K+3 \).

Thus, by the definition of \( H(M_r,p), r \in H(M_r,p) \).

Step 2: Let \( M_r = \text{MakeHistMDD}(r, rid, cid, fid) \) for some values \( rid, cid, fid \), and some rule \( r \). Let \( p \) be a packet such that \( r \in H(M_r,p) \). Then, \( r \in H(p) \).

Since \( r \in H(M_r,p) \), there is a path \( P = (e_0, e_1, \ldots, e_{K+3}) \) from the root of \( M_r \) to \( MATCHES \) such that \( L(e_0) = rid \), \( L(e_1) = cid \), \( L(e_2) = fid \), and \( L(e_j) = p[j-3] \) for \( 2 < j \leq K+3 \).

To show that \( p \in M(r) \), we must show that \( pr\text{.low} \leq p[k] \leq pr\text{.high} \) for each value \( 0 \leq k \leq K \). Consider that the only way \( \text{MakeHistMDD} \) can create an arc at levels 3 through \( K+3 \) is in line 15. But the if statement in line 14 guarantees that this can only happen when \( pr\text{.low} \leq p[k] \leq pr\text{.high} \). Since we know that there is a path from the root of \( M_r \) to \( MATCHES \), we know that \( pr\text{.low} \leq p[k] \leq pr\text{.high} \) for all \( 0 \leq k \leq K \). Therefore, \( p \in M(r) \).

This demonstrates that given a rule \( r \), the history MDD, \( M_r \), is a faithful representation of the history mapping for that rule. From this and from the proof of correctness of the MDD union and intersection operations given in chapters 2 and 3, it is trivial to show that the history MDD for a chain is correct and that the intersection of the fail set history MDD with the history MDD for a chain produces an MDD which maps packets in the fail set to all of the rules which they match.
6.8 Directed Repair and Equivalence Classes

It is often much easier to use assertions than to perform a manual inspection of the policy. For one thing, the rules in a policy interact with each other in ways that can be confusing to the user. One rule in the policy might mask another rule or cause the rule to be applied only in certain, unusual, circumstances.

Because assertions are independent of each other, writing and understanding a list of assertions is often easier than manually correcting the rule set. More importantly, it is possible to construct a partial or high-level specification of the policy using assertions. This partial specification can ignore many of the details of the policy, which allows it to be simpler than the rule set to which it is applied.

Nevertheless, debugging the firewall using assertions has certain limitations. There is a tradeoff between the completeness of a specification and how easy the specification can be constructed. Deriving assertions that are both useful and effective can be a very challenging task.

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>source</th>
<th>destination</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP</td>
<td>all</td>
<td>192.168.1.0/24</td>
<td>anywhere</td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>all</td>
<td>anywhere</td>
<td>192.186.2.0/24</td>
<td>tcp dpt:22</td>
</tr>
</tbody>
</table>

Figure 6.13: A fault that history mapping misses

Another limitation of the assertion approach is that certain kinds of faults cannot easily be identified using history maps for an assertion. The policy in figure 6.13 is supposed to protect a secure subnet 192.168.2.0/24 from intrusions on an untrusted network 192.168.1.0/24.

```
ASSERT HISTORY TO 192.168.2.* AND
FOR TCP 22 AND
NOT FROM 192.168.1.*
SUBSET OF ACCEPTED forward;
```

Figure 6.14: Combining assertions with history

An assertion checking that legitimate SSH traffic can reach the protected network is
also given in figure 6.14. A typo in rule 2 causes the assertion to fail. Unfortunately, the history map for the assertion will show only the default policy. None of the other rules in the policy match any packets in the fail set. In particular, rule 2, which contains the fault, does not match any packets from the 192.168.2.0/24 subnet and, therefore, is not listed.

One way to address this problem is to create a new assertion that checks whether packets from 192.186.2.0/24 are accepted. The history map for such an assertion would immediately identify the typo in rule 2. The problem with this is that the system administrator has no way of knowing such an assertion is needed. It is not practical to create assertions for all of the possible typos in a policy, since doing so would require at least as much work as manual inspection of the policy.

A better way to address the problem is to extend the technique described in the previous chapter to provide history information that can be used to discover faults in the policy. Figure 6.15 lists three classes derived from the assertion in figure 6.13. Class 2 corresponds to the untrusted subnet 192.168.1.0/24. Class 3 is an anomalous class of hosts caused by the typo in rule 2. The existence of this class is an immediate clue to the system administrator that the firewall policy contains a serious error. Class 1 corresponds to all other hosts on the network.

<table>
<thead>
<tr>
<th>QUERY HISTORY CLASSES;</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are 3 total host classes:</td>
</tr>
<tr>
<td>Class 1:</td>
</tr>
<tr>
<td>Class 2:</td>
</tr>
<tr>
<td>Class 3:</td>
</tr>
</tbody>
</table>

Figure 6.15: Equivalence class decomposition of a policy

As described in the previous chapter, partitioning the hosts on a network into equivalence classes allows us to generate a "policy map" that shows functional groupings of the hosts on a network. When the policy contains a fault, it will often be manifested in the policy map
as a missing class or by the presence of an unexpected class of hosts. Unfortunately, while the policy map assists the system administrator in detecting these problems, it provides him with little information that can be used to identify the rules that must be changed to repair the issue.

![History MDD for class three](image)

**Figure 6.16:** History MDD for class three

We can enhance the policy map by annotating each class of hosts with a list of rules that match packets to and from a host in the class. To do this, we extend each class MDD with wildcard nodes. The resulting graph is similar in structure to the history MDDs used to represent the fail set of an assertion, but has wildcards at every level except the source address levels. This MDD matches the set of all packets whose source address matches a host in the class. We then repeat the procedure to produce an MDD with wildcards everywhere except the destination address levels. We can now intersect these class MDDs with the history MDDs for each chain to determine which rules match these packets. This intersection generates a result MDD which can be translated into a human-readable history map.

An MDD representing all packets with source address from class 3 is given in figure 6.16. The top four levels of the MDD correspond to source addresses on subnet 192.186.2.0/24. The remaining levels contain wildcard nodes.

A portion of the history map for the equivalence classes of the policy in figure 6.13 is given in figure 6.17. The existence of an anomalous class containing hosts from the
Class 3:  
Firewall 0 Chain 1 Default Policy.

Firewall 0 Chain 1 Rule 2:  
ACCEPT all -- * * 0.0.0.0/0 192.186.2.0/24  
tcp dpt:22

Figure 6.17: History Map for class three

192.186.2.0/24 subnet immediately alerts the system administrator to a serious error. A quick glance at the history map for class 3 reveals that only two rules are of interest: the default policy and rule 2. The system administrator now takes a careful look at rule 2 and discovers the fault, which enables her to repair the policy.
Chapter 7

Conclusion and Future Work

In this dissertation, we considered the problem of constructing a formal model of a firewall policy using multi-way decision diagrams. We presented several techniques for creating and analyzing such a model. The most significant theoretical contributions of this dissertation are a quasi-reduced multi-way decision diagram representation of a firewall policy and an algorithm for deriving classes of equivalent hosts from the MDD representation. We applied these techniques to the development of tools for analyzing, testing, and repairing a firewall policy.

We also demonstrated that network address translation can be incorporated into the MDD model by application of a special MDD operator. We show that the MDD representations of connected firewalls can be combined to allow for analysis of distributed firewalls.

We extended this work to provide counter-examples and rule history, which enable a directed repair of a firewall policy. These tools enable a system administrator to trace an error in the policy to the particular faults that cause those errors.

We applied these techniques to the development of ITVal, a tool for testing and repairing iptables firewalls. This analysis can be performed on policies that use advanced features such as state-based filtering and packet-mangling. The equivalence class techniques all an ITVal user to perform an analysis without generating a large and complicated set of test cases or queries. This means that, using ITVal, a Linux firewall administrator can quickly and easily discover errors in a firewall policy without a tedious manual inspection of the
rule set.

There are several interesting areas in which both the theory and application of this work might be extended. One possible extension of this work is support for proxy firewalls. The work described in this thesis addresses packet filtering firewalls in which filtering is performed at the data and transport layers. Proxy firewalls operate at higher levels of the protocol stack and provide a system administrator with a lot more power in deciding what traffic should be filtered. Because the MDD model described in this paper provides a fixed number of levels, it is not suitable for representing application-level data in which there can be a varying number of fields (keywords of various sizes, for instance). A new data structure, based on the filtering MDD, but which allows for a flexible number of levels, could address these needs. Extending these techniques to application layer information would also require the development of new types of queries and new operations on the MDD.

The composition operator which enables analysis of connected firewalls currently only supports firewalls connected in series. By using the algorithm repeatedly with different inputs, it is possible to perform an analysis on more general topologies, but this is awkward and inefficient. Extending the to more general topologies would require some innovations in the design of the composition operation, but would significantly improve the usability of the algorithm.

Dynamic firewalls, which adapt to changes in network conditions by modifying the policy, are becoming increasingly popular. The query and assertion language we use to analyze the firewall provides MDD operations for basic logic manipulations. This is suitable for answering basic questions about the behavior of a static firewall. Expanding the language and the model to allow temporal logic queries would enable our model to analyze dynamic firewalls.

There are several areas in which the equivalence class generation techniques might be extended. For instance, packet classifiers, which organize packets into related streams for traffic shaping define an equivalence relation which is very similar to the host-equivalence relation for packet filtering. It might be possible to adapt our MDD algorithms to provide
correctness and performance analysis of packet classifiers.

It might also be interesting to explore the use of the equivalence class technique with other analysis strategies. For instance, while active testing tools have many disadvantages, they can be used to solve problems that passive testing tools cannot, such as verifying that the firewall software (rather than just the policy of the firewall) is correct. We have already investigated the combination of equivalence class generation with passive testing tools, but have not yet developed techniques for using the equivalence classes with active testing tools.

An interesting hybrid between active and passive tools is Russell's netfilter simulator [44]. The simulator is intended to be used for debugging kernel hooks in netfilter, and provides very low-level access to the internals of netfilter, so it is not by itself suitable as a query tool for non-developers, but could perhaps be used as the basis of a more general query library. Integrating the simulator with the equivalence class technique could produce a hybrid testing technique that combines elements of active analysis with passive analysis.

Another area in which the work might be extended is to further develop the possibilities of the guided-repair techniques. For instance, it might be possible to develop, from an assertion's critical set, a list of candidate solutions for repairing the firewall. The user could then select an appropriate remedy from these choices, which could be immediately applied to the firewall. Providing a reasonably short list of candidate solutions would require some means of winnowing down the extremely large number of permutations of rules to a few likely solutions. This might be done by combining the equivalence class analysis with the counter-example generation technique.

There are also many ways in which the development of ITVal can be extended. One area in which the tool could be significantly improved is visualization of query results. Queries often generate a significant amount of unstructured output which can be difficult for a human reader to parse. The output of the equivalence class queries is usually much more structured and easier to read. Developing a technique which uses the equivalence classes to add structure to the output of other queries could lead to significant enhancements in the usability of the tool. Another possible solution to this problem is the development of a
graphical interface.

Currently, ITVal supports basic data/link layer filtering and network address translation. There are many other features of iptables that are potentially amenable to analysis. For instance, it is possible to use the netfilter framework to create rules that match against the MAC address, or to modify the TTL of a packet. Extending ITVal to accommodate more of the features of the firewall is another potential direction for future work.

Another way in which ITVal could be enhanced is by extending it to new types of firewalls on a more diverse set of platforms. Although the tool currently supports only iptables firewalls, a user can perform analysis on ipfwm and ipchains firewalls using scripts [49] which translate between the various formats. Supporting other firewalls, such as Checkpoint’s FW-1 firewall, or BSD’s “pf” firewall, would open the tool up to a wider user base.

One way to do that would be to convert firewall policies into a common intermediate representation that could be easily parsed by ITVal and other firewall tools. We are currently developing an application which will convert iptables firewalls to an XML-based format used by the Redseal Security Risk Manager [40] and it should be possible to create a parser for their XML format which would allow ITVal to support all of the firewalls supported by their tool, including Cisco PIX firewalls. An alternative solution would be support for the MDL language used by the Firmato [5] framework.
Appendix A

Query Selection for Effective Analysis

Designing effective queries for a passive testing tool can be challenging and error-prone. Queries that are too narrow can miss important anomalies in the firewall policy. Queries that are too broad can produce too much information to be usefully analyzed.

In this appendix, we present some solutions for constructing a broad range of firewall policy tests. These paradigms are drawn from well known firewall design practices. We implement several of these tests as a set of generic query files that can be easily adapted to the needs of a particular network or system.

A.1 Using ITVal

In this work, we will concentrate on how to use ITVal effectively, rather than on its implementation and design. With a little work, it should be possible to use our examples with other tools, such as active testing utilities.
A.2 Constructing Queries

There is a fairly sizable body of work available on configuring a packet filter to avoid various kinds of vulnerabilities. Rather than simply repeat what is readily available, we will focus on how to use these sources of information to derive tests for the firewall policy.

In order to illustrate our technique, we use the example network shown in figure A.1. The example network has two mail servers and a large number of client machines. The firewall acts as a perimeter defense separating servers and clients on the 128.40.10.0/24 subnet from the outside world.

![Figure A.1: A simple perimeter firewall](image)

Suppose that we know that the packet filter should block packets of type $X$ given a certain condition $Y$. There are several ways to construct a query that tests whether the packet filter meets this criteria. One possibility is to produce a list of hosts or services that satisfy the criteria. A better way is to produce a list of all hosts that fail to meet the criteria.

To see this, consider figure A.2. In order to prevent client machines from becoming spam relays, it is usually advisable to prevent outgoing SMTP traffic from any host but the mail server. This can be stated as: "The packet filter should always block packets of type
SMTP given the condition that the source address of those packets is not a mail server.\footnote{14}{1} Figure A.2 gives two possible queries for checking this property.

\begin{verbatim}
QUERY DPORT FROM 128.140.10.* AND
   NOT FROM 128.140.10.2 AND
   NOT FROM 128.140.10.3 AND ACCEPTED forward;

QUERY SADDY FROM 128.140.10.* AND
   FOR TCP 25 AND ACCEPTED forward;
\end{verbatim}

**Figure A.2:** Queries for finding rogue mail servers

Both queries can be used to check whether hosts other than the mail server are allowed to send SMTP traffic on the network describe in figure A.1. The first query produces a list of allowed destination ports from hosts other than the mail server. If the SMTP port does not appear in this list, the firewall is correctly configured. The second query produces a list of hosts that are allowed to send SMTP traffic. If any host but the mail server appears in the output, the firewall is incorrectly configured. While either query could correctly determine whether a client machine is allowed to send mail, the second query generates far less output and identifies which hosts the firewall incorrectly allows to send mail traffic.

### A.3 Avoiding pitfalls

Although creating queries for testing best practice is usually straightforward, there are a few pitfalls to avoid. Consider for instance, the simple ITVa1 query file shown in figure A.3. This query set is a more general version of the queries shown above that test whether the example firewall contains adequate protections against internal and external spam relays. Line 1 defines a named group of hosts that represents the mail servers of the network. Line 2 defines a named group of mail related services. The query in line 3 lists all services provided by the mail server other than mail service. Line 4 lists all hosts, other than the mail servers themselves, that are allowed to transmit mail related traffic that is not destined for one of the mail servers.
GROUP mailservers 128.140.10.2 128.140.10.3;
SERVICE mail TCP 25 TCP 110 TCP 993;

QUERY DPORT TO mailservers AND
   NOT FOR mail AND ACCEPTED forward;

QUERY SADDY FOR mail AND
   NOT FROM mailservers AND
   NOT TO mailservers AND ACCEPTED forward;

Figure A.3: Example ITVAl query

This query file can be easily adapted to fit almost any network by changing the addresses in lines 1 and 2 and will catch some interesting behaviors of a poorly configured firewall. For instance, it is good practice to secure important servers so that only services provided by that server are allowed from outside the network (see for example, [8], pp. 56, 75–76, and [48], pp. 183). The query on line 3 lists all services allowed to the mail server that are not strictly necessary for mail service.

A well-configured packet filter can block most traffic from spam relays by only accepting mail traffic that originates from the mail servers. However, for the mail server to function, incoming traffic to the mail server must also be permitted. The query on line 4 checks whether the firewall meets these specifications. If mail traffic is allowed from a host not listed in the mailservers list, that host will appear in the query results.

While these queries are very useful for ensuring that the firewall controls unwanted traffic, both of these queries have significant problems. The query on line 3 will return a large number of false positives on a stateful firewall. The query on line 4 ensures that the firewall defeats most spam relays, but doesn’t check whether a host that uses spoofing can circumvent that protection.

A.3.1 Accounting for State

Many packet filters are configured to accept traffic from established or related connections. Accepting established connections allows the firewall to handle incoming replies
caused by legitimate outgoing traffic. Accepting related connections allows protocols such as FTP to successfully navigate the packet filter.

```
QUERY DPORT TO mailservers AND
    NOT FOR mail AND
    IN NEW AND ACCEPTED forward;
```

**Figure A.4:** Adding state to a typical query

Since established and related connections can potentially use any source or destination port, the query on line 3 of figure A.3 will list every network port even if the firewall is adequately protected against spam relays. To eliminate these false positives, we must be careful to consider only new connections. In ITVal, we can do this by adding an `IN NEW` condition as shown in figure A.4. This forces the query to match only incoming new connections.

### A.3.2 Accounting for Spoofing

Because a packet filter has no way of identifying the authentic source of a packet, address spoofing can defeat any filtering policy. A common strategy for preventing most spoofing attempts is to drop packets that arrive on a network interface and have source addresses from the wrong side of the interface [8]. Before we test for individual vulnerabilities, we should check to make sure that the firewall has this kind of spoofing protection in place. The easiest way to test for this is to write a query for each interface. The query will list all hosts that can access the network from the wrong side of the interface.

```
QUERY SADDY FROM 128.140.10.* AND
    INFACE eth0 AND ACCEPTED forward;
QUERY SADDY NOT FROM 128.140.10.* AND
    INFACE eth1 AND ACCEPTED forward;
```

**Figure A.5:** Queries to check for spoofing on the example network

Figure A.5 introduces two queries that could be used to verify that the example network is protected from spoofing attacks. Another well known precaution against spoofing attacks
is to block access from non-routable addresses and other illegal addresses such as broadcast addresses (see [48], pp. 186–190). These addresses are commonly used as spoofed source addresses by malicious programs.

```
Grupo illegal *.*.*.255 10.*.*.*
172.[16-31].** 192.168.**
169.254.** 224.**.* 0.***
[240-255].**.* 127.***

Query SADDY FROM illegal AND
((INFACE eth0 AND OUTFACE eth1) OR
(INFACE eth1 AND OUTFACE eth0)) AND ACCEPTED forward;
```

Figure A.6: Queries to check for illegal source addresses

It is usually very simple to check that the firewall correctly restricts packets from these addresses. Figure A.6 illustrates how this might be done for the example network. Line 1 creates a named group consisting of special addresses to check. Line 2 tests whether the firewall blocks network traffic from those addresses that crosses the network perimeter.

### A.4 Putting it All Together

We now illustrate our technique with a sophisticated example derived from guidelines in “Linux Network Security” [48] and “Firewalls and Internet Security” [8]. The example includes the previously mentioned checks for spoofing and use of illegal addresses and also verifies that the firewall adequately protects three important servers.

```
Grupo firewall 128.40.10.1;
Grupo dns_server 128.40.9.101;
Grupo mailservers 128.40.10.2
  128.40.10.3;
Grupo webserver 128.40.10.4;

Grupo illegal *.*.*.255 10.*.*.*
```
172.[16-31].*.* 192.168.*.*
169.254.*.* 224.*.*.* 0.*.*.*
[240-255].*.*.* 127.*.*.*;

#Check for packets that appear on the each
#interface with the wrong addresses.

QUERY SADDY INFACE eth0 AND
    FROM 128.40.10.* AND ACCEPTED forward;

QUERY SADDY OUTFACE eth0 AND
    NOT FROM 128.40.10.* AND ACCEPTED forward;

QUERY SADDY INFACE eth1 AND
    NOT FROM 128.40.10.* AND ACCEPTED forward;

QUERY SADDY OUTFACE eth1 AND
    FROM 128.40.10.* AND ACCEPTED forward;

#Check that broadcast addresses are
#blocked. This can
#discourage SMURF and FRAGGLE attacks.
#Also check other non-routable and
#illegal addresses.

QUERY SADDY FROM illegal AND
    ((INFACE eth0 AND
      OUTFACE eth1) OR
(INFACE eth1 AND
OUTFACE eth0)) AND ACCEPTED forward;

#Ensure that UDP traffic to and from the
#firewall box itself is blocked, except
#for NTP traffic and DNS.

QUERY SADDY (FOR UDP * AND
NOT FOR UDP 123 AND
NOT FOR BOTH 53) OR
(ON UDP * AND
NOT ON UDP 123 AND
NOT FOR BOTH 53) AND
IN NEW AND ACCEPTED input;

QUERY SADDY (FOR UDP * AND
NOT FOR UDP 123) OR
(ON UDP * AND
NOT ON UDP 123) AND
IN NEW AND ACCEPTED output;

#Check that SSH to the firewall box
#is only allowed from internal
#hosts.

QUERY SADDY (FOR TCP 22 AND
TO firewall AND
NOT FROM eth0)
AND IN NEW AND ACCEPTED input;

#Check that DNS traffic can only
#come from the correct external
#server.

QUERY DADDY (FOR BOTH 53) AND
    NOT (TO dns_server OR
    FROM dns_server) AND
    IN NEW AND ACCEPTED forward;

#Check that Only allow HTTP and
#HTTPS traffic is allowed into
#the webserver.

QUERY SADDY TO webserver AND
    (NOT FOR TCP 80 AND
    NOT FOR TCP 443 AND
    NOT FOR BOTH 53) AND
    IN NEW AND ACCEPTED forward;

#Check that only SMTP and POP
#traffic is allowed to
#the mailservers (and DNS replies).

QUERY SADDY TO mailservers AND
    (NOT FOR TCP 25 AND
    NOT FOR TCP 110
This example can be used as a boilerplate for creating security policy validation queries. It can be easily adapted to a new network simply by changing the IP addresses in the predefined groups. By making slight modifications to some of the queries, the system administrator can tweak the example to account for variations in the security policy (to allow the web server to provide additional services, perhaps).

A.5 Conclusion

In addition to the sources already mentioned, there are several other places to find good information on locking down a firewall. Among them is a list of guidelines published by the CERT coordination center at http://www.cert.org/tech_tips/packet_filtering.html and a list of ports that most firewalls should block at http://www.doshelp.com/Ports/Trojan_Ports.htm. Information from these sources can be easily converted into firewall compliance checks.

While these techniques can significantly reduce the risk of leaving a significant hole in the firewall policy, they also have some disadvantages. Traditional penetration testing tools can test both the firewall and the server for vulnerabilities. They can test whether buggy firewall software has compromised the security policy. Rigorous testing should use both techniques side by side to ensure that the firewall correctly applies the security policy.

The latest version of ITVal and several more examples of how to use it for verifying a security policy can be obtained from http://itval.sourceforge.net. The examples are designed to be easily modified for use in a variety of settings and provide checks for many known vulnerabilities.
Bibliography


[41] REG QUINTON. Using Solaris ipfilters.  

[42] RUSTY RUSSEL AND MICHAEL NEULING. iptables source code.  
http://lxr.linux.no/source/include/linux/netfilter/x.tables.h.


[49] BILL STEARNS. i2i: The firewall conversion routines, September 1999.  
http://www.stearns.org/i2i/.


[51] BOB TODD. SARA man page.  


http://rocky.eld.leidenuniv.nl.


VITA

Robert Mathias Marmorstein

Robert Marmorstein was born in Sacramento, California on March 6, 1979. After graduating from Central High School in Aberdeen, South Dakota, he enrolled at Washington and Lee University. In May 2000, He graduated with a Bachelor of Arts in Computer Science and a Bachelor of Arts in Mathematics. He completed his Master of Science at the College of William and Mary in the spring of 2003.