Firewall Policy Diagram: Novel Data Structures and Algorithms for Modeling, Analysis, and Comprehension of Network Firewalls

By

Patrick G. Clark

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of
Doctor of Philosophy.

Dr. Arvin Agah, Chairperson

Dr. Swapan Chakrabarti

Committee members

Dr. Prajna Dhar

Dr. Jerzy W. Grzymala-Busse

Dr. Bo Luo

Date defended: ____________________

The Dissertation Committee for Patrick G. Clark certifies
that this is the approved version of the following dissertation :

Firewall Policy Diagram: Novel Data Structures and Algorithms for Modeling, Analysis, and Comprehension of Network Firewalls

Dr. Arvin Agah, Chairperson

Date approved:

Abstract

Firewalls, network devices, and the access control lists that manage traffic are very important components of modern networking from a security and regulatory perspective. When computers were first connected, they were communicating with trusted peers and nefarious intentions were neither recognized nor important. However, as the reach of networks expanded, systems could no longer be certain whether the peer could be trusted or that their intentions were good. Therefore, a couple of decades ago, near the widespread adoption of the Internet, a new network device became a very important part of the landscape, i.e., the firewall with the access control list (ACL) router. These devices became the sentries to an organization's internal network, still allowing some communication; however, in a controlled and audited manner. It was during this time that the widespread expansion of the firewall spawned significant research into the science of deterministically controlling access, as fast as possible. However, the success of the firewall in securing the enterprise led to an ever increasing complexity in the firewall as the networks became more inter-connected. Over time, the complexity has continued to increase, yielding a difficulty in understanding the allowed access of a particular device.

As a result of this success, firewalls are one of the most important devices used in network security. They provide the protection between networks that only wish to communicate over an explicit set of channels, expressed through the protocols, traveling over the network. These explicit channels are described and implemented in a firewall using a set of rules, where the firewall implements the

will of the organization through these rules, also called a firewall policy. In small test environments and networks, firewall policies may be easy to comprehend and understand; however, in real world organizations these devices and policies must be capable of handling large amounts of traffic traversing hundreds or thousands of rules in a particular policy. Added to that complexity is the tendency of a policy to grow substantially more complex over time; and the result is often unintended mistakes in comprehending the complex policy, possibly leading to security breaches. Therefore, the need for an organization to unerringly and deterministically understand what traffic is allowed through a firewall, while being presented with hundreds or thousands of rules and routes, is imperative.

In addition to the local security policy represented in a firewall, the modern firewall and filtering router involve more than simply deciding if a packet should pass through a security policy. Routing decisions through multiple network interfaces involving vendor-specific constructs such as zones, domains, virtual routing tables, and multiple security policies have become the more common type of device found in the industry today. In the past, network devices were separated by functional area (ACL, router, switch, etc.). The more recent trend has been for these capabilities to converge and blend creating a device that goes far beyond the straight-forward access control list.

This dissertation investigates the comprehension of traffic flow through these complex devices by focusing on the following research topics:

- Expands on how a security policy may be processed by decoupling the original rules from the policy, and instead allow a holistic understanding of the solution space being represented. This means taking a set of constraints on access (i.e., firewall rules), synthesizing them into a model that represents an ACCEPT and DENY space that can be quickly and accurately analyzed.

- Introduces a new set of data structures and algorithms collectively referred to as a Firewall Policy Diagram (FPD). A structure that is capable of modeling Internet Protocol version 4 packet (IPv4) solution space in memory efficient, mathematically set-based entities. Using the FPD we are capable of answering difficult questions such as: what access is allowed by one policy over another, what is the difference in spaces, and how to efficiently parse the data structure that represents the large search space. The search space can be as large as 2^{88}; representing the total values available to the source IP address (2^{32}), destination IP address (2^{32}), destination port (2^{16}), and protocol (2^{8}). The fields represent the available bits of an IPv4 packet as defined by the Open Systems Interconnection (OSI) model. Notably, only the header fields that are necessary for this research are taken into account and not every available IPv4 header value.
- Presents a concise, precise, and descriptive language called Firewall Policy Query Language (FPQL) as a mechanism to explore the space. FPQL is a Backus Normal Form (Backus-Naur Form) (BNF) compatible notation for a query language to do just that sort of exploration. It looks to translate concise representations of what the end user needs to know about the solution space, and extract the information from the underlying data structures.
- Finally, this dissertation presents a behavioral model of the capabilities found in firewall type devices and a process for taking vendor-specific nuances to a common implementation. This includes understanding interfaces, routes, rules, translation, and policies; and modeling them in a consistent manner such that the many different vendor implementations may be compared to each other.

Acknowledgements

It has been a great pleasure working with the faculty, staff, and students at the University of Kansas during my tenure as a graduate student. I would like to thank Professors Swapan Chakrabarti, Prajna Dhar, and Bo Luo for serving on my committee.

Furthermore, this work would never have been possible without the flexibility to pursue my own research interests and the guidance provided by my long-time advisor and committee chair, Arvin Agah. I truly appreciate his ability to bring a fresh perspective as he served a vital role in helping to direct my research. I will be forever grateful for the lessons about academic life, and what it means to be a professor, teacher, and research scientist.

I would also like to acknowledge the efforts invested into my graduate experience by Jerzy Grzymala-Busse. Not only did I gain a deep knowledge about data mining and rough set theory, he has also shared his extensive experience with solid investigative methods and the proper ways to present research results to the community. I have thoroughly enjoyed working with him and his research team as we blended theoretical research with practical implementations. I will always truly appreciate the lessons learned.

Finally, I would like to thank my parents Ken and Pat Clark, my sister Kristen Matthews, her husband Andrew Matthews, my niece Ella Kate Matthews, the entire Clark, Matthews, Jones extended family, and my friends for all their support. Without their encouragement and involvement the long process of making this all possible would have been immeasurably more difficult.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer networking has arguably been one of the most important advancements in modern computing. Allowing disparate applications to trade information, conduct business, exchange financial transactions, and even the routine act of sending an email are some of the most common things we do with computers today. Even with the advancement of ever faster computer chips, the trend continues to connect devices at an astounding rate. In addition, there is also a thriving mobile device market, thus increasing the amount of traffic flowing between systems. An important aspect of this interconnected system is security. Without security, the convenience and speed of networked transactions would present more risk than the majority of applications could handle. In order to mitigate the risk and provide a much more secure communication channel, the firewall device was designed and deployed. It is one of the most widely used and important networking tools, existing in virtually every organization connected to a network. In fact, over the past two decades the landscape of network security has come to rely heavily on this single type of device. The primary purpose of a firewall is to act as the first line of defense against malicious and unauthorized traffic, keeping the information that the organization does not want out, while allowing approved access to flow [Cheswick et al., 2003]. Figure 1.1 illustrates how a firewall is commonly used in a network environment.

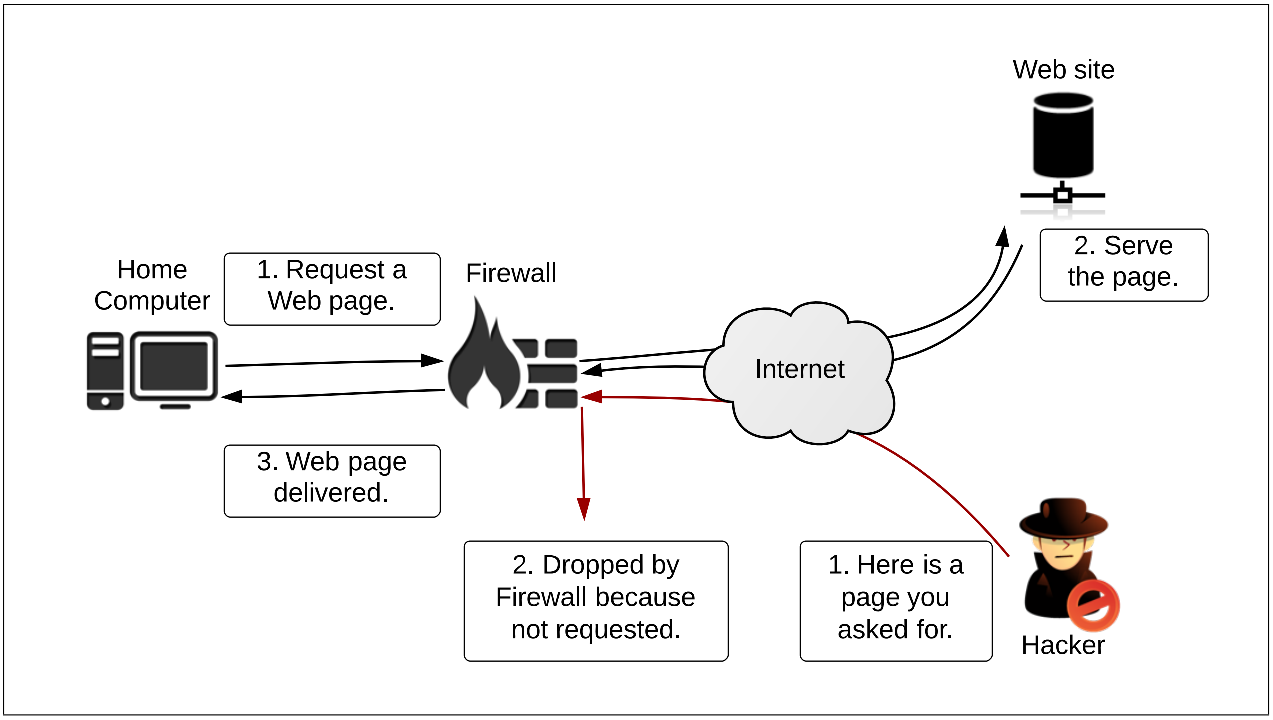


Figure 1.1: Common firewall placement in a network.

Since the introduction of the first packet filtering software, the configuration of the software has been built upon the concept of a firewall rule. A firewall rule, in general, is what most rules represent, an explicit regulation governing the conduct within a particular activity. In this case the rule states what traffic will be allowed to pass. As with most configurable software, firewall rules start off as an "easy to understand" set of instructions. However, time and use result in increased complexity, especially if that use is not well managed and understood. As a result, in most medium to large organizations, firewall configurations consist of hundreds and sometimes thousands of individual rules that must be managed by a team of security professionals in the most efficient manner possible. The fact is that an organization's network security is only as good as its firewall configuration; and as discussed in subsequent sections, the security is often lacking. This is not a lack of effort on the part of the security or management team, instead it is a shortcoming in algorithms available to understand the complexities. Therefore, the motivation for this research effort is the fact that firewall configuration and the research underlying comprehension of access policy have not advanced to address the algorithmic problem and the size of the solution space.

1.2 Firewall Basics

Firewalls have become an important part of the network landscape because they will often provide the first defense against malicious attacks and unauthorized traffic. Firewalls allow two entities to connect their networks together through existing infrastructure and protocols, while securing the private networks behind them [Chapman and Zwicky, 1995; Chapman, 1992]. The common placement of a firewall is at the entry point into a network so that all traffic must pass through the firewall in order to enter the network. The traffic that passes through the firewall is derived from existing packet-based protocols, and a packet can be thought of as a tuple with a set number of fields [Chapman and Zwicky, 1995; Chapman, 1992]. Examples of these fields are the source/destination IP address, port number, and the protocol field. A firewall will inspect each packet that travels through it and decide if it should allow that packet to pass based on a sequence of rules. This sequence of rules is made up of individual rules that follow the general form:

$$\langle \mathit{predicate} \rangle \rightarrow \langle \mathit{decision} \rangle$$

The *predicate* defines a boolean expression over the fields in a packet tuple that are evaluated and the physical network interface from which the packet arrives. For example, source IP is 10.2.0.1 and destination IP address is 192.168.1.1 on eth0 (a common label for a linux interface). Then the *decision* portion of a rule is what happens if the predicate matches to a true evaluation. A *decision* will likely be accept or deny with the possibility of additional actions, such as an instruction to log the packet [Chapman and Zwicky, 1995; Chapman, 1992]. However, for the purpose of this research we are only concerned with the accept or deny decision.

A firewall policy is made up of an ordered list of these rules such that as a packet is processed by the firewall, it attempts to match the packet to the predicate. Matching the packet means that the firewall evaluates a packet based on the fields in the rule tuple, a packet matches the rule if it matches all the fields identified in the predicate of the rule. The predicate does not necessarily need to contain a value for all possible fields and can sometimes contain the "any" variable in a field to indicate to the rule processing software that this is a "do not care" condition or U of the predicate. This means that any value for that variable will match. It must completely match all the fields for the firewall to take the appropriate action. These rules are processed in order, until the firewall finds a match, at which time it will take the appropriate action as identified by the decision.

1.2.1 Interfaces

An interface on a firewall is the physical network connection to the wire (or radio) that will transmit the traffic to and from the parties communicating [Kurose and Ross, 2007]. In the strictest definition of the firewall and packet filtering device, two network interfaces exist. This represents a physical separation between one network and another, requiring the traffic to flow through the firewall from one interface to the other, and vice versa. However, as discussed in [Chapple et al., 2009], the firewall will commonly have many more than two interfaces, which means that a firewall is no longer just a filtering device, but must also decide the appropriate egress (outbound) interface for traffic. The firewall will then use a routing table to find the egress interface.

1.2.2 Routes

In this work a route can be defined as a simple one tuple rule with a decision being the egress interface. The one tuple of the traffic being processed is the destination. For a particular routing rule, the destination can be identified as an IP Address, address range, or Classless Inter-Domain Routing (CIDR) format. Therefore, in a similar manner as security rules, the solution space can be split as the traffic is processed. Traffic is attempted to be matched from the top to bottom in the routing table.

1.2.3 Rules and Policies

Firewalls commonly have multiple ingress (inbound) and egress (outbound) network connections. The purpose of multi-homed (multi-interface) firewalls is to allow the physical separation between networks, forcing the data to be processed by that firewall device, with no other route around the device. Each interface of the firewall relates to a policy such that some firewall vendors allow the definition of multiple policies based on the interface where the packet was received. A policy is then "applied" to the packet and a routing decision is made based on that application.

The landscape of the enterprise network will feature multiple entry and exit points, as well as varying levels of access throughout the internal network. This multi-point egress and hierarchical model require that the organization employ multiple firewalls to physically secure those points of access and ensure that the organization's or sub-organization's security policy is enforced. These multiple firewalls are often acquired over time and often from different vendors. With these different devices from different origins comes a difference in methods of configuration and skill-sets required to properly program them to accurately and securely protect the internal network. The difference might exist in the technical aspects of representing the policy, but the overall organizations policy is, in itself, generalized. Therefore part of the challenge to complexity is not only the number of devices, but the types as well [Wang et al., 2006].

1.2.4 Network Address Translation

A final broad element of a modern firewall is Network Address Translation (NAT). The primary purpose of a NAT capable firewall is to allow an address on one side of the network to be translated as it travels through the firewall. NAT implementation will often function as a translation table such that the inbound traffic will match an entry (either source or destination address) to be translated to another address on the egress side. The firewall device is then required to keep track of that conversation in order for response packets to have the reverse translation applied and arrive at the appropriate destination. There are usually three types of NAT: Source Address Translation (SNAT), Destination Address Translation (DNAT) and Port Address Translation (PAT). Each of these are roughly the same idea, and each will require a translation table, yet they will operate on different fields of the packet [Kurose and Ross, 2007].

1.3 Hierarchical Data Sets and Structures

The Internet protocol addressing and routing scheme as defined by the OSI model: Internet Protocol (IP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP); represent data packet based routing using different fields of binary, numeric data. Each of these fields of binary data represent a sequence of available numbers, therefore they can be considered hierarchical data sets available to be represented as a binary tree. Figure 2.1 illustrates the concept. The background section will cover the topic in more detail. Some of the unique properties of representing binary data as a tree is to know that fully formed child trees of a particular node assure the model that a range of values is covered and that the node can be compressed. In addition, the hierarchical nature of the data set can be further compressed using a Reduced Ordered Binary Decision Diagram (ROBDD) [Bryant, 1992]. In addition to capturing the innate hierarchy contained in binary data as a compressed format, the ROBDD posses the capability of performing various mathematical SET operations on the data without having to decompress the structure. Therefore this work uses these types of data structures to aid in comprehension firewall policies. The Firewall Policy Diagram data structure is explained in detail in chapter 3.

1.4 Problem Statement and Key Contributions

Although the deployment of network security devices and firewall technology is a very important step in securing the communication of the enterprise, it is only the initial step. As with many technologies, adoption only begins the process of securing the enterprise and the true measurement of cost to the organization is in the proper management and understanding of the technology itself. In this case it is ensuring that the firewall rule sets accurately and deterministically represent the policy defined by the organization as a whole and ensuring that as the organization changes over time, the change in the security devices is accurate and consistent.

Over the past decade firewall rule-bases have grown out of control. In a study finished in 2001, it was discovered that the average organization typically has 200 firewalls under the control of its network consisting of an average of about 150 rules per device [Wool, 2004]. In addition, these rule sets have been shown to grow to thousands of rules controlling routing between as many as 13 distinct networks [Wool, 2004]. More recent statistics further support that the growth has only accelerated. In a study finished in 2009, it was determined that the growth in complexity has out paced the growth in the organization's ability to synthesize and comprehend the changes [Chapple et al., 2009]. The average number of rules has substantially increased from 150 in 2001 to 793, with the largest rule set found comprised of 17,000 rules [Chapple et al., 2009]. The study did not discuss the number of firewalls deployed, but in the unlikely case that firewall deployment growth stopped and the number of firewalls at an average organization stayed at 200 [Wool, 2004], then approximately 160,000 rules (200×793) would be under active management. In addition, the study also discovered that the average rule turnover (change) rate for an organization is 9.9% of the rules per month [Chapple et al., 2009]. This means that the average firewall administration team has to accurately manage about 160,000 rules where 16,000 of those rules are changing on a monthly basis. Therefore, the ability to accurately and confidently understand the modern firewall configuration and know what changes have occurred are more difficult than ever, and continue to increase in complexity.

On the surface this may seem like a problem where the organization or community is losing focus due to the speed of change in the environment. However, upon closer inspection the problem is exposed as a theoretical puzzle where just adding more human management, more computing power, or more workers will never bring the sort of understanding necessary. What is needed is a different way to analyze the modern firewall and the organizational security policy, a method to provide accurate software modeling and comprehension of the device. This area of research involves areas not covered completely in related work and we present the results of the research effort providing a number of key contributions.

The basis of the research presented is a novel set of data structures, together called a Firewall Policy Diagram (FPD). These data structures seek to solve the problem of large network access comprehension, as it relates to firewall policies, in several areas:

- De-correlation of the firewall policy from the source rule set in order to gain a holistic view of access. This will remove any overlapping rules that will often exist in a firewall policy [Yuan et al., 2006] and the resulting data structure will model the actual ACCEPT and DENY spaces.

- Provide the ability to perform arbitrary mathematical set based operations such as *intersection*, *union*, and *complements*. These operations will assist in reasoning about firewall policy changes over time. They will also provide the foundation for many other firewall operations, such as understanding the functional differences between two policies. In addition, these operations will provide the base for querying an arbitrary policy.

- Provide the data structure basis for the implementation of an approach to query the policy.

- Once a policy has been decomposed into a FPD, allow the reconstitution of that policy into a human comprehensible form, similar to an equivalent policy rule set.

- Experiments will show that these operations can be executed in seconds of computation time even on large policies (up to 10,000 rules).

The research presented then builds on the FPD foundation a means in which to gain a better understanding of large network access comprehension through a domain specific query language named Firewall Policy Query Language (FPQL), with the following properties:

- Provide an expressive query language that a team of firewall administrators would be able to use to answer important questions about what is contained in a large, and often very difficult to understand, set of firewall policies.
- Design a syntax similar to the familiar Structured Query Language (SQL) [Codd, 1970] such that the users can describe what they desire in a declarative manner, allowing the system to find and return the information. This is in contrast to a more procedural style of telling the system how to find the information.
- Provide experimental evidence of the efficacy of this language, presenting the results of experiments run against large policies (up to 20,000 rules) showing how individual results can be returned in approximately 120 microseconds of processing time.
- Present how FPQL is capable of formal verification of single and multiple policies in a manner similar to FIREMAN [Yuan et al., 2006].

Finally, this research further builds upon the existing base FPD by providing a formal structure for abstracting a modern firewall into its common elements so that vendor nuances may have a common software simulated implementation. We present a framework called Behavior Abstraction Modeling in which to simulate individual firewalls in software such that analysis and other formal methods may be used with the model. Some examples of industry uses for modeling networks and firewalls in software include:

- Network trace analysis for understanding how a packet will traverse the device without physically sending the packet. In addition to an individual packet, a packet *space* in the form of an FPD may traverse the network to understand what will make it from point A to point B.

- Logical comparisons of firewall vendor implementations. If two firewalls are configured to behave the exact same way but are from two different implementations, modeling the behavior of each vendor in software allows testing that each ingress and egress FPDs are identical. Subsequently, if they are not identical, the FPD used to traverse the spanning graph can represent what is different from what is potentially a large solution space.

- Behavior abstraction modeling of specific firewall vendors serve as the basis for automated translation from one vendor configuration to another with certainty of how the device will behave.

- Participate in a larger software modeled network for more comprehensive simulations without requiring the physical cables and radios.

- This research shows Behavior Abstraction Modeling in conjunction with the FPD is capable of modeling very large firewalls for analysis in milliseconds of time. Allowing fast verification of changes to the firewall and testing of those changes entirely in software.

1.5 Dissertation Organization

This dissertation is organized into six chapters. In chapter 2 we discuss background topics including Internet Protocol basics, firewall history, basic firewall design and modeling the IP address space as binary trees. Chapter 3 discusses the details of the Firewall Policy Diagram, the foundation of our work for human comprehension of a firewall. Then, chapter 4 builds on that foundation with the design and development of the Firewall Policy Query Language, demonstrating the useful functionality of querying a policy for various access capabilities.

Chapter 5 outlines a firewall behavior abstraction model, allowing a methodology independent of vendor and implementation to simulate and comprehend modern firewalls. Chapter 6 concludes this dissertation.

Chapter 2

Background

Computer networking and today's inter-connected system of computers can trace their inception to research done in the 1960's by telecommunications companies. During that time frame the telephone network was the primary means in which voices where connected over an audio signal that was a direct circuit. This meant that one long wire through one or many switch boards allowed the two ends communication through an analog signal. While this was fairly common and stood the test of time when the communication was most often voice, data began being carried over these lines as a means in which to connect two physically distance computers [Kurose and Ross, 2007].

This era was also the time of the early computer. These early computing machines were very large and expensive to operate, therefore it was very common for the computer to be given multiple tasks during the same time frame and the operating system time shared the computer between multiple tasks [Kurose and Ross, 2007]. It was perhaps natural that the designers and users of these time-shared computers would see the committed circuit allocated between only two machines as inefficient and began to devise ways to improve the efficiency of the link by sharing it with multiple computers [Kurose and Ross, 2007]. Since a computer was already processing data in discrete elements, most of the ideas were to divide the information flowing between the two computers into discrete packets, hence the beginning of packet networks [Kurose and Ross, 2007].

As more and more computers at research facilities were being linked together, other ideas surfaced to help make those networks more efficient. Packet switching and routing became more common and the redundant nature of a switched network went on to lead Advance Research Projects Agency (ARPA) to fund the first packet-switched computer network and a direct ancestor of the public Internet [Kurose and Ross, 2007].

During the 1970's and 1980's the United States and the world became more interconnected. Protocols and standards emerged to facilitate a distributed growth strategy and allow disparate systems to communicate just using explicit binary protocols to handshake [Kurose and Ross, 2007]. These protocols evolved into the three key Internet protocols seen today: Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Internet Protocol (IP).

IP forms the basis on which two devices are connected to each other through a computer network or the Internet. It defines the standard format that a packet of information is comprised and also describes the standard headers, which allow the packet to get from one computer to another. Table 2.1 describes the general format of an IP version 4 datagram.

<table>
<tr><th colspan="5">32 bits of data</th></tr>
<tr><td>Version</td><td>Header length</td><td>Type of service</td><td colspan="2">Datagram length (bytes)</td></tr>
<tr><td colspan="3">16-bit Identifier</td><td>Flags</td><td>13-bit Fragmentation offset</td></tr>
<tr><td colspan="2">Time-to-live</td><td>Upper-layer protocol</td><td colspan="2">Header checksum</td></tr>
<tr><td colspan="5">32-bit Source IP address</td></tr>
<tr><td colspan="5">32-bit Destination IP address</td></tr>
<tr><td colspan="5">Options (if-any)</td></tr>
<tr><td colspan="5">Data</td></tr>
</table>

Table 2.1: General format of an IP version 4 Packet.

Portions of these protocols form some of the basics of this research and allow the determination of reach-ability and other means of a specific host on the network. Because this work is focused on levels of network access as they relate to routing and switching, the most important fields are the source IP address, source port, destination IP address, destination port, and protocol. The remaining fields are primarily useful for efficiency in the network itself rather than security or reach-ability of the network.

Two fields just cited are not defined in the IP version 4 (IPv4) protocol stack, they are the source and destination port. These two fields were built on top of the IP protocol in the form of Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The data in these two protocols is layered on top of the IP protocol in the Data field (Table 2.1). Tables 2.2 and 2.3 describe the layout of those packets and the location of the port values.

<table>
<tr><th colspan="9">32 bits of data</th></tr>
<tr><td colspan="8">Source port #</td><td>Dest port #</td></tr>
<tr><td colspan="9">Sequence number</td></tr>
<tr><td colspan="9">Acknowledgment number</td></tr>
<tr><td>Header length</td><td>unused</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Receive window</td></tr>
<tr><td colspan="8">Internet checksum</td><td>Urgent data pointer</td></tr>
<tr><td colspan="9">Options</td></tr>
<tr><td colspan="9">Data</td></tr>
</table>

Table 2.2: General format of a TCP segment.

<table>
<tr><th colspan="2">32 bits of data</th></tr>
<tr><td>Source port #</td><td>Dest port #</td></tr>
<tr><td>Length</td><td>Checksum</td></tr>
<tr><td colspan="2">Application data (message)</td></tr>
</table>

Table 2.3: General format of a UDP segment.

It is important to note that IP version 4 is still currently widely used, however IP version 6 has started adoption in the industry, therefore further work can be done using that protocol. The premise is mostly the same, however the 32-bit address scheme of IPv4 has been nearing exhaustion due to the large number of computers networked together on the public Internet and IP version 6 allows a much larger addressing scheme (among other improvements). However, the large majority of internal networks still make use of the IP version 4 address scheme.

2.1 Firewall History

News reports over the last decade about hacking attempts at many prominent organizations demonstrate that the world of an inter-connected network is not always a safe place for important data. The anonymity and decentralization that the network protocols have provided are great for the rapid adoption of a very powerful technology, however they also provide opportunities for nefarious adversaries to conduct malicious acts against the unsuspecting public. In many organizations, measures have been taken to protect the most important assets of their computer system. These measures will almost always involve greater safety through network devices known as firewalls [Kurose and Ross, 2007].

As with most physically secure facilities, like a building or a compound, there are only a few points of access to the entity. These points of access are tightly guarded with protection measures put into place that allow the inspection of personnel crossing that access point. In a similar manner, a computer network can employ a high level of security between the *trusted* and *untrusted* zones by placing a device that physically separates the two networks [Kurose and Ross, 2007]. This device is most often called a firewall. Figure 1.1 depicts what this placement might look like.

In general, a firewall is a hardware or software device that will isolate an organization's network from the Internet or other untrusted network. It allows some packets to pass, those that adhere to the security policy defined by the organization, and denies others. The firewall provides the network administrator a tool in which to control access to resources within the organization to only those specified by the overall security policy. In general, a firewall has three goals [Kurose and Ross, 2007]:

- All traffic from *untrusted* to *trusted* and the reverse, must pass through the firewall device. Just as a few access points must control access to a secure facility, all traffic must pass through the firewall for adherence to the organizational security policy.

- Authorized traffic is the only traffic allowed to pass through the firewall. Traffic is compared against an allowed policy and only those packets that adhere to the policy are allowed to pass.

- The firewall itself is immune to compromise. This means that the firewall itself is a device connected to the network that is running software and hardware. Therefore measures must be taken by the network and firewall administrators to ensure that it cannot be penetrated.

2.2 Packet Filters

Firewall operation and its ability to keep out unauthorized traffic is based on the IP protocol and the structure of data packets that are flowing over an IP network. Therefore, the most basic level of control that a firewall must exhibit is a packet filter, where each packet is examined in isolation. The firewall then determines if the packet should be accepted or denied based on a list of administrator-defined filtering rules, most often derived from an organizational policy [Kurose and Ross, 2007]. Filtering decisions will commonly be based on [Kurose and Ross, 2007]:

- IP source or destination address
- Protocol type: TCP, UDP, ICMP, etc.
- TCP or UDP source and destination port
- ICMP message type
- Different rules for ingress or egress routing
- Different rules for different router interfaces

A network administrator configures the firewall based on the security policies defined by the organization. For example, the policy may take into account the e-commerce site that the organization operates and allow access to the Web server handling that traffic. It may also consider that no direct connection to a desktop computer is allowed from the untrusted network and therefore that traffic would be denied. Table 2.4 demonstrates an access list that might be defined for a particular organization. It shows access rules 1 and 3 grant traffic to internal hosts from any computer in the 192.168.0.0/16 range (65,536 hosts) over port 80 and 53 from any port greater than 1023. Access rules 2 and 4 grant response traffic back to the originating hosts. Finally, rule number 5 automatically denies any traffic that is not accepted by a previous rule. Most modern packet filtering firewalls will automatically put the final rule because it is viewed as an industry best practice to deny any traffic that is not explicitly allowed.

rule	**action**	**src address**	**dest address**	**protocol**	**src port**	**dest port**	**flag**
1	allow	192.168/16	outside of 192.168/16	TCP	> 1023	80	all
2	allow	outside of 192.168/16	192.168/16	TCP	80	> 1023	ACK
3	allow	192.168/16	outside of 192.168/16	UDP	> 1023	53	all
4	allow	outside of 192.168/16	192.168/16	UDP	53	> 1023	all
5	deny	all	all	all	all	all	all

Table 2.4: Example access control list for a firewall interface.

In this research effort only certain fields will be analyzed and modeled. The source address, destination address, protocol, and destination port will be used. This is because most common reach-ability and vulnerabilities are not involved in the use of source port and flag. The source port is often not used in most firewall products and the flag is only at the network layer for setup and tear down of TCP connections.

2.3 Firewall Rule Processing

2.3.1 Policies and Rules

A firewall is primarily a physical device that purposely interrupts the flow of data through a network link in order to ensure that the data adhere to a defined prescript most often outlined by an organization's security group. There are hardware and software based firewalls, but the general idea behind all the implementations is enforcement of allowed traffic.

This device may segregate one link or many through different interfaces. These interfaces are commonly setup to have their own subset of the IPv4 address space and route from one area of the network to another, inspecting the traffic along the way. Each ingress and egress interface combination will often have a policy associated with that pair and define what information is allowed to flow. So, a firewall can have multiple policies and in each policy is an ordered set of rules. As a packet enters the interface and a policy is applied, the application of the policy attempts to match the packet rules with the predicate of the rule. As the ordered set of rules is processed, top to bottom, the first rule that matches the packet is acted upon.

2.3.2 Redundant, Shadowed, and Conflicting Rules

An important concept that serves as the basis for understanding firewall policies as it relates to their composition (rules) is how representing rule processing as a set of spaces can provide insight into a policy, thus assisting in comprehension. An example of this idea is that since the firewall policy is an ordered list of rules, there can sometimes be an overlap of two rules, which means that a packet may match more than one rule in the policy. As a result it is said that a firewall has *conflicting* rules if two rules overlap but have different decisions [Al-Shaer et al., 2005; Liu and Gouda, 2005]. The premise is that discerning that conflict can be accurately and quickly found by modeling the rule set in set-based data structures [Yuan et al., 2006].

In order to gain this sort of insight and model the policy, it is necessary to introduce the potential input to a policy represented by I, which is a collection of packets that can possibly arrive at this policy. For the jth rule $\langle P_j, action_j \rangle$ in the policy, the current state is defined as $\langle A_j, D_j \rangle$, where A_j and D_j denote network traffic accepted and denied before the jth rule, respectively, i.e., the ACCEPT and DENY spaces. Then R_j denotes the collection of remaining traffic that can possibly arrive at the jth rule. As a result R_j can be found using the input I and the current state of information [Yuan et al., 2006].

$$R_j = I \cap \neg(A_j \cup D_j) \tag{2.1}$$

For the first rule in the rule-set, the initial values are $A_1 = D_1 = \emptyset$ and $R_1 = I$. As the processing of each rule proceeds, the state of each variable is updated. The data are processed left to right, so the new state of R reflects the change to A and D of that same line [Yuan et al., 2006]:

$$\begin{cases} \langle A, D, R \rangle, \langle P, \mathit{accept} \rangle \vdash \langle A \cup (R \cap P), D, I \cap \neg(A \cup D) \rangle \\ \langle A, D, R \rangle, \langle P, \mathit{deny} \rangle \vdash \langle A, D \cup (R \cap P), I \cap \neg(A \cup D) \rangle \end{cases} \tag{2.2}$$

After processing each rule the ACCEPT and DENY spaces are accurately represented through A and D, respectively. This premise allows conclusions to then be drawn about a rule being added to spaces based on the action. The analysis is started by first setting I to the entire set of possible address space represented by the tuples of a rule, and $A_1 = D_1 = \emptyset$. Each rule is then processed sequentially based on the type [Yuan et al., 2006].

For $\langle P, \mathit{accept} \rangle$ rules:

1. $P_j \subseteq R_j \Rightarrow$ good: This rule is valid and defines an action for a new set of packets. There is no overlap with preceding rules.

2. $P_j \cap R_j = \emptyset \Rightarrow$ masked rule: Defines an error in the rule as it does not match any packet because a previous rule has hidden its execution. There are three forms of masking:

shadowing, redundancy, and correlation.

   (a) $P_j \subseteq D_j \Rightarrow$ shadowing: The rule accepts some packets that have been denied by previous rules. This represents a misconfiguration.

   (b) $P_j \cap D_j = \emptyset \Rightarrow$ redundancy: This rule will not expand the access of the policy and therefore is unnecessary.

   (c) else $\Rightarrow$ redundancy and correlation: Some of the packets allowed by this rule have been denied. Others are either accepted or will not follow this path.

3. $P_j \not\subseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule:

   (a) $P_j \cap D_j \neq \emptyset \Rightarrow$ correlation: Some of the packets have been denied by preceding rules.

   (b) $\forall x < j, \exists \langle P_x, \mathit{deny} \rangle$ such that $P_x \subseteq P_j \Rightarrow$ generalization: Rule x matches a subset of the current rule j but defined a different action.

   (c) $P_j \cap A_j \neq \emptyset$ and $\forall x < j, \exists \langle P_x, \mathit{accept} \rangle$ such that $P_x \subseteq P_j \Rightarrow$ redundancy: the previous rule $\langle P_x, \mathit{accept} \rangle$ can be removed and the rule P_j will still accept those packets.

In a similar manner for $\langle P, \mathit{deny} \rangle$ rules:

1. $P_j \subseteq R_j \Rightarrow$ good.

2. $P_j \cap R_j = \emptyset \Rightarrow$ masked rule:

   (a) $P_j \subseteq A_j \Rightarrow$ shadowing: the rule denies some packets that were accepted by a previous rule. This could indicate some sort of security violation.

   (b) $P_j \cap A_j = \emptyset \Rightarrow$ redundancy: This rule will not expand the denied space of this policy and therefore is unnecessary.

(c) else $\Rightarrow$ redundancy and correlation: Some of the packets denied by this rule have already been accepted. Others are either denied or will not follow this path.

3. $P_j \not\subseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule:

   (a) $P_j \cap A_j \neq \emptyset \Rightarrow$ correlation: Some of the packets have been accepted by preceding rules.

   (b) $\forall x < j, \exists \langle P_x, accept \rangle$ such that $P_x \subseteq P_j \Rightarrow$ generalization: Rule x matches a subset of the current rule j but defined a different action.

   (c) $P_j \cap A_j \neq \emptyset$ and $\forall x < j, \exists \langle P_x, deny \rangle$ such that $P_x \subseteq P_j \Rightarrow$ redundancy: the previous rule $\langle P_x, deny \rangle$ can be removed; and the rule P_j will still be deny those packets.

The result of the modeling capability discussed in [Yuan et al., 2006] is not only to detect policy anomalies, but provide the capability to accurately model the system. This concept is important because the policy spaces that result from a rule validation exercise precisely represent the system, but are devoid of the original rules. The research presented in this dissertation will take the model of the policy and allow interesting data to be derived, for example, true difference between two policies and various querying activities against the policy model.

2.3.3 Rule Data Modeling

Firewall rules are made up of five elements: source IP address, source port, destination IP address, destination port, and service. These are all numeric values and are directly mapped from the lower level transport layer represented in the TCP/IP protocol stack. In practice only four of those elements are commonly handled when processing rules, the source IP address, destination IP address, destination port, and service. Therefore, this dissertation will focus only on those elements when modeling and processing a rule. The interesting attribute of those fields is that by representing each of them as binary numbers, a hierarchy emerges that allows for the representation of those numbers in a smaller memory structure than the entire space. For a small example, considering the destination port value, according to the TCP/IP specification [Kurose and Ross, 2007], this value is 8 bits and will be comprised of the integer range 0 to 65535, therefore have an binary representation of:

$$00000000 \rightarrow 11111111$$

Taking this a step further, this number can be represented as a binary tree where the children of a particular node will be a numeric subset of that node. Before arriving at that point, a binary tree is considered where the edges represent a 0 or a 1, a left edge of a node is considered the value 0 and the right edge is considered a 1. Figure 2.1 is a binary tree that represents a set of values contained in the range from 0 to 15.

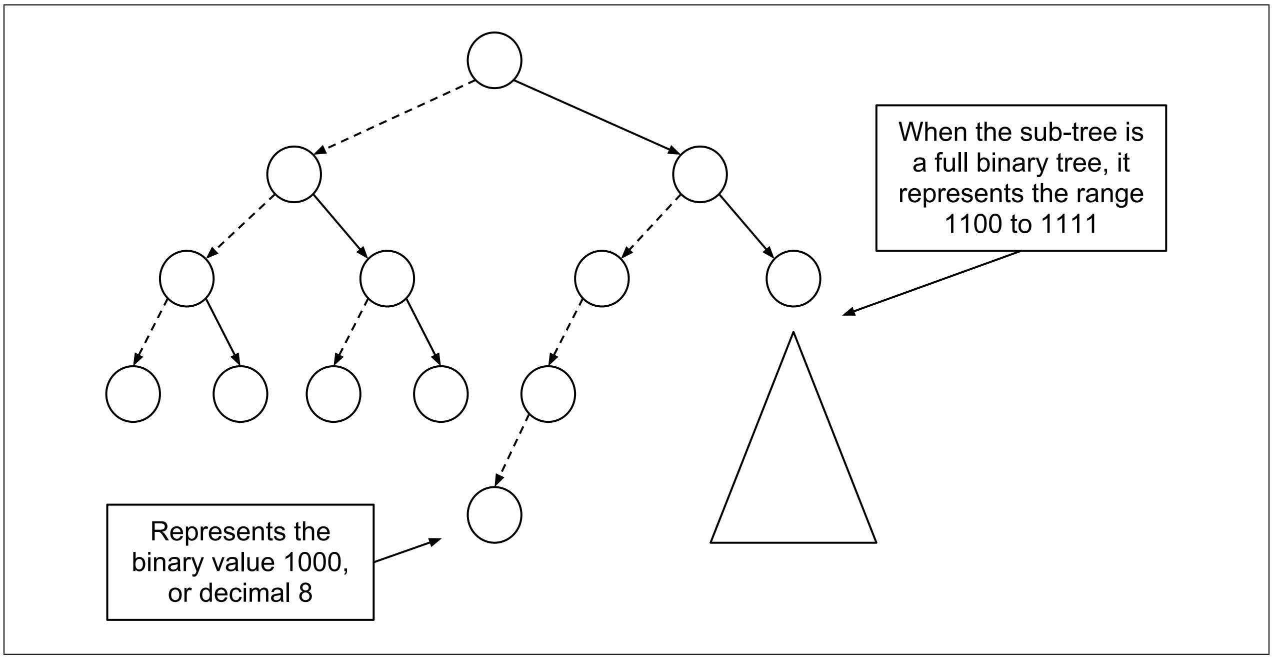

When the sub-tree is a full binary tree, it represents the range 1100 to 1111
Represents the binary value 1000, or decimal 8

Figure 2.1: Hierarchical data set in a binary tree.

2.3.4 Directed Acyclic Graph

A common model used in computer science for representing data is called a graph. A graph is a set of nodes, or vertices, connected to each other by edges. Figure 2.2 is an example graph and illustrates the concept of interconnected nodes by edges. Many problems in computer science can be decomposed into a graph model, and as a result there are many efficient algorithms allowing the manipulation of graph models in very useful ways [Cormen et al., 2009].

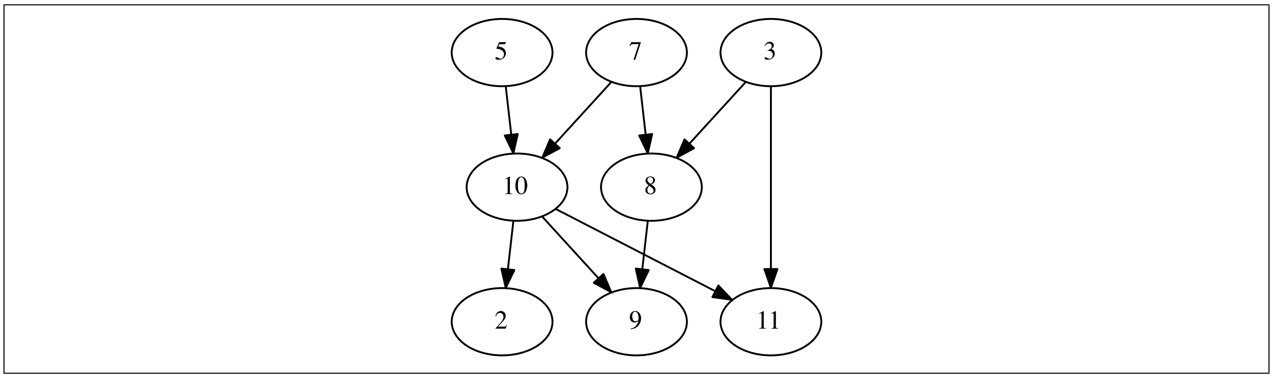
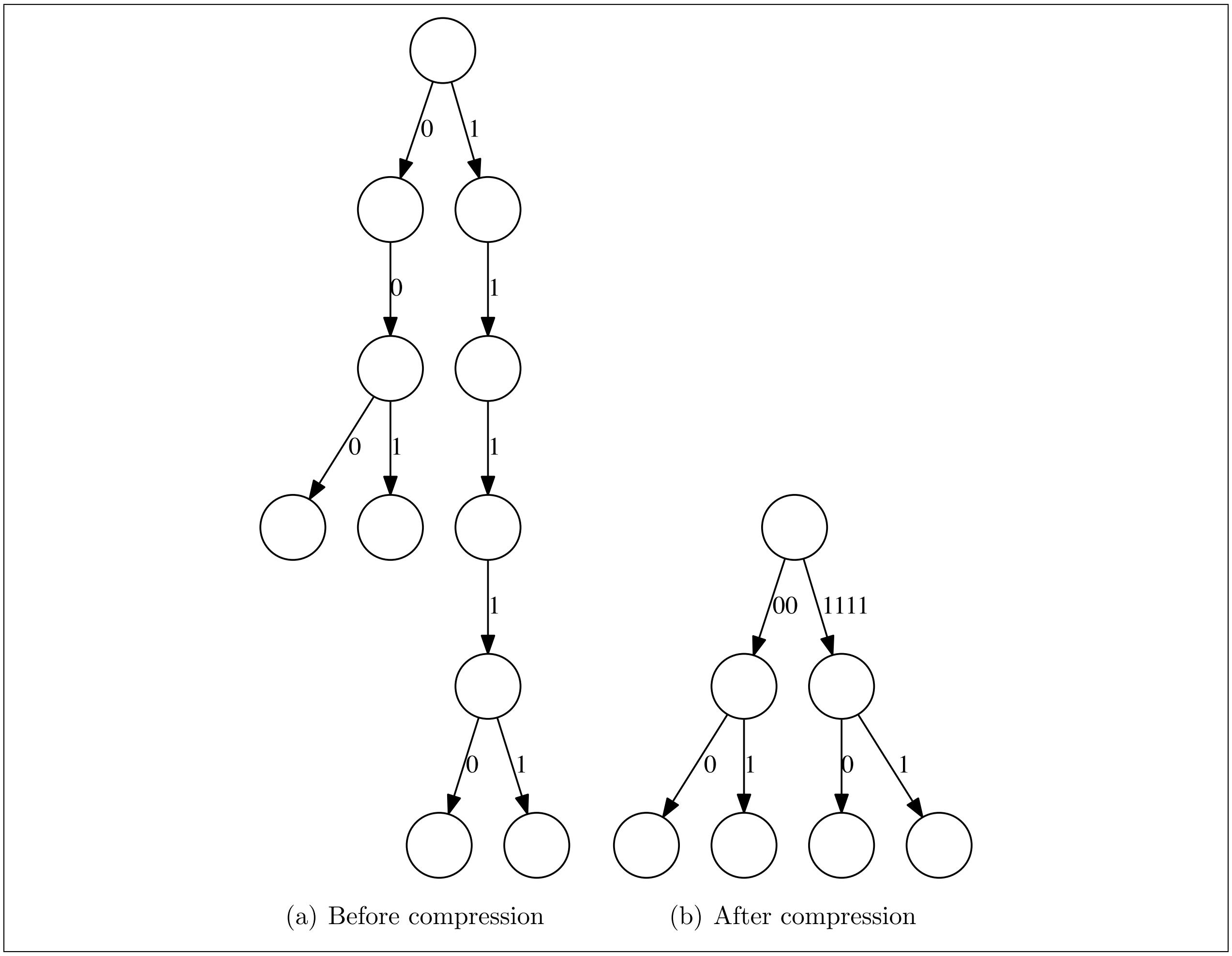


Figure 2.2: Example of a directed acyclic graph (DAG).

A specific kind of graph is one that only allows directed edges that do not result in cycles. This is referred to as a directed acyclic graph (DAG) and Figure 2.2 is also a representation of this type of graph. The edges have arrows indicating the direction of connectivity from one node to another. By following every edge to its stopping point, the same node is never visited twice. This particular property of a graph is called acyclic. A tree data structure is a more specific type of DAG that not only enforces directional edges and zero cycles, it also introduces a hierarchy through the parent vertex and child vertices relationship [Cormen et al., 2009].

2.3.5 Tries

Firewall researchers have recognized the hierarchical nature of a binary IP address, and furthermore, the benefit of representing the number in a binary tree data structure [Hazelhurst et al., 2000]. The focus of these research efforts was to improve the performance of packet classification through matching the incoming packet to a set of filters (or rules). It can be said that a packet matches a filter if each field of the packet matches the corresponding field of the filter. This can be an exact, prefix, range match [Hazelhurst et al., 2000]. A filter in the policy is associated with a non-negative number, *cost(F)*, and the aim of this research was to find the least cost filter matching algorithm when matching a packet's header.



(a) Before compression (b) After compression

Figure 2.3: A standard compression algorithm merging a single branch into a trie node.

Beyond a binary tree representation, two primary techniques were used, namely, backtracking and path compression. Both of these techniques reduced the storage cost by a factor of 4 to 12, and reduced the look up time by a factor of 2 to 5 [Hazelhurst et al., 2000]. The concept of path compression is an improvement over the straight representation of a policy as a binary tree and moves toward the use of binary decision diagrams as an even more efficient data structure. Figure 2.3 demonstrates the effect of path compression such that the fully binary tree seen in Figure 2.3(a) can have the redundant paths compressed into edges that represent the sequence of binary digits. The sequence *00* and *1111* are subsequently reduced and compressed into one edge from the previous multiple edge representation. This compressed representation of a binary tree is called a trie. The effect of the compression operation is to reduce the number of edges to be processed when matching a packet to the trie and can be seen in Figure 2.3(b).

2.3.6 Reduced Ordered Binary Decision Diagrams

The Binary Decision Diagram (BDD) is a graph-based data structure first devised in 1986 as a representation of a boolean equation for use in formal circuit verification, computer aided design, and fault tree analysis. The primary idea of the model is based off of Shannon's expansion, a mathematical method in which a boolean function can be represented by the sum of two sub-functions of the original function [Bryant, 1986, 1992]. By modeling a data set as a boolean function, these systems can be represented as directed acyclic graphs with internal vertices corresponding to the variables over which the function is defined [Bryant, 1992]. A boolean function $f(x_1, x_2, x_3)$ with the truth table defined in Figure 2.4 illustrates this idea.

x1	**x2**	**x3**	***f***
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

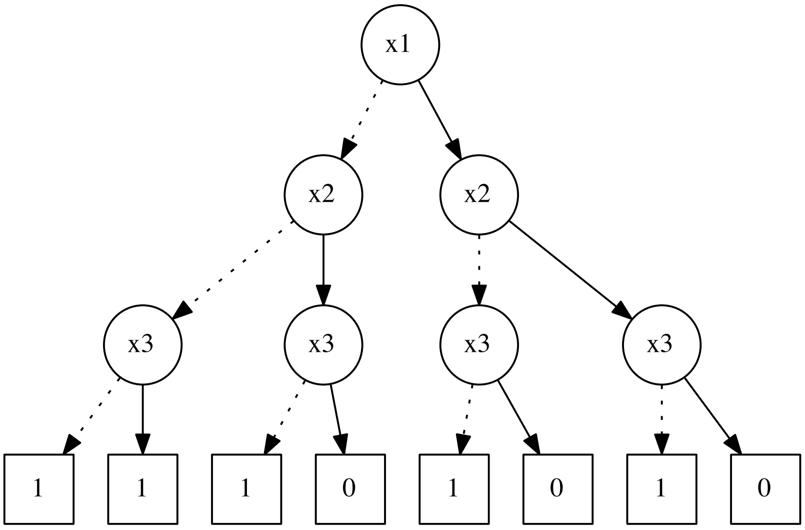


Figure 2.4: Truth table and decision tree representation of a boolean function.

Each non-terminal vertex v is labeled with the variable *var(v)* and consists of two arcs directed to children. There is a low path, *low(v)* (dashed line), and a high path, *high(v)* (solid line), where they represent a 0 and 1, respectively. The resulting truth table is represented by the decision tree and for a given boolean assignment to the variables, the value found at the terminal is what is yielded by the function. For example, if $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$ then the function would yield 1.

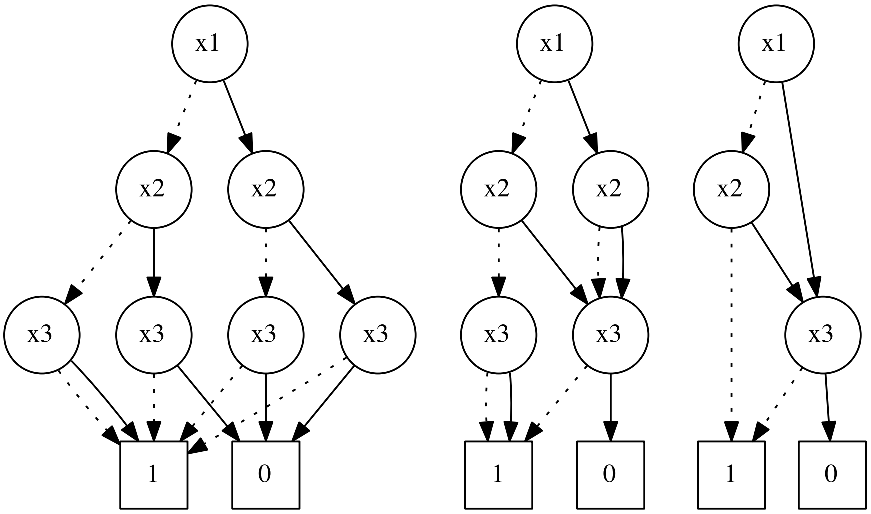
While representing a particular truth table as a decision tree is useful for comprehension, it does not really gain much in terms of space or reduced computing complexity. However, what researchers found is that by ordering the variables and reducing the tree, the representation of the truth table can be much more compact. As a result the Reduced Ordered Binary Decision Diagram (ROBDD) data structure was derived from the Binary Decision Diagram to represent and include these properties. Three transformation rules were defined that allow reduction, and did not alter the function represented [Bryant, 1992]:

- Remove Duplicate Terminals: Since the BDD deals with boolean expressions and truth assignments, the result of a set of variables is either zero or one. Therefore reducing the terminal nodes to two reduces the tree depth by one.

- Remove Duplicate Non-terminals: If non-terminal vertices u and v have *var(u)* = *var(v)*, *lo(u)* = *lo(v)*, and *hi(u)* = *hi(v)*, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex. This has the effect of removing redundant nodes horizontally.

- Remove Redundant Tests: If non-terminal vertex v has *lo(v)* = *hi(v)*, then eliminate v and redirect all incoming arcs to v to *lo(v)*. This has the effect of removing a redundant node vertically. For example, *hi(x1)* points to *x2* and *lo(x2)* = *hi(x2)* = *x3*, therefore redirect *hi(x1)* to *x3*.

One can then start with any BDD satisfying the ordering property and reduce its size through repeated application of the transformation rules. Figure 2.5 shows the progression of the decision tree from Figure 2.4 through the three reduction rules, first removing duplicate terminals, then removing duplicate non-terminals, and finally removing redundant tests.



(a) Remove duplicate terminals

(b) Remove duplicate non-terminals

(c) Remove duplicate tests

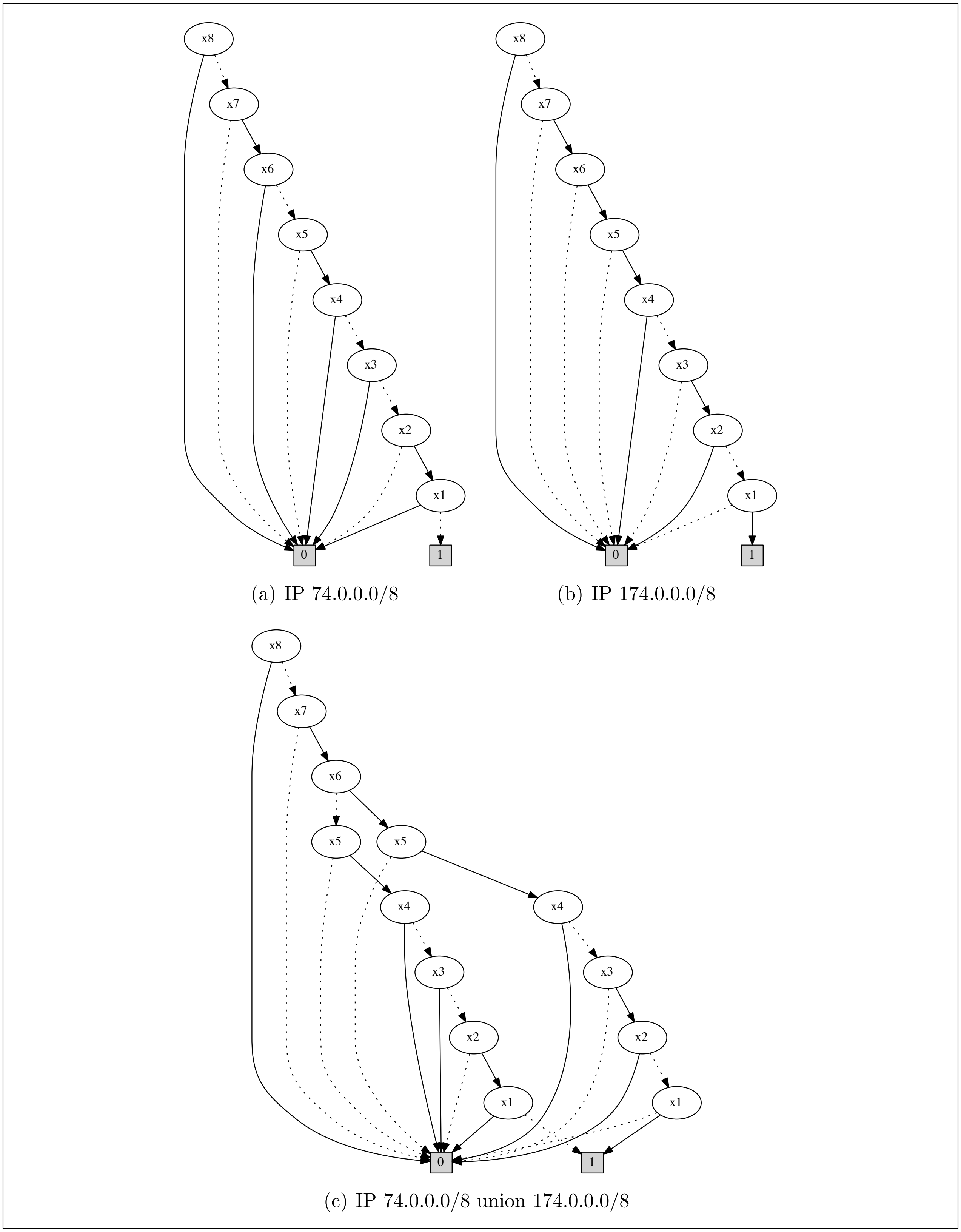
Figure 2.5: Reduction of the decision tree from Figure 2.4 into a ROBDD.

At an abstract level, a ROBDD is considered a compressed representation of sets. This means that membership can be modeled and set operations can be performed on the data structure, but unlike other compressed relationship representations, a ROBDD does not need to be decompressed in order to apply an operation. This is accomplished through two functions that can be performed on a ROBDD, namely, APPLY and RESTRICT [Bryant, 1986]. The APPLY operation allows the generation of boolean functions by applying boolean algebraic operations to other functions. For example, given the functions f and g with the boolean operator $\langle op \rangle$, the APPLY will return the function $f\langle op \rangle g$ [Bryant, 1986]. Using the available operators *and*, *or*, and *not*, over all the variables represented in the ROBDD, other more complicated boolean operations can be derived. The RESTRICT operation allows the ROBDD to be represented in its most canonical form. It will "restrict" the values represented for a particular variable if it is a "do not care" condition, thus having the effect of removing unnecessary nodes as the operation is applied in a depth first walk of the tree [Bryant, 1986]. These available functions are the basis for accurate and efficient operations when applied to understanding firewall access rules [Hazelhurst et al., 2000].

2.3.7 Modeling Firewall Rules as ROBDDs

ROBDDs are designed to represent a set of binary variables that yield a binary decision. With that in mind, the firewall rule can also be modeled in a similar manner by using the firewall rule tuples as the variables in the ROBDD and the resulting binary decision is whether the packet is accepted or denied. In this research, a firewall rule consists of four tuples, source IP, destination IP, destination port, and protocol. Depending on the configuration of the rule, each of these values can be represented as a single number or as a range. Considering the example of a source IP address block 74.0.0.0/8 that is of the Classless Inter-Domain Routing (CIDR) format. A CIDR format allows the expression of four octet IP address with a routing prefix divided by a forward slash (/). The routing prefix is a number 0-32 that identifies how many bits of the IP address are not masked out. Therefore, in the example of 74.0.0.0/8, it means that the first octet is not masked resulting in the representation of the network address space of 74.0.0.0 to 74.255.255.255, or potentially 16,777,216 individual hosts. A naive implementation using straight binary trees for this data set would involve a large number of nodes, however, using ROBDDs each bit in the address can be a variable and the compressed data structure would resemble Figure 2.6(a). Because of the compression properties, only the variables $x_1 \rightarrow x_8$ are represented in the form $x_1'x_2'x_3'x_4'x_5x_6'x_7x_8'$. Figure 2.6(a) and Figure 2.6(b) demonstrate the representation of IP address 74.0.0.0/8 and 174.0.0.0/8 as ROBDD variables.

In addition to efficiently modeling hierarchical data sets, ROBDDs provide the ability to perform set operations such as *intersection*, *union*, and *complements*. For example, Figure 2.6(c) is the union of IP 74.0.0.0/8 and 174.0.0.0/8 and is a canonical form for the union. The representation of firewall rules can then be extrapolated to contain the source IP, destination IP, port, and protocol in such a way that a large address space can be represented in a concise and correct form.



(a) IP 74.0.0.0/8

(b) IP 174.0.0.0/8

(c) IP 74.0.0.0/8 union 174.0.0.0/8

Figure 2.6: ROBDD representation of IP addresses: (a) 74.0.0.0/8, (b) 174.0.0.0/8 and (c) 74.0.0.0/8 union 174.0.0.0/8.

2.3.8 Rule Processing Conclusion

This research utilizes the ROBDD data structure as the basis for firewall rule and policy analysis, comparison, and human comprehension. Taking an abstract policy as it is represented by rules, and then transforming that representation into a compressed data structure capable of efficient mathematical set operations, such as the ROBDD, facilitates investigation into policy differences and other interesting operations. By extending related work with ROBDDs as it relates to firewall rule processing, this research seeks to advance the comprehension of the resulting policy and the ability to extract information from the compressed data structures.

2.4 Backus-Naur Form

In the 1950s two scientists conducting unrelated research efforts generated similar meta-language description forms which became the most widely used method for formally representing a programming language syntax [Sebesta, 2009]. These efforts resulted in the foundation for formalization of how an artificial language can be described in a method referred to as context-free grammar, which is capable of describing the syntax of an entire programming language. In addition, the mathematical correctness of language design is achieved using the generalization of a particular syntax into token abstractions.

During the same time frame, additional language formalization work occurred with the design of the ALGOL 58 programming language. John Backus produced research introducing a new formal notation for specifying programming language syntax and a description using the new *meta-language* for the design of ALGOL 58 [Sebesta, 2009]. In subsequent work, the new meta-language was extended and improved by Peter Naur. The revised method of syntax description serves as the the basis for the Backus-Naur Form (BNF). The context-free grammar and BNF are remarkably similar and have been merged into the same conceptual research area as it relates to programming language formalization.

$$\begin{aligned}
< assign > &\rightarrow < id > = < expr > \\
< id > &\rightarrow \ C \mid A \mid B \\
< expr > &\rightarrow < id > + < expr > \\
&\quad \mid \ < id > * < expr > \\
&\quad \mid (< expr >) \\
&\quad \mid \ < id >
\end{aligned}$$

Figure 2.7: Grammar for a simple assignment statement.

$$\begin{aligned}
< assign > &\Rightarrow < id > = < expr > \\
&\Rightarrow \ C \ = < expr > \\
&\Rightarrow \ C \ = < id > * < expr > \\
&\Rightarrow \ C \ = \ A * < expr > \\
&\Rightarrow \ C \ = \ A \ * \ (< expr >) \\
&\Rightarrow \ C \ = \ A \ * \ (< id > + < expr >) \\
&\Rightarrow \ C \ = \ A \ * \ (C + < expr >) \\
&\Rightarrow \ C \ = \ A \ * \ (C + < id >) \\
&\Rightarrow \ C \ = \ A \ * \ (C + B \ )
\end{aligned}$$

Figure 2.8: Derivation using the BNF grammar from Figure 2.7.

The BNF meta-language uses abstractions for syntactic structures. Compilers then use these syntactic structures to ensure language correctness during compilation to machine code. The abstractions in BNF descriptions, or grammar, are composed of *non-terminal* and *terminal* symbols. The non-terminal symbols are assignments or variables, such as *expr* and terminal symbols represent lexemes and tokens of the rules themselves. Therefore, a collection of these grammar rules comprises a full BNF description. Figure 2.7 is an example BNF grammar for a simple assigment statement.

Taking that simple assignment statement BNF, the conversion of the actual language construct

$$C = A * (C + B)$$

is generated by the derivation shown in Figure 2.8. The assignment statement is read like a derivation, it begins with the start symbol *assign*, and processed through the BNF grammar. Figure 2.8 follows the processing of the language and the symbol $\Rightarrow$ is read as “derives”.

A BNF formalized language benefits from being syntactically correct and statically verifiable. Therefore, the resulting language does not suffer from ambiguity and is better able to represent the expressiveness needed by the language [Sebesta, 2009].

Chapter 3

Firewall Policy Diagram (FPD): Structures for Firewall Behavior Comprehension

Communication security and regulatory compliance have made the firewall a vital element for networked computers. They provide the protections between parties that only wish to communicate over an explicit set of channels, expressed through protocols, traveling over a network. These explicit set of channels are described and implemented in a firewall using a set of rules. The firewall implements the will of the organization through an ordered list of these rules, collectively referred to as a policy. In small test environments and networks, firewall policies may be easy to comprehend and understand; however, in real world organizations these devices and policies must be capable of handling large amounts of traffic traversing hundreds or thousands of rules in a particular policy. Added to that complexity is the tendency of a policy to grow substantially more complex over time and the result is often unintended mistakes in comprehending what is allowed, possibly leading to security breaches. Therefore, it is imperative that an organization is able to unerringly and deterministically reason about network traffic, while being presented with hundreds or thousands of rules.

This chapter covers the Firewall Policy Diagram, a set of structures that is an effort to advance the state of large network behavior comprehension.

3.1 Key Contributions

This chapter presents a novel set of data structures, together called a Firewall Policy Diagram (FPD). These data structures seek to solve the problem of large network access comprehension as it relates to firewall policies in several key areas:

- De-correlation of the firewall policy from the source rule set to gain a holistic view of behavior. This will remove any overlapping rules that will typically exist in a firewall policy [Yuan et al., 2006] and the resulting data structure will model the actual ACCEPT and DENY space.
- Provide the ability to perform arbitrary mathematical set based operations like *and*, *or*, and *not*. These operations will assist in reasoning about firewall policy changes over time. They will also provide the foundation for many other firewall operations, such as understanding the functional differences between two policies. In addition, these operations will also provide the base for querying an arbitrary policy.
- Provide the foundation data structure for the implementation of a method to query the policy.
- Once a policy has been decomposed into an FPD, allow the reconstitution of that policy into a human comprehensible form, like an equivalent policy rule set.
- Finally, the experiments will show that these operations can be executed in seconds of computation time even on large policies (up to 10,000 rules).

The remainder of the chapter is organized as follows: We begin with an overview of the FPD data structure and a description of how it is constructed, operated on, and translated into a set of de-correllated rules. We will then present the results of performance related experiments in terms of creation, SET operations, and reconstitution of firewall policies of various sizes. In the final two sections, related work and conclusions about FPDs will be covered.

3.2 Firewall Policy Diagram

A Firewall Policy Diagram is a set of data structures and algorithms used to model a firewall policy into an entity allowing efficient mathematical SET operations. The entity also has the ability to reconstitute the policy into a set of human comprehensible rules.

Table 3.1 demonstrates an access list that might be defined for a particular organization [Kurose and Ross, 2007]. As shown, firewall policy traditionally consists of a list of rules. These rules are comprised of a subset of the fields in the Internet Protocol version 4 (IPv4) packet as defined by the Open Systems Interconnection (OSI) model [Kurose and Ross, 2007]. In this research effort only certain fields will be analyzed and modeled. The source address, destination address, protocol, and destination port will be used. The decision was made for two reasons, the source port is not often used in most firewall products and the flag is primarily used at layer three for setup and tear down of TCP connections [Kurose and Ross, 2007]. However, if source port is necessary, it is a straightforward process to extend an FPD to include an additional 16 variables.

rule	**action**	**src address**	**dest address**	**protocol**	**dest port**
1	allow	192.168/16	outside of 192.168/16	TCP	80
2	allow	outside of 192.168/16	192.168/16	TCP	> 1023
3	allow	192.168/16	outside of 192.168/16	UDP	53
4	allow	outside of 192.168/16	192.168/16	UDP	> 1023
5	deny	all	all	all	all

Table 3.1: Example access control list for a firewall interface.

The internal storage mechanism of an FPD uses Reduced Ordered Binary Decision Diagrams (ROBDD or BDD) [Bryant, 1986, 1992; Shannon, 1938]. These data structures were introduced as an efficient way to capture hierarchical binary data and related works have described their use in firewall policy validation [Hazelhurst et al., 2000; Ingols et al., 2009; Yuan et al., 2006]. In addition to those that support the use of the BDD compressed data structure, there are research efforts that argue against its use in favor of other combination data structures [Liu and Gouda, 2008]. However, in our work, the BDD provides the efficient storage and the necessary operations that allow our algorithms to reason about policy changes over time and differences between policies. Also, using network address translation (NAT) methods presented by [Ingols et al., 2009], multi-firewall behavior over time can be modeled using BDDs, a missing research component in other policy comprehension work to date.

In a similar manner to the FIREMAN system [Yuan et al., 2006], policies and rules are modeled as variable sets represented as BDDs. Using a BDD is an efficient way to represent a Boolean expression, like $(a \vee b) \wedge c$. Extending this concept to firewall policies, the variables in the expression become the bits of the associated IPv4 field. In this research, 32 bits representing the source address, 32 bits representing the destination address, 8 bits representing the protocol, and 16 bits representing the destination port. This means that for a particular ACCEPT space, there are 88 variables and 2^{88} potential combination of variable values.

3.2.1 Creation and Decomposition

When an FPD is initialized and created, the process begins by iterating over the policy rule set one rule at a time. Each rule is decomposed into its constituent parts relevant to our research: source, destination, protocol and destination port. At this point, each bit of each field is converted into an input vector of an appropriate size for the field. For example, the source field is 32 bits and therefore the vector is of size 32. Once the input vectors are constructed, the four input vectors are appended and added as a constraint in the underlying BDD [Yuan et al., 2006].

3.2.2 Operations

Based on SET mathematics, if a data structure can accurately implement the *union*, *intersection*, and *complement* operation, other operations can be derived. For the purposes of this research and experimentation there are two primary operations that are being studied, namely, Difference and Symmetric Difference.

DIFFERENCE is used in the situation where there exists a base policy P and the desire is to understand what has changed in a later version of the policy, P'.

$$\begin{aligned} \triangle &= P' - P \\ &= P' \wedge \neg P \end{aligned}$$

SYMMETRIC DIFFERENCE is used in the situation where there exists two policies $P1$ and $P2$ and the desire is to know what is not shared between the two policies.

$$\begin{aligned} \triangle &= (P1 - P2) \vee (P2 - P1) \\ &= (\neg P2 \wedge P1) \vee (\neg P1 \wedge P2) \end{aligned}$$

Using these two operations with basic SET functions, we are able to reason about policy discrepancies and understand how one policy is related to another arbitrary policy. In addition, these operations can be chained together to produce a FPD' such that changes to a policy over time or as a set of access flows through multiple policies, the data structure can then be used to extract the resulting allowed (or denied) rules in a human comprehensible form.

3.2.3 Human Comprehension

Using the resulting policy represented by $\triangle$, the procedure to reconstitute that policy into a human consumable set of rules involves transforming the BDD. The first step in the algorithm is to only explore the space necessary for the operation, typically the ACCEPT space. Therefore, starting at the root node the graph is traversed to the accept node and avoids having to explore the potentially large opposite space. During this traversal, the BDD is transformed into human comprehensible rules. To accomplish this sort of rule extraction without having to completely decompress the data structure, two additional data structures and algorithms are used.

The first data structure is a Ternary Tree, which, as the name suggests, consists of three child nodes for every parent. The purpose behind this tree is to take the fully compressed BDD and decompress portions of it without the full cost of the worst case of 2^{88} leaf nodes being represented. There is a low, high, and combination child node, such that the low represents a 0, the high represents a 1, and the combination represents both 0 and 1. The idea behind using a Ternary Tree was inspired by [Bryant, 1986, 1992] and other systems allowing the processing of a "do not care" variable, such as ternary content-addressable memory (TCAM). Therefore, for a particular sequence of ordered variables in a BDD, solutions that occupy the same space (i.e., ACCEPT) have variables of three potential values: 0, 1, or both. The representation of the potential values is *0*, *1*, and X, respectively. The definition of the *0* and *1* are the same as a binary tree, and the variable represented in the node assumes that value. The new edge value of X represents that it is both 0 and 1 for that particular variable at that particular node in the tree. The result is compression in the size of the tree by removing the need for a left and right sub tree.

Figure 3.1 illustrates the concept and what the model would resemble when a binary tree is represented as a Ternary Tree. The tree still maintains the hierarchical nature of the data, but in a more compressed format. There are two important differences as a binary tree transforms to a Ternary tree. The first is the representation of the values of a particular

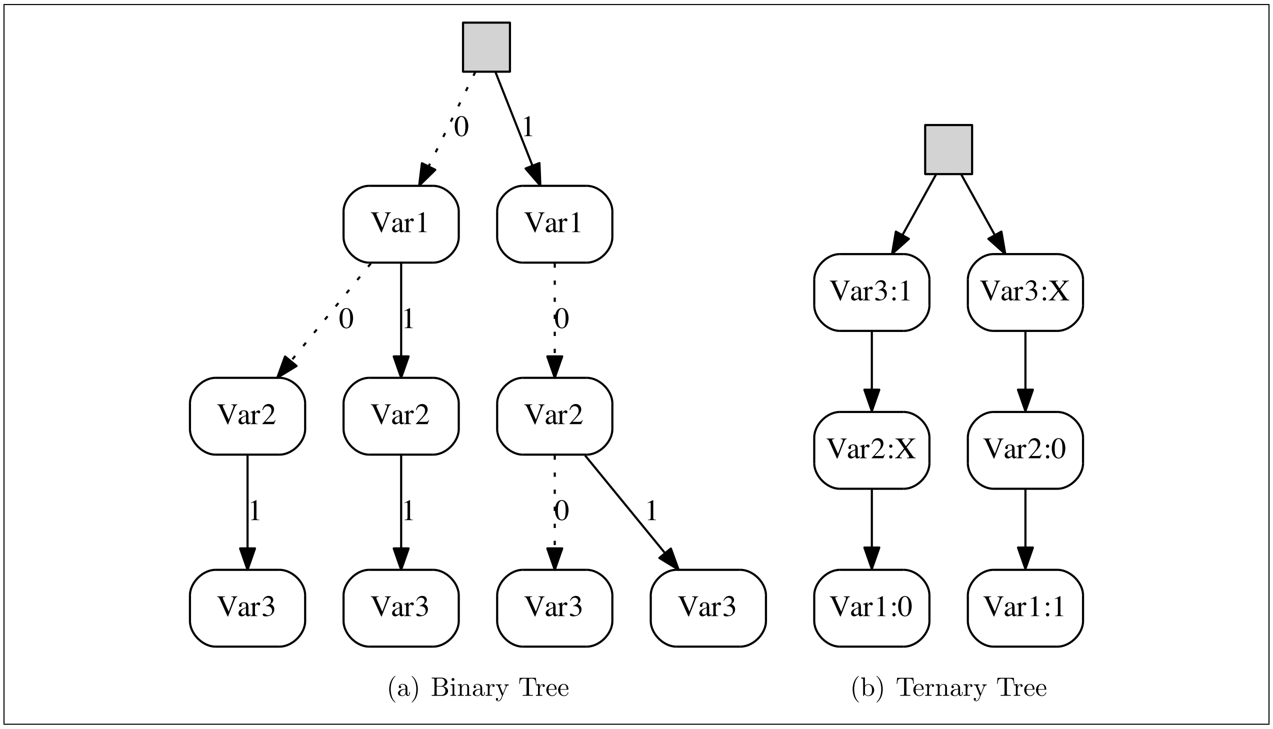


Figure 3.1: Identical binary numbers in a (a) binary tree and (b) Ternary tree.

variable (identified by bit location) is stored with the node. The second difference is the order of the variables in the Ternary Tree, as they are representative of how a tree formed from the algorithm shown in Figure 3.2 resulting in the least significant bit (LSB) variable at the root. These differences allow the tree to be pruned in reverse such that the process begins at the root and generates intervals as it traverses to a leaf. The binary numbers represented in these identical trees are 1, 3, 4, and 5.

The Ternary Tree provides an intermediary between the BDD and the pruned rules by allowing ranges in data to be represented and subsequently combined as the tree is pruned. This is information that cannot be easily ascertained from a canonical BDD representation. The algorithm to transform the ROBDD into a Ternary tree is shown in Figure 3.2.

An additional concern is that a true tree structure has the potential to have a higher storage cost because of replicated data nodes in child trees when the pattern could be shared where appropriate. This has been addressed in the Ternary Tree by allowing it to share nodes where the underlying pattern is shared. The resulting rule set deals with any collisions that may occur when pruning by copying intervals. For example, if the pattern starting at a certain variable is common and shared by parent variables, then the Ternary Tree will share those nodes by an edge pointing to those nodes. This accomplishes the goal of reducing space requirements without sacrificing the expressiveness of the data structure.

```
Input: ROBDD $Bdd$
Output: A fully formed Ternary Tree $T$
procedure TRANSLATE($Bdd$)
                                                        ▷ Bdd Variables labels start with 0
    $T \leftarrow newTernaryTree$
    for all $R \in RootNodes(Bdd)$ do
        $N \leftarrow createTernaryRoot(T)$
        WALKEDGE($R.low, N$)
        WALKEDGE($R.high, N$)
    end for
end procedure
procedure WALKEDGE($bN, tN$)
    for $bN.parent.var$ to $bN.var - 1$ do
        $tN \leftarrow tN.middle \leftarrow newTernaryNode(X)$
    end for
    if $bN.var = Bdd.One$ then return
    end if
    if $bN.low \neq Bdd.Zero$ then
        $tN.left \leftarrow newTernaryNode(0)$
        WALKEDGE($bN.low, tN.left$)
    end if
    if $bN.high \neq Bdd.Zero$ then
        $tN.right \leftarrow newTernaryNode(1)$
        WALKEDGE($bN.high, tN.right$)
    end if
end procedure
```

Figure 3.2: Algorithm for translation of a ROBDD to Ternary tree.

3.2.4 Pruning the Ternary Tree

The Ternary tree acts as an intermediate data structure where the primary purpose is to allow a second algorithm to collapse the tree into a set of human comprehensible data structures. The second algorithm is the pruning procedure that starts at the place-holder root of the

```
for $i = Min;\ i <= Max;\ i += \ 2^{Conversion\ Factor}$ do
    $Value \leftarrow i$
end for
```

Figure 3.3: Algorithm for expressing all interval values.

Ternary tree and then traverses the tree to the leaves, pruning and generating intervals. The resulting data structure captures an interval and a count of bits in a conversion factor. The interval has a starting and ending number, with the conversion factor identifying how the interval sequence progresses. As an example, the source IP address Ternary Tree that represents an odd number of IP addresses is considered. In this example, the interval would be the starting and ending values of the range with a conversion factor of 1. All individual values in an interval can be expressed using the procedure shown in Figure 3.3.

As the tree is walked to the heuristically defined leaves of the tree, the interval is transformed by bit shifting and sometimes cloning of the interval to represent a transformation from a hierarchical data set into a rule. Additional details of the algorithm are identified in Figure 3.4. As the algorithm progresses, a number of these intervals are captured and get merged when appropriate. The internal representation of the ranges makes use of a red-black tree algorithm [Cormen et al., 2009] to maintain a balanced structure while the intervals are assembled into a human readable form.

Notably, the entire rule is represented on the originating BDD; and therefore heuristics are used to help separate the data and make the rules more human comprehensible. This means that for a particular policy model, the data structure represents the concatenation of the source IP, destination IP, destination port, and protocol. The algorithms will separate the data structure into three separate Ternary Tree boundaries during the processing of the algorithm, namely, a source IP boundary, destination IP boundary, and service boundary. The process of extracting human comprehensible rules from an FPD runs at $O(V+E)$ where V and E are the number of vertices and edges in the Ternary trees, respectively.

The upper bound on the number of copies an interval must endure is 16. This is because a copy must occur when a variable transitions from 0 or 1 to X, and for a 32 variable number that can only occur a maximum of 16 times. This is also an upper bound on the number of intervals that could potentially exist at 2^{16}.

The final useful function achieved by using the interval data structure with conversion factor is a simple way to determine the number of addresses, ports or services in a particular rule. For an interval in a rule, it results in the equation:

$$\left[\frac{(Interval\ Maximum - Interval\ Minimum)}{2^{\ Conversion\ Factor}}\right] + 1$$

The result of the pruning is a list of de-correlated rules with three intervals, the source IP range, destination IP range, and service range. An interesting result of the way that we have chosen to order the BDD variables is that the source IP to destination IP variable transition drives the number of distinct rules present in the final reconstituted rule set. This means that when that boundary in the variables is crossed while traversing the Ternary Tree, a new rule is generated and the remaining destination IP and service ranges are collected in that rule. Additional research could be done using the BDD variable reordering capability to find the most concise rule set representing a particular policy.

3.2.5 Heuristics Applied to Policies

Knowledge about the data set being modeled is important to the conversion and pruning algorithms. The heuristics provide the knowledge about where the hierarchical data sets begin and allow the summary of those data as the tree is pruned.

In this work the size of each of the fields drives the separation of the trees such that the 32nd, 64th, and 88th variables divide the hierarchy of a solution into the source IP address, destination IP address, and service (a combination of protocol and port). Notably, these algorithms can be applied to other hierarchical data sets in different domains. They may be especially useful when dealing with large solution spaces and multiple hierarchies being combined to provide the composition of a desired “space”.

```
Input: Ternary Tree
Output: List of Rules that composed the SPACE
procedure PruneRules($T$)
    for all $L \in Children(T.root)$ do
        $Interval \leftarrow create(from = 0, to = 0)$
        $Depth \leftarrow 0$
        ParseNode($L, Interval, 0$)
    end for
end procedure
procedure ParseNode($N, Interval, lnVal$)
    if $N.value = X$ and $Interval.factor\ empty$ and $lnVal \neq X$ then
        $Interval.factor \leftarrow Depth$
    end if
    if $N.value = X$ then
        $Interval.to \leftarrow to \mid 1 << depth$
    end if
    if $N.value = 1$ then
        $Interval.from \leftarrow from \mid 1 << depth$
        $Interval.to \leftarrow to \mid 1 << depth$
    end if
    if $N$ is boundary then
        $Rules \leftarrow Interval$
        $Interval \leftarrow create(from = 0, to = 0)$
        $Depth \leftarrow 0$
        ParseNode($N.parent, Interval, 0$)
    else if $N\ not\ root$ then
        $Depth = Depth + 1$
        if $lnVal \neq X$ and $N.value = X$ then
            $Interval' \leftarrow clone(Interval)$
            ParseNode($N.left, Interval', N.value$)
            ParseNode($N.right, Interval', N.value$)
            ParseNode($N.middle, Interval', N.value$)
        end if
        ParseNode($N.parent, Interval, N.value$)
    end if
end procedure
```

Figure 3.4: Algorithm for rule extraction

3.2.6 Generalized Example

This section will step through an example of how the algorithms of the FPD function on a smaller solution space in an effort to both show how the boundary heuristics may be applied to other domains, as well as a better description of how the algorithms function. A example is considered where we represent the solution space as 2^8, with a fictitious rule being made up of two fields of 4 binary variables, A and B, which subsequently form an 8 binary variable solution space. As the stored ROBDD is generated, A will represent the decimal values 2 through 12 and B will represent the decimal values 3 through 13. Figure 3.5 visualizes the canonical ROBDD data structure representing A and B with the LSB at the root, and referred to as variable zero. As the ROBDD is traversed from root to leaf, the tree is transformed into the graph as seen in figure 3.6. The variable transition values *high* and *low* are shifted to the variable values in the individual nodes. The new root node with label *1* is created and the tree is rooted at that node. In addition, the solution space removes the need for the edges leading to the *zero* value as we do not care about those numbers when extracting the stored data. The change from one tree to another is what is described in the algorithm shown in Figure 3.2.

As indicated in earlier explanations of the Ternary tree, it acts as an intermediary data structure as human comprehensible intervals are extracted. Therefore, the next step of process is to traverse the Ternary Tree from root to leaf in an effort to generate a set of *rules* with the described intervals that make up those rules. We have already identified our single boundary heuristic in this generalized example as variable 4 of our 8 variables (using a zero index). Therefore as the algorithm shown in Figure 3.4, begins with that knowledge by traversing the root to leaf in an effort to create intervals.

The initial result is two groupings or *rules* being generated. This can be visually confirmed in the Ternary tree with the existence of two nodes for variable 4. Subsequently, in each rule there are two sets of intervals, those representing the variables 0 through 3 and those representing the variables 4 through 7. Rule 1, segment 1 (variables 0-3):

$$2 \textit{ to } 10 \textit{ every } 2^3$$

$$3 \textit{ to } 11 \textit{ every } 2^3$$

$$4 \textit{ to } 12 \textit{ every } 2^3$$

$$5, 6, 7, 8, 9$$

Rule 1, segment 2 (variables 4-7):

$$4 \textit{ to } 12 \textit{ every } 2^3$$

$$6, 8, 10$$

Thus, the intervals may be merged into segment 1 (variables 0 through 3) being represented by the decimal numbers 2 through 12. Subsequently the segment 2 interval may be merged into 4 *to* 12 *every* 2, so *even* numbers 4 through 12. Based on a similar interval merging procedure for Rule 2, segment 1 (variables 0-3) becomes:

$$2 \textit{ to } 10 \textit{ every } 2^3$$

$$3 \textit{ to } 11 \textit{ every } 2^3$$

$$4 \textit{ to } 12 \textit{ every } 2^3$$

$$5, 6, 7, 8, 9$$

Rule 2, segment 2 (variables 4-7):

$$3 \textit{ to } 11 \textit{ every } 2^3$$

$$5 \textit{ to } 13 \textit{ every } 2^3$$

$$7, 9$$

The intervals may then be merged into segment 1 being represented by the decimal numbers 2 through 12. Then segment 2 may be merged into 3 *to* 13 *every* 2, so *odd* numbers 3 through 13.

The final merge may then be reviewed between two rules such that when a segment of a rule matches another segment, as it does with segment 1, it may become one rule with the segment that is not matching (segment 2), being combined and reviewed for merges. Therefore, after rule 1 merges with rule 2, and since the segment 2 intervals represents overlapping odd and even ranges, i will become a contiguous decimal interval from 3 to 13. This reconstitutes our original rule that created the BDD: A = 2-12, B = 3-13; which is fit for human comprehension.

This algorithm is in contrast to a more naive, brute force approach to processing all known "solutions" to the BDD. In the example presented here, there are 64 8-bit numbers that will need to be parsed back into the known intervals that created the BDD from the start. While that number may seem small, the example and solution space are small by design. The important difference is between the number of solutions and the number of nodes in the Ternary tree. In addition, the brute force method removes the hierarchical relationships between the nodes, i.e. the heuristic value of node 4, further complicating reconstituting related intervals.

3.2.7 De-correlation

When a list of rules in a policy is decomposed into an FPD, a reconstituted policy that covers the same equivalent ACCEPT or DENY space will result in a rule set with none of the resulting rules overlapping in any area. The primary reason for this behavior is that as a policy is decomposed one rule at a time, all inconsistent or overlapping rules are removed and

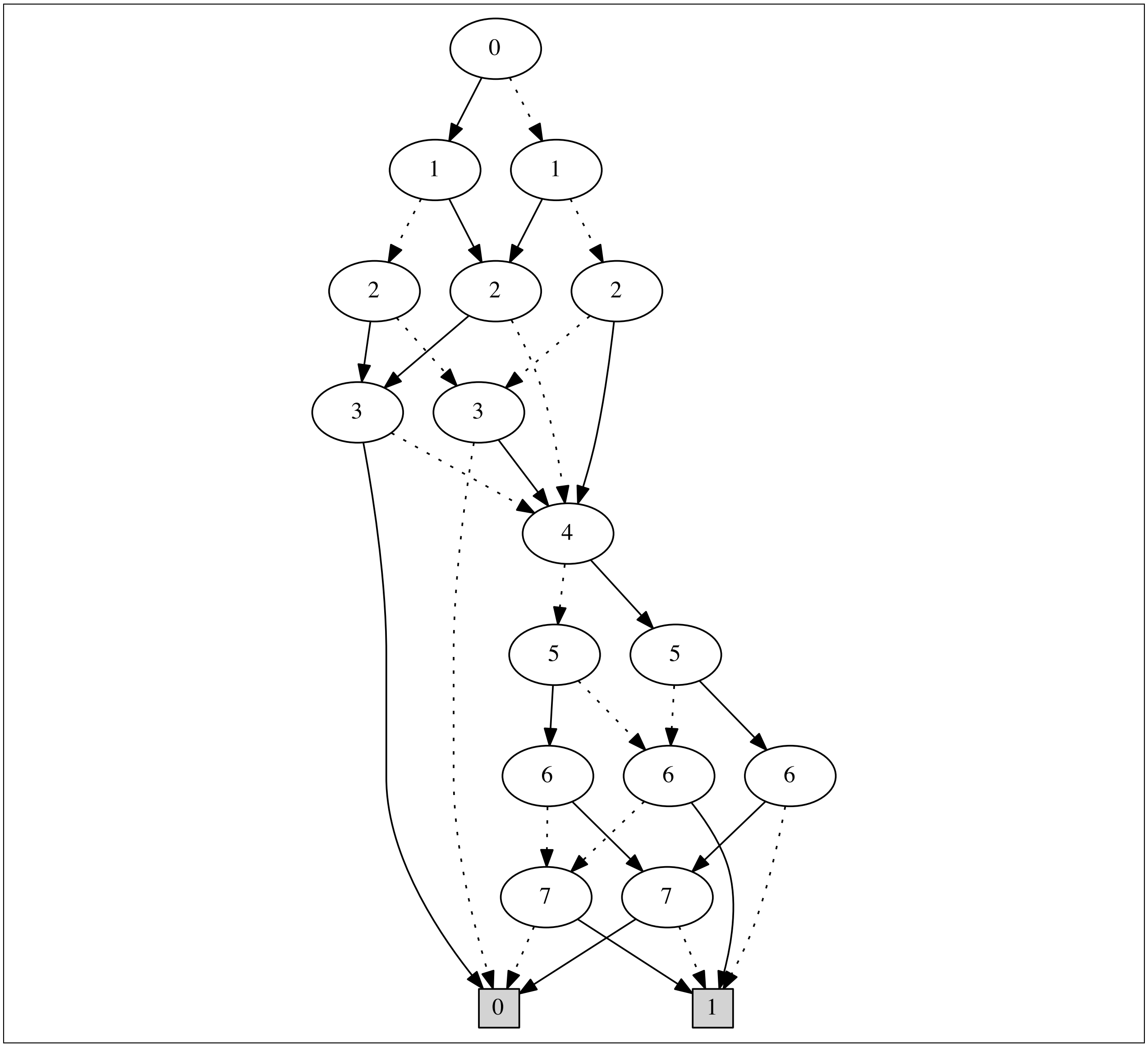


Figure 3.5: ROBDD generalized example.

just the space is represented. The de-correlation property is useful in a number of scenarios:

- A policy no longer has a need to be processed as an ordered set of rules, since the FPD removes any overlapping rules. As a result, if the FPD is built by the rules in the policy from last to first, the resulting system can match an incoming packet to all rules simultaneously.
- A policy may be substantially smaller and take much less time to process once it has been de-correlated. This behavior is the effect of the procedure that converts rules into

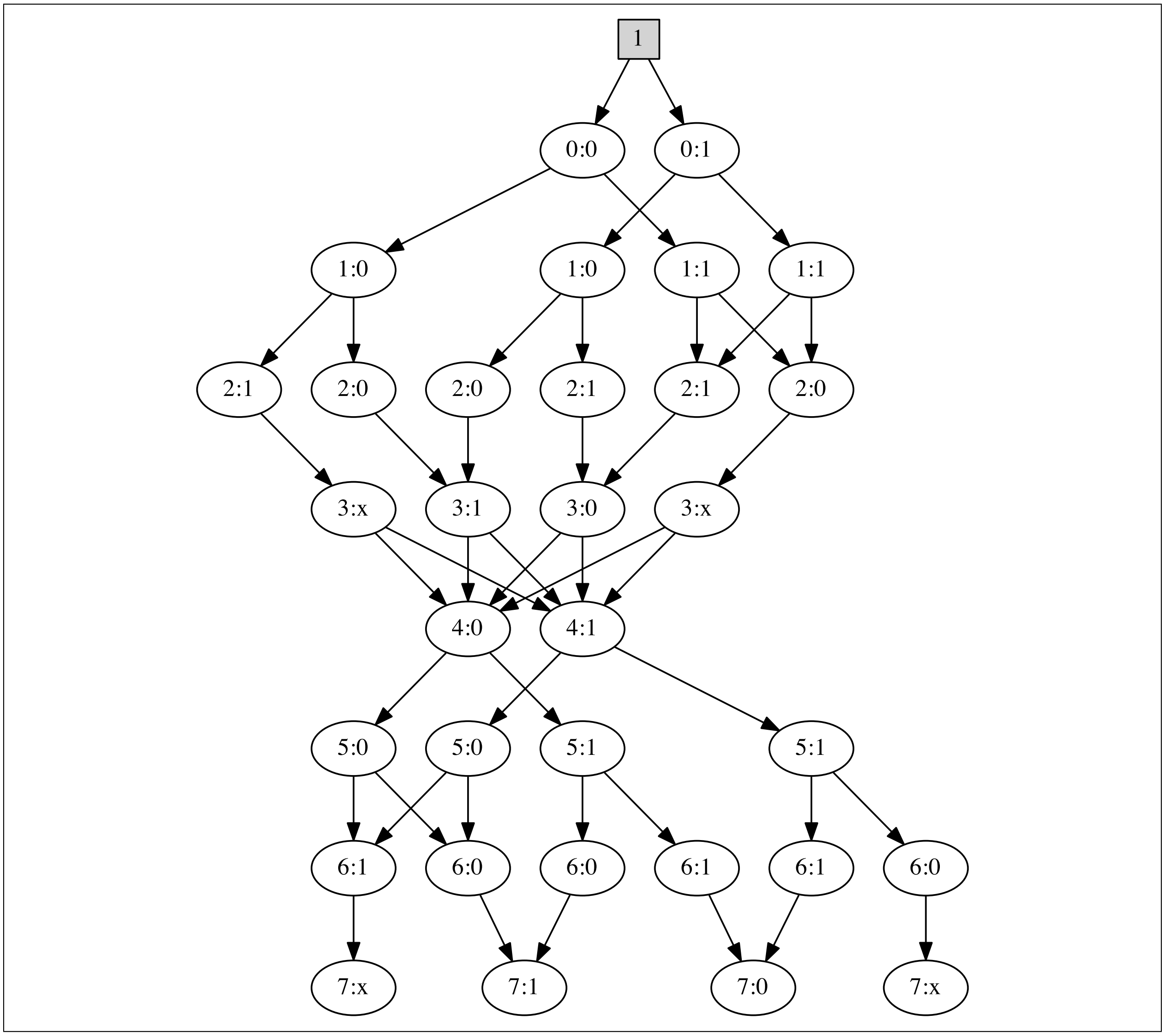


Figure 3.6: ROBDD as Ternary tree.

the FPD where it has also merged adjacent rules and removed any redundancies. In addition, the matching operation of a rule can be performed in constant time. This is because matching a rule involves walking the data structure from root to result, which is a constant 88 elements.

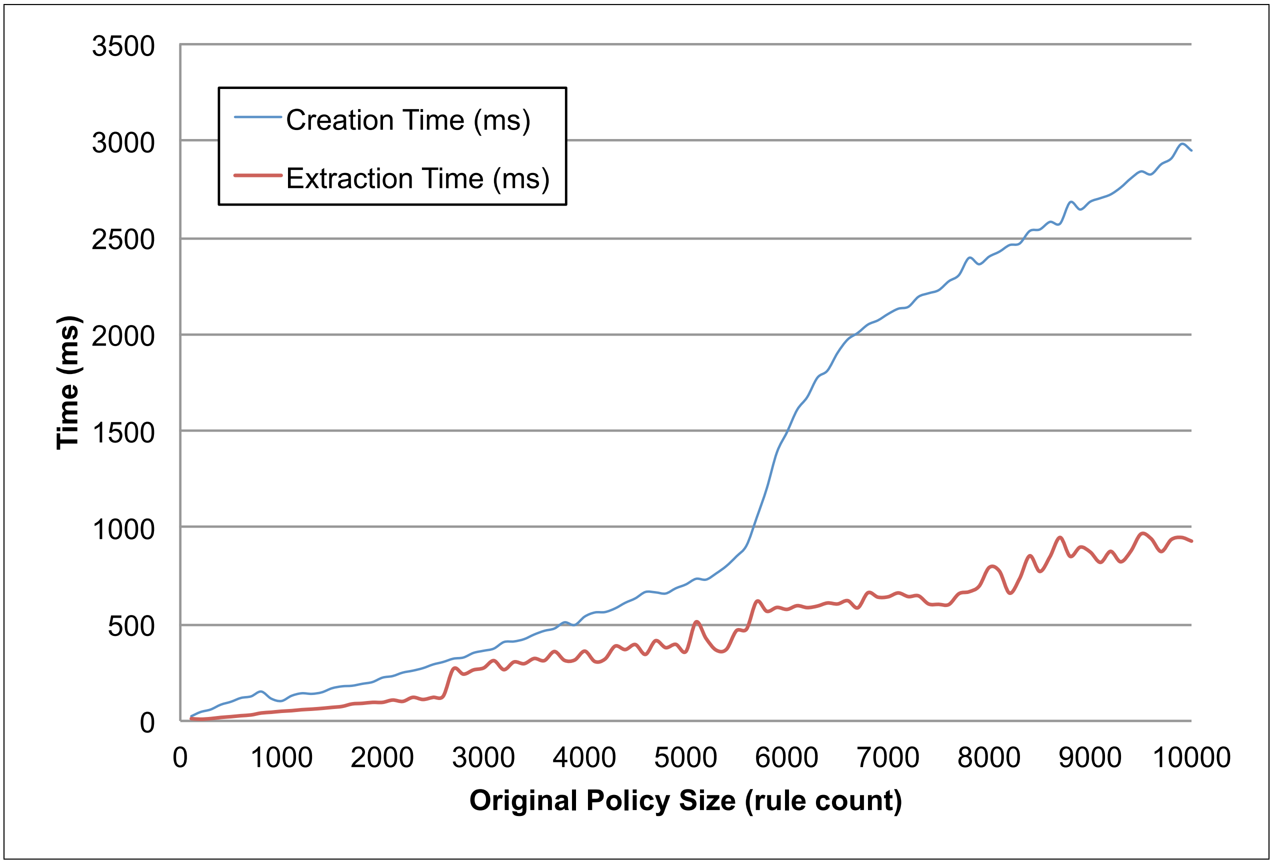


Figure 3.7: FPD generation and extraction of rules for policy sizes 100 to 10,000 rules.

3.3 Experiments

The experiments designed for our work seek to review and address problems seen in the outside industry, i.e., large and difficult to understand firewall policies. We will first measure the time required to construct an FPD from rule sets of various sizes. We will then measure the time to extract a set of human comprehensible rules from an FPD. Finally, we will review the results of the DIFFERENCE and SYMMETRIC DIFFERENCE operations between two policies that share from 0% to 90% of the same space. The FPD data structure and algorithms are implemented in the Java^{TM} 6 programming language and the tests are run on a Mac Book Pro laptop computer. For ROBDD software, the Buddy and JavaBDD libraries were used [Lind-Neilsen, 2004; Whaley, 2007].

For the data sets used to test creation and extraction of rules, policies of sizes 100 to 10,000 with 100 rule increments were created. When testing the DIFFERENCE and SYMMETRIC

DIFFERENCE operations, we created two randomly generated 200 rule policies that share from 0% to 90% of the same space. Security reasons prevented us from being able to gain access to actual industry rule sets, as the majority of companies with firewall policies of those sizes are hesitant to allow outside parties access.

Figure 3.7 charts the performance of the FPD data structure for creation of a policy from a set of rules through to extraction of the policy into an equivalent set of rules. For creation of the FPD from a set of rules, the performance is consistently below one second for policies up to 5,500 rules. The creation times then begins to take more than a second until finally approximately 3 seconds to create a policy that originated from 10,000 rules. The extraction stays consistently less than a second to produce an equivalent set of rules and stays less than 500 ms for the majority of the data sets.

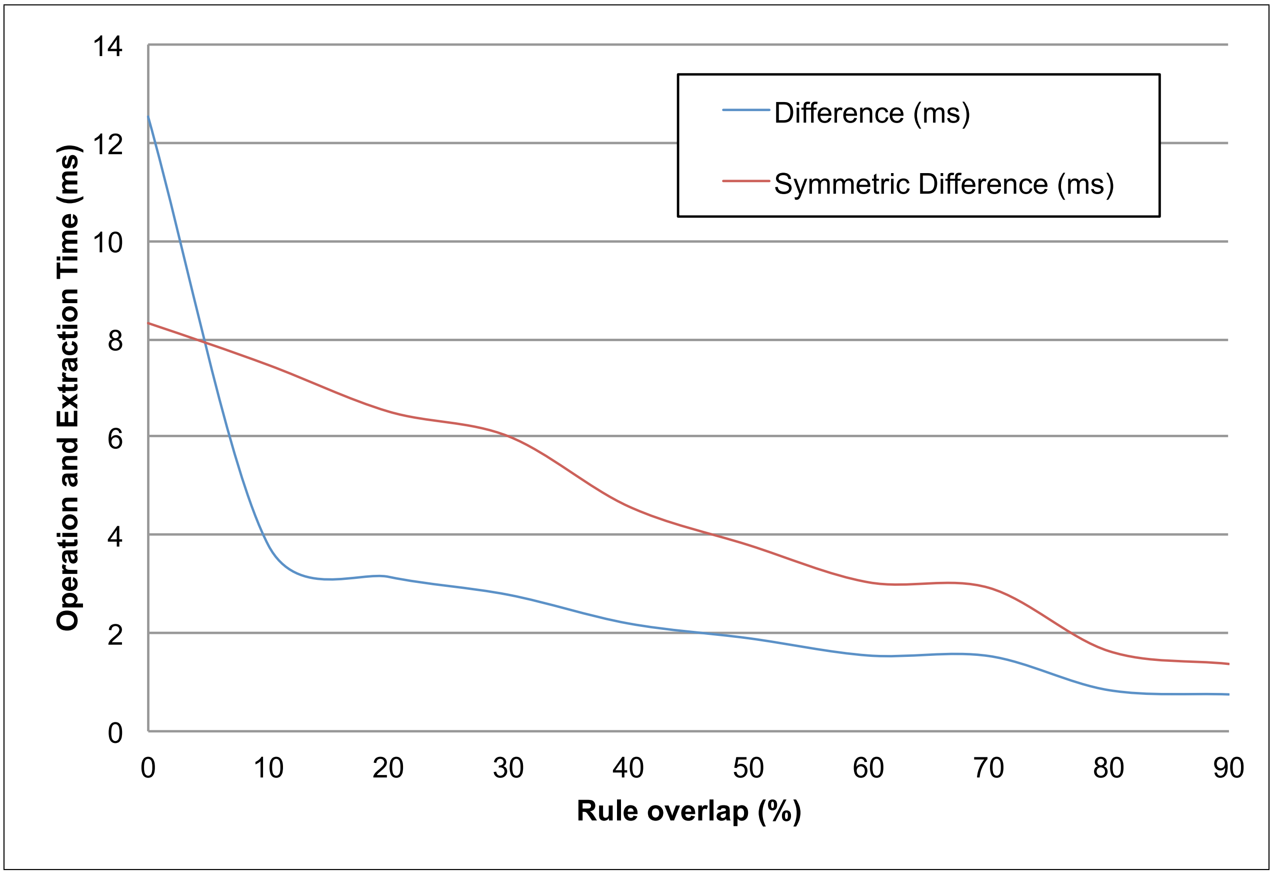


Figure 3.8: FPD difference and symmetric difference operations for a 200 rule policy.

Figure 3.8 charts the performance of a DIFFERENCE operation and SYMMETRIC DIFference operation between two 200 rule policies. The experiment involved executing the appropriate operation; and then extracting the human comprehensible rules from the FPD. The performance of the algorithms takes the most time when the overlap of the space is zero or very small. This is indicative of those results being the largest resulting solutions space but also demonstrates the algorithms are fast enough to be used in larger simulations and larger solution spaces. Symmetric difference operations exhibited slower average processing performance due to the size of the resulting space represented by that operation. Difference operations yielded smaller resulting data set sizes and reflected that fact in the computation times.

3.4 Related Work

The first work to discuss binary decision diagrams in the context of solving interesting problems, such as constraint satisfaction (SAT) and boolean expressions is [Bryant, 1986]. This is the basis for some of the previous algorithmic discussions and covers a couple of important items. The first are the manipulation algorithms necessary to accomplish ROBDD behaviors reduce and apply. The behaviors are the basis for a compressed data structure, thus saving space; and the mathematical set operations and and or, thus providing the more advanced set manipulation. These operations are described in full and the paper covers the binary manipulation that is based on the mathematical concept called the Shannon expansion [Bryant, 1986; Shannon, 1938]. The second item is a discussion of the worst case running time, and how a compressed data structure typically does not hit the worst case of a fully decompressed structure because it is not required to touch every decompressed node in the ROBDD. Thus, the time complexity of ROBDD operations is proportional to the size of the graphs being manipulated. A later work, advanced the state of the binary decision diagram by introducing variable ordering and reducing operations for further efficiencies of the model [Bryant, 1992].

Qiu *et al.* (2001) were interested in how to implement data structures in the firewalls for the most efficient processing of the packet as it entered the inbound interface. The research focused on how to implement the algorithms that made packet analysis data structures as fast as possible. The work combined backtracking search backed by database technology and pruning tries. The techniques allowed routers based on software and hardware to improve how packets are analyzed [Qiu et al., 2001]. So, while this effort did not expand on policy comprehension, the work provides some background on how existing devices quickly process packets and serves as the basis to differentiate between packet matching and policy comprehension. It also introduced modeling binary numbers as fast lookup trees.

One of the first papers to address large rule sets was published in 2000 [Hazelhurst et al., 2000]. It is the primary data structure paper that has allowed the analysis of large and complex rule sets in much faster time. Essentially it is a demonstration of how ROBDDs are used to improve the look up algorithm without sacrificing performance; and allowed for the completely accurate validation of rules sets to ensure that the ACCEPT and DENY spaces were as desired. This is the first paper to discuss using a compressed data structure to handle the IPv4 large search space. In addition, the paper built a query engine on top of a formal verification system named Voss [Hazelhurst et al., 2000]. They implemented a simple functional language that is capable of modeling a firewall policy in a formal verification system. While the underlying Voss hardware verification system used ordered binary decision diagrams to model a rule set, it is unclear as to the method in which the results were extracted from the canonical BDD form. It appears that a brute force method was implemented to parse through the entire solution space resulting from a query. The algorithm complexity was reported to be linear with respect to the number of rules for creation of the policy and constant time with respect to the number of variables in the BDD for any SET operations on the policy [Hazelhurst et al., 2000]. This is consistent with our work on processing FPDs. However, that analysis neglects to take into account the processing time related to extracting human comprehensible data from a binary decision diagram. This is a large segment of processing and could involve millions of individual BDD solutions. Our analysis shows that the complexity should have been a function of the resulting BDD and the number of distinct solutions, a potentially much larger complexity that is not correlated to the number of rules.

As research progressed on the use of ROBDDs for modeling policies, additional work was done using a new data structure called a Firewall Decision Diagram (FDD) to simplify a rule set. This first paper appears to be a blend of the ROBDD reduction algorithms to remove unnecessary rules from the policy [Gouda and Liu, 2004]. Of note, the FDD does not seek to provide the most canonical form of a policy space. As such, the FDD is not a compressed data structure. Furthermore, the work is strictly focused on modeling a security policy and is not shown to represent routing space or NAT. In a subsequent work, they used a special type of FDD called the Firewall Decision Tree (FDT) for use in detecting redundant rules [Liu and Gouda, 2005]. As indicated, this is an alternate means outlined by [Hazelhurst et al., 2000] and instead of using ROBDD data structures, devises a new FDD derived data structure.

Gouda and Liu further expanded on their previous work with the FDD data structure to allow a policy to be compared to another similar policy, and rules to be extracted from the merging of those two policies [Liu and Gouda, 2004]. The ideas are rooted in the premise of better methods to design firewall rule sets and attempt to apply programming techniques and static analysis used in redundant system design to find the best solution to a particular overall security policy design [Engler et al., 2001; Feamster, 2004; Feamster and Balakrishnan, 2005; Liu, 2007; Mahajan et al., 2002; Maltz et al., 2004; Spring et al., 2004; Sung et al., 2009]. It is not clear in the original work if the algorithms could be used for arbitrary policies to be compared, nor what sort of information comes from the comparison, just that it could be used to determine the differences of when a particular decision for a packet is different that another policy [Liu and Gouda, 2004]. In later experiments from the same FDD data structure, they further describe FDDs as being able to compare two firewalls that have been synthetically changed to have 0-50% of the rules changed or removed; however the two policies always had the same origin [Liu and Gouda, 2008]. It is in this work that they specifically state problems with using a ROBDD as the core of the data structure [Liu and Gouda, 2008], however our work seeks to overcome those concerns.

The FDD data structure is most functionally similar to what the FPD seeks to achieve [Gouda and Liu, 2004; Liu and Gouda, 2005, 2008]. The goal of the data structure is similar to ours, but they chose a different internal representation of a policy, specifically a combination of prefix and interval trees with additional data structures. It is capable of some DIFFERENCE operations, and is portrayed as a tool in which to achieve more accurate firewall design and change impact analysis. In addition, it is not clear that they are capable of handling more complicated situations, such as network address translation or routing tables, key components in analyzing modern firewall configurations. In addition, the data structure is very specific to firewalls and it does not appear to handle arbitrary mathematical SET operations or arbitrary hierarchical data sets, both functionalities the FPD is capable of modeling.

Wang *et al.* (2006) and Yin *et al.* (2006) also developed a comparison model between two policies. The work introduces a new data structure called Firewall Policy Tree (FPT), but appears to be a derivation of the FDT. The efforts take a firewall policy and transforms it into a FPT, and from that state it is able to be compared to another FPT to discern the differences between the two policies [Wang et al., 2006; Yin et al., 2006]. The resulting construction and comparison algorithms appear to be less efficient than our work. The work also does not cover NAT or routing related to modern firewall technology.

Horowitz and Lamb (2009) also designed a policy extraction framework for taking a particular policy rule set and de-correlating any overlapping, or correlated, rules. The data structures are not covered, but the algorithm is a rule by rule comparison for those rules that are deemed as correlated. How the correlation is determined is not clear, but the assumption is that related work on correlated rule sets is used. This is another step in the direction of human comprehension of the difference between two arbitrary firewalls and introduces the notion that a rule set does not have to be ordered if the rules are all completely disjoint and de-correlated. The methods are strictly for de-correlating rules with an exponential running time correlated to the size of the original policy and no overall policy comprehension [Horowitz and Lamb, 2009].

Other work has been done modeling firewall policies, however, most of the models reflect their intended use and not all are capable of the sort of operations described in the FPD. Much of the research has focused on rule processing and validation of those rules where the goal is to identify hidden, shadowed, and inconsistent rules [Al-Shaer and Hamed, 2002, 2004a,b; Bartal et al., 1999; Chao, 2011; Eronen and Zitting, 2001; Golnabi et al., 2006; Gouda and Liu, 2004; Liu and Gouda, 2005; Yuan et al., 2006]. In general the focus in those efforts is on algorithms for finding policy anomalies both from a single policy model to a multi-policy model. A portion of the related research introduced the use of BDDs for the models and became the foundation for some of the algorithms in our work [Bryant, 1986, 1992; Hazelhurst et al., 2000; Shannon, 1938; Yuan et al., 2006].

The latest firewall access data structure to emerge is referred to as Rule Anomaly Relation (RAR) tree [Chao, 2011]. This data structure and work is another way to detect and display anomalies and inconsistencies in firewall policy design. The system allows a single or multiple firewall security policy to be modeled as a RAR and successfully detect hidden, shadowed, redundant, or correlated rules. The work is similar to that of FIREMAN in that it seeks to find anomalies in a security policy [Yuan et al., 2006].

3.4.1 Rule Processing and Validation

Application of these policy modeling data structures was first reviewed in the Firmato research project [Bartal et al., 1999]. The Firmato firewall management toolkit is some of the first work that focuses on the complexity of an organization's firewall policy. Broadly speaking, it was a research tool for generalizing firewall policies from different vendors, applying a entity relationship diagram (ERD) model to the system, and allowing some analysis of the policy through the ERD. This research did not really address the complexity of the rules, but instead began the work of a generalized management structure of the firewall policy. It attempted to separate the security policy design from the actual network topology through the use of an ERD and modeling language that could be statically checked for correctness [Bartal et al., 1999].

Firewall Policy Advisor is an early effort to find hidden, shadowed, and otherwise misconfigured rules in a particular policy [Al-Shaer and Hamed, 2004a]. They utilize their own tree algorithms and finite state machine for identifying problems. The data structures do not compress, therefore the space and time complexity will grow with the number of rules in the policy [Al-Shaer and Hamed, 2002]. They continued working on the Firewall Policy Advisor and improved some of the analysis to identify general anomalies in a policy. They continue to use the same data structures, and improved on some of the available functionality [Al-Shaer and Hamed, 2004a,b]. In a paper they extended the model to include distributed firewalls, extending the reach to the organizations [Al-Shaer and Hamed, 2004b].

A problem that had typically been addressed by manual configuration and faster hardware is the problem of traffic flow analysis and rule ordering. The basis behind rule ordering is to place the most used rules at the top of the policy so that less time is spent finding a match. [Hamed and Al-Shaer, 2006] researched the ability to autonomously optimize rule order through statistical distributions of traffic analysis followed by automatically ordering the rules. They made use of FDD data structures to ensure that the policy is consistent after reordering (as best as we can tell). It is not explicitly spelled out, but they must have had a mechanism to ensure that re-ordering rules does not alter the firewall behavior.

FIREMAN is a project that allowed the checking of the ACCEPT and DENY space as it relates to an entire policy and checked the policy for inconsistencies and misconfiguration [Yuan et al., 2006]. The primary contributions of this work was to represent the firewall rules in a ROBDD, and then show how performing different set operations on the data sets as a rule was added to the space allowed the fast and accurate detection of rule anomalies

[Yuan et al., 2006]. The work accomplished with FIREMAN is the basis for the research in our work. The construction of the ROBDD in relation to the firewall rule sets starts the process of comprehending a policy and understanding what is allowed or denied.

Two works use data mining algorithms on firewall traffic logs to find a more efficient and possibly smaller set of rules. One looks at mining log frequency to deduce a better policy based on that frequency, and follows that up with a filtering rule generalization algorithm to reduce the size of the rule set [Abedin et al., 2010]. In addition to making the policy more efficient, the research use FDDs [Gouda and Liu, 2004] for testing the policy for consistency. The second work developed a model to check the correctness of a firewall policy by mining the firewall log data [Abedin et al., 2010]. The effort looks to ensure that policy anomalies are identified and corrected.

Work has also been done on an automated testing framework for ensuring policy compliance [El-Atawy et al., 2007]. This work does not accomplish this effort by brute-force testing of all possible combinations, as the time for that sort of a test is unreasonable. Instead, they use a Backus-Naur Form (BNF) compliant language to express the policy and static analysis to ensure that the policy complies to the intent [El-Atawy et al., 2007].

3.4.2 Reachability and Access

Fang is based on the Firmato engine and simulates a firewall based on generalized configurations [Mayer et al., 2000]. Therfore, it is able to build the entire model of the network in memory and then run various “what if” scenarios through the simulated network to find the answer to the questions. The work is similar to what is presented in our work; however, they did not include the analysis engine that handles policy differences [Mayer et al., 2000]. In addition the methods and data structures appear to involve “brute force” testing of all possible combinations, or holding rule information at each firewall. This issue could lend itself to undesirable running times.

Gouda and Liu (2004) covers network reachability theory that acts as a basis for the algorithmic complexity to calculating routes. The work covers more than just firewalls and instead reviews both access control lists (ACL) and physical routes in a network. In addition, the research covers reasoning about how different network elements affect the routing. The most important work is packet transformation through network address translation. Finally, the work provides an algorithm for calculating reachability in a network. While the algorithm is inefficient and at worst case is exponential, it provides a mathematical framework for reachability calculations [Gouda and Liu, 2004].

Liu *et al.* (2004) also presented the first research effort for a query language of reachability and firewall access. The primary purpose was to find a structured and consistent way to ask questions about access. An example presented in that work is “Which computers in the private network can receive packets from a known malicious host in the outside Internet?” To solve this generic sort of problem they addressed two challenges: how to describe a firewall query, and how to process a firewall query. The resulting language was similar to Structured Query Language (SQL), a declarative language used to handle relational data. The language was named Structured Firewall Query Language (SFQL) [Liu et al., 2004]. The work included using Firewall Decision Trees (FDT), a derivative of FDD, with the query language. The results show that large volumes of rules can be processed using FDTs, but the research did not address difficult issues such as policy comparison, multiple firewalls, and network address translation [Liu et al., 2004].

3.5 Internet Protocol Version 6

Internet Protocol Version 6 (IPv6) is the latest iteration in the Internet protocol routing stack. It is an improvement to the current protocol IPv4, with the primary difference related to our research being the unique address space being expanded from 32 bits to 128 bits. This increase will allow many more devices to communicate and connect on the Internet. However, this also represents a challenge for firewalls and firewall administration teams. The growth of the addressable space means that these teams will need better and faster tools in which to comprehend how the firewalls under their control are configured and what sort of changes are happening to the security policies over time. This trend in the market space also supports the need for structures such as FPD for allowing the comprehension of these policies. While the experiments in this dissertation are on 32 bit addresses, expanding the FPD to use 128 bit addresses is a straightforward process and is planned for future experimentation. There is no reason that the algorithms would not function, although the size of the search space would become at least 2^{192} larger. This expansion is a result of the source and destination IP address space growing from 2^{32} to 2^{128}, therefore the number of bits needing to be represented grows $2^{96} \times 2^{96} = 2^{192}$.

3.6 Chapter Summary

In this chapter we presented the Firewall Policy Diagram, an efficient and accurate data structure that serves as a basis for reasoning about firewall policy behavior and change. There are four primary contributions in this work:

- Data structures to efficiently model firewall policies that can be used to reason about them over time and modification.
- A data structure to act as the basis for the implementation of a means in which to query the policy.
- A set of algorithms in which to extract understandable rules from the FPD.
- Experimental evidence that these algorithms can perform the appropriate operations in a seconds even on large firewall policy rule sets.

Chapter 4

Firewall Policy Query Language (FPQL) for Behavior Analysis

In the last chapter we introduced the Firewall Policy Diagram as a mechanism to represent a firewall policy in a concise and efficient data structure. The structure facilitates the canonical representation of a policy as well as human comprehension of the policy. This chapter builds upon that data structure to provide a language for querying the data structure about the space that is represented in a policy, either the accepted, denied, or remaining traffic. The novel Firewall Policy Query Language (FPQL) is a language developed in this work, loosely modeled after the Structured Query Language often seen related to database systems and relational algebra. At its core FPQL provides a group of boolean-based operators for deriving knowledge from a very large solution space. This work seeks to provide a simple, yet powerful, query language that is useful for human comprehension of a firewall policy as represented by an FPD.

On the surface, firewall rule sets are relatively easy to understand. In the context of this research each rule consists of four fields and when the set contains a few rules, an individual firewall administrator can comprehend the access level, and immediately know the appropriate location for granting additional access for a given request. However, because the

IP address, protocol, and port solution space can cover a very large number of permutations, an administrator's ability to fully understand access diminishes as a rule set grows.

4.1 Key Contributions

This chapter presents a query language for comprehension of large network access. The language extends the FPD and provides an expressive query language that is utilized by a team of firewall administrators to answer important questions about what is contained in a large, and often very difficult to understand, set of firewall policies. In a syntax similar to the familiar Structured Query Language (SQL) [Codd, 1970], the user describes what is desired in a declarative manner, allowing the system to find and return the information. This is in opposition to a more procedural style of telling the system how to find the information.

In addition to describing the tenets of the language, this dissertation presents the results of experiments run against large policies (up to 20,000 rules) showing how individual results can be returned in approximately 120 microseconds of processing time. Finally, we present how this language is capable of formal verification of single and multiple policies in a manner similar to FIREMAN [Yuan et al., 2006].

The remainder of the chapter is organized as follows: We begin with a description of the Firewall Policy Query Language (FPQL) and how it may be used to generate, manipulate, and query a firewall policy, as represented in an FPD data structure. We will then present the results of performance-related experiments in terms of query operations on large ruleset firewall policies. Next, formal verification of policies will be demonstrated using FPQL. In the final two sections, related work and conclusions about FPQL will be covered.

4.2 Firewall Policy Query Language

The intent behind the design of FPQL is to provide a mechanism for firewall administrators and other security professionals to take a known set of complex firewall policies, load them into an FPD database, and query the policies and across policies to uncover valuable information about access.

4.2.1 Policy Database

Figure 4.1 illustrates the general architecture of our policy database, which is based on the FPD data structure. The database interaction is managed with a FPQL processor such that the language may be used to create and manipulate firewall policies stored within. The important notion here is that the generation and storage of multiple policies in one system enables comparisons and manipulations between the stored policies. Examples include storing versions of a policy over time to understand how it has changed, or storing policies from multiple vendors to better understand the differences.

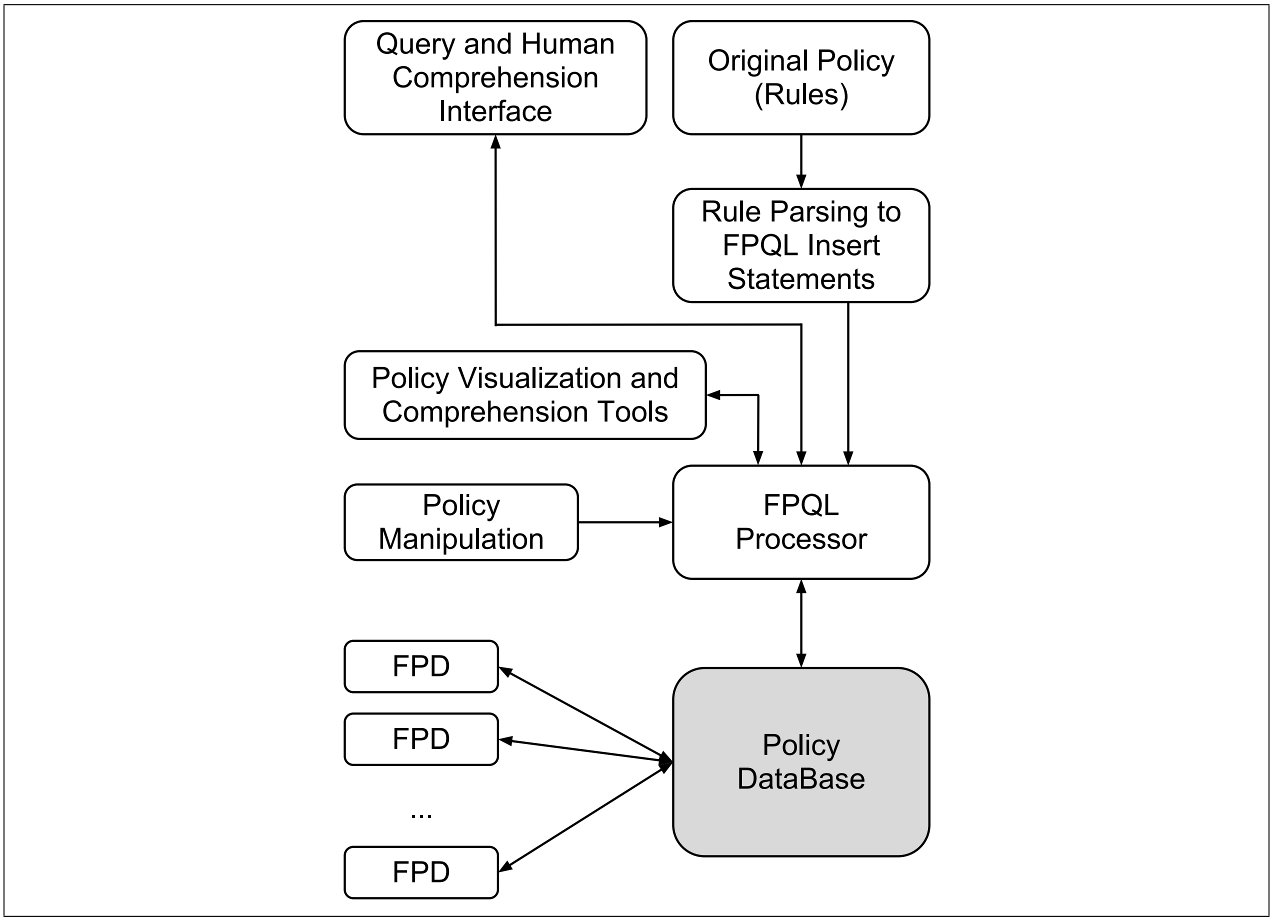

Query and Human Comprehension Interface
Original Policy (Rules)
Rule Parsing to FPQL Insert Statements
Policy Visualization and Comprehension Tools
Policy Manipulation
FPQL Processor
FPD
FPD
...
FPD
Policy DataBase

Figure 4.1: Architecture of the FPQL processor and policy database.

4.2.2 FPQL Grammar

In the subsequent sections of this chapter Extended Backus-Naur Form (EBNF) [Sebesta, 2009] is used to accurately describe the grammar of FPQL. The EBNF grammar of common elements used in the more specific *Insert*, *Query* and *Delete* grammar is described by:

$$\begin{aligned}
< alpha > &\rightarrow\ a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \\
&\quad \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z \\
&\quad \mid A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \\
&\quad \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z \\
< digit > &\rightarrow\ 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0 \\
< id > &\rightarrow\ (< alpha >\ \mid\ < digit >\ \mid \_\,),\ \{\ (< alpha >\ \mid\ < digit >\ \mid \_\,)\ \} \\
< field > &\rightarrow\ S \mid D \mid Prot \mid Port \mid Pol \\
< var > &\rightarrow < id >\ .\ < field > \\
< val > &\rightarrow\ (\ < digit >\ \mid . \mid / \mid -),\ \{\ (\ < digit >\ \mid . \mid / \mid -)\ \} \\
< lparen > &\rightarrow\ (\\
< rparen > &\rightarrow\ ) \\
< comma > &\rightarrow\ ,
\end{aligned}$$

The EBNF meta-language uses abstractions for syntactic structures. The abstractions in EBNF descriptions, or grammar, are composed of *non-terminal* and *terminal* symbols. The non-terminal symbols are the EBNF descriptions, such as *var*, *id* and *field* such that they are composed of other non-terminal or terminal symbols. Terminal symbols are the individual characters, combination characters (strings), punctuation marks or digits; together called the *lexemes* or *tokens*. Terminals are considered the lowest level of derivation and may be matched with the actual input. Therefore, a collection of these grammar rules comprises a full EBNF description. The syntax statements are read like a derivation, beginning with the start symbol of the particular grammar, and is processed through the BNF structure.

To further explain the notation, we will use the following definitions:

$$\begin{aligned}
S &= \text{Source IP Space}\\
D &= \text{Destination IP Space}\\
Prot &= \text{Protocol}\\
Port &= \text{Destination Port}\\
Pol &= \text{Entire policy Space}\\
alpha &= \text{Identifies } a \text{ through } z \text{ or } A \text{ through } Z\\
digit &= \text{Identifies } \textit{0} \text{ through } \textit{9}\\
var &= \text{An FPD name } (id) \text{ and } \textit{field}\\
val &= \text{A field value that will an address or number format,}\\
&\quad \text{depending on the field being manipulated}\\
id &= \text{A user selected identifier composed of } \textit{alpha}, \textit{ digit}, \text{ or underscores}\\
field &= \text{The known fields of an FPD, } \textit{S}, \textit{ D}, \textit{ Prot}, \textit{ Port} \text{ or } \textit{Pol}\\
lparen &= \text{A left parenthesis ``(''}\\
rparen &= \text{A right parenthesis ``)''}\\
comma &= \text{A comma ``,''}
\end{aligned}$$

Most of the common elements are straight forward in the descriptions; however, the *var* definition will benefit from some additional explanation. The *var* element is made of two sub elements, *id* and *field*, separated by a “dot”. Using these sub elements in-conjunction with a “dot” notation allows the language to dereference a policy identifier with the policy element for use in the operation. For example, in the query one might define a policy as *p1_accept* and add known *rules* to the policy in the policy database. In subsequent queries of the policy database, the policy elements are referenced by *p1_accept.S* representing the source IP address space for the accept *p1*. For example:

p1_accept.S

Another element that requires some additional discussion is *val*. While the grammar allows *val* to be a *digit*, forward slash (/), period (.) or dash (-); the correct *val* identifier format is dependent on the *field* being manipulated. For example, for *S* or *D*, then the language would expect an IP address or CIDR form for the correct interpretation. In this work, the validation is processed when the parse tree is traversed for interpretation. It is at this time the FPQL Processor, identified in Figure 4.1, returns a parse error back to the calling program if invalid identifiers are used in the FPQL statement.

4.2.3 Policy Generation and Manipulation

Before a policy can be queried or visualized, it must first be created and loaded. Using FPQL this can be done with the *Insert* command and keywords. The intent behind the insertion syntax is to linearly process a firewall ruleset and insert each rule, one at a time. The following EBNF describes the grammar of the *Insert* statement:

$$\begin{aligned} < insert > &\rightarrow \ insert\ into\ \ < id > \\ &\quad < insertExpr > \\ < insertExpr > &\rightarrow < field > == < insVal > \\ &\quad \{ < comma > < field > == < insVal > \} \\ < insVal > &\rightarrow < lparen > < val > \ \{ < comma > < val > \} \ < rparen > \end{aligned}$$

The following FPQL inserts a rule into the FPD identified by *p1_accept* such that the rule source is host address *192.168.1.1*, the rule destination is the network address *10.1.1.0/24*, the rule protocol is *TCP (6)*, and the rule destination port range is *80* to *90*.

insert into p1_accept S == (192.168.1.1), D == (10.1.1.0/24),

Prot == (6), Port == (80-90)

This example also demonstrates the concept of how FPQL and the underlying FPD policy database interact to capture the action of the original security rule. In this work the action for a particular rule is constrained to either *accept* or *deny*, therefore to capture the different spaces for a policy, it is strictly a policy naming standard to append the action to the policy name. Not only does this allow for the easy identification of the *space* being manipulated, it also allows other actions to be represented in extended works, such as *encrypt* or *log*, without modifying the grammar.

A second example demonstrates inserting into a policy where the destination will not factor into the matching decision. In this case, the destination becomes a "do not care", i.e., *any*. The remaining fields identify the rule as being for *UDP (17)* and a port of *21* or *22*.

insert into p2_accept S == (6.5.4.3),

Prot == (17), Port == (21,22)

4.2.4 Queries

Retrieving policy information from a populated policy database is accomplished using the policy query grammar. In addition to identifying what sort of data are wanted from the policy database, the grammar provides operators that can be applied across defined policies. The FPD database can be queried and compared because each operation or grouping that is defined in a FPQL statement results in an FPD. This means that as *sub-expressions* and other nested operations are executed from the language parse tree, an FPD is constructed and manipulated until the tree is traversed from leaf up and reaches the root. Not only does this simplify reasoning about how the results are constructed, it also allows a common expected result from each operation.

The grammar of a *Query* statement uses the same *id* and *field* definitions as the *Insert* grammar, allowing consistency in use of the language. The following EBNF describes the grammar of the *Query* statement:

$$\begin{aligned}
<query> \rightarrow\ & (count\ \ <field>\ \mid\ <field>) \\
& where\ \ <expr> \\
<expr> \rightarrow\ & <comparison> \\
& \{\ (\ \&\ \mid or\ )\ \ <comparison>\ \} \\
<comparison> \rightarrow\ & <var>\ \mid\ <lparen><expr><rparen> \\
& \mid\ <varStat> \\
<varStat> \rightarrow\ & <var><op> \\
& <lparen><val>\ \ \{<comma><val>\}\ \ <rparen> \\
<op> \rightarrow\ & ::\ \mid !::\ \mid\ \tilde{}\ \mid !\tilde{}\ \mid\ == \\
& \mid\ !==\ \mid \&\ \mid or \mid -
\end{aligned}$$

One small difference from the actual implementation is the use of the *or* operator. In order to make the EBNF easier to read in this work, the word *or* is used, however the actual implementation uses a pipe character. The $< op >$ operators are defined as:

$$\begin{aligned}
:: &\rightarrow \textit{In (subset)} \\
!:: &\rightarrow \textit{Not In} \\
\tilde{} &\rightarrow \textit{Contains (superset)} \\
!\tilde{} &\rightarrow \textit{Not Contains} \\
== &\rightarrow \textit{Equals} \\
!== &\rightarrow \textit{Not Equals} \\
\& &\rightarrow \textit{And} \\
or &\rightarrow \textit{Or} \\
- &\rightarrow \textit{Difference}
\end{aligned}$$

As an example of how the query expression grammar may be used, is the case of an administrator who wants to know the sources allowed to access an important server at 10.2.1.100 over tcp/80 from a policy loaded into the Policy Database named *policy1*.

S where policy1.D == (10.2.1.100) &

policy1.Prot == (6) & policy1.Port == (80)

The second example is an administrator who would like to know all destinations that are allowed by a source 192.168.1.1 or 14.1.20.21. This particular example has a couple of solutions:

D where policy1.S == (192.168.1.1) |
policy1.S == (14.1.20.21)

D where policy1.S :: (192.168.1.1, 14.1.20.21)

A more complicated way to use a policy database and FPQL interpreter is to track a policy as it changes over time. We suppose that a policy *policy_september* is loaded into the database and the administrator would like to know how the policy changes over the next month. One way for an administrator to find this information is to go back through the potentially thousands of changes that may have occurred in the month of October (up to 16,000 [Chapple et al., 2009]). However, using FPQL, the same policy from the next month (*policy_october*) can be loaded into the policy database and the following FPQL will list the differences between the two policies, thus providing the administrator with a list of human comprehensible rules representing access changes that are different from the previous month.

Pol where policy_october.Pol – policy_september.Pol

Notably, the query will represent the *space* that is not a subset of *policy_september*, i.e., the difference operation as implemented in FPQL is a relative complement. Therefore, if the level of access between September and October was removed and *policy_october* is a subset of *policy_september*, then the administrator may want to understand the symmetric difference such that the query would become:

Pol where (policy_october.Pol – policy_september.Pol)
| (policy_september.Pol – policy_october.Pol)

4.2.5 Delete

The final manipulation language grammar is for deleting portions of *space* from a policy that has been loaded into the policy database. While not useful for direct comprehension of a policy as it relates to the query language, it is necessary for formal verification of a particular policy for anomalies such as those checked by FIREMAN [Yuan et al., 2006]. The following EBNF describes the grammar of the *Delete* statement:

$$\begin{aligned} < deleteFromPol > \rightarrow \ & delete\ from\ policy \\ & < id > \ where \ < expr > \end{aligned}$$

If one wants to remove a particular rule from an existing policy, the *Delete* grammar allows selectively removing a portion of the *space* from a policy.

delete from policy p1 where p1.S == (192.168.1.1)

& p1.D == (10.1.1.0/24) & p1.Prot == (6)

& p1.Port = (80-90)

4.2.6 Miscellaneous Policy Operations

In addition to the more formal *Insert*, *Query*, and *Delete* language syntax, FPQL also provides the ability for the administrator to check what policies have been loaded into the policy database using the *list policies* command and delete those policies using the *delete policy* command. This work uses these commands as examples of many needed for general maintenance of the policy database in operational situations by firewall administrators. Other commands such as these may be useful for the user to accomplish day to day tasks.

4.2.7 Syntax Tree Parsing

When a FPQL query is parsed, an Abstract Syntax Tree (AST) is produced such that the leaves are processed and transformed as the language is traversed to its root. As each node

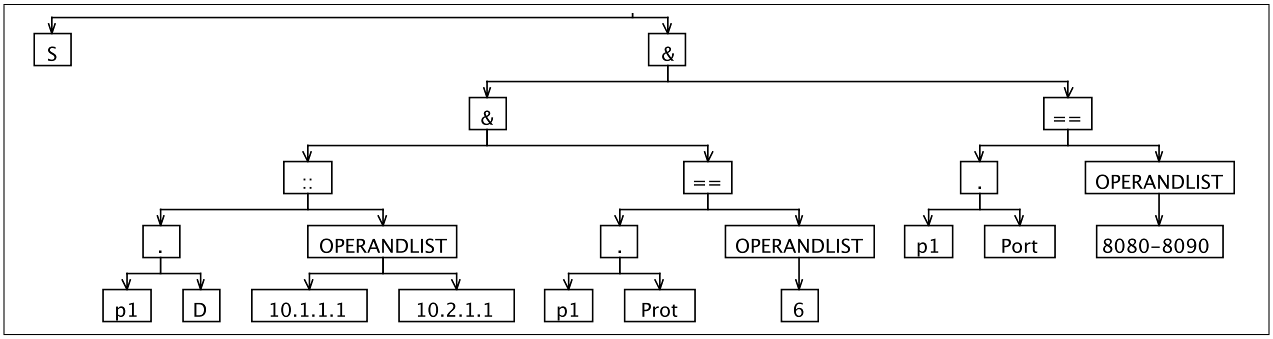


Figure 4.2: An example AST parsed from a FPQL grammar.

is visited, an FPD is produced that will then be used as an operand in the next operator. An example is a simple query for discovering the sources which are allowed access to the destinations 10.1.1.1 or 10.2.1.1, over service tcp/8080-8090:

S where p1.D :: (10.1.1.1,10.2.1.1) & p1.Prot == (6)

& p1.Port == (8080-8090)

Figure 4.2 shows the abstract syntax tree that is produced as a result of processing the EBNF grammar of the example query. As the tree is traversed from leaf to root, nodes representing operators are acting on operands that are interpreted as portions of an existing policy (as in the case of *p1.D*) or lists (as in the case of '10.1.1.1,10.2.1.1'). These constructs are combined as the tree is pruned into FPD data structures where finally the results are extracted from the final FPD at the root, based on the requested field (*S,D,Prot,Port,Pol*) at the left child and the FPD produced from the right sub-tree.

4.3 FPQL and Policy Database Performance

The experiments designed for this work seek to evaluate the performance of FPQL when dealing with policies of sizes ranging from 1,000 to 20,000 rules and focuses on processing and executing FPQL statements against the policy database. Security reasons prevented us from being able to gain access to industry rule sets, as the majority of companies with firewall policies of those sizes are hesitant to allow outside parties access. Therefore, the rulesets are randomly generated and are generally distributed over the solution space. The FPQL and FPD data structures and algorithms are implemented in the Java$^{\text{TM}}$ 6 programming language with the tests being run on a Mac Book Pro laptop computer. For ROBDD software the Buddy and JavaBDD libraries were used [Lind-Neilsen, 2004; Whaley, 2007], for the FPQL language parsing and manipulation this work uses the ANTLR tool [Parr and Quong, 1995].

Figure 4.3 charts the performance of FPQL inserting and querying operations on the policies. The insertion operations averaged approximately 500 microseconds with no growth as the number of rules increased. The query operation performance averaged 120 microseconds, again with no growth as the number of rules increased. The constant processing time reflects the constant time performance of the underlying ROBDD data structures in the FPD. However, while the operations and testing appear to be constant, that is a reflection of the queries being run against the system having a constant seven operators. Therefore, the actual complexity is related to the number of operators present in the query and will grow linearly with those.

This work focuses on an extension of a policy comprehension model FPD. Because comprehension is the end goal, it is very important to provide very fast access to ad-hoc queries described by FPQL. The performance numbers presented here achieve this goal by allowing the firewall administration team to very quickly and unerringly gain an understanding of what is allowed through the security policies implemented in their organizations.

4.4 FPQL Policy Anomaly Detection

FIREMAN is a related work that analyzed intra-firewall and inter-firewall anomalies with a framework and algorithm for identifying those inconsistencies [Yuan et al., 2006]. A good firewall configuration is consistent with the administrator's intention and the assertion that FIREMAN makes is that inconsistencies in the ruleset are an indication of mistakes or misconfigurations. FIREMAN defines four primary inconsistencies based off of related work

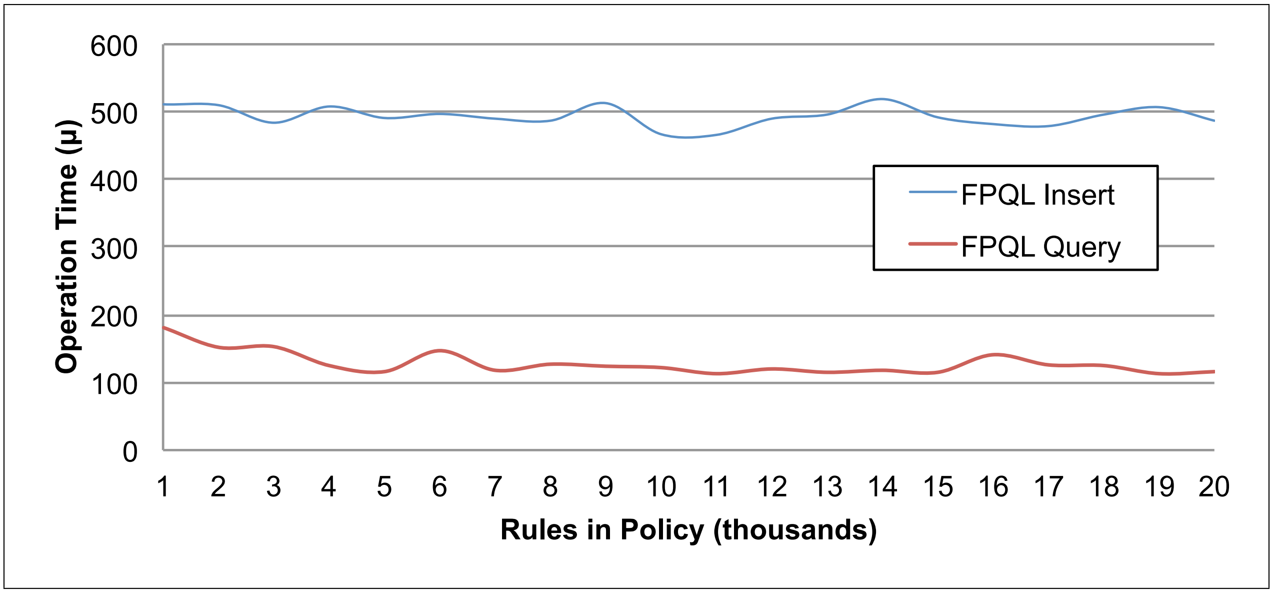


Figure 4.3: FPQL performance with 1,000 to 20,000 rule policies.

[Al-Shaer and Hamed, 2004a,b] and we show a FPQL algorithm that is able to identify those anomalies.

Number	Action	Predicate
1	deny	tcp 15.21.10.0/25 any
2	accept	tcp any 16.5.1.0/24
3	deny	tcp 15.21.10.128/25 any
4	deny	udp 10.10.2.0/25 172.16.0.0/16
5	accept	tcp 15.21.10.0/24 any
6	deny	tcp 192.168.1.0/24 16.5.0.0/16
7	accept	udp 10.10.2.0/24 any
8	accept	udp 10.10.2.1/32 12.168.3.0/24

Table 4.1: Sample policy with anomalies.

1. *Shadowing:* An inconsistency in a firewall policy such that the entire *space* a rule represents and the associated action is contained in one (or a combination of) previous rules with the opposite action. This is identified as an "error" because the intended action will not be taken and the rule is guaranteed to never be exercised because of the previous rules. An example of *shadowing* can be seen in Table 4.1 by rule number 5 being shadowed by a combination of rules 1 and 3.

2. *Generalization:* A problem that identifies a case where a portion or subset of the *space* represented by a rule had been previously matched by one or multiple rules with the opposite action. An example of *generalization* can be seen in Table 4.1 where rule 7 is a superset of rule 4 with an opposite action.

3. *Correlation:* Represents a problem where the *space* of the current rule intersects with one or multiple previous rules with the opposite action. The *space* represented by the current rule and previous ones intersect but are not a subset or superset of one another. Table 4.1 demonstrates this anomaly with rule 6 being correlated to rule 2. The intersection between the two rules is “tcp 192.168.1.0/24 16.5.1.0/24” with rule 2 determining the action of the traffic.

4. *Redundant:* A situation identifying that a previous rule or rules already handled the current rule *space* with the same action. Therefore, this rule would never be matched and is considered redundant. It is the case where removing the rule from the policy will not change the behavior of the firewall. While this will not cause a security breach by misconfiguration, it can be viewed as an inefficiency. Cleaning up *redundant* rules is a method to improve the firewall processing speed by removing an unused rule that would otherwise need to be reviewed by the firewall. Table 4.1 has rule 8 as being *redundant* because rule 7 matches the traffic for rule 8 with the same action.

FIREMAN identifies *shadowing* as an error, but considers *generalization* and *correlation* as potentially not being an error because administrators may be using these more broadly defined networks as a way to keep the policy sizes smaller. However, it is warned that use of these techniques introduces risk, and requires prior knowledge of the intention of the administrator to ensure that the intent of the policy is understood to future change agents [Yuan et al., 2006]. Therefore, our work will also identify the anomalies as warnings. *Redundant* rules are considered an error in this situation because it reflects a rule that will never be used. Leaving *redundant* rules in policy makes the policy unnecessarily larger,

```
Input: Policy P to test for inconsistencies
Output: Inconsistent rules and the type of anomaly
procedure TestPolicy($P$)
    FPQL: insert into remain $S == (0.0.0.0/0)$,
      $D == (0.0.0.0/0),\ Prot == (0 - 255)$,
      $Port == (0 - 65535)$
    for all Rule $R \in P$ do
        TestRule($R$)
    end for
end procedure
```

Figure 4.4: FPQL Intra-firewall anomaly detection.

increasing processing time of the firewall and decreasing the comprehension of the policy.

4.4.1 FPQL Intra-firewall Anomaly Detection

An algorithm may be used in conjunction with FPQL to identify if a rule is *shadowing*, *generalization*, or *correlation* in an individual firewall policy. The initialization of the policy database begins with three policies represented as FPDs: *accept*, *deny*, and *remain*. The *accept* and *deny* policies are initialized to $\emptyset$, with the *remain* policy starting off as U (meaning, every possible rule field combination). In addition, each rule R of the test policy P contains the known FPQL fields S, D, *Prot*, and *Port* with two new fields:

$$A = \text{Action: } \textit{accept} \text{ or } \textit{deny}$$

$$R = \text{Rule as an FPD: } S + D + \textit{Prot} + \textit{Port}$$

Rule R fields are dereferenced by a dot (i.e. $R.S$). Each rule R in policy P is processed linearly from top to bottom as to model how an actual firewall processes a packet for a matching rule. Notably, not all parts of the algorithm are FPQL. Some of the operations are based on FPD SET capabilities and are predicated on FPQL processing returning an FPD. Figures 4.4 and 4.5 details the algorithms for computing intra-firewall anomaly detection.

```
procedure TestRule($R$)
    $accept \leftarrow accept.Pol$ from the policy database
    $deny \leftarrow deny.Pol$ from the policy database
    $remain \leftarrow remain.Pol$ from the policy database
    if $remain = \emptyset$ then
        if $R.R \subseteq accept$ then
            if $R.A =$ "accept" then $R$ is Redundant
            else $R$ is Shadowed
            end if
        else if $R.R \subseteq deny$ then
            if $R.A =$ "deny" then $R$ is Redundant
            else $R$ is Shadowed
            end if
        else $R$ is Correlated
        end if
    else if $R.R \cap remain = \emptyset$ then
        if $R.R \subseteq accept$ then
            if $R.A =$ "accept" then $R$ is Redundant
            else $R$ is Shadowed
            end if
        else if $R.R \subseteq deny$ then
            if $R.A =$ "deny" then $R$ is Redundant
            else $R$ is Shadowed
            end if
        else $R$ is Correlated
        end if
    else if $R.R \cap remain \neq \emptyset$ and $R.R \nsubseteq remain$ then
        if $R.A =$ "accept" then
            if $R.R \cap deny \neq \emptyset$ and $R.R \nsubseteq deny$ then
                $R$ is Correlated
            end if
        else if $R.A =$ "deny" then
            if $R.R \cap accept \neq \emptyset$ and $R.R \nsubseteq accept$ then
                $R$ is Correlated
            end if
        end if
    end if
```

Figure 4.5: TestRule function.

```
    if $R.A$ = "$accept$" then
        FPQL: $insert\ into\ accept\ S == (R.S),$
          $D == (R.D),\ Prot == (R.Prot),$
          $Port == (R.Port)$
    else
        FPQL: $insert\ into\ deny\ S == (R.S),$
          $D == (R.D),\ Prot == (R.Prot),$
          $Port == (R.Port)$
    end if
    FPQL: $delete\ from\ policy\ remain\ where$
      $S == (R.S)\ \&D == (R.D)\ \&$
      $Prot == (R.Prot)\ \&\ Port == (R.Port)$
end procedure
```

Figure 4.5: TestRule function (continued).

4.4.2 FPQL Inter-firewall Anomaly Detection

FIREMAN identifies the reality of many firewalls existing in a network and the potential for data to flow through those multiple policies [Yuan et al., 2006]. A computer network is a graph of nodes (firewalls, routers, switches, hosts, etc.) and edges (traffic flow connections). A well designed network includes multiple paths from one node to another. By modeling the network as a graph, and using DFS or BFS algorithms, a network can be converted into a spanning tree.

Figure 4.6 shows a spanning tree of a network providing all known directed paths from a *starting* node to a resulting *ending* node. This spanning tree can subsequently be used to find anomalies in the network filtering rules. For large and dense graphs, a large number of paths might result in the subsequent spanning tree. These paths may go through routers and other non-filtering devices used primarily for traffic management. However, because we are only concerned with the filtering devices in the network, i.e., firewalls, the size of the resulting spanning tree can be greatly reduced by only including firewall nodes [Yuan et al., 2006]. Figure 4.6 is a reformulation of the solution presented in [Yuan et al., 2006] to identify the *starting* node once and build the spanning tree out from that location to multiple *ending* nodes. This simplifies the algorithm [Yuan et al., 2006] presented such that the processing

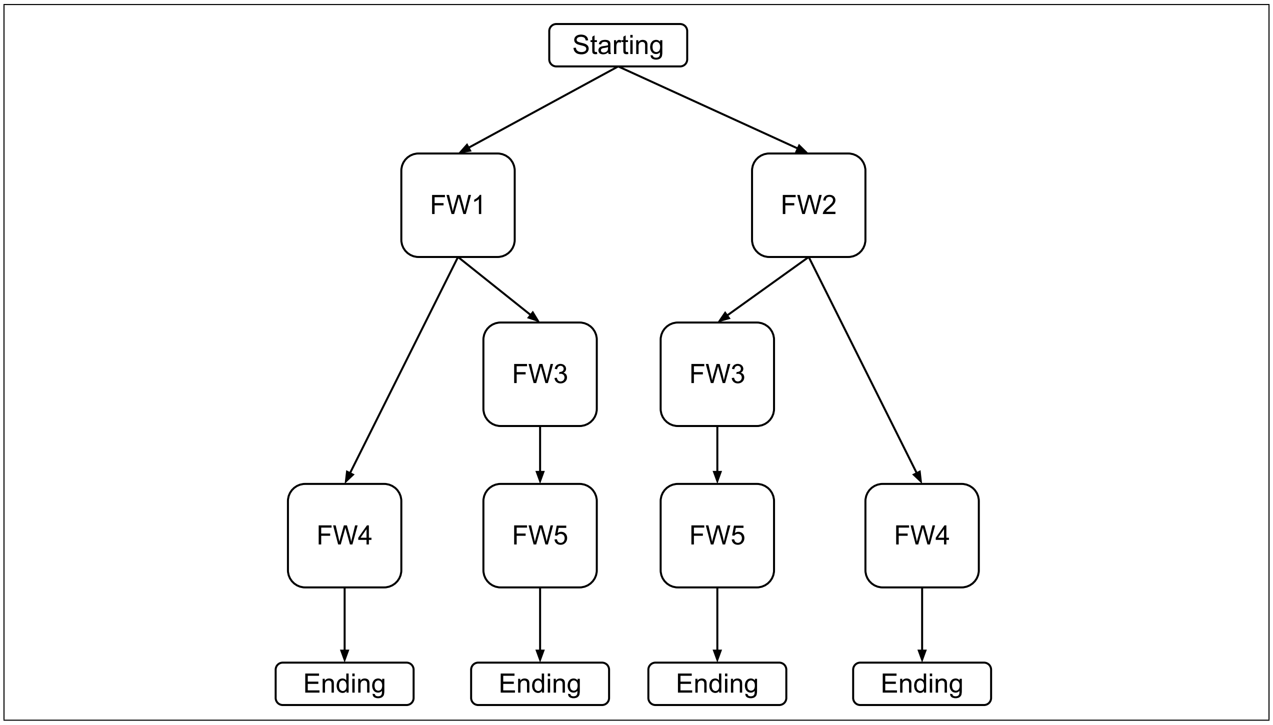


Figure 4.6: Network spanning tree.

at each node is just the inbound solution space I having been potentially manipulated by its parents. The FPQL reformulation is presented in Figure 4.7 and traverses a depth first search from the root node *starting* to the leaf nodes *ending*, identifying anomalies as the graph is traversed.

An additional classification used when processing Inter-firewalls for anomalies is the identification of a *raised security level*. This classification primarily identifies those packets that were accepted by a previous firewall, but denied downstream. A situation where this may not be considered an anomaly, but because traffic was allowed through one firewall and denied by the next, should be reviewed by firewall administrators to ensure that this does not indicate unintended access in a previous firewall.

4.4.3 FPQL Policy Anomaly Detection Performance

The purpose of the intra-firewall and inter-firewall formal verification model presented in this section is to demonstrate the formal verification capabilities of the FPQL language. This

```
Input: Root Node P of Spanning Tree
Output: Inconsistent rules and the type of anomaly
procedure TestInterFirewall($P$)
    FPQL: insert into $I$ $S == (0.0.0.0/0)$,
      $D == (0.0.0.0/0)$, $Prot == (0 - 255)$,
      $Port == (0 - 65535)$
    for all Node $C \in Children(P)$ do
        ProcessNode($C, I$)
    end for
end procedure
procedure ProcessNode($P, I$)
    if $P \in ending$ then return
    end if
    $I' \leftarrow I \cap U$
    for all Rule $R \in P.Policy$ do
        if $R.R \subseteq I$ then
            if $R.A =$ "deny" then $R$ raised security level
            end if
        else if $R.R \subseteq \neg I'$ then
            if $R.A =$ "accept" then $R$ is Shadowed
            else if $R.A =$ "deny" then $R$ is Redundant
            end if
        end if
        FPQL: delete from policy $I'$ where
          $S == (R.S)$ &$D == (R.D)$ &
          $Prot == (R.Prot)$ & $Port == (R.Port)$
    end for
    for all Node $C \in Children(P)$ do
        ProcessNode($C, I'$)
    end for
end procedure
```

Figure 4.7: FPQL Inter-firewall anomaly detection.

section has shown that FPQL and the Policy Database architecture is capable of formally modeling policies in both an individual and networked environment. In general, the performance of the presented algorithms have a computational complexity similar to the original FIREMAN system. The complexity is bound by the number of rules being verified and will grow in relation to that number. Therefore the algorithms presented were a reformulation of those presented in other works with the exception of one algorithm. The computation of the network spanning tree for inter-firewall verification included an improvement based on only having to process a single input FPD space I.

4.5 Related Work

Over the past decade there have been many research efforts that involved analyzing and understanding firewall policies, both from a single and multiple policy level. Fewer involve allowing ad-hoc querying of a policy and allow comprehension of large policies.

One of the earliest works, and the most closely related to ours, built a query engine on top of a formal verification system called Voss [Hazelhurst et al., 2000]. They implemented a simple functional language that is capable of many set operations and selection of specific fields in which to return from the query. The query language is somewhat difficult to use, but is expressive. While the underlying Voss hardware verification system used ordered binary decision diagrams to model a rule set, the method in which the results were extracted from the canonical BDD form is unclear. It appears that a brute force method was implemented to parse through the entire solution space resulting from a query. The algorithm complexity was cited to be linear with respect to the number of rules for creation of the policy and constant time with respect to the number of variables in the BDD for queries of the policy [Hazelhurst et al., 2000]. This is consistent with our work on processing FPQL. However, the analysis in [Hazelhurst et al., 2000] does not take into account the processing time related to extracting human comprehensible data from a binary decision diagram. This is a large segment of processing and could involve millions of individual BDD solutions. Our analysis shows that the complexity should have been a function of the resulting ROBDD and the number of distinct solutions, a potentially much larger complexity that is not correlated to the number of rules, rather the *space* those rules represent. Chapter 3 discussed our work with the FPD, a core element of the FPQL system, allowing the human comprehension of the results from a FPQL query.

Liu *et al.* (2005) propose a query language called Structured Firewall Query Language that is executed on a data structure called a Firewall Policy Tree. While similar to this work, the effort differs in the method in which the policy is modeled and how the query is processed. They cite "rule-based processing" and linearly process each rule on the query [Liu et al., 2005]. Their later efforts implemented the same language on a different policy data model called a Firewall Decision Diagram [Liu and Gouda, 2009]. This improved the running time of the query language from $O(n)$ to $O(log^n)$ where n is the number of rules in a policy. A key differentiator to our work is that the FPQL processing time is based on the number of operators in the query and does not change with the size of the input policy.

Mayer *et al.* (2000) present a three field query interface is presented to interrogate a software modeled network and ACL graph. The performance of the system is not clearly stated, nor are the algorithms for efficiently handling the very large solution space discussed. It appears that once a query is issued, a brute force analysis of all possible packets traverse the simulated network [Mayer et al., 2000]. While this may work for a single packet, the computation time would quickly become unreasonably large for anything but the most trivial networks and firewall policies.

In addition to static model checking, such as FPQL describes, tools such as Nessus employ active probing [Deraison, 2005]. What is ascertained from active vulnerability testing could be valuable for understanding real threats to a network. However, the act of running these tests involve a brute force method of trying all possibilities, a very time consuming and potentially disruptive process to an active network.

Other associated efforts that do not include a query language have been done by modeling firewall policies; however, most of the models reflect their intended use and not all are capable of the sort of operations described in this work. Much of the research has focused on rule processing and validation of those rules where the goal is to identify hidden, shadowed, and inconsistent rules [Al-Shaer and Hamed, 2002, 2004a,b; Bartal et al., 1999; Chao, 2011; Gouda and Liu, 2004; Liu and Gouda, 2005; Yuan et al., 2006]. In general the focus is on algorithms for finding policy anomalies both from a single policy model to a multi-policy model. A portion of the related research introduced the use of BDDs for the models and became the foundation for some of the algorithms in our work [Bryant, 1986, 1992; Hazelhurst et al., 2000; Shannon, 1938; Yuan et al., 2006]. Horowitz and Lamb (2009) present a rule de-correlation algorithm with efforts to extract rules from existing firewall policies. The methods appear to be strictly for de-correlating rules with an exponential running time correlated to the size of the original policy and no overall policy comprehension [Horowitz and Lamb, 2009]. In addition, some works propose design methods for firewalls, where this work discusses analyzing rules already designed [Guttman, 1997; Guttman and Herzog, 2005].

4.6 Extended Fields

It is important to note that while this work focuses on four tuples of a firewall rule, there are potentially more tuples to include. In addition, next generation firewalls have begun to expand their application layer filtering to include fields at layers higher in the protocol stack. Both FPQL and the underlying policy database with FPDs are capable of being extended to include fields in future work. The processing time complexity becomes larger; however, it is still bound by the depth of the ROBDD in the case of queries.

4.7 Chapter Summary

In this chapter we presented FPQL (Firewall Policy Query Language), an efficient and powerful language built on top of a policy database and the Firewall Policy Diagram data structure. There are three primary contributions in this work:

- Provided an expressive language for a firewall administrator to understand access through a policy or set or policies.
- Demonstrated that FPQL can run in microseconds of time, even against very large rulesets. This speed encourages comprehension of policies through ad-hoc queries, further supporting the overall goal of understanding network access.
- Demonstrated how FPQL could be used in formal verification of both intra and inter firewall policies.

Chapter 5

Modeling Firewalls for Behavior Analysis

Historically, attempts to model firewall behavior have focused on the security rules contained in a single access control list. However, modern firewalls and filtering routers involve more than simply deciding if a packet should pass through a security policy. Routing decisions through multiple network interfaces involving vendor specific constructs such as zones, domains, virtual routing tables, and multiple security policies have become the more common type of device found in the industry today. In the past, network devices were separated by functional area (ACL, router, switch, etc.). The more recent trend has been for these capabilities to converge and blend, creating a device that goes far beyond the straight-forward single access control list.

This chapter presents a behavioral model of the capabilities found in firewall type devices and a process for taking vendor specific nuances to a common implementation. This includes understanding interfaces, routes, rules, translations, and policies; modeling them in a consistent manner such that the many different vendor implementations may be compared to each other. In addition, a process is presented for allowing the comprehension of two vendor device configurations with the means to precisely compare configurations from one device to another. This chapter builds on the foundation of the FPD such that it is used to test the entire potential solution space in milliseconds of time.

5.1 Key Contributions

This chapter presents a framework in which to model individual firewalls such that software based analysis and other formal analysis methods may be used with the model. Some examples of industry uses for modeling a network and firewalls in software include:

- Network trace analysis for understanding how a packet will traverse the device without physically sending the packet. In addition to an individual packet, a packet *space* in the form of an FPD may traverse the network to understand what will make it from point A to point B.

- Logical comparisons of firewall vendor implementations. If two firewalls are configured to behave the exact same way but are from two different implementations, modeling the behavior of each vendor in software allows for the formal verification that each ingress and egress FPD are identical. Subsequently, if they are not identical, the FPD used to traverse the spanning graph can show what is different from what is potentially a large solution space.

- Behavior Abstraction Modeling of specific firewall vendors serves as the basis for automated translation from one vendor configurations to another with certainty of how the device will behave.

- Participate in a larger software modeled network for more comprehensive simulations.

5.2 Modeling Firewall Behavior

In a broad description, a firewall is a network device that is used to isolate an organization's internal network from outside networks, allowing some packets to pass and blocking others. Firewalls allow two entities to connect their networks together through existing infrastructure and protocols, while securing the private networks behind them [Liu and Gouda, 2005]. The typical placement of a firewall is at the entry point into a network so that all traffic must pass through the firewall to enter the network. The traffic that passes through the firewall is typically based on existing packet-based protocols, and a packet can be thought of as a tuple with a set number of fields [Liu and Gouda, 2005]. Examples of these fields are the source/destination IP address, port number, and the protocol field. The firewall is a combination of hardware and software such that the hardware will typically be the physical connection to the networks and the software will implement the filtering, routing, and NAT capabilities [Kurose and Ross, 2007]. The portions that are implemented in hardware or software vary by vendor and product design. In general, a firewall has three goals [Kurose and Ross, 2007]:

- All traffic from one network to another that must be filtered passes through the firewall and the internal mechanisms of the firewall. So, the traffic will enter on the ingress interface. Once inside the firewall, it will pass through the internal routing, filtering, and NAT mechanisms. Finally, the traffic will exit the egress interface.

- When the traffic is being processed inside the firewall, only authorized traffic as defined by the local security policy will be allowed to pass.

- The firewall itself is assumed to be immune to penetration. While this is not always reality, the goal of firewall behavior abstraction is not to find flaws in the firewall itself, but to verify and simulate the configuration of the firewall.

5.2.1 Behavioral Components of Modern Firewalls

Traditional firewall capabilities were strictly focused on filtering traffic and applying the local security policy of an organization. Some firewalls were more sophisticated than others, but in general the filtering focused on an individual packet (packet filters), stateful packet flow (stateful filters), or application related traffic (application gateways) [Kurose and Ross, 2007]. All of these capabilities are built on top of the known TCP/IP stack and the only goal is to accurately allow only the communication channels defined in the local policy.

However, firewall vendors have continued to increase the scope of what defines a firewall. Looking at any modern firewall product today, one will typically find a combination of router, network address translation, and filtering capabilities. In addition, each of these sub-components may be broken down further into other elements such as virtual routers, embedded NAT inside of rules, and multiple filtering policies applied at different places. Therefore, for any abstraction of a firewall there must exist a mechanism to represent these capabilities so that accurate results may be computed.

5.2.1.1 Interfaces

An interface on a firewall is the physical network connection to the wire (or radio) that will transmit the traffic to and from the parties communicating [Kurose and Ross, 2007]. In the strictest definition of the firewall and packet filtering device, two network interfaces exist. This represents a physical separation between one network and another, requiring the traffic to flow through the firewall from one interface to the other and vice versa. However, as discussed in [Chapple et al., 2009], the firewall will typically have many more than two interfaces, which means that a firewall is no longer just a filtering device, but must also decide the appropriate egress interface for traffic.

5.2.1.2 Security Policies

A firewall will have single or multiple security policies to be applied to incoming or outgoing traffic. These policies are typically an ordered sequence of rules that follow the general form:

$$\langle predicate \rangle \rightarrow \langle decision \rangle$$

The *predicate* defines a boolean expression over the fields in a packet tuple that are evaluated and the physical network interface from which the packet arrives. For example, source IP is 10.2.0.1 and destination IP address is 192.168.1.1 on eth0 (a common label for a linux interface). Then the *decision* portion of a rule is what happens if the predicate matches to a true evaluation. A *decision* is typically accept or deny with the possibility of additional actions, such as an instruction to log the action [Liu and Gouda, 2005].

A firewall policy is made up of an ordered list of these rules such that as a packet is processed by the firewall, it attempts to match the packet to the predicate one rule at a time, in order. Matching the packet means that the firewall evaluates a packet based on the fields in the rule tuple, a packet matches the rule if it matches all the fields identified in the predicate of the rule [Kurose and Ross, 2007]. The predicate does not necessarily need to contain a value for all possible fields and can sometimes contain the "any" variable in a field to indicate to the rule processing software that this is a "do not care" condition of the predicate and any value for that variable will match. It must completely match all the fields for the firewall to take the appropriate action. These rules are processed in order until the firewall finds a match, at that time it will take the appropriate action identified by the decision [Kurose and Ross, 2007].

5.2.1.3 Routes

For this work, a route can be defined as a simple one tuple rule with a decision being the egress interface. The one tuple of the traffic being processed is the destination. For a particular routing rule, the destination can be identified as an IP Address, address range, or Classless Inter-Domain Routing (CIDR) format. Therefore, in a similar manner as security rules, the solution space can be split as the traffic is processed. Traffic is attempted to be matched from top to bottom in the routing table. For example, a routing table may look like:

$$\begin{aligned}\langle Destination\rangle &\rightarrow \langle Interface\rangle\\ 192.168.2.0/24 &\rightarrow eth0\\ 10.20.0.0 - 10.20.255.255 &\rightarrow eth1\\ 0.0.0.0/0 &\rightarrow eth2\end{aligned}$$

For instance, if traffic from 10.20.5.5 is being processed through the routing table, then route 2 would match and send the traffic out of eth1.

5.2.1.4 Network Address Translation

Network address translation (NAT) is a feature often offered by firewall and router vendors to solve a few problems. The first is to allow private and public IP addresses to communicate. In the current IPv4 address space there exists several realms of addresses that are not routable on the public internet. RFC 1918 non-routable address space include 10.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. The reasoning is to allow private networks that do not communicate directly with other private networks to share these address spaces without fear of collision. Therefore NAT provides the means for two private networks with colliding address space to communicate through a border device, like a firewall [Kurose and Ross, 2007].

The second reason is that public Internet service providers typically charge for each public address and have a finite number of them. Therefore, for flexibility and cost savings it is often advantageous to use the one to many relationship from external to internal or only allowing outbound traffic. This sort of configuration allows the internal network to appear as one device, when in reality the border device is hiding many private hosts.

Another reason NAT can be used is to provide a layer of security to the devices in the private network. By disguising the true location of secure resources an organization provides one more level of security to its assets [Kurose and Ross, 2007].

In addition to these three reasons, there are various other reasons an organization would want to hide private address locations behind a border device and as a result NAT is a very important and very useful network tool [Kurose and Ross, 2007].

NAT implementation will typically function as a translation table such that the inbound traffic will match an entry (either source or destination address) which is translated to another address on the egress side. The firewall device is then required to keep track of that conversation in order for response packets to have the reverse translation applied and arrive at the appropriate destination. There are typically three types of NAT: source address translation (SNAT), destination address translation (DNAT), and port translation (PAT). Each of these are roughly the same idea, and each will require a translation table, yet they will operate on different fields of the packet [Kurose and Ross, 2007]. For example, a DNAT table will typically look like the following:

$$\langle Destination\ Address\rangle \rightarrow \langle Translated\ Address\rangle$$

$$192.168.2.1 \rightarrow 74.125.228.39$$

$$10.1.1.10 \rightarrow 157.56.237.251$$

$$... \rightarrow ...$$

Then, when a packet arrives on an interface, it will look at where the packet is going and replace the destination of the packet with the appropriate translated address.

In general, this sort of translation may take place immediately on the packet being processed through the firewall. However, there are certain situations where the replacement is delayed until the egress interface is known. This is an example of a *hide* translation where the outgoing packet source address will assume the address of the egress interface, making the packet appear to have originated from the firewall, and subsequently hiding the true origination. Furthermore, when the response is seen by the firewall, it will reverse the translation and send the traffic to the originating host, i.e., the intended recipient.

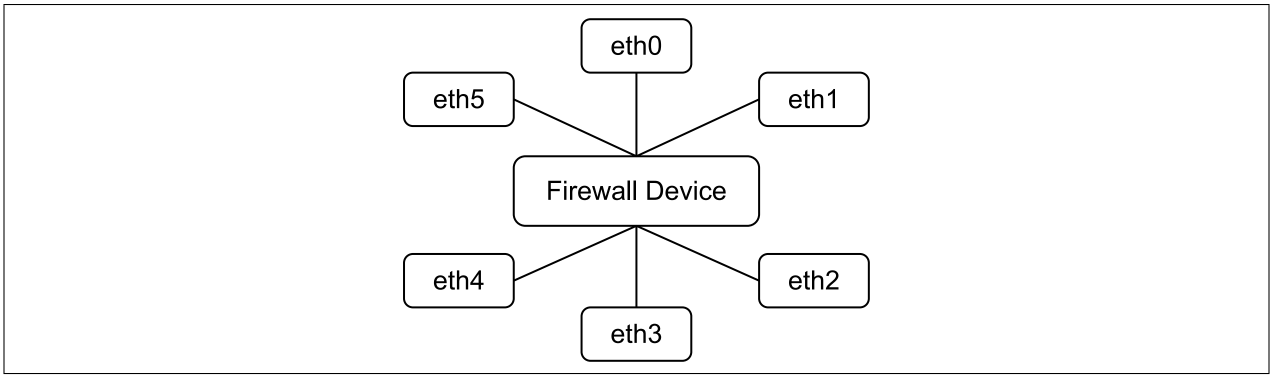


Figure 5.1: Linking a firewall to the network.

5.3 Firewall Behavior Chains and Spanning Graphs

When dealing with a firewall or network device, the entire goal of the broader network is to provide a path from point A to point B. From a physical perspective this is a cable or radio providing a path from one device interface to another, hopping along until the data reach the final destination. Because the firewall is responsible for taking traffic in from one interface and sending it out another, there are internal paths that take the data through the various control and routing structures, as defined in the previous section. These paths and structures can be abstracted into behavior rules, behavior chains, and a spanning graph, the links between the behavior chains.

5.3.1 Behavior Rules

As a first step when considering how a particular firewall might process data, one typically thinks about the physical interfaces. Figure 5.1 shows how the interfaces might appear as spokes off of a firewall allowing it to participate in a larger network. To model the internals, one must consider the elements we have defined: routes, security rules, and NAT. Essentially all three items may be thought of as *predicates* and *actions*. The predicate of all three types are over the known *tuples* or fields of a Internet Protocol and potentially TCP and UDP port, i.e., the source address, destination address, source port and destination port. The actions that may be defined when the predicate matches are *accept*, *deny*, or *next* action; with two additional state transition operators: *translate* and *egress interface*. When a predicate matches, only one action may be applied (accept, deny, or next), however, any (or all) of the state transition operators may be applied. Using these elements and the initial link from the ingress interface, an internal spanning graph may be constructed generating a data path through the device and out the egress interface that may then be linked to the overall network spanning graph. These $\langle predicate \rangle \rightarrow \langle action \rangle$ constructs are identified as *Behavior Rules*.

5.3.2 Behavior Chain

A behavior chain is defined as a set of behavior rules such that they are processed top-down and the first matching predicate for a particular individual packet performs the associated action. The common use for a behavior chain is the encapsulation of the larger ideas of a firewall. For example, it is common when modeling a particular vendor firewall to identify a *routing* behavior chain or a *security policy* behavior chain such that all corresponding routes or security rules are in that chain, and may potentially be processed as one entity.

In addition to providing a grouping mechanism for behavior rules, each behavior chain possesses three overall actions: *accept*, *deny*, and *default*. These actions may be linked to the next chain in the spanning graph, or potentially to the egress interface. The behavior chain accept action will be applied to a traversing packet when the packet has matched a behavior rule predicate and the action is *accept*. In a similar manner, the behavior chain deny action will follow the same logic but takes the deny path. Finally, the behavior chain default action will be applied if no behavior rule predicate matches the packet.

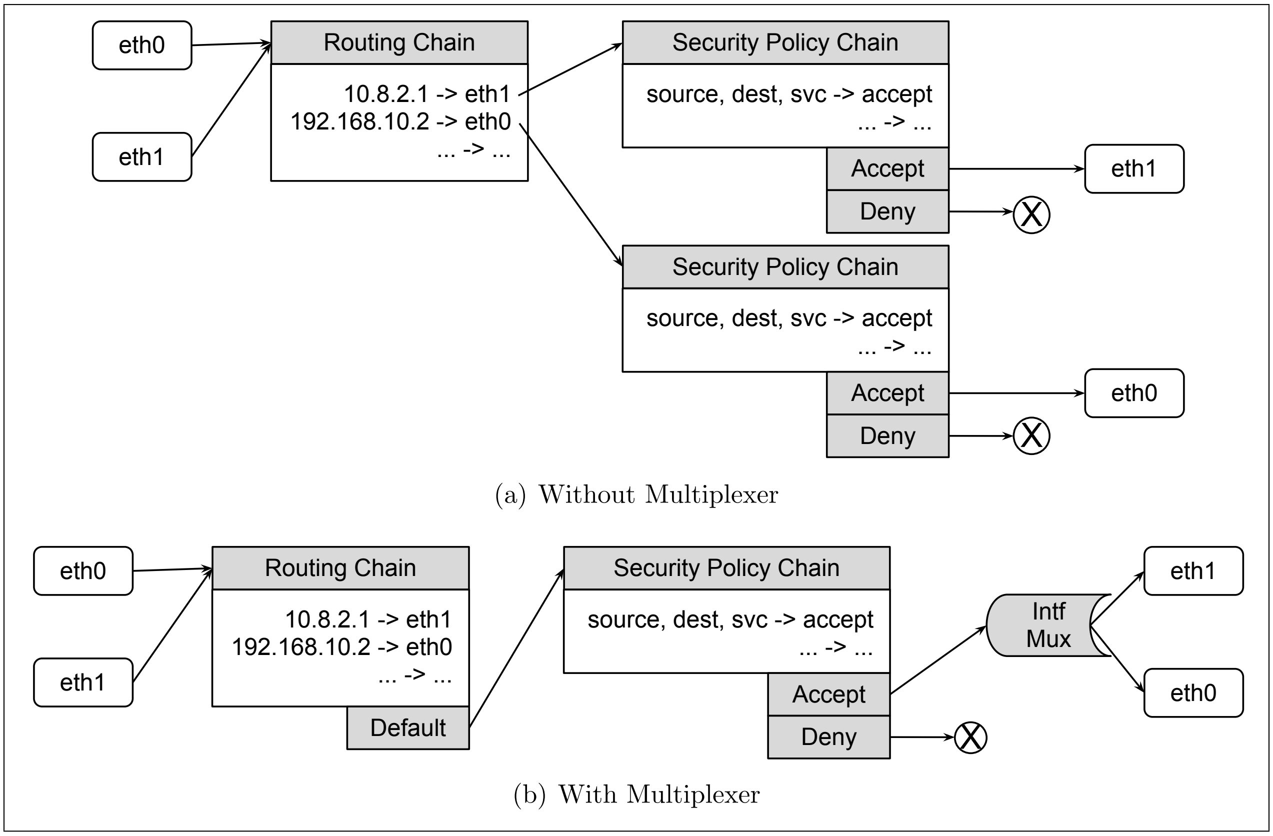
5.3.3 Interface Multiplexer

An important concept we present to reduce the number of paths that must be represented in a spanning graph is the use of an *ingress interface*, an *egress interface action*, and an *interface multiplexer*. Multiplexers may be placed in the behavior chain model to act upon two elements. The first is the inbound interface the traffic passes. The second is a matching state transition behavior rule that identifies the egress interface at some point in the spanning graph. The reason for providing this capability is that different firewall vendors make egress interface decisions at different times throughout the control flow. The egress interface sometimes will not impact the internal behavior chains traversed until the final step. Without having the means to act later on a state that was set early in the path, many duplicate paths and chains would be needed. Figure 5.2 demonstrates a spanning graph without and with an interface multiplexer such that without an interface multiplexer, the security policy chain is duplicated. By using the multiplexer in Figure 5.2(b), the potentially large security policy chain will only need to be modeled once. Figure 5.2(b) also demonstrates how the egress interface is selected in the routing chain and then acted upon at the interface multiplexer.

Another reason that interface multiplexers are useful is for firewalls that employ *zone* definitions. A zone in firewall parlance is typically the grouping of a number of interfaces into a logical area of the network. For example, eth0 and eth1 are considered an *internal zone* with eth2 and eth3 in the *unsafe zone*. A vendor may then identify a security policy when the traffic is passing from zone-to-zone and is specific to that zone-to-zone transition. In this example, there may exist a security policy that will need to be applied if the traffic arrives in the internal zone and is destined for the unsafe zone. Without an interface multiplexer there would need to be a path for every interface to interface combination. This is regardless if those interfaces shared the same zones, with the result being duplicated behavior chains and paths.

5.3.4 Spanning Graph

All of the elements described form a directed acyclic graph rooted at the ingress interfaces with the leaves identified as the egress interfaces. This allows simulation of the traffic to be based on interface origination. This is also a mechanism to compare two firewall types that process traffic differently, but expect the same external results. Figure 5.3 is an example of



(a) Without Multiplexer

(b) With Multiplexer

Figure 5.2: Behavior spanning graph (a) without and (b) with a multiplexer.

the behavior elements coming together to form a behavior spanning graph. In subsequent sections, traversing this spanning graph with the correct data structure will allow for a better understanding of the traffic that may pass the firewall.

5.3.5 Modeling Sub Elements

Until this point we have covered the general elements of the behavior abstraction model such that there was a single routing chain, security policy chain, or destination NAT chain. However, the reality of modern firewalls is that they are often constructed of smaller elements that may be linked and reused. Constructs such as virtual routers and zone policies may easily be represented as their own behavior chains that are linked. For example, a virtual router is typically a routing table with an action of "next", taking the processing to another chain until finally an *egress interface* decision is made. Figure 5.4 diagrams this behavior

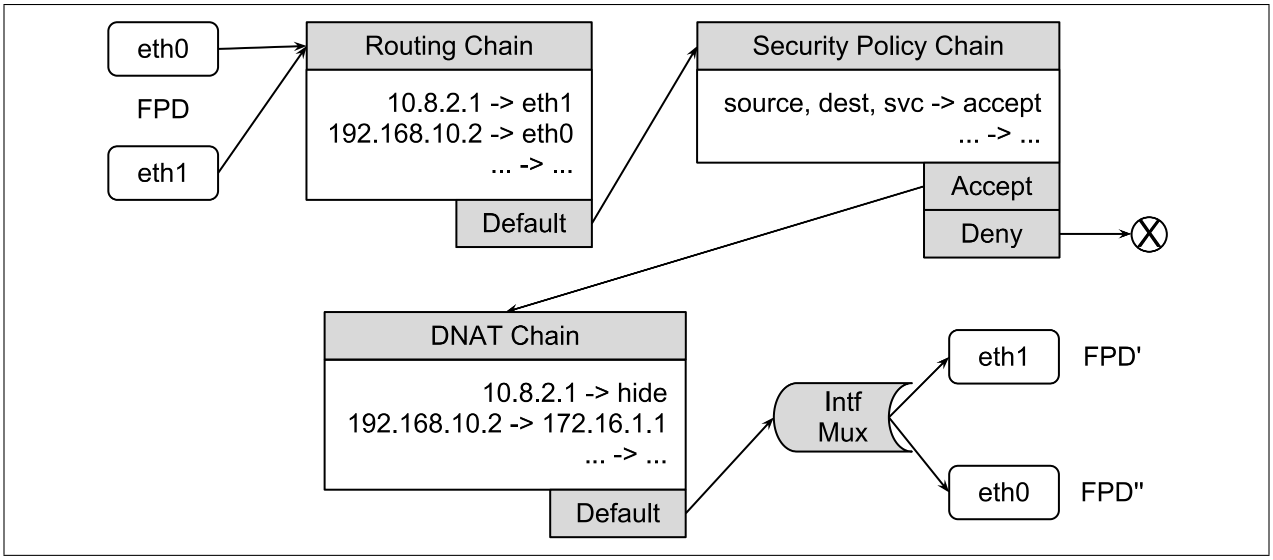


Figure 5.3: Example behavior spanning graph with FPD.

chain configuration such that the first routing chain uses *next* action that matches the packet predicate in order to direct the processing path to the virtual router chain. Once in the virtual router chain, the *egress* interface is selected and processing continues to the next chain.

Furthermore, zone-to-zone policies may be represented by using an interface multiplexer before selecting the security policy to be processed. Figure 5.5 demonstrates an example configuration where the interface multiplexer is used once to select the zone policy, and then again to select the egress interface. In addition, Figure 5.5 illustrates the expressiveness of the behavior chain model such that zone and global security policies may be represented as stand alone chains that are linked together. This is in contrast to a more naive abstraction that requires constructing the security rules that will be applied for all possible interface to interface combinations.

5.3.6 Behavior Translation

It is important to note that understanding and modeling a particular vendor's firewall as accurately as possible will lead to the best results. This means using the vendor provided documentation and tools to know when certain elements of the processing stack are applied. For example, Cisco Systems "packet tracer" gives a good understanding of what operations happen in order, providing the means to replicate that process in common behavior chains [Cisco, 2013]. Therefore, each vendor product will most likely require its own translator from the raw configurations of security rules, routes, and NATs into a cogent approximation of behavior chains.

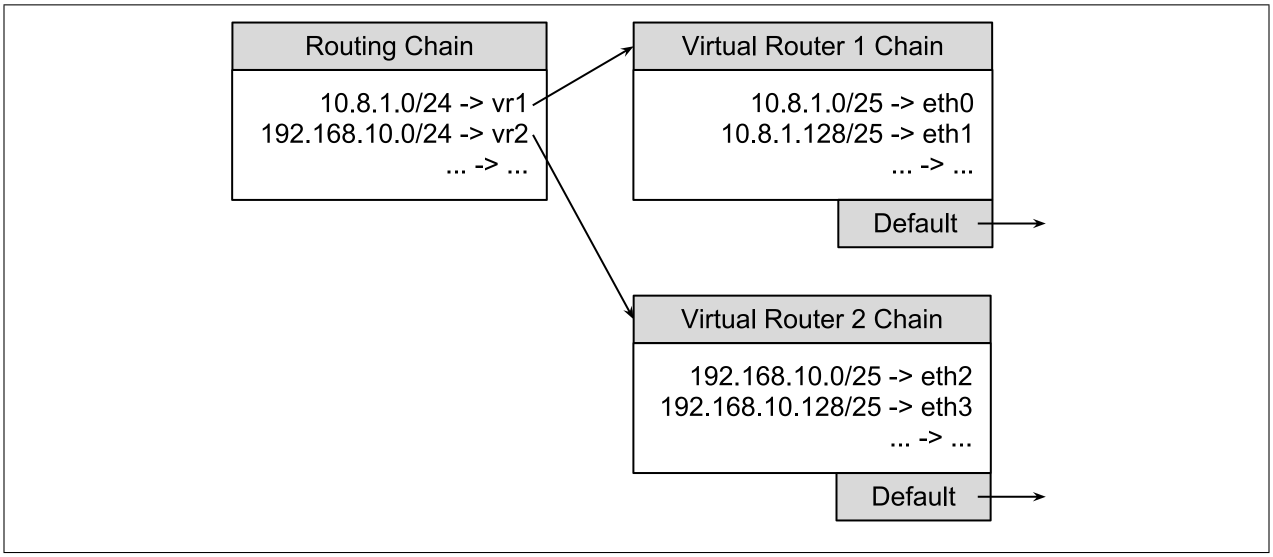


Figure 5.4: Example virtual router configuration.

5.4 Modeling Traffic Solution Space

In the previous sections we have gone through a detailed explanation of how individual elements of a firewall may be abstracted into a directed acyclic graph that may be traversed from inbound interface to outbound interface. This is useful for individual packet trace, however, the more important capabilities come when used with a data structure capable of representing the entire solution space that a packet may represent, i.e., the Firewall Policy Diagram (FPD).

5.4.1 Walking the Spanning Graph

When considering how data will pass through a firewall device, they must first be accurately modeled using behavior chains. The next step to fully understand what will pass through a firewall configuration is to construct an FPD representing the entire solutions space, U, at each interface root. Then the spanning graph is traversed from root to depth with the FPD splitting and mutating as each behavior chain is processed until it reaches the egress interface leaf. Each leaf FPD result is OR'd together to produce the final FPD at that leaf, representing an accurate space of what traffic can pass through the firewall and out of that interface. The process tests all possible scenarios through all inbound interfaces. It is this sort of formal verification that allows a firewall configuration team to be confident that they are not allowing any more access than the business requires and managing risk appropriately. Figure 5.3 demonstrates this operation visually by starting with an FPD at the root of the spanning graph, eth0 and eth1. As the FPD traverses the behavior chains a portion of the space ends up at the leaf interfaces, eth1 and eth0. The resulting FPDs at each interface are then combined with an OR operation that results in FPD' at eth1 and FPD" at eth0, representing the traffic space that has passed through the device and out those individual interfaces.

5.4.2 Performance: Avoiding the Linear Case

When taking the behavior chains and behavior rules that make up each of the individual firewall entities, the initial ideas of how traffic may be processed is linear. Meaning, treating traffic as a true firewall does, attempting a match in a linear method, one behavior rule at a time to one packet at a time. While this is straightforward, it is also performance prohibitive as the full testing of a firewall configuration requires $2^{88} \times br$. Where br is the number of behavior rules that exist in the device and the 2^{88} is the solution space represented by an IPv4 packet. It becomes exponentially worse when dealing with IPv6.

With a reformulation of each behavior chain, the processing time may be bound to the number of decisions that must be made, as opposed to the number of behavior rules. This will be substantially smaller than the number of rules and in most cases be considered constant time. For instance, a behavior chain is formed from a single security policy of a firewall

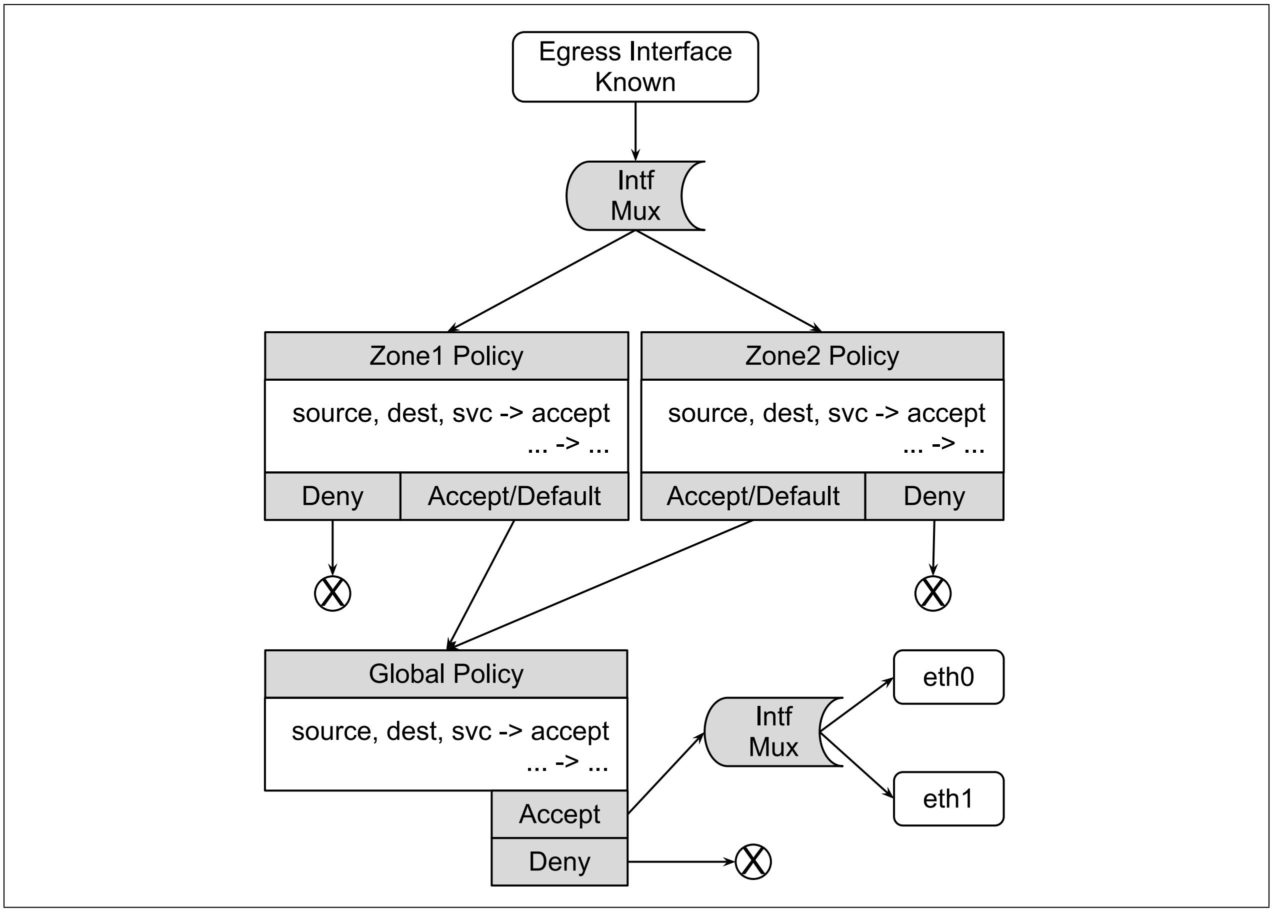


Figure 5.5: Example security zone configuration.

containing 1,000 security rules. Each security has one of two decisions, *accept* or *deny.* The linear time processing would put the cost at $2^{88} \times 1,000$. However, if we instead model the behavior chain as two FPDs, one for the *accept* traffic and one for the *deny* traffic, the results are $2^{88} \times 2$. Furthermore, we can make the entire operation constant by modeling the solution space as an FPD as well, replacing 2^{88} with a constant 88 thus making it a constant time operation to know what is an accept and what is a deny and follow the paths appropriately.

This same model may then be replayed for behavior chains that represent routing tables and NAT tables. The formulation of the problem is different, but is still a factor of the number of decisions that must be made. In a review of the routing behavior chain, the decision to be made is what egress interface matches the traffic. Therefore, there an FPD is produced for each egress interface and the processing time is a factor of that number. Because most firewalls have a fixed number of interfaces, typically less than 15 [Wool, 2004], the processing time will almost always be better than the number of routes. In a similar manner, NAT behavior rules may be grouped and stored based on the field being translated to and the value of that field. An example is grouping an FPD based on the Destination fields with the same value.

Notably, what was presented in this section still requires linearly processing the behavior rules once in order to build the FPD data structures in memory. In addition, the behavior chain FPDs may be cached to avoid a reprocessing cost. However, by using the FPD linearly processing all possible traffic may be avoided and reduced to a constant operation, by far the largest percentage of time.

5.4.3 Understanding Traffic Flow

The performance improvements in the previous section are very important for understanding access using the entire solution space. However, it will most likely only act as an initial pass of the data in the steps to fully understand why traffic flowed the way it did. The second step involves taking a subset of the traffic that was an anomaly or require further attention and sending it back through the device configuration purposely processing each behavior rule and creating a spanning tree where the nodes are the steps taken through the device. This sort of in depth analysis allows the understanding of why the traffic flowed the way it did, what rules it matched, etc.

5.5 Experiments

An experiment to test the performance of a common filtering and routing firewall was designed containing 10,000 routes and 10,000 security rules with two interfaces, eth0 and eth1. Figure 5.6 is a graphical depiction of the behavior chains and where the two FPDs represent-

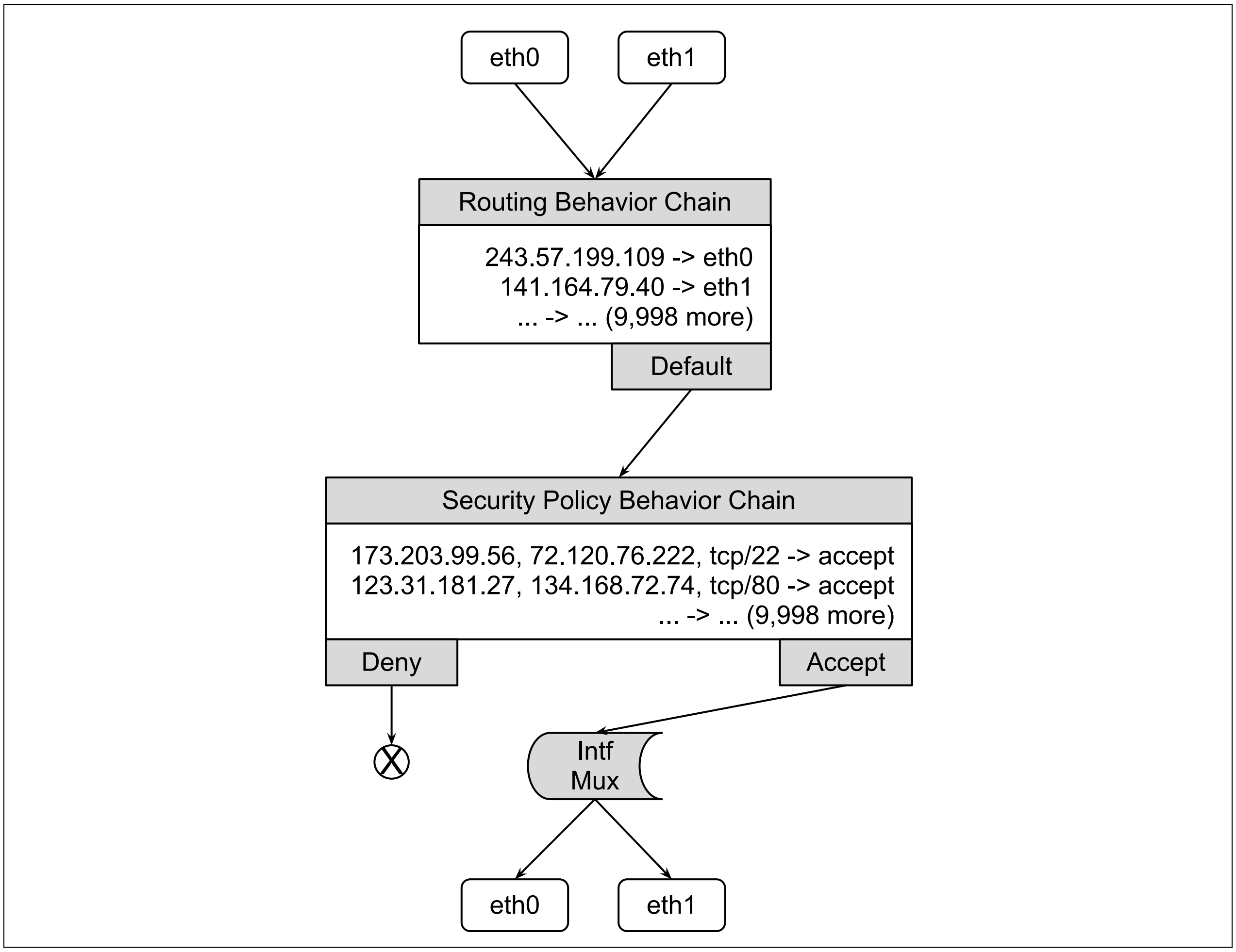


Figure 5.6: Experimental setup.

ing all possible traffic, U, are sent through from each ingress interface. In the experiment, the cost of loading each behavior chain FPD was measured in addition to the time taken for the simulated traffic to flow through the device and collect in the egress interfaces.

The results show that preloading the behavior chain data structures took 4.7 seconds with the majority of the time being the linear scan of the 20,000 behavior rules (10,000 security rules and 10,000 routes). The time taken to test all possible packets through the simulated device was 210 milliseconds. This number is many orders of magnitude better than the brute-force testing methods found in industry today. The results support the use of this behavior abstraction model and simulation in industry firewall configuration simulations. A sub-second simulation number is important because it will support further expansion of the behavior abstraction model such that it may participate in the larger network topology, both helping to verify and secure the organization.

5.6 Related Work

ACL modeling is a widespread area of study in computer networking; however, very few works attempt to model all the nuances of a modern firewall and fewer still include NAT modeling. It is very useful to include the ability of accurately representing a system of interconnected devices within software with the intent of validation and verification checking. Not only is it less expensive than the more common task of testing connectivity, but the correct data structures allow the formal testing of all possible paths without the cost of actually testing paths individually, as presented in this chapter. Until this point, brute force testing of all possible paths through a real network has been the method which related works have verified device configuration behavior.

One of the earliest attempts to model a firewall is based on the capability of generating individual packet test cases for testing a real firewall [Jürjens and Wimmel, 2001]. They use a general CASE (computer aided software engineering) tool in which they derive test cases to check for firewall vulnerabilities. The method is useful for verification of the firewall software function; however, it is different from what we present, as their goal was to verify firewall function rather than configuration. In addition, the model checking neglects to address other functions of a modern firewalls, namely, routes, NAT, and multiple internal policies. However, this is one of the first attempts to model a network with ACLs and other actors as a state transition diagram.

Brucker *et al.* (2008) presents a simple network modeling formalization with the common packet tuples found in many other works [Al-Shaer and Hamed, 2002, 2004a,b; Bartal et al., 1999; Brucker et al., 2008; Chao, 2011; Gouda and Liu, 2004; Liu and Gouda, 2005; Yuan et al., 2006]. Brucker *et al.* (2008) uses a model based testing tool in conjunction with a testing framework that analyzes a single policy rule set and generates a list of test cases to be run against the actual firewall with an expected result [Brucker et al., 2008]. The work is not capable of modeling or measuring output from multiple interfaces, routing tables, or network address translation rules. It appears to be strictly focused on what a single policy allows. In addition, the work does not appear capable of handling the size of the solution *space* of the industry, where experimentation of policies with seven rules were used. Industry firewalls will average more than a hundred times that number, thus it is difficult to say if the method is applicable.

In a similar manner to our work, [Jeffrey and Samak, 2009] presents a firewall modeling framework based on Netfilter [Russell, 2011] and the chaining mechanism. However, Jeffrey *et al.* (2009) make the argument against using BDDs for processing through the chains, and instead builds a decision graph representing all possible ways through an individual policy and then broadened out to a network. The work also uses a portion of what a BDD is capable of representing, a SAT solver. This work considers routing topologies and presents a method to condense a multi-policy and multi-device topology into a single policy with the representation of interfaces in the tuples of the model. While the work shows comparable performance numbers with the FPD, it does not include modeling a very important portion of modern networks, namely, source and destination network address translation. The performance comparisons of a SAT model to a BDD SAT model also overlook the performance benefits of a BDD with a security policy. Since the policy can be distilled into an *accept* and *deny* decision BDD, the matching of traffic can be done in constant time and does not grow with the size of the policy. Finally, it not clear from the work if the entire *space* represented by a packet can be quantified and subsequently fully understand what is considered reachable through a network.

Turley and White (2009) discuss a broad idea that firewalls have common elements such as an ingress and egress point; and within the firewalls the packet is manipulated and subsequently forwarded or dropped [Turley and White, 2009]. This is defined as behavior, and while a general concept of a filtering network device is presented, specifics on how that is modeled and accomplished are not discussed.

El-Atawy *et al.* presents a framework to generate test data for testing an actual firewall device. The work focuses on the ACL and does not cover the many other capabilities of modern firewalls (routes, NAT, etc.). While they focus on generating the smallest number of test cases based on network segmentation, there is also a behavioral modeling component to the effort. They use BDDs to check the rules for the appropriate network segments to include in the test, although it is not clear how the data are extracted from the underlying model [El-Atawy et al., 2005].

[El-Atawy et al., 2007; Tuglular et al., 2009] present a framework that is capable of generating validation traffic for a particular firewall. In a similar manner to other topology mapping programs [Govindan and Tangmunarunkit, 2000; Xie et al., 2005], the work sends actual traffic through the system, introducing the potential for disruption. The intent of the work is to generate a list of test cases that will then be able to exercise a firewall's internal filtering to ensure consistency. It is not intended to abstract behavior for formal verification of the security policy. It also does not cover any other behaviors common to a modern firewall, routing and network address translation.

[Zaliva, 2008] presents a survey of firewall modeling. This work demonstrates that the focus is typically on the individual firewall policy and some efforts on modeling the security posture as a broader inter-connected network. [Bartal et al., 1999; Mayer et al., 2000; Wool, 2001] discuss analysis engines; and while they all advance the state of behavior modeling of firewalls, they neglect certain portions of actual firewalls. [Bartal et al., 1999] is unable to accurately model NAT and [Mayer et al., 2000] does not model routing decisions, which are both important elements in attempting to understand the accuracy of traffic through ingress to egress interfaces. [Wool, 2001] strictly models single device, single policy ACLs.

FIREMAN focuses on finding configuration anomalies in firewall policies. The primary similarity with our work is the extension of a *chain* model that is also extended from Linux

Netfilter [Russell, 2011]. The work advocates converting individual and networked policies into a linear chain of processing that includes rules and links between firewalls [Yuan et al., 2006]. Our work extends some of the concepts presented to cover NAT, interfaces, and routing tables. In addition, instead of enumerating all possible linear activities which might occur in a complex network, our work focuses on sharing paths through a spanning directed acyclic graph structure.

The work most closely related to ours involves modeling attack graphs for exploitation of individual hosts in a network [Ingols et al., 2009]. The effort uses the chains concept [Russell, 2011] in portions of the work to reflect how a virtualized *space* might flow through the network. In addition to ACLs, the work is able to model NAT and routes. However, the primary difference from our work is the use of the data structure that is processed and the enhancements in our work to the chains concept that allows selection and identification of policies based on ingress and egress interfaces. This is something required for modeling modern firewalls. In addition, the spanning graph that is produced in our work differs from the reduced attack graph represented in [Ingols et al., 2009], both in the goal of the intermediate representation of the network, and its purpose.

5.7 Chapter Summary

In this effort we have presented a framework for modeling the known capabilities of modern firewalls such that the devices may be abstracted from the manufacturer specific implementation details. This abstraction allows a firewall to be modeled in a common manner in software, such that functional and configuration oriented testing may be achieved with more sophisticated plans than using a brute force, test all possibilities, approach that is seen today. In addition, this framework may be extended to assist in translation and migration from one vendor specific implementation to another, thus allowing the verification of the translation of complicated firewalls with certainty.

Chapter 6

Conclusions

In this dissertation many elements of firewalls and networks were covered with the goal of providing a framework for simulating and comprehending the large solution space represented by the addressable network space. Chapter 3 began by defining the foundations of data structures and algorithms necessary for human comprehension of the large solution space, the Firewall Policy Diagram. Chapter 4 built on that foundation by providing a formal language for querying the solution space and a mechanism to manipulate the space in a policy database. Chapter 5 discussed how a modern firewall may be abstracted into a set of common elements, such that those elements may be linked together and tested using the entire solution space represented by the FPD.

The following contributions were researched and discussed:

- Presented a set of data structures and algorithms for human comprehension of the addressable network space.

- Discussed firewall rule de-correlation as it relates to the policy gaining a holistic view of access. We demonstrated how the FPD removes any overlapping rules such that it models the actual ACCEPT and DENY space.

- Demonstrated FPD capabilities to perform the basic set operations *and*, *or*, and *not*. Using these basic operations on the data structure allows reasoning about firewall policy changes over time. In addition, they provide the foundation for FPQL and Behavior Abstraction Models.

- Presented an algorithm to reconstitute a set of human comprehensible rules from decomposed hierarchical data set representing the solution space.

- Designed and implemented an expressive query language that a team of firewall administrators would be able to use to answer important questions about what is contained in a large, and often very difficult to understand, set of firewall policies.

- Provided a syntax similar to Structured Query Language (SQL) such that the users describe what they desire in a declarative manner, allowing the system to find and return the information. This is in contrast to a more procedural style of telling the system how to find the information.

- Demonstrated experimentally that FPQL and the policy database are capable of handling industry caliber data sets, presenting the results of experiments run against large policies (up to 20,000 rules) showing how individual results can be returned in microseconds of processing time.

- Demonstrated the benefits of FPQL for formal verification efforts with single and multiple firewalls.

- Designed and implemented a Behavior Abstraction Model for the modern firewall device. This model allows implementations of vendor specific nuances in software, as well as functional testing of those capabilities in software.

- Described uses of the Behavior Abstraction Model such as network trace analysis for understanding how a packet will traverse the device without physically sending the packet. In addition to an individual packet, a packet *space* in the form of an FPD may traverse the network to understand what will make it from point A to point B.

- Provided experimental evidence that the linear case of testing all possible combinations through the model can be avoided by using the FPD. The canonical data structure and behavior abstraction models are capable of modeling very large firewalls for analysis in milliseconds of time, allowing fast verification of changes to the firewall and testing those changes completely in software.
- Finally, we discussed how the behavior abstraction models may participate in a larger software modeled network for more comprehensive simulations without requiring the physical cables and radios.

Future work will focus on modeling the disparate networking systems seen today as a software model. The intent is to include all decision making and control elements found in a network at various levels in the IP stack. The importance of the endeavor is evidenced by the large increase in the number of networked devices and the inadequate tools possessed by the industry today in understanding what is communicating. In addition, the emergence of Software Defined Networking holds the potential to further increase the control devices, making data structures and algorithms helping humans comprehend the addressable space an ongoing and interesting challenge.

Bibliography

Abedin, M., Nessa, S., Khan, L., Al-Shaer, E. S., and Awad, M. (2010). Analysis of firewall policy rules using data mining techniques. *International Journal of Internet Protocol Technology*, 5(1-2):3–22.

Al-Shaer, E. S. and Hamed, H. H. (2002). Design and implementation of firewall policy advisor tools. Technical report, School of Computer Science, Telecommunications and Information Systems, DePaul University, Chicago, USA.

Al-Shaer, E. S. and Hamed, H. H. (2004a). Discovery of policy anomalies in distributed firewalls. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2605–2616.

Al-Shaer, E. S. and Hamed, H. H. (2004b). Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, 1(1):2–10.

Al-Shaer, E. S., Hamed, H. H., Boutaba, R., and Hasan, M. (2005). Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications*, 23(10):2069–2084.

Bartal, Y., Mayer, A., Nissim, K., and Wool, A. (1999). Firmato: a novel firewall management toolkit. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 17–31.

Brucker, A., Brügger, L., and Wolff, B. (2008). Model-based firewall conformance testing.

In *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 103–118. Springer Berlin Heidelberg.

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

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318.

Chao, C. (2011). A flexible and feasible anomaly diagnosis system for internet firewall rules. In *Proceedings of the 13th Asia-Pacific Network Operations and Management Symposium*, pages 1–8.

Chapman, B. and Zwicky, E. (1995). *Building Internet Firewalls*. O'Reilly.

Chapman, D. B. (1992). Network (in)security through ip packet filtering. In *Proceedings of the 3rd Usenix Unix Security Symposium*, pages 63–76.

Chapple, M. J., D'Arcy, J., and Striegel, A. (2009). An analysis of firewall rulebase (mis)management practices. *ISSA Journal*, pages 12–18.

Cheswick, W. R., Bellovin, S. M., and Rubin, A. D. (2003). *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison Wesley.

Cisco (2013). Cisco systems packet tracer.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. McGraw-Hill Higher Education.

Deraison, R. (2005). Nessus vunerability scanner. http://www.tenable.com/products/nessus.

El-Atawy, A., Ibrahim, K., Hamed, H., and Al-Shaer, E. (2005). Policy segmentation for intelligent firewall testing. In *Proceedings of the 1st IEEE ICNP Workshop on Secure Network Protocols*, pages 67–72.

El-Atawy, A., Samak, T., Wali, A., Al-Shaer, E. S., Lin, F., Pham, C., and Li, S. (2007). An automated framework for validating firewall policy enforcement. *IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 151–160.

Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. (2001). Bugs as deviant behavior: A general approach to inferring errors in systems code. *SIGOPS Operating System Review*, 35:57–72.

Eronen, P. and Zitting, J. (2001). An expert system for analyzing firewall rules. In *Proceedings of the 6th Nordic Workshop on Secure IT-Systems*, pages 100–107.

Feamster, N. (2004). Practical verification techniques for wide-area routing. *ACM SIGCOMM Computer Communication Review*, 34(1):87–92.

Feamster, N. and Balakrishnan, H. (2005). Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 43–56.

Golnabi, K., Min, R., Khan, L., and Al-Shaer, E. (2006). Analysis of firewall policy rules using data mining techniques. In *Proceedings of the 10th IEEE Conference on Network Operations and Management Symposium*, pages 305–315.

Gouda, M. G. and Liu, A. X. (2004). Firewall design: Consistency, completeness, and compactness. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 320–327.

Govindan, R. and Tangmunarunkit, H. (2000). Heuristics for internet map discovery. In *Pro-*

ceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, volume 3, pages 1371–1380.

Guttman, J. D. (1997). Filtering postures: Local enforcement for global policies. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 120–129.

Guttman, J. D. and Herzog, A. L. (2005). Rigorous automated network security management. *International Journal of Information Security*, 4:1–2.

Hamed, H. H. and Al-Shaer, E. S. (2006). On autonomic optimization of firewall policy organization. *Journal of High Speed Networks*, 15(3):209–227.

Hazelhurst, S., Attar, A., and Sinnappan, R. (2000). Algorithms for improving the dependability of firewall and filter rule lists. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 576–585.

Horowitz, E. and Lamb, L. (2009). A hierarchical model for firewall policy extraction. In *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications*, pages 691–698.

Ingols, K., Chu, M., Lippmann, R., Webster, S., and Boyer, S. (2009). Modeling modern network attacks and countermeasures using attack graphs. In *Proceedings of the 2009 Computer Security Applications Conference*, pages 117–126.

Jeffrey, A. and Samak, T. (2009). Model checking firewall policy configurations. In *Proceedings of the 2009 IEEE International Conference on Policies for Distributed Systems and Networks*, pages 60–67.

Jürjens, J. and Wimmel, G. (2001). Specification-based testing of firewalls. In Bjørner, D., Broy, M., and Zamulin, A., editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 308–316. Springer Berlin Heidelberg.

Kurose, J. F. and Ross, K. W. (2007). *Computer Networking: A Top-Down Approach*. Addison Wesley, 4th edition.

Lind-Neilsen, J. (2004). Buddy version 2.4. http://sourceforge.net/projects/buddy.

Liu, A. (2007). Change-impact analysis of firewall policies. In *Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 155–170. Springer Berlin / Heidelberg.

Liu, A. X. and Gouda, M. G. (2004). Diverse firewall design. In *Proceedings of the 2004 IEEE International Conference on Dependable Systems and Networks*, pages 595–604.

Liu, A. X. and Gouda, M. G. (2005). Complete redundancy detection in firewalls. In *Proceedings of the 19th Annual IFIP Conference on Data and Applications Security*, pages 196–209.

Liu, A. X. and Gouda, M. G. (2008). Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1237–1251.

Liu, A. X. and Gouda, M. G. (2009). Firewall policy queries. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):766–777.

Liu, A. X., Gouda, M. G., Ma, H. H., and Ngu, A. H. (2004). Firewall queries. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, pages 124–139.

Liu, A. X., Gouda, M. G., Ma, H. H., and Ngu, A. H. (2005). Firewall queries. In Higashino, T., editor, *Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg.

Mahajan, R., Wetherall, D., and Anderson, T. (2002). Understanding bgp misconfiguration. In *Proceedings of the 2002 ACM SIGCOMM*, pages 3–16.

Maltz, D., Xie, G., Zhan, J., Zhang, H., Hjalmtysson, G., and Greenberg, A. (2004). Routing design in operational networks: A look from the inside. *ACM SIGCOMM Computer Communication Review*, 34:27–40.

Mayer, A., Wool, A., and Ziskind, E. (2000). Fang: A firewall analysis engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 177–187.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Qiu, L., Varghese, G., and Suri, S. (2001). Fast firewall implementations for software-based and hardware-based routers. *SIGMETRICS Performance Evaluation Review*, 29:344–345.

Russell, R. (2011). The netfilter.org project. http://www.netfilter.org.

Sebesta, R. W. (2009). *Concepts of Programming Languages*. Pearson, 9th edition.

Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723.

Spring, N., Mahajan, R., Wetherall, D., and Anderson, T. (2004). Measuring isp topologies with rocketfuel. *IEEE/ACM Transactions on Networking*, 12:2–16.

Sung, Y. E., Lund, C., Lyn, M., Rao, S. G., and Sen, S. (2009). Modeling and understanding end-to-end class of service policies in operational networks. *ACM SIGCOMM Computer Communication Review*, 39:219–230.

Tuglular, T., Kaya, O., Muftuoglu, C., and Belli, F. (2009). Directed acyclic graph modeling of security policies for firewall testing. In *Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 393–398.

Turley, P. and White, E. (2009). System and method for behavior-based firewall modeling. Patent. US 7610621.

Wang, W., Chen, W., Li, Z., and Chen, H. (2006). Comparison model and algorithm for distributed firewall policy. In *Proceedings of the 2006 International Conference on Intelligent Computing: Part II*, pages 545–556.

Whaley, J. (2007). Javabdd version 1.0b2. http://javabdd.sourceforge.net.

Wool, A. (2001). Architecting the lumeta firewall analyzer. In *Proceedings of the 10th conference on USENIX Security Symposium*, volume 10, pages 85–97.

Wool, A. (2004). A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67.

Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. (2005). On static reachability analysis of ip networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 2170–2183.

Yin, Y., Bhuvaneswaran, R., Katayama, Y., and Takahashi, N. (2006). Inferring the impact of firewall policy changes by analyzing spatial relations between packet filters. In *Proceedings of the 2006 International Conference on Communication Technology*, pages 1–6.

Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C., and Mohapatra, P. (2006). Fireman: A toolkit for firewall modeling and analysis. *IEEE Symposium on Security and Privacy*, pages 199–213.

Zaliva, V. (2008). Firewall policy modeling, analysis and simulation: a survey. *Source-Forge, Tech. Rep*, pages 1–11.