FIREWALL POLICY OPTIMIZATION AND MANAGEMENT

By

ASHISH TAPDIYA

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

December 2008

Winston-Salem, North Carolina

Approved By:

Errin W. Fulp, Ph.D., Advisor ____________________

Examining Committee:

David J. John, Ph.D., Chairperson ____________________

Todd C. Torgersen, Ph.D. ____________________

Acknowledgements

I would like to thank my advisor, Dr. Errin W. Fulp for his patience, guidance, and the innumerable hours that he has spent with me during the entire research process to bring this thesis to its current state. My gratitude towards him cannot be expressed in words. I would also like to thank my co-advisor, Dr. Todd C. Torgersen for taking time out of his busy schedule and helping me develop an insight into the problem which has been instrumental in the success of this project.

I will always be grateful to Dr. William H. Turkett who has always been ready to help me, and has been a continous source of inspiration in the last two years. I would also like to thank Dr. David J. John for personal support and shielding me from any kind of administrative hassle.

I would like to thank Robert Anderson and Great Wall Systems, Inc for providing me with the opportunity to work on interesting projects during the summer of 2008, which helped in my professional development as well as writing my thesis.

Finally, I would like to thank my family for supporting me all the way through. I would also like to thank my friends Eddie, Matt, Nikhil, Samrat and Sebastian. My life as a graduate student would not have been the same without these friends.

Table of Contents

Acknowledgements ..... **ii**

List of Tables ..... **v**

List of Figures ..... **vi**

Abstract ..... **vii**

Chapter 1 Introduction **1**
- 1.1 The Case for Network Security 2
- 1.2 The Case for Network Firewall 3
  - 1.2.1 Firewall Services 3
- 1.3 Firewalls and High Speed Networks 4

Chapter 2 Overview of Network Firewalls **6**
- 2.1 What is a Firewall Rule? 6
- 2.2 Network Firewall Policies 7
- 2.3 Packet Matching Process 8
  - 2.3.1 Best Match 9
  - 2.3.2 Last Match 9
  - 2.3.3 First Match 10

Chapter 3 Policy Management **11**
- 3.1 Rule Intersection 11
- 3.2 Modeling Precedence Relationships 12
- 3.3 Rule Subsets and Anomalies 14

Chapter 4 Optimal Rule Ordering **18**
- 4.1 Optimal Rule Ordering 19
  - 4.1.1 Single Machine Job Scheduling with Precedence Constraints 19
  - 4.1.2 Finding The Optimal Firewall Rule Set Ordering 20
  - 4.1.3 Single Machine Job Scheduling Reduces To Optimal Firewall Rule Ordering 22

Chapter 5 Firewall Policy Sorting Algorithms **24**
5.1 The α Algorithm 24
5.1.1 Example That Results In A Non-Optimal Ordering 27
5.2 The β Algorithm 28
5.2.1 Example That Results In A Non-Optimal Ordering 31
5.3 The γ Algorithm 33
5.3.1 Example That Results In An Optimal Ordering 34
5.3.2 Examples That Results In Non-Optimal Ordering 36
5.4 Integrity Validation Post Optimization 39

Chapter 6 Results and Analysis **41**
6.1 Edge Density Versus Number of Breaks 41
6.2 Break Cases Overlap 43
6.3 Average Delay versus Edge Density 44
6.4 Zipf Probability Distribution 46

Chapter 7 Conclusions and Future Work **49**
7.1 Conclusions 49
7.2 Future Work 51

References **53**

Vita **55**

List of Tables

2.1 IP address representations. 6
2.2 Port address representations. 6
2.3 Example rule actions. 7
2.4 Example security policy consisting of multiple ordered rules. 8
2.5 Example packet header information. 9

3.1 Rules r_2, r_4 : Corresponding parameter intersection values. 11
3.2 Example security policy demonstrating rule shadow. 15
3.3 Example security policy demonstrating set shadow. 15

5.1 Example security policy consisting of multiple ordered rules. 25
5.2 Example security policy for which α ordering is not optimal. 28
5.3 Example security policy for which the β ordering is not the optimal ordering. 31
5.4 Example security policy for which the γ ordering is not the optimal ordering. 37
5.5 Example security policy for which γ ordering is not optimal ordering. 38

List of Figures

3.1 DAG for policy in table 2.4. 13
3.2 DAG generator algorithm. 13
3.3 Shadow detection algorithm. 16

5.1 Algorithm α. 24
5.2 Policy DAG representations of the rules given in Table 5.1. 26
5.3 Policy DAG representations of the rules given in Table 5.2. 27
5.4 Algorithm β. 29
5.5 Different policy DAG representations of the firewall rules given in table 5.3. 33
5.6 Algorithm γ data structures. 34
5.7 Algorithm γ. 35
5.8 Different policy DAG representations of the firewall rules given in 5.4. 36
5.9 Different policy DAG representations of the firewall rules given in table 5.5. 37
5.10 Algorithm for integrity validation. 39

6.1 Average number of breaks versus Edge density for different policy sizes. 42
6.2 Number of edges versus Number of rules versus Standard deviation of probability distribution. 44
6.3 Edge density versus Average number of rule comparisons. 45
6.4 Zipf distribution. 48

Abstract

Firewalls enforce a security policy by inspecting packets arriving or departing a network. This is accomplished by sequentially comparing the policy rules with the header of an arriving packet until the first match is found. This process becomes time consuming as policies become larger and more complex. For example, a firewall connecting two high speed networks is responsible for processing heavy network load and can easily become a bottleneck. Therefore determining the appropriate action for arriving packets must be done as quickly as possible.

The process of packet header matching can be improved if more popular rules appear earlier in the policy. Unfortunately, a simple sorting algorithm is not possible, since the relative order of certain rules must be maintained in order to preserve the original policy intent. Using directed acyclical graphs to represent the firewall policy, this thesis shows that determining the best order of firewall rules is equivalent to job-shop scheduling, a known $\mathcal{NP}$-hard problem. The sorting techniques are novel in that they consider sub-graphs of rules (inter-related by precedence constraints) and compare the advantage of placing and merging the nodes that comprise them. For policy management, a shadow detection algorithm is presented to detect anomalies.

Chapter 1: Introduction

With the increase in global Internet connectivity and the innumerable attacks that a network is susceptible to, network security has become the cynosure in both research and industrial communities. Networking technologies such as *firewalls* form the first line of defense for a network and enforce a security policy by filtering malicious or unsolicited packets from incoming packet stream.

With the increase in transmission medium speeds, a firewall connecting two high speed networks is responsible for processing a heavy network load and hence can easily become a bottleneck. This problem is aggravated by the growth and continuous addition of network services, requiring a more detailed firewall policy for providing comprehensive defense against adversaries. As the size of a policy increases incoming packets will have to compared against a large set of rules and packet inspection will require more work.

This thesis builds upon previous work performed in the area of firewall policy optimization and management, specifically presenting new algorithms for reordering the rules in firewall policy. Although an $\mathcal{NP}$-hard problem, the proposed methods allow for lower delays in packet inspection while maintaining the integrity and intent of the original policy. In addition to the algorithms for reordering the rules, this paper presents an algorithm for detecting anomalies in a policy, resulting in enhanced policy management.

1.1 The Case for Network Security

Networks allow computers to share data and perform distributed computing. Unfortunately networks can potentially serve as a way to break into a host, enabling an attacker to steal, alter or destroy information, or prevent a legitimate user from accessing resource by performing denial of service attack. Corporations often have confidential documents, trade secrets which are highly coveted by competitors. The CRS report for congress [9], describes cybercrime and cyberattack as a growing threat to the national security and critical infrastructures pertaining to government as well as civilians.

A variety of security models can be deployed which differ in their placement and the range of vulnerabilities that they protect against. No security is the simplest model and involves putting no effort in the security. Another model referred to as security through obscurity - it presumes a system to be secure since nobody knows about it. The host security model enforces security of each host separately. One of the obvious shortcoming of this approach is its inability to scale to a large number of machines. Network security approach is more amenable to scalability and controls the network access to hosts. However, any single security model or approach is not comprehensive by itself. Hence a layered security model that combines different approaches should be used [24].

A firewall gives an enormous amount of leverage for network security by enforcing a security policy, which allows approved services to pass through. Therefore, an optimal policy which provides low average delay per packet is extremely important from a network security and performance standpoint since it can quickly find the appropriate action for any given packet.

1.2 The Case for Network Firewall

Interconnection of computers into a network forms the foundation for sharing information resources such as electronic mail, hypertext documents, etc. The popularity of the information superhighway is increasing everybody's desire to use it. The risk associated with its popularity is also obvious and demands for the protection of internal networks from miscreants and adversaries.

A firewall between two networks is analogous to the moat of a medieval castle and restricts all the traffic to enter or leave at a controlled point. As a single choke point firewalls enable us to focus our security efforts at this point where our internal network connects to the external network. Since all traffic passes through the firewall it provides to be a pertinent place to collect log data about network use and/or misuse.

Firewalls provide effective protection against a wide range of network attacks. However, there are limitations and it's not a comprehensive security solution that can prevent all possible threats. As new threats come to light, firewalls need to be updated accordingly; hence firewalls cannot protect against novel attacks. A firewall cannot protect against a malicious insider as traffic pertaining to internal network does not pass through it.

1.2.1 Firewall Services

In addition to packet filtering, firewalls provide several other services including Quality of Service (QoS), Network Address Translation (NAT) and stateful packet inspection. IPV4 Internet is a 'best effort' packet delivery system, however interactive multimedia applications like as Voice Over IP (VOIP), streaming video place bounds on the timing of data as late data is considered irrelevant data. Hence, a level of service must be guaranteed by classifying and prioritizing the network traffic. Also, this mechanism

can be used in deciding which traffic should be dropped first in case of heavy network load.

Network Address Translation is a technique that enables a large number of hosts to connect to the Internet using a small number of allocated addresses. It can also allow a network that's configured with unroutable addresses to connect to the Internet using valid addresses. It is not a security technique, although it enhances the security by obfuscating the network topology [24].

Stateful packet inspection allows a firewall to keep information about the state of transactions. As this technology enables the firewall to see the contents of the packet, its behavior changes depending on the traffic it sees.

1.3 Firewalls and High Speed Networks

Firewalls have proved to provide effective protection against a gamut of network attacks that originate outside the network. With the increase in the number of vulnerabilities and the growing need for securing networks from attackers, firewalls are becoming ubiquitous and indispensable to the operation of network. The continuous growth of Internet and the increasing sophistication of attacks is imposing stringent demands on firewall performance. A firewall with large policy can easily become a bottleneck under attack or heavy network load [10, 11]. Hence, as the transmission speeds, size of networks and computing power of networked hosts increases, firewalls must process packets at higher speeds [5, 18, 24].

As a policy gets larger and more complex, suboptimal placement of rules in the policy can greatly reduce the efficiency of the firewall. Hence firewall rule reordering and management is extremely crucial for the performance of firewalls that connect the next generation of high speed networks. Optimal rule ordering reduces the latency experienced by the passing traffic which in turn also improves the throughput of

the system. Unfortunately finding the best order will be shown to be an $\mathcal{NP}$-hard problem. This thesis develops different approaches and algorithms for firewall policy reordering which reduce the average number of rule comparisons while maintaining the intent of the policy, as well as demonstrates the performance improvement achieved by optimal rule placement while maintaining the precedence relationships between them.

The size and complexity of policies not only impact performance, but may also make finding anomalies more difficult [3, 4, 10]. Firewall rule parameters can be ranges, hence it is possible for a incoming packet to match multiple rules in the policy. Therefore ordering among the rules is important. If a general rule is placed above a specific rule then the specific rule will never be reached, which is an anomaly. Also, a set of rules above a certain rule can shadow it which should also be categorized as an anomaly. The effectiveness of the security provided by a firewall policy is dependent on the addition, deletion and rearrangement of rules in correct place such that it does not introduce any anomaly. This thesis presents an algorithm for detecting shadows in policies that can help network administrators in managing and deploying policies as intended.

Chapter 2: Overview of Network Firewalls

This section explains the building blocks of network firewalls. This includes defining firewall policies, network packets, and firewall rules. In addition, this chapter presents the transformation of a traditional list-based security policy into a policy model which is better suited for visualization and optimization.

2.1 What is a Firewall Rule?

A firewall rule r is an ordered set of parameters $r = (r[1], ..., r[k])$, where $k > 1$. The upper bound on the value of k is network specific. For the Internet firewall rules are commonly represented as an ordered set of 5 parameters comprising: protocol, source IP address, source port, destination IP address, destination port [23, 24]. In addition to these parameters each rule r has an associated action as seen in Table 2.3.

Notation	Example	Explanation
CIDR	192.1.1.6/24	Represents a range of IP addresses between 192.1.1.0 and 192.1.1.255
Wild Card	192.1.*	Represents a range of IP addresses between 192.1.0.0 and 192.1.255.255
Range	192.1.1.5-192.1.1.230	Represents a range of IP addresses between 192.1.1.5 and 192.1.1.230
Regular	192.1.1.1	Represents a single IP address 192.1.1.1

Table 2.1: IP address representations.

Notation	Example	Explanation
Wild Card	*	Represents a range of ports between 0 and 65535
Range	20-190	Represents a range of ports between 20 and 190
Regular	80	Represents a single port 80

Table 2.2: Port address representations.

Action	Explanation
Accept	Forward the packet
Deny	Discard the packet
Log,Accept	Log and forward the packet
Log,Deny	Log and discard the packet

Table 2.3: Example rule actions.

The rule action is only performed after an incoming packet matches that particular rule and therefore the rule action is not part of rule packet comparison. Each parameter $r[l] \in r$ represents a set of values. For example the rule protocol parameter can have any value which is a permissible value for protocol field in IP packet format documented in RFC 791 [1]. Example values are TCP, UDP, ICMP and RSVP. In addition to the permissible values mentioned earlier, the protocol field can also have IP as a value which represents a wild card (*) and will match any incoming packets protocol parameter value.

Every incoming packet has a header and a payload. The header indicates the source and destination of the packet while the payload is the actual data. The header of an incoming packet d has the same parameters as a firewall rule, as seen in Tables 2.1 and 2.2. Hence, the set of packet header parameters corresponding to each rule parameter is denoted by $(d[1], ..., d[k])$, where k is the number of rule parameters. Table 2.5 represents an example packet header with parameters corresponding to and relevant for the rule packet match operation.

2.2 Network Firewall Policies

A firewall policy is traditionally an ordered list of n rules, denoted as R = $\{r_1$, r_2, r_3,..., $r_n\}$. In many implementations, the rule set is stored internally as a linked list [23]. Each incoming packet is compared sequentially to the rules, starting with the first, until a match is found. After finding a matching rule, the associated rule action

is performed on the packet.

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	190.1.1.*	*	*	80	deny	0.05
2	UDP	210.1.*	*	*	88-90	accept	0.10
3	TCP	180.*	*	180.*	90	accept	0.15
4	TCP	210.*	*	220.*	87-89	accept	0.20
5	UDP	190.*	*	*	*	accept	0.20
6	IP	*	*	*	*	deny	0.30

Table 2.4: Example security policy consisting of multiple ordered rules.

A firewall policy R is considered comprehensive if for every possible packet d, a match is found in R. In order to make a security policy comprehensive, often a default rule is added to the end of the policy that matches every incoming packet and performs the deny action on it. Table 2.4 represents an example comprehensive security policy, where r_6 is the default rule.

2.3 Packet Matching Process

As described in [10, 11] a match occurs between a rule and a packet if each parameter value in the packet header is a proper subset of the corresponding parameter value in the rule. Let the packet d and rule r_i match be denoted as,

$$d \Rightarrow r_i \quad \text{iff} \quad \forall \; l, \quad d[l] \subseteq r_i[l], \quad l = 1, ..., k$$

Since each parameter value in a rule is compared against corresponding parameter value in the incoming packet, therefore the order among rule parameters is important as well. For example, if the source and destination IP addresses are interchanged in the rule then the packet might or might not match the rule and the resulting action performed on the packet might be different than expected.

	Source		Destination	
Proto.	IP	Port	IP	Port
UDP	190.1.2.1	22	148.1.1.5	1142

Table 2.5: Example packet header information.

Given a policy R and packet d, multiple matches may be possible. The 'match policy' describes which of the multiple matching rules will be applied. There are three commonly used matching mechanisms 'first match', 'last match' and 'best match' which are described in the following subsections.

2.3.1 Best Match

An incoming packet d is compared against the entire policy. The rule that matches the closest to the packet header is selected and the action associated with it is performed on the packet. The closeness is measured by the narrowness of the rule's parameters, or the longest matching prefix [6, 7, 8, 19]. For example, the packet in Table 2.5 matches rules r_1, r_5 and r_6 from the policy in Table 2.4. However, the best match is rule r_1 and is same as the first match. As rule matching is independent of the order of rules, the default rule can be placed any where. This matching mechanism is used by the routers in routing packets.

2.3.2 Last Match

An incoming packet d is sequentially compared against each rule $r_i \in$ R, beginning at the first rule in policy, until the end of the policy. The match operation performs action associated with the last rule in policy that matched the packet. Thus for each packet entire policy is reviewed and will result in high latency as compared to the first match policy. As an example of last match consider the packet from Table 2.5 which matches rule $r_6 =$ (IP, *, *, *, *) from Table 2.4. Note, the default rule should be

placed at the beginning of the policy to prevent the denial of all traffic as otherwise every packet will match the default rule.

2.3.3 First Match

An incoming packet d is sequentially compared against each rule $r_i \in \text{R}$, beginning at the first rule in policy, until a match is found ($d \Rightarrow r_i$). The match operation ends after finding the first rule that matches the packet and the associated rule action is performed on the packet. As an example of first match, the packet from Table 2.5 matches rule r_5 and rule r_6 from Table 2.4, but rule r_5 is the appropriate match since it appears before rule r_6. First match is the most popular method for network firewalls as it provides low latency and is simple to implement, and will be the assumed matching policy for the rest of this paper.

Since a packet can match multiple rules in the policy, the precedence among the rules should be maintained. Otherwise if the order among the rules is reversed then different action may be performed on the incoming packet and will result in a violation of policy integrity. Integrity refers to maintaining the original intent of policy. Let R represents an original policy and $\text{R}^{'}$ is a reorder of R. If for every possible legal packet d the same action is performed by the two rule lists R and $\text{R}^{'}$, then the policy integrity is maintained.

Chapter 3: Policy Management

Policy management involves understanding the precedence relationship between rules, which can help identify anomalies and better orderings. This chapter describes rule intersection which forms the basis for transformation of a firewall policy into a policy Directed Acyclic Graph (DAG). It also introduces a mechanism for detecting and removing shadows in the policy which enables a policy to be anomaly free.

3.1 Rule Intersection

An important characterstic of firewall rules is precedence, which is determined by intersection of rules. Two rules r_i and r_j intersect with each other if their parameter values have nonempty intersection across all corresponding parameters. Intersection among pair of rules is possible because of the allowance of range representation across different parameters comprising a rule. The rule r_i and the rule r_j intersection can be denoted as,

$$r_i \Downarrow r_j \quad \text{iff} \quad \forall \;\; l, \quad r_i[l] \cap r_j[l] \neq \phi, \quad l = 1, ..., k$$

As an example consider the rules r_2 = (UDP, 210.1.*, *, *, 88-90) and r_4 = (UDP, 210.*, *, 220.*, 87-89) from Table 2.4.

Intersection Operation	Value
r_2[protocol] $\cap$ r_4[protocol]	{UDP}
r_2[sourcecIP] $\cap$ r_4[sourceIP]	{210.1.0.0,...,210.1.255.255}
r_2[sourcePort] $\cap$ r_4[sourcePort]	{0,...,65535}
r_2[destinationIP] $\cap$ r_4[destinationIP]	{220.0.0.0,...,220.255.255.255}
r_2[destinationPort] $\cap$ r_4[destinationPort]	{88,89}

Table 3.1: Rules r_2, r_4 : Corresponding parameter intersection values.

Table 3.1 shows that across all parameters, the intersection of parameter values yield a non null value and therefore $r_2 \Downarrow r_4$. The existence of intersections may limit orderings and cause anomalies. This is discussed further in the next section.

3.2 Modeling Precedence Relationships

As described in the previous chapter, a firewall policy R is an ordered list of rules and a packet d is sequentially compared against the rules in R beginning at the first rule until a match is found. Although described as a list, a DAG G = (R,E), can be used to model a firewall policy, where R is the set of rules in policy and E represents the set of precedence relationships between rules as described in the section 2.3.3 .

Using a DAG to model a firewall policy is valuable for two reasons. First, certain rules have precedence relationships between them and if the nearness of these rules relative to the beginning of the policy is interchanged then, different actions might be performed on the packet d. Therefore DAG enables us to capture and represent this precedence between rules by having a edge between them. Second, the problem of determining optimal rule set ordering is analogous to the job scheduling problem subject to precedence constraints and DAG has been successfully used to represent the job scheduling problem. Therefore same model will be pertinent for modeling firewall policy.

As an example of a policy DAG consider Figure 3.1 for the policy given in Table 2.4, where R $= (r_1, r_2, r_3, r_4, r_5, r_6)$ and E $= ((r_1, r_5), (r_1, r_6), (r_2, r_6), (r_3, r_6), (r_4, r_6), (r_5, r_6))$. Each edge (r_i, r_j) represents the existence of a precedence relationship between rules r_i and r_j in the policy.

For policy rules that intersect, exchanging their relative order may change the intent of policy. For example, consider the rule r_1 and the rule r_6 ($r_1 \Downarrow r_6$) from the policy given in Table 2.4. If the rule r_1 = (UDP, 190.1.1.*, *, *, 80) is swapped with

rule r_6 = (IP, *, *, *, *) then, the outcome will be a denial of all incoming packets to this firewall. Hence the swapping of rules introduces an anomaly in the policy. However, if rule r_2 is interchanged with rule r_3, the intent of policy is maintained because $r_2 \not\Downarrow r_3$. Thus certain rules have precedence relationships between them which need to be preserved in any reordering of the rules.

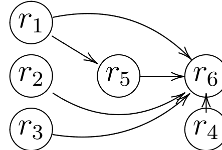


Figure 3.1: DAG for policy in table 2.4.

The precedence relationship between rules in a policy can be modeled using DAG G = (R,E) where vertices are the rules and edges are the precedence relationships [10, 11]. An edge exists between rule r_i and rule r_j, if $i < j$ (i appears before j in the original policy) and rules intersect.

```
1   // Let R be the ruleset corresponding the security policy
2   // Let n be the number of rules in the security policy
3   // Let G be the resulting DAG
4
5     function generator(R)
6         for( i = 1; i ≤ n; i++ )
7             for( j = i+1; j ≤ n; j++ )
8                 if( r_i ⇓ r_j ) then
9                     add edge ( r_i , r_j ) to G
10                end
11            end
12        end
```

Figure 3.2: DAG generator algorithm.

For example, Figure 3.1 represents the DAG generated on the execution of the algorithm given in Figure 3.2, with the ruleset corresponding the security policy presented in Table 2.4 as an input. There exist an edge between the rule r_3 and the rule r_6 in Figure 3.1 because $r_3 \Downarrow r_6$. However, there does not exist an edge between the rule

r_3 and the rule r_4 in Figure 3.1 because $r_3 \not\Downarrow r_4$. All linear permutations of the DAG will maintain the integrity, this is proved in [10, 11].

3.3 Rule Subsets and Anomalies

The shadowing of rules has been defined only in terms of two rules. A general rule earlier in policy shadows a specific rule later in the policy; however, there may be a group of rules earlier in the policy that can shadow a rule later in the policy. This complicates the policy management(anomaly discovery) and reordering process.

A rule r_i is a subset of rule r_j if for each parameter l, the value of parameter $r_i[l]$ is a subset of the value of parameter $r_j[l]$. The subset relationship between rule r_i and the rule r_j can be denoted as,

$$r_i \subseteq r_j \quad \text{iff} \quad \forall \ l, \quad r_i[l] \subseteq r_j[l], \quad l = 1, ..., k$$

As an example consider the rules from Table 3.2. Rule r_1 is a subset of rule r_3. Since each parameter of r_1 is a subset of the corresponding parameter of r_3. If a rule r_i is a subset of rule r_j where r_i appears later than r_j then, rule r_i is termed as a shadow of rule r_j [3, 4, 10, 11]. Shadowing is an anomaly since all the packets destined for rule r_i will match rule r_j only. Hence, rule r_i will never be used. Also, if the shadowed and the shadowing rules have the same rule actions then intent of policy remains same but the policy will have shadowed rule as a redundant rule. However, if the shadowed and the shadowing rules have different rule action then this should be considered as an anomaly since it will result in the acceptance of traffic that should have been denied or vice versa. Therefore, when a new rule is added to the policy its position should always be decided by checking that it does not shadow any existing rule.

The previous section described how a single rule can be shadowed by a rule that appears before it in the policy. However, there can also be a scenario in which a rule

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	190.*	*	*	*	accept	0.20
2	UDP	190.1.1.*	*	*	80	deny	0.05
3	IP	*	*	*	*	deny	0.30

Table 3.2: Example security policy demonstrating rule shadow.

r_q is shadowed by a set of rules S which appear before the rule in question in the policy [14] and intersect with it. Hence a rule r_q is set-shadow of rules in s if,

$$r_q \subseteq \bigcup_{r_k \in S} r_k$$

As an example consider rules r_1, r_2 and r_3 from Table 3.3. For r_3 source IP range = {0.0.0.2-0.0.0.6} and source Port = {1}. Incoming packets with source IP in the range {0.0.0.2-0.0.0.5} and source Port = {1} matches rule r_1 and packets with source IP = {0.0.0.6} and source Port = {1} matches rule r_2. Also, all three rules have remaining corresponding parameter values as same. Hence, any possible packet that can match rule r_3 will either match rule r_1 or rule r_2. Therefore, rule r_3 is set-shadowed by rules $\{r_1,r_2\}$. Also, both r_1, r_2 intersects with r_3 but neither of them independently shadows it.

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	0.0.0.2-0.0.0.5	*	*	*	accept	0.20
2	UDP	0.0.0.4-0.0.0.7	1-3	*	*	deny	0.05
3	UDP	0.0.0.2-0.0.0.6	1	*	*	deny	0.30

Table 3.3: Example security policy demonstrating set shadow.

A simple approach to detect a set shadow would be to have two 5-dimensional matrices where each dimension corresponds to a parameter in a rule. A matrix Q for the rule in question r_q and a matrix S for the set of rules s above the rule in question which intersect with it. Each cell in the matrix represents a discrete set of

values corresponding to a rule. Also, the IP address can be mapped to a linear range of integers between 0 and 4294967295. If a rule has wild cards and or ranges for a parameter then it would be represented by multiple cells in the matrix.

```
1   Let r_q be the rule in question
2   Let s be the set of rules above r_q in the policy and intersects with it.
3   r_q[i] <== s[i] represents the subset check and is true if r_q[i] ⊂ s[i]
4   else is false , where i is any single rule parameter
5
6   function shadowDetector(r_q, s)
7   for(i=1;i ≤ k; i++)
8       bool shadow = false
9       if((r_q[i] ==> s[i]) ∩ ( ⋂_{j=1:k and j≠i} (r_q[j] <== s[j]))) then
10          shadow = true
11          print("r_q is a shadow")
12          exit
13      end
14  end
15  print("r_q is not a shadow")
16  end
```

Figure 3.3: Shadow detection algorithm.

For any rule, the cell bit is set if the cell index is present in the rule. As an example if rule r_3 from Table 3.3 is the rule in question then Q[0][2][1][62][3] will be set to 1 and Q[0][2][5][62][3] will be set to 0, where UDP is mapped to a integer value of 0. Q is populated with the rule in question r_q and S is populated with the rules in s by sequentially taking one rule at a time and overwriting to the same existing matrix S until all rules from s are considered. Finally, for each cell in Q which is set to 1 if the corresponding cell in R is also set to 1 then rule r_q is shadowed by rules in set s otherwise not. The shadow check operation between r_q and set s can be denoted as $r_q[\text{ALL}] \Longrightarrow s[\text{ALL}]$, where ALL denotes across all parameters. If rule r_q is shadowed by rules in s then $r_q \Longrightarrow s$ is true otherwise false.

This matrix approach is simple but is not practical since a single source IP parameter requires 512 MB of memory to represent entire IP range if each bit represents

a single value. Hence all 5-parameters will make this approach infeasible from the memory requirements standpoint. Hence, Figure 3.3 represents a shadow detection algorithm which has the same effect as matrix approach. However it requires less memory since it does shadow check across a single parameter at a time as compared to the matrix approach which does shadow check across all the parameters at the same time.

Chapter 4: Optimal Rule Ordering

Suboptimal placement of rules can greatly reduce the efficiency of a firewall policy and make the firewall susceptible to Denial of Service attacks. Hence firewall policy reordering is extremely crucial for the performance of firewalls that connect the next generation of high speed networks. A rule order that minimizes the average number of rule comparisons for each incoming packet is defined as an optimal rule order.

Although all possible linear permutations of the policy DAG maintain integrity, this Chapter will show that finding the best permutation is a $\mathcal{NP}$-hard problem.

Certain firewall rules have a higher probability of matching a packet than others. Hence, it is possible to develop a policy profile over time that indicates frequency of rule matches. Let $\text{P} = [p_1, p_2, p_3, ..., p_n]$ be the policy profile, where p_i is the prior probability that a packet will match rule i and n is the number of rules in the policy. Optimal rule ordering can be determined by utilizing the probability associated with each rule as a comparison criteria for reordering the rules.

Given a policy DAG $\text{G} = (\text{R},\text{E})$ and a policy profile $\text{P} = [p_1,p_2,p_3,...,p_n]$, a linear arrangement π of the rules in R is sought that minimizes equation,

$$\text{R}(\pi) = \sum_{i=1}^{n} p_i^{'} \; t_i \ .$$

where t_i is the total time to match a packet against the rule r_i.

In the absence of precedence relationships, the rules are completely independent of each other and the average delay per packet is minimized by sorting the rules in non-increasing order according to the probabilities [20]. Precedence relationships cause the problem to be more realistic; however, it also makes determining the optimal rule

ordering more problematic.

4.1 Optimal Rule Ordering

The problem of determining optimal firewall rule set ordering can be viewed as a single machine job scheduling problem with precedence constraints. The notation for such scheduling problems is a three field classification $\alpha \mid \beta \mid \gamma$, where α represents the machine environment including the number of machines, β represents the job characteristics including presence (or absence) of precedence constraints between jobs, the completion time for jobs, and γ represents the optimality criterion [13]. Ensuing is the description for the problem of single machine job scheduling with precedence conditions, the problem of determining optimal firewall rule set ordering and a reduction of the problem of determining optimal firewall rule set ordering from the problem of single machine job scheduling with precedence conditions.

4.1.1 Single Machine Job Scheduling with Precedence Constraints

Assume J = $[j_1,j_2,j_3,...,j_n]$ denotes a set of n jobs that have to be completed on a single machine M. Assume M can complete at most one job at a time and each job consists of a single operation to be performed on machine M. Also, jobs cannot be preempted. Let w_i be the weight of job j_i ; w_i is indicative of the relative importance of job j_i. Let the required time for each job j_i be 1 unit time and c_i be the completion time for the job j_i, such that

$$\mathrm{c}_i = \sum_{k=1}^{i} 1 = \mathrm{i}\ .$$

Let a precedence relationship exist between certain jobs, since certain jobs must be completed before others. The precedence relation is modeled using a directed acyclic

graph. Let G=(J,E) represents a directed acyclic graph for jobs J = $[j_1,j_2,j_3,...,j_n]$ where J is the set of jobs that need to be scheduled and edges E are the precedence relationships between jobs and if G contains a directed edge from job j_k to job j_l then it is required that job j_k is completed before job j_l.

Let $J' = [j_1',j_2',j_3',...,j_n']$ be a linear arrangement of J. Then, $[j_1',j_2',j_3',...,j_n'] = \pi[j_1,j_2,j_3,...j_n]$ and $[w_1',w_2',w_3',...,w_n'] = \pi[w_1,w_2,w_3,...,w_n]$, where π denotes some permutation. Hence, the optimality criteria is

$$J(\pi) = \sum_{i=1}^{n} w_i' c_i \ .$$

Thus the 3 field classification $\alpha \mid \beta \mid \gamma$ for single machine job scheduling problem has $\alpha = 1$, since the number of machines is 1, $\beta = \{G, 1\}$, since certain jobs have precedence constraints and the required time for each job is 1 unit time and $\gamma = \sum_{i=1}^{n} w_i' c_i$, since it minimizes the total cost of completing n jobs. Thus, the 3 field classification for optimal rule ordering is

$$1 \mid \{G, 1\} \mid \sum_{i=1}^{n} w_i' c_i \ .$$

4.1.2 Finding The Optimal Firewall Rule Set Ordering

Let R = $[r_1,r_2,r_3,...,r_n]$ be an ordered set of n rules in the firewall policy, and an incoming packet is compared against the rule set R on a single processor m. Let m can compare at most one packet at a time, processing packet cannot be preempted and each packet comparison is considered as a single operation to be performed on processor m.

Assume for a policy profile P, p_i is the probability that a packet will match rule r_i

and is indicative of the priority of rule r_i, such that

$$\sum_{i=1}^{n} \mathrm{p}_i = 1 \text{ and } 0 \leq \mathrm{p}_i \leq 1 \ .$$

Therefore if the policy is comprehensive, a packet will match at least one rule in R. Let the time that processor takes to compare an incoming packet against each rule r_i be 1 unit of time and t_i be the total time that processor takes so as to match a packet against the rule r_i, then

$$\mathrm{t}_i = \sum_{k=1}^{i} 1 = i \ .$$

Let G_r=(R,E) represents a directed acyclic graph for the rule list R, and let $\mathrm{R}^{'} = [r_1^{'},r_2^{'},r_3^{'},...,r_n^{'}]$ be a linear arrangement of R. Then, $[r_1^{'},r_2^{'},r_3^{'},...,r_n^{'}] = \pi[r_1,r_2,r_3,...,r_n]$ and $[p_1^{'},p_2^{'},p_3^{'},...,p_n^{'}] = \pi[p_1,p_2,p_3,...,p_n]$. Hence, the optimality criteria is

$$\mathrm{R}(\pi) = \sum_{i=1}^{n} p_i^{'} \ t_i \ .$$

Thus the 3 field classification $\alpha \mid \beta \mid \gamma$ for optimal firewall rule set ordering problem has $\alpha = 1$, since the number of processors is 1, $\beta = \{\mathrm{G}_r,\ 1\}$, since certain rules have precedence constraints and the comparison time for each rule against a incoming packet is 1 unit time. Also, $\gamma = \sum_{i=1}^{n} \mathrm{p}_i^{'}\mathrm{t}_i$, since it minimizes the average number of rule comparisons.

$$1 \mid \{\mathrm{G}_r, 1\} \mid \sum_{i=1}^{n} \ p_i^{'} \ t_i \ .$$

4.1.3 Single Machine Job Scheduling Reduces To Optimal Firewall Rule Ordering

The problem of single machine job scheduling subject to precedence constraints with all jobs having a constant required time is $\mathcal{NP}$-hard [15, 17]. If number of jobs to be scheduled is equal to the number of rules in the firewall policy whose linear arrangement is sought, then

$$\mid \mathrm{J} \mid = \mid \mathrm{R} \mid \text{ and } \mathrm{j}_i = r_i \ . \tag{4.1}$$

All jobs have to be completed on a single machine M and all firewall policy rules have to be compared against a packet on a single processor m, therefore

$$\mid \mathrm{M} \mid = \mid m \mid \ . \tag{4.2}$$

Jobs are modeled using directed acyclic graph and a directed edge exists between job j_k and job j_l, if job j_k has to be scheduled before job j_l. Similarly, rules in a firewall policy are modeled using directed acyclic graph and a directed edge exists between rule r_k and rule r_l, if rule r_k and rule r_l intersect, and rule r_k has to be compared prior to rule r_l to each incoming packet, therefore for $\mid \mathrm{J} \mid = \mid \mathrm{R} \mid$

$$\mathrm{G} \equiv \mathrm{G}_r \ . \tag{4.3}$$

The required time each job is 1 unit time and the time to compare a packet against each rule in the firewall policy is also 1 unit time, therefore the completion time c_i for job j_i is equal to the total time t_i that processor takes so as to match a packet against rule r_i, hence

$$\mathrm{c}_i = \mathrm{t}_i = \sum_{k=1}^{i} 1 = \mathrm{i} \ . \tag{4.4}$$

w_i is the weight of job j_i and it indicates the priority of job j_i, p_i is the probability that packet will match rule r_i and it indicates the priority of rule r_i, and

$$\sum_{i=1}^{n} \mathrm{p}_i = 1 \text{ and } 0 \leq \mathrm{p}_i \leq 1 \ . \tag{4.5}$$

Given a set of firewall rules R and a policy profile P, let W be the set of weights for jobs J, then every probability from set P can be mapped to a weight from set W using a invertible mapping function,thus

$$\forall \ \mathrm{p} \in \mathrm{P} \text{ and } \mathrm{w} \in \mathrm{W} \ \exists \ \mathrm{f}(\mathrm{p}_f) = \mathrm{w}_g \text{ and } \mathrm{f}^{-1}(\mathrm{w}_g) = \mathrm{p}_f \ . \tag{4.6}$$

Thus from (12) and (13) the optimality criteria for the problem of single machine job scheduling with precedence constraints and the optimality criteria for the problem of finding optimal firewall rule set ordering are equivalent,hence

$$\mathrm{J}(\pi) \equiv \mathrm{R}(\pi) \ . \tag{4.7}$$

Thus from 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 the 3 field classification $\alpha \mid \beta \mid \gamma$ for the problem of finding optimal firewall rule set ordering and the problem of single machine job scheduling with precedence constraints are exactly analogous,

$$1 \mid \{\mathrm{G}, 1\} \mid \sum_{i=1}^{n} \mathrm{w}_i^{\prime} \mathrm{c}_i \ \equiv \ 1 \mid \{\mathrm{G}_r, 1\} \mid \sum_{i=1}^{n} \mathrm{p}_i^{\prime} \mathrm{t}_i \ .$$

Thus the problem of finding optimal firewall rules set ordering is a reduction from the problem of single machine job scheduling with precedence constraints. Since single machine job scheduling problem with precedence conditions has been proved to be $\mathcal{NP}$-hard thus the problem of optimal firewall rule set ordering is also $\mathcal{NP}$-hard. Therefore the optimal firewall rule ordering problem has a solution if and only if the single machine job scheduling problem has a solution.

Chapter 5: Firewall Policy Sorting Algorithms

This chapter introduces three different algorithms to minimize the average number of rule comparisons.

5.1 The α Algorithm

```
1   Let S be the sorted policy represented by a FIFO Queue, and initially
2   S = φ.
3   Let Q be the List of rules to sort, and initially Q = R.
4   Let L be the List of rules that have no incoming edge to them.
5   Let G be the DAG corresponding to the input firewall policy.
6
7   function policySort(Q, S, G)
8   initialize L with rules from Q that have no incoming edge to them.
9   do
10      // find the best rule in L to place in S
11      set r_b to a rule in L
12      for(∀ r_j ∈ L and r_j ≠ r_b)
13         if( (P(ST(r_b)) / |ST(r_b)|) < (P(ST(r_j)) / |ST(r_j)|) ) then
14            r_b = r_j
15         end
16      end
17      add r_b to S and remove r_b from Q
18      remove r_b from L
19      update L with new candidates from Q, if any
20  while(Q ≠ φ)
21  end
```

Figure 5.1: Algorithm α.

The α algorithm is a heuristic firewall policy sorting technique. It has the ability to sort the rules in R that have precedence relationships between them, while maintaining the integrity of the input firewall policy. Thus α is an improvement over the sorting algorithm presented in [10, 11] which can only sort independent rules contained in R.

Let the Spanning Tree (ST) of rule r_i (ST(r_i)) be the set of rules in the Depth

First Traversal (DFT) of G with r_i as root node. For example in Figure 5.2(a) ST(r_3) = {r_3, r_4, r_5}. Let |ST(r_i)| denote the cardinality of set ST(r_i). Now, let P(ST(r_i)) be the sum of probability of all the rules in set ST(r_i). Therefore,

$$P(ST(r_i)) = \sum_{\forall\, k\, \in\, ST(r_i)} p(k)\ .$$

where p(k) is the probability of rule k.

For example P(ST(r_2)) = p(r_2) + p(r_5) = 0.152667 + 0.000001 = 0.152668. Hence, P(ST(r_i))/|ST(r_i)| represents the average probability of the spanning tree obtained after DFT of G with r_i as the root node.

		Source		Destination			
No.	Proto.	IP	Port	IP	Port	Action	Prob.
1	UDP	190.1.1.*	*	*	*	deny	0.252512
2	UDP	190.1.2.*	*	*	90	accept	0.152667
3	TCP	190.1.*	*	180.1.*	65-90	accept	0.594802
4	TCP	*	*	180.1.45.27/14	80	deny	0.000018
5	IP	*	*	*	*	deny	0.000001

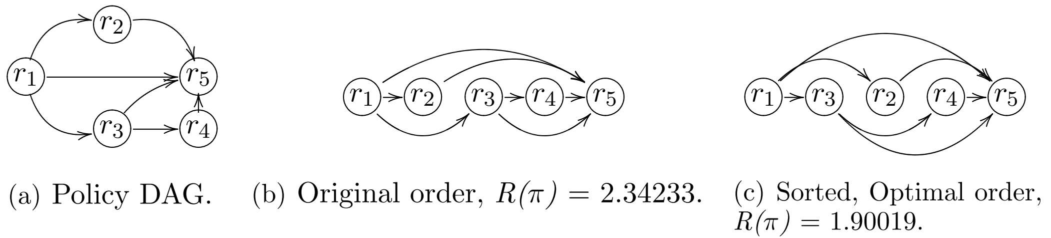
Table 5.1: Example security policy consisting of multiple ordered rules.

The α algorithm takes DAG G as input and initially generates a list L of independent rules. A rule that does not have any incoming edge incident to it is regarded as independent. Also, let S be the queue of sorted rules and let Q be the list of rules to be sorted.

During each pass from all the rules in L, the rule r_b with the highest average spanning tree probability is selected. Rule r_b is then inserted in the sorted queue S and is removed from Q. Subsequently r_b is removed from L and new candidates generated, if any after removal of r_b are added to L. This process is repeated until there is no rule left in Q to be sorted.

Thus α sort works by inserting the rule with the highest average spanning tree probability out of all possible rules that can be inserted at that particular point in

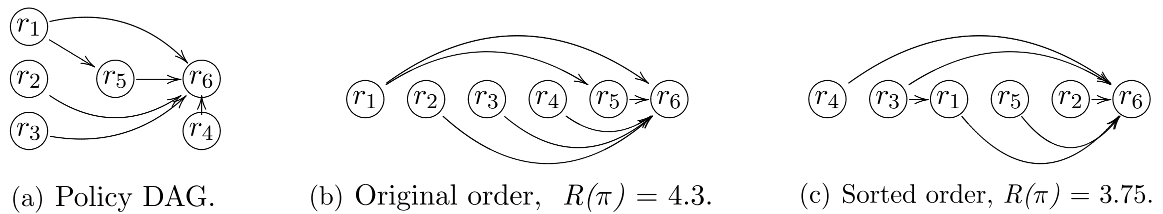
time. It reasons that if any rule is selected whose spanning tree does not have highest average probability, then it will push down each rule in the spanning tree of rule which has highest average probability by at least one place. α algorithm maintains the integrity of the policy since during each pass it selects a rule that has no edge incident to it and hence precedence relationships are always maintained.



(a) Policy DAG. (b) Original order, $R(\pi) = 2.34233$. (c) Sorted, Optimal order, $R(\pi) = 1.90019$.

Figure 5.2: Policy DAG representations of the rules given in Table 5.1.

For an example of how α sort works, consider the policy given in Table 5.1 and the associated DAG in Figure 5.2(a). In the initial pass L $= \{r_1\}$ and since L contains a single rule r_1, it is inserted in S and removed from Q and L. After the removal of r_1 L is updated and two new candidate rules r_2, r_3 are inserted in L, hence L $= \{r_2, r_3\}$. In the second pass P(ST(r_3))/ |ST(r_3)|=0.29741 and is the highest average spanning tree probability among all rules present in L, thus is inserted into S and is removed from Q.L is updated with removal of r_3 and addition of r_4. In subsequent pass P(ST(r_2))/ |ST(r_2)|=0.076334 is maximum, hence it is inserted in S and removed from Q. In the current pass after removal of r_2, new candidates are not generated and hence L just contains r_4. In next pass r_4 is the only competing rule,hence is selected and inserted in S. After removal of r_4, r_5 is added to L. Finally r_5 is the only rule left in Q and thus sorting culminates with insertion of r_5 in S.



(a) Policy DAG. (b) Original order, $R(\pi) = 4.3$. (c) Sorted order, $R(\pi) = 3.75$.

Figure 5.3: Policy DAG representations of the rules given in Table 5.2.

5.1.1 Example That Results In A Non-Optimal Ordering

Consider the policy given in Table 5.2. Applying the α algorithm, [r_4,r_3,r_2,r_1,r_5,r_6] rule ordering is obtained where as the optimal ordering is [r_4,r_3,r_1,r_5,r_2,r_6]. Initially L=$\{r_1,r_2,r_3,r_4\}$ and since r_4 has the highest average spanning tree probability of 0.25, it gets selected. In the second pass L=$\{r_1,r_2,r_3\}$ and P(ST(r_3))/ |ST(r_3)|=0.225 is maximum. Hence, r_3 is inserted into S. In the subsequent pass L=$\{r_1,r_2\}$ and r_2 outperforms r_1 and gets selected.

The α algorithm makes a mistake in the third pass when r_2 is selected ahead of r_1 since,

$$\mathrm{P}(\mathrm{ST}(r_2))/\ |\mathrm{ST}(r_2)| > \mathrm{P}(\mathrm{ST}(r_1))/\ |\mathrm{ST}(r_1)|$$

Rule r_1's spanning tree has rules r_1, r_5 and r_6 in it. Now, since there is a precedence edge between r_1 and r_5, r_1 will have to precede r_5 in any ordering that maintains the integrity of policy. p(r_5) = 0.20 and p(r_2) = 0.10, however p(r_1) = 0.05 which lowers the average probability of rule r_1's spanning tree. Therefore rule r_2 is selected and rule r_5 is pushed down in the ordering at least by one place despite of it having high individual probability. α algorithm selects r_2, followed by r_1, followed by r_5 whereas optimal ordering selects r_1, followed by r_5, followed by r_2 in the corresponding passes and since,

$$r_1 * 3 + r_5 * 4 + r_2 * 5 < r_2 * 3 + r_1 * 4 + r_1 * 5$$

Therefore, the average delay for ordering obtained by following α algorithm is higher than the average delay for optimal ordering.

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	190.1.1.*	*	*	80	deny	0.05
2	UDP	210.1.*	*	*	90	accept	0.10
3	TCP	180.*	*	180.*	90	accept	0.15
4	TCP	210.*	*	220.*	80	accept	0.20
5	UDP	190.*	*	*	*	accept	0.20
6	*	*	*	*	*	deny	0.30

Table 5.2: Example security policy for which α ordering is not optimal.

The α algorithm fails since during any pass if there are multiple rules in L to select from and if one or more rules from L denoted by L', which does not have the highest average spanning tree probability happen to have very small difference between their average spanning tree probability and the average spanning tree probability of the rule with the highest average spanning tree probability. Then, if one or more rules in L' has higher standard deviation among the probabilities of the rules in the spanning tree as compared to the rule with highest average spanning tree probability then it is suggestive of the fact that these rules in L' have certain rules in their spanning tree with very high probability and certain rules with very low probability. Hence, when these rules with high probability are pushed back in the policy it could result in a ordering which is not same as the optimal ordering. The α algorithm runs in polynomial time but the rule ordering problem is shown to be $\mathcal{NP}$-hard.

5.2 The β Algorithm

The β algorithm is a heuristic based recursive sorting method, developed by B. C. Carruth while he attended Wake Forest University. During each pass it selects the rule with the highest average subgraph probability from the graph of rules available

```
1   Let S be the sorted policy represented by a FIFO Queue, and initially S = φ
2   Let Q be the List of rules to sort, and initially Q = R.
3   Let G be the DAG corresponding to the input firewall policy.
4   Let T be the transitive closure of G.
5
6   function policySort(Q, S)
7   do
8       // find the best rule in Q to place in S
9       set r_b to a rule in Q
10      for(∀ r_j ∈ Q and r_j ≠ r_b)
11         if( (P(G(r_b)) / |G(r_b)|) < (P(G(r_j)) / |G(r_j)|) ) then
12            r_b = r_j
13         end
14      end
15      //If number of rules dependent on r_b is 0
16      if(|G(r_b)| == 1) then
17         add r_b to S and remove r_b from Q
18         update T
19      //If number of rules dependent on r_b is greater than 0
20      else
21         policySort(G*(r_b), S)
22         add r_b to S
23      end
24  while(Q ≠ φ)
25  end
```

Figure 5.4: Algorithm β.

during that pass. The average subgraph probability for any rule is the average of prior probability of rules present in the subgraph of that rule. The *selected rule* is then inserted in the list of sorted rules if it has no edges incident on it, otherwise the subgraph, excluding itself, is sorted recursively and inserted in the list of sorted rules followed by the insertion of the *selected rule*. The β algorithm uses the same heuristic as α algorithm and selects the rule with the highest average subgraph probability from the set of candidate rules. Another important observation is that, if we reverse the direction of edges in the graph then the spanning tree of each rule is same as the subgraph of that rule before reversing. However, the β algorithm is different from α algorithm since during any pass in α algorithm, selection candidates are only the rules

which have no edges incident on them whereas in β algorithm, selection candidates are all the rules in the graph available during that pass. Hence β algorithm has more candidates to choose from during each pass as compared to α algorithm. Therefore, β algorithm has more available options and hence, finds an optimal ordering in more cases than the α algorithm.

Let $\mathrm{G}(\mathrm{r}_i)$ be the set of rules in the subgraph of rule r_i, including r_i. For example in Figure 5.8(a) $\mathrm{G}(\mathrm{r}_6)=\{\mathrm{r}_1,\mathrm{r}_2,\mathrm{r}_3,\mathrm{r}_4,\mathrm{r}_5\}$. Let $\mathrm{G}^*(\mathrm{r}_i)$ be the set of rules in the subgraph of rule r_i excluding itself. Thus in Figure 5.8(a) $\mathrm{G}^*(\mathrm{r}_5)=\{\mathrm{r}_1\}$. Suppose $|\mathrm{G}(\mathrm{r}_i)|$ be the cardinality of set $\mathrm{G}(\mathrm{r}_i)$. Now, let $\mathrm{P}(\mathrm{G}(\mathrm{r}_i))$ be the sum of probability of all the rules in set $\mathrm{G}(\mathrm{r}_i)$. Therefore,

$$\mathrm{P}(\mathrm{G}(\mathrm{r}_i)) = \sum_{\forall\, k\, \in\, \mathrm{G}(r_i)} \mathrm{p}(k)\ .$$

where $\mathrm{p}(k)$ is the probability of rule k.

For example $\mathrm{P}(\mathrm{G}(\mathrm{r}_5)) = \mathrm{p}(\mathrm{r}_1) + \mathrm{p}(\mathrm{r}_5) = 0.05 + 0.20 = 0.25$. Hence, $\mathrm{P}(\mathrm{G}(\mathrm{r}_i))/|\mathrm{G}(\mathrm{r}_i)|$ is the average probability of the subgraph $\mathrm{G}(\mathrm{r}_i)$. The algorithm takes the DAG as input and for each rule r_i in the graph, average probability $\mathrm{P}(\mathrm{G}(\mathrm{r}_i))/|\mathrm{G}(\mathrm{r}_i)|$ of rule r_i's subgraph is calculated. The rule with the highest average subgraph probability is selected. If the selected rule has no incident edges then it is inserted in the sorted queue S and is removed from Q, else $\mathrm{G}^*(\mathrm{r}_b)$ is sorted recursively using the same heuristic.

This process is repeated until there is no rule left in Q to be sorted. Thus β sorts and inserts in sorted policy list, the rule subgraph that has higher average probability as compared to all other rule subgraphs present in the unsorted policy list. It takes into account that if any rule subgraph is selected that does not have highest average probability, then it will push each rule in the rule subgraph that has highest average probability further down in the sorted policy list at least by the number of rules in the selected rule subgraph.

For an example of how β algorithm works, consider the policy given in Table 5.2

and the associated DAG in Figure 5.8(a). In the initial pass $P(G(r_4))/\ |G(r_4)| = 0.20$ and is the highest average subgraph probability, thus is inserted into S and is removed from Q. In subsequent pass $P(G(r_6))/\ |G(r_6)|$=0.1666 is maximum. Since it has rules dependent on it, $G^*(r_6)$ is sorted recursively and inserted before r_6 is inserted in S and removed from Q. In the next pass r_3 scores over $\{r_1,r_2,r_5\}$ with $P(G(r_3))/|G(r_3)|$ = 0.15 and is pushed into S. Now, S holds $[r_4,r_3]$ and Q has $\{r_1,r_2,r_4,r_5,r_6\}$. In the next pass r_5 is selected but since it has r_1 as dependent on it, thus $G^*(r_5)$ having r_1 is recursively sorted and r_1 is inserted in S. After that r_5 is added to S when recursive call is returned. Now, r_2 is the only rule that can be sorted before recursive call can be returned to r_6. Thus, r_2 is pushed in S. Finally r_6 is the only rule left and is inserted into S after top level recursive call is returned.

		Source		Destination			
No.	Proto.	IP	Port	IP	Port	Action	Prob.
1	UDP	190.1.*	*	*	90	accept	0.0645
2	UDP	190.1.1.*	*	*	90-94	deny	0.161
3	UDP	190.1.1.2	*	*	94	deny	0.2258
4	UDP	190.1.2.*	*	*	*	accept	0.2587
5	*	*	*	*	*	deny	0.29

Table 5.3: Example security policy for which the β ordering is not the optimal ordering.

5.2.1 Example That Results In A Non-Optimal Ordering

Given the difficulty of the problem, the β algorithm will not always find the optimal ordering. For the DAG in figure 5.5 β ordering is $[r_1,r_2,r_4,r_3,r_5]$ where as the optimal ordering is $[r_1,r_3,r_2,r_4,r_5]$. In the first pass average probability of the subgraph of rule r_5 is 0.2 and is maximum. Since it has rules dependent on it, $G^*(r_5)$ has to be sorted and pushed in S before r_5 can be pushed. $G^*(r_5) = \{r_1,r_2,r_3,r_4\}$, in the next pass subgraph of rule r_4, $G(r_4)=\{r_1,r_2,r_4\}$ has the highest average probability

and is equal to 0.1614, also subgraph of rule r_3, $G(r_3)=\{r_1,r_3\}$ has the second highest average probability and is equal to 0.14515. Subgraphs of rule r_4 and r_3 have r_1 as the common rule in between them. Thus following the algorithm $[r_1,r_2,r_4,r_3,r_5]$ ordering is obtained where as the optimal ordering is $[r_1,r_3,r_2,r_4,r_5]$.

The β algorithm makes a mistake in finding the optimal ordering because when $G^*(r_5)$ having r_1,r_2,r_3 and r_4 in it, contend in the second pass

$$\frac{P(G(r_4))}{\mid G(r_4) \mid} > \frac{P(G(r_3))}{\mid G(r_3) \mid} > \frac{P(G(r_2))}{\mid G(r_2) \mid} > \frac{P(G(r_1))}{\mid G(r_1) \mid}$$

and thus r_4 is selected, however since r_4 has dependent rules, $G^*(r_4) \neq \phi$. Therefore, $G^*(r_4)$ is sorted recursively and r_1 is pushed into the sorted policy S. After r_1 is removed from Q, algorithm continues to sort the remaining rules in $G^*(r_4)$. However, average probability of subgraph of rule r_4 and rule r_3 changes after removal of rule r_1, since the graphs have this rule in common.

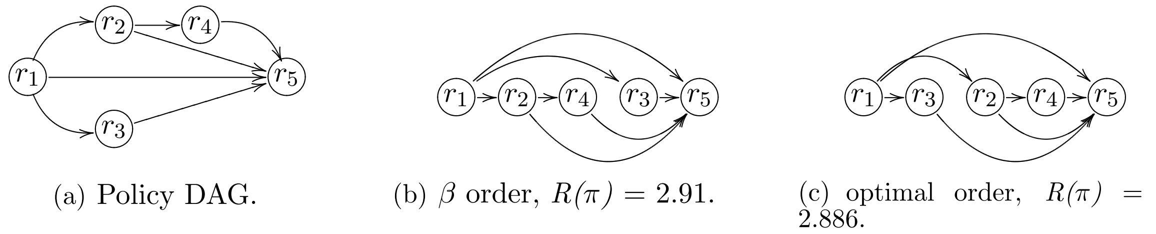
$$\frac{p(r_2) + p(r_4)}{2} < \frac{p(r_3)}{1} \text{ and hence,}$$

$$\frac{P(G(r_4))}{\mid G(r_4) \mid} < \frac{P(G(r_3))}{\mid G(r_3) \mid}$$

Where $G(r_4) = \{r_2,r_4\}$ and $G(r_3) = \{r_3\}$. Thus optimal ordering has r_3 before r_2 ,r_4. However, since β algorithm continues to sort entire subgraph of r_4 before considering any other rule not present in its subgraph. Hence r_2,r_4 are pushed before r_3.

In general, let r_i be a common rule in the subgraph of two or more rules, such that $r_i \in G(r_m)$, $r_i \in G(r_n)$, ..., $r_i \in G(r_p)$, and

$$\frac{P(G(r_m))}{\mid G(r_m) \mid} > \frac{P(G(r_p))}{\mid G(r_p) \mid} > ... > \frac{P(G(r_n))}{\mid G(r_n) \mid}$$



(a) Policy DAG.

(b) β order, $R(\pi) = 2.91$.

(c) optimal order, $R(\pi) = 2.886$.

Figure 5.5: Different policy DAG representations of the firewall rules given in table 5.3.

Thus, r_m is selected ahead of set of rules $\text{J} = \{r_p, ..., r_n\}$ which had smaller average subgraph probability as compared to r_m. After removal of r_i if any of rule $r_j \in \text{J}$ has higher average subgraph probability than r_m, such that

$$\frac{\text{P}(\text{G}(r_j) - r_i)}{\mid \text{G}(r_j) - r_i \mid} > \frac{\text{P}(\text{G}(r_m) - r_i)}{\mid \text{G}(r_m) - r_i \mid}, \text{where } r_j \in \text{J}$$

Then algorithm makes a mistake in finding the optimal ordering; since the average probability of subgraph changes.

5.3 The γ Algorithm

The γ algorithm is a heuristic approach that is similar to the β algorithm. However, it is different since in every pass the β algorithm finds the rule with the highest average subgraph probability and inserts that rule in S if it has no dependents on it. Otherwise, it recursively sorts the subgraph of its dependents and eventually pushes the entire subgraph followed by itself in S.

The γ algorithm also finds the rule with the highest average subgraph probability in each pass and inserts that rule directly into S if it has no dependents on it, otherwise it recursively sorts the subgraph of its dependents until it finds a rule that has no dependents and inserts that rule in S. Hence in each pass β algorithm inserts subgraph of a rule which might or might not have dependents on it, where as the γ algorithm in

each pass inserts the subgraph of a rule that has no dependents on it. Since during any pass γ algorithm has all the rules in the graph as candidates whereas β algorithm only has a subgraph of the original graph as candidates except for the initial non-recursive call. Therefore, γ algorithm makes more informed decision and finds optimal ordering in more cases than β algorithm.

5.3.1 Example That Results In An Optimal Ordering

For an example of how γ sort works, consider the policy given in Table 5.3. In the first pass rule r_5 has the highest average subgraph probability of 0.2000. Since it has rules dependent on it,$G^*(r_5) = \{r_1, r_2, r_3, r_4\}$ has to be sorted recursively until a rule is found that has no dependent. In the recursive call r_4 has the highest average but it too has r_1 and r_2 as dependents. Thus in the next recursive call r_1, r_2 contend and r_2 gets selected, however it has r_1 as dependent thus r_1 alone contends in the next recursive call. Finally r_1 gets selected and has no dependents, hence is inserted into S. In the second pass r_2, r_3, r_4 and r_5 contend and this process is repeated until Q is empty. Thus at the end of each pass one rule is removed from Q and inserted into S. Following the algorithm [r_1,r_3,r_2,r_4,r_5] ordering of rules is obtained and is same as the optimal ordering.

```
1  Let S be the sorted policy represented by a FIFO Queue, and initially S = φ.
2  Let X be the List of rules from which the best node to add to S is selected
3  during the top level(non-recursive) function call, and initially X = R.
4  Let Q be the subgraph of rules from which the best node to add to S is selected
5  recursively until a node with no edges incident on it is found, and initially Q=R.
6  Let recursive be a boolean variable indicating whether the function call is
7  recursive or top level (non-recursive), initially recursive = false.
```

Figure 5.6: Algorithm γ data structures.

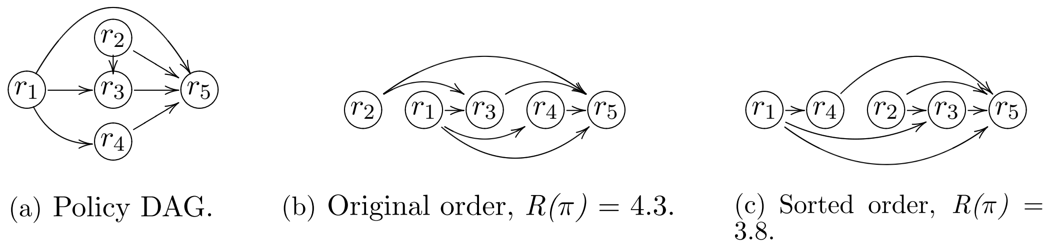
```
1   function policySort(Q, S, X, recursive)
2   do
3       //If non-recursive(top level) call
4       if(recursive == false) then
5           set r_b to a rule in X
6           for(∀ r_j ∈ X and r_j ≠ r_b)
7             if( (P(G(r_b)) / |G(r_b)|) < (P(G(r_j)) / |G(r_j)|) ) then
8               r_b = r_j
9             end
10          end
11      end
12      //If recursive call
13      else
14          set r_b to a rule in Q
15          for(∀ r_j ∈ Q and r_j ≠ r_b)
16            if( (P(G(r_b)) / |G(r_b)|) < (P(G(r_j)) / |G(r_j)|) ) then
17              r_b = r_j
18            end
19          end
20      end
21
22      //If number of rules dependent on r_b is 0
23      if(|G(r_b)| == 1) then
24          add r_b to S and remove r_b from X
25          if(recursive == true) then
26              return
27          end
28      end
29      //If number of rules dependent on r_b is greater than 0
30      else
31          //If recursive call
32          if(recursive == true) then
33              policySort(G*(r_b), S, X, true)
34              return
35          end
36          //If non-recursive call
37          else
38              policySort(G*(r_b), S, X, true)
39          end
40      end
41  while(X ≠ φ)
42  end
```

Figure 5.7: Algorithm γ.

5.3.2 Examples That Results In Non-Optimal Ordering

This section describes two cases where the γ algorithm is unable to find the optimal ordering.

A Simple Case



(a) Policy DAG. (b) Original order, $R(\pi) = 4.3$. (c) Sorted order, $R(\pi) = 3.8$.

Figure 5.8: Different policy DAG representations of the firewall rules given in 5.4.

For the DAG in Figure 5.8, γ ordering is [r_2, r_1, r_3, r_4, r_5] where as the optimal ordering is [r_1, r_4, r_2, r_3, r_5]. In the first pass rule r_5 has the highest average subgraph probability of 0.2 . Since it has dependent rules, its subgraph excluding itself is sorted recursively. Hence in the recursive call rules r_4, r_3, r_2, r_1 contend. r_3 is selected as its average subgraph probability is highest. Now, since r_3 has r_2 and r_1 as its dependents, recursive sorting is continued. In the next recursive call, r_2 gets selected over r_1 and since it has no dependents, is inserted into the queue holding sorted rules. Thus γ algorithm selects r_2 in the first pass where as optimal ordering has r_1 in the first place.

The γ algorithm makes a mistake during the recursive call in which rules r_2 and r_1 contend. Rule r_2 has higher average subgraph probability than r_1 and hence gets selected. This pushes rule r_4 down by at least one place because r_4 has r_1 as a dependent on it which did not get removed in the current pass. However, if r_1 is selected ahead of r_2 when r_1 and r_2 contend, then it allows r_4 to get selected in the

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	IP	210.4.5.*	44-50	*	*	accept	0.1998
2	UDP	190.1.1.*	*	*	80	deny	0.1297
3	UDP	190.1.1.55-190.1.3.*	*	*	40-94	accept	0.2200
4	UDP	190.1.1.24-190.1.1.48	*	*	75-92	accept	0.2597
5	UDP	190.1.1.34	*	*	91	deny	0.1898
6	IP	*	*	*	*	deny	0.0010

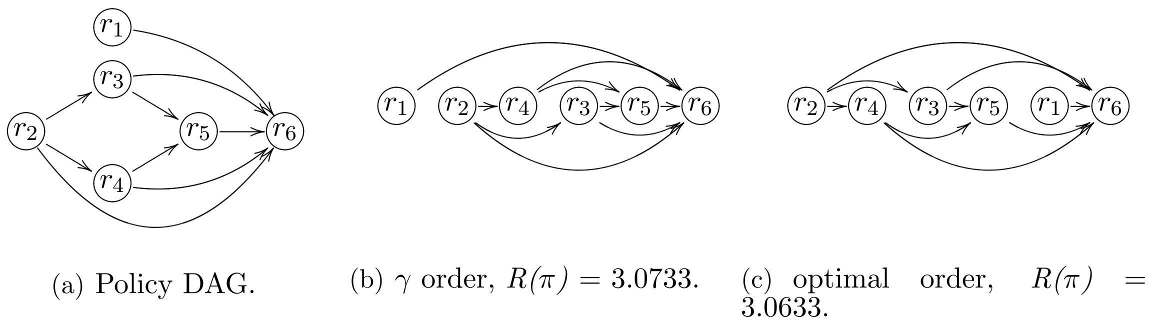
Table 5.4: Example security policy for which the γ ordering is not the optimal ordering.

next pass. As r_4 has high individual probability, its selection in the second pass as compared to γ algorithm's fourth pass, results in a smaller average delay.

A DAG With A Diamond Structure

For the DAG in Figure 5.9, in the first pass rule r_1 and r_5 have the same and highest average subgraph probability of 0.1998. If the tie is broken in favor of rule r_1 then the γ ordering is [r_1, r_2, r_4, r_3, r_5, r_6] where as the optimal ordering is [r_2, r_4, r_3, r_1, r_5, r_6].

However, if the tie is broken in favor of rule r_5 then γ ordering is [r_2, r_4, r_3, r_1, r_5, r_6] and is same as the optimal ordering.



(a) Policy DAG. (b) γ order, $R(\pi) = 3.0733$. (c) optimal order, $R(\pi) = 3.0633$.

Figure 5.9: Different policy DAG representations of the firewall rules given in table 5.5.

If we have a diamond structure of rules in the policy and the ultimate rule (For

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	190.1.*	*	*	90	accept	0.0585333
2	UDP	190.1.1.*	*	*	88	deny	0.0728863
3	UDP	190.1.1.2	*	*	88-94	deny	0.156762
4	UDP	190.1.2.*	*	*	*	accept	0.123346
5	*	*	*	*	*	deny	0.5884724

Table 5.5: Example security policy for which γ ordering is not optimal ordering.

example rule r_5 in Figure 5.9(a)) shares the highest average subgraph probability with one or more rules in any pass during the sorting then,

Case I : If p$(r_{ultimate}) < \frac{P(G(r_{ultimate}))}{|G(r_{ultimate})|}$

Then there will be only one rule, in favor of whom if the tie will be broken, then optimal ordering will be obtained else if the tie is broken in favor of any other rule then the optimal ordering will be different from γ ordering and algorithm will make a mistake.

Case II : If p$(r_{ultimate}) \geq \frac{P(G(r_{ultimate}))}{|G(r_{ultimate})|}$

Then it does not matter which rule the tie is broken in favor of, there will be multiple optimal orderings, with each rule having the same average probability in the contention place.

For the DAG in Figure 5.9 if the probabilities of rules r_2, r_3, r_4 and r_6 are kept constant and r_1 is increased to anywhere between [0.1998 - 0.2031] and r_5 is decreased to any where between [0.1965 - 0.1998] while keeping the $\sum_{k=1}^{6} p(r_k) = 1$ then for the entire range of r_1 as well as the entire range of r_5 γ algorithm fails to find the optimal rule set order. These ranges represents the cases where r_1 has highest average subgraph probability and thus γ ordering has it inserted ahead of r_5 where as optimal ordering has r_5 inserted ahead of it.

5.4 Integrity Validation Post Optimization

As mentioned in the section 3.2, firewall rules can have precedence relationships between them. Hence, in any reordering of the rules these relationships should be maintained so as to preserve the intent of the policy. This section presents a validation algorithm which checks whether the rule ordering obtained post optimization maintains the integrity of the policy or not. Validator is independent of the optimization algorithm used and utilizes the DAG of the original policy before optimization and the rule ordering obtained after optimization to classify whether the intent of the policy is maintained or not.

```
1   Let S is an array that holds the rule ordering post optimization.
2   Let TEMP is an associative container.
3   Let A is an upper triangular matrice corresponding DAG G.
4
5   function Validator(A, S)
6   bool violation = false
7   //Iterate through S
8   for(i = 1; i ≤ n; i++)
9       // Add S_i to TEMP
10      TEMP.insert( S_i )
11      //Iterate through the S_i^th column of matrice A
12      for(j = 1; j ≤ n; j++)
13          if ( A_{j,S_i} == 1) then
14              //isAbsent() returns true if j is not present in TEMP else returns false
15              violation = isAbsent(TEMP , j)
16              if(violation == true) then
17                  print("Integrity Violated")
18                  exit
19              end
20          end
21      end
22  end
23  print("Integrity Maintained")
24  end
```

Figure 5.10: Algorithm for integrity validation.

A policy DAG G = (R, E) can be represented as an upper triangular matrix $A_{n,n}$

where n is the number of rules in the policy. $\forall$ $A_{i,j} = 1$ where $i < j$, $\exists$ a directed edge from rule r_i to rule r_j in G and $\forall$ $A_{i,j} = 0$ where $i < j$, $\exists$ no directed edge from rule r_i to rule r_j in G. $A_{i,j} = 0$,where $i > j$, since a DAG does not allow for an edge from a higher node number to a smaller node number. $A_{i,j} = 0$,where $i = j$, since a DAG does not allow for an edge from a node to itself. Also, $1 \leq i \leq n$ and $1 \leq j \leq n$.

Validator works by iterating over the rule ordering obtained after applying optimization algorithm to the input policy. For each rule r_i it checks whether the rules directly dependent on it have already appeared in the optimal ordering or not. Rules directly dependent on r_i will be a set of rules D_i representing the rows in A for which $A_{j,i} = 1$, where $1 \leq j \leq n$. For each rule r_i, if any of the rule in D_i has not appeared in the optimal ordering before rule r_i then the integrity is not preserved.

Chapter 6: Results and Analysis

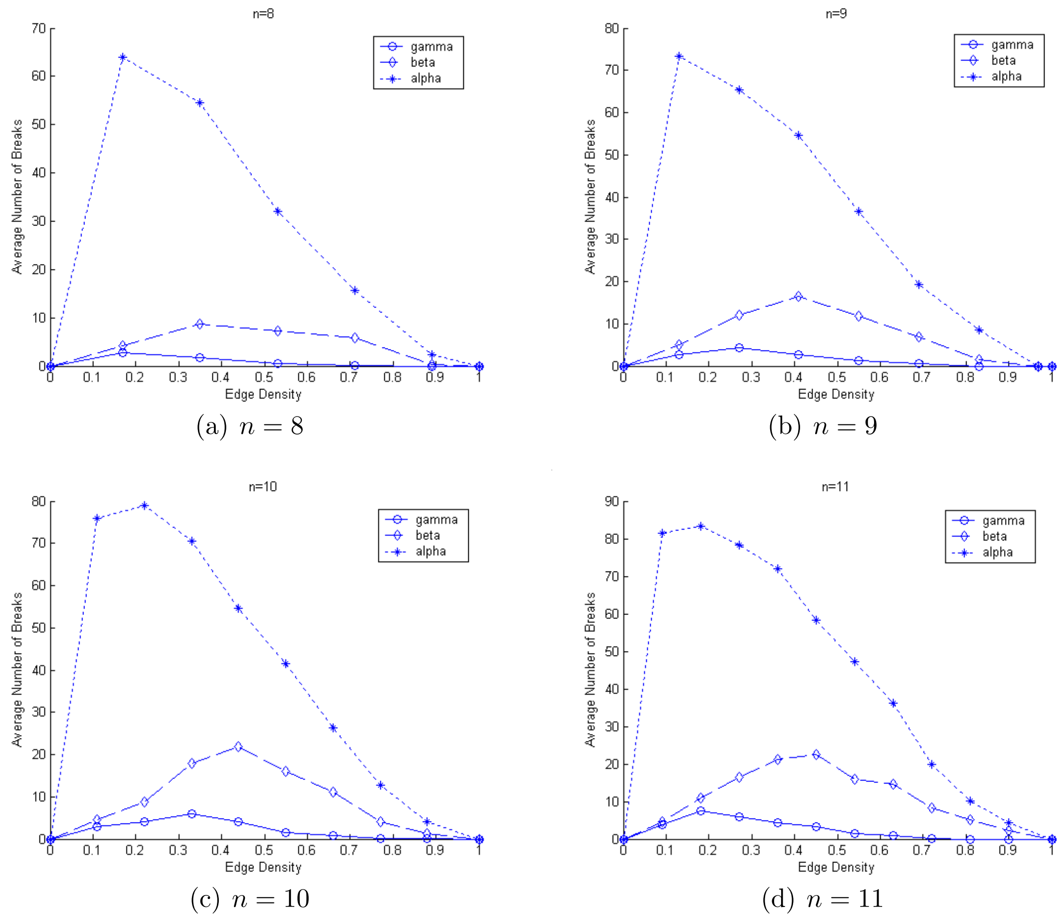
In this chapter the performance of the algorithms is measured empirically using simulation. We are interested in measuring the number of times each algorithm fails to find the optimal ordering on small policies, as well as the average number of rule comparisons for the sorted policy. In addition this chapter also presents the setup for different experiments and includes the graphs corresponding these experiments.

A firewall policy can be characterized by three parameters, the policy size n, the number of edges e, and the policy profile P. Policy size refers to the number of rules in the policy. Also, as mentioned in Chapter 4, a policy profile P is the set of hit probabilities for each rule in the policy. For simulations, these three parameters will vary to represent different policies and the DAG model will be used to represent the firewall policies.

6.1 Edge Density Versus Number of Breaks

This experiment compares average number of cases for which α, β and γ algorithms fail to find optimal order, as the number of precedence edges increases. This will be referred to as break. Given the difficulty of the problem, determining the optimal ordering for large policies is infeasible; hence, experiments were performed with only small policy sizes.

In these experiments the number of edges e were varied from 0 to e_{max} with increments of 5. The maximum number of edges, e_{max} for n rules is $(n^2 - n)/2$. For each number of edges 100 runs were performed, where the edge placement and hit probabilities were randomly selected using a uniform distribution. The average number of breaks was recorded for the given edge density, which is the percentage of



(a) $n = 8$ (b) $n = 9$

(c) $n = 10$ (d) $n = 11$

Figure 6.1: Average number of breaks versus Edge density for different policy sizes.

total number of edges possible. This was repeated for policies with 8, 9, 10, and 11 rules.

The results presented in the graphs in figure 6.1 indicate that both the β and γ algorithm performs better than α algorithm. This performance increase is because during any pass of the α algorithm the candidate rules are only the rules that have no edges incident on them whereas, β algorithm has all the rules in the DAG as candidate rules during the non-recursive calls. Also, the γ algorithm performs better than the β algorithm since during any pass it has all the rules in the DAG to select from whereas β algorithm has all the rules in the DAG as candidate rules only during the non-recursive

calls. Hence, the γ has more possible orderings to select from.

The graphs in figure 6.1 show that the maximum number of breaks for the β algorithm occur between the edge density range of 0.3 to 0.5. The rationale behind this behavior is that, with low edge density the rules are mostly independent and can be sorted in decreasing order of probabilities where as with high edge density the rules have high dependencies between them and hence only few orderings are possible. Therefore, with very low and very high edge density, the number of breaks are fewer. It can also be inferred that edge density range of 0.3 to 0.5 indicates the prevalence of cases for which β algorithm fails to find the optimal ordering.

6.2 Break Cases Overlap

In this section the overlap of the break cases are investigated to help determine when one algorithm has difficulty finding optimal ordering as compared to the others.

The number of edges e were varied from 0 to e_{max} with increments of 5. For each experiment 100 runs were performed at each point in the edge range, where edges were selected pseudo randomly. Also, during each experiment the hit probability of each rule is selected pseudo randomly and remains constant throughout the experiment. Hence this experiment is different from the last experiment since, in this experiment probability distribution remains constant for all the runs whereas, in the last experiment probability distribution varies for each run.

For each run the α, β, γ and brute force algorithms were executed. The number of rules n was varied in the range from 8 to 11, with increments of 1. For each integer $n \in [8, 11]$, 5 experiments were performed. Also, for each experiment, break cases for α, β and γ were recorded.

The graph in Figure 6.2 represents the cases for which α, β and γ ordering was not the same as optimal ordering. The x axis represents the number of rules for

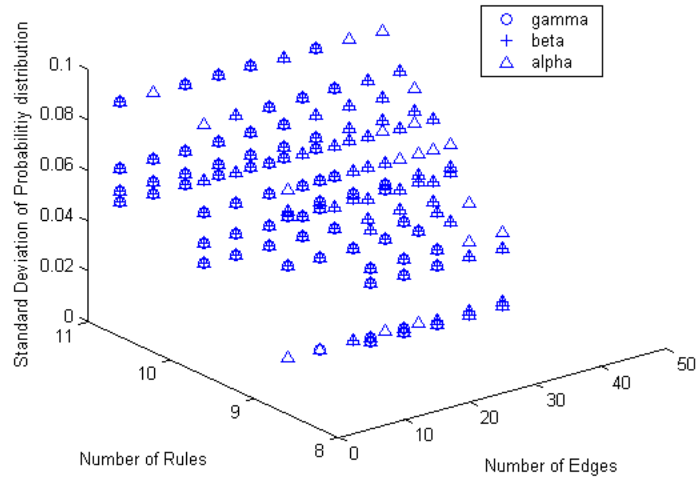
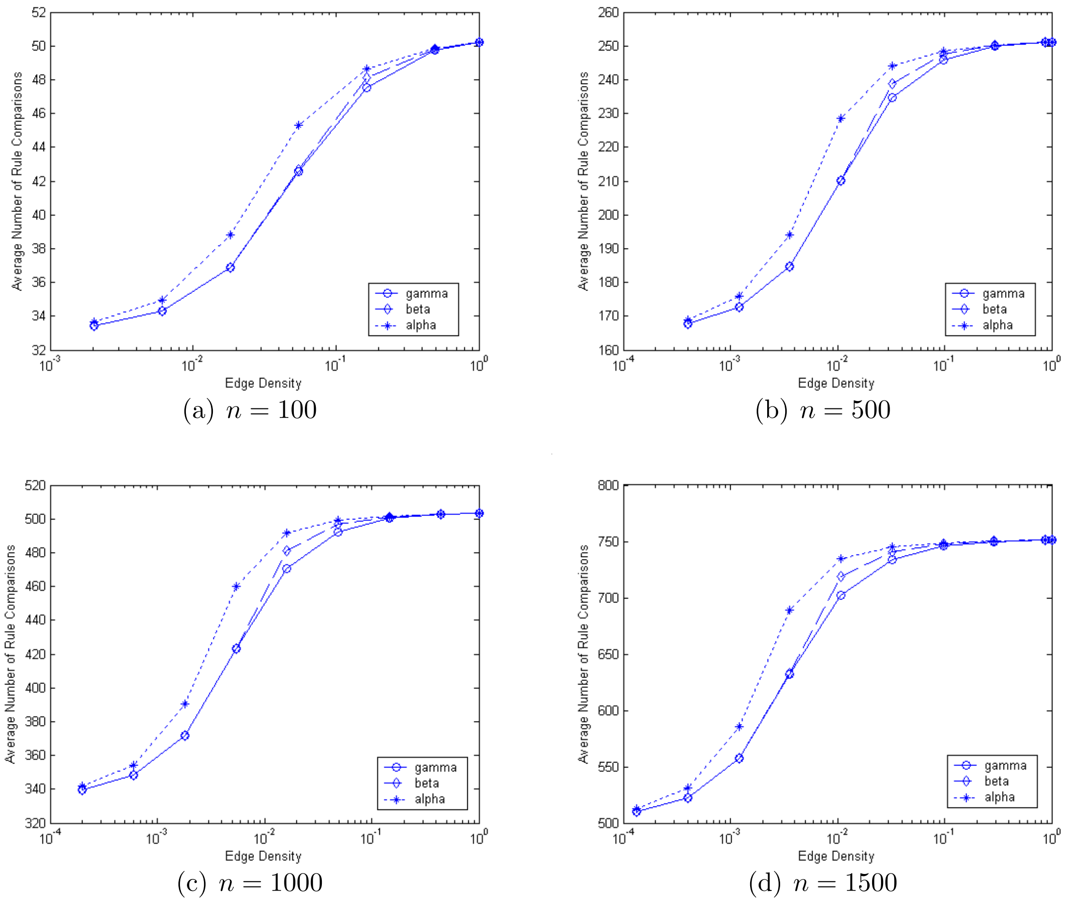


Figure 6.2: Number of edges versus Number of rules versus Standard deviation of probability distribution.

the break case; the y axis represents the edge density for the break case; and the z axis represents the standard deviation of probability distribution for each break case. This graph enables us to visualize the algorithmic overlap for each break case. As an example, the cases with $n = 9$ and $e = 10$ and a certain probability distribution suggests that out of at least 100 runs, the β algorithm did not fail whereas the α and the γ algorithms failed in one or more cases. Therefore, the simulation results indicate that the γ algorithm, on average, breaks fewer times as compared to the β algorithm; however, there exists cases for which γ algorithm can fail but β algorithm can find optimal ordering.

6.3 Average Delay versus Edge Density

The previous experiments compared the sorted policy orderings with the optimal ordering. This is computationally not feasible for policies with more than 12 rules. Therefore the experiments in this section consider large policies, but only compare



(a) $n = 100$ (b) $n = 500$

(c) $n = 1000$ (d) $n = 1500$

Figure 6.3: Edge density versus Average number of rule comparisons.

the average number of rule comparisons.

This experiment presents the variation in average number of rule comparisons as the edge density increases. The number of edges e were varied from 0 to e_{max} such that,

$$e_1 = 0, e_2 = n/10 \text{ and } e_x = 3 * e_{x-1}, \text{for } x = 3, ..., y \text{ such that } e_y \leq e_{max}$$

The number of edges are varied using a geometric function since, the difference between the minimum and the maximum number of possible edges is very large. Also, during each experiment 10 runs were performed at each point in the edge range. This experiment was performed for policy size n equal to 100, 500, 1000 and 1500. Also,

for each policy size experiment was repeated 3 times. The average number of rule comparisons at each point in the edge range per experiment was recorded. Each single run comprised execution of α, β and γ algorithms. Placement of edges was selected pseudo randomly. Also, during each experiment the hit probability of each rule is selected pseudo randomly and remains same throughout the experiment.

The graphs in Figure 6.3 indicate that as the edge density increases both β and γ algorithms perform better than α algorithm. Also, it can be seen that as the edge density increases performance difference decreases and can be attributed to the inverse relationship between the number of possible ordering and the edge density. The performance difference between β and γ algorithm becomes more prominent at lower edge densities as the number of rules increase. This behavior can be explained with the number the breaks versus edge density graphs in Figure 6.1. The number of breaks for β algorithm peaks in the middle. Hence, it can be inferred that the difference is because of the prevalence of these break cases.

6.4 Zipf Probability Distribution

The previous experiments used a uniform distribution for the policy profile, which may not be always commiserate with actual policies. The Zipf distribution for firewall policy profile has been shown to closely resemble the actual distributions of firewall hit probabilities [2, 12, 16, 22]. Hence, in this experiment prior probability for each rule is selected such that it follows Zipf distribution. Also, the probability remains same throughout the experiment.

Similar to the last experiment, this experiment presents the variation in average number of rule comparisons with the increase in the edge density. These experiments were performed with policy size n equal to 1000, 2000, 3000, 4000 and 5000. The

number of edges e were varied such that,

$$e_1 = 0, e_2 = n/10 \text{ and } e_x = 3 * e_{x-1}, \text{for } x = 3, ..., 7$$

Also, during each experiment 10 runs were performed at each point in the edge range. Each single run comprised execution of α, β and γ algorithms. Edges were selected pseudo randomly and the average number of rule comparisons at each point in the edge range was recorded.

For all the graphs in Figure 6.4, as the edge density increases both β and γ algorithms perform better than α algorithm. However as the number of edges increase the percentage difference decreases, which is because of the inverse relationship between the number of edges and the number of different possible rule orderings. Also, the difference between β and γ orderings is visible only in the last point of the edge range across all graphs. It implies the prevalence of cases where β algorithm fails and γ does not, resulting in a substantial difference between two orderings.

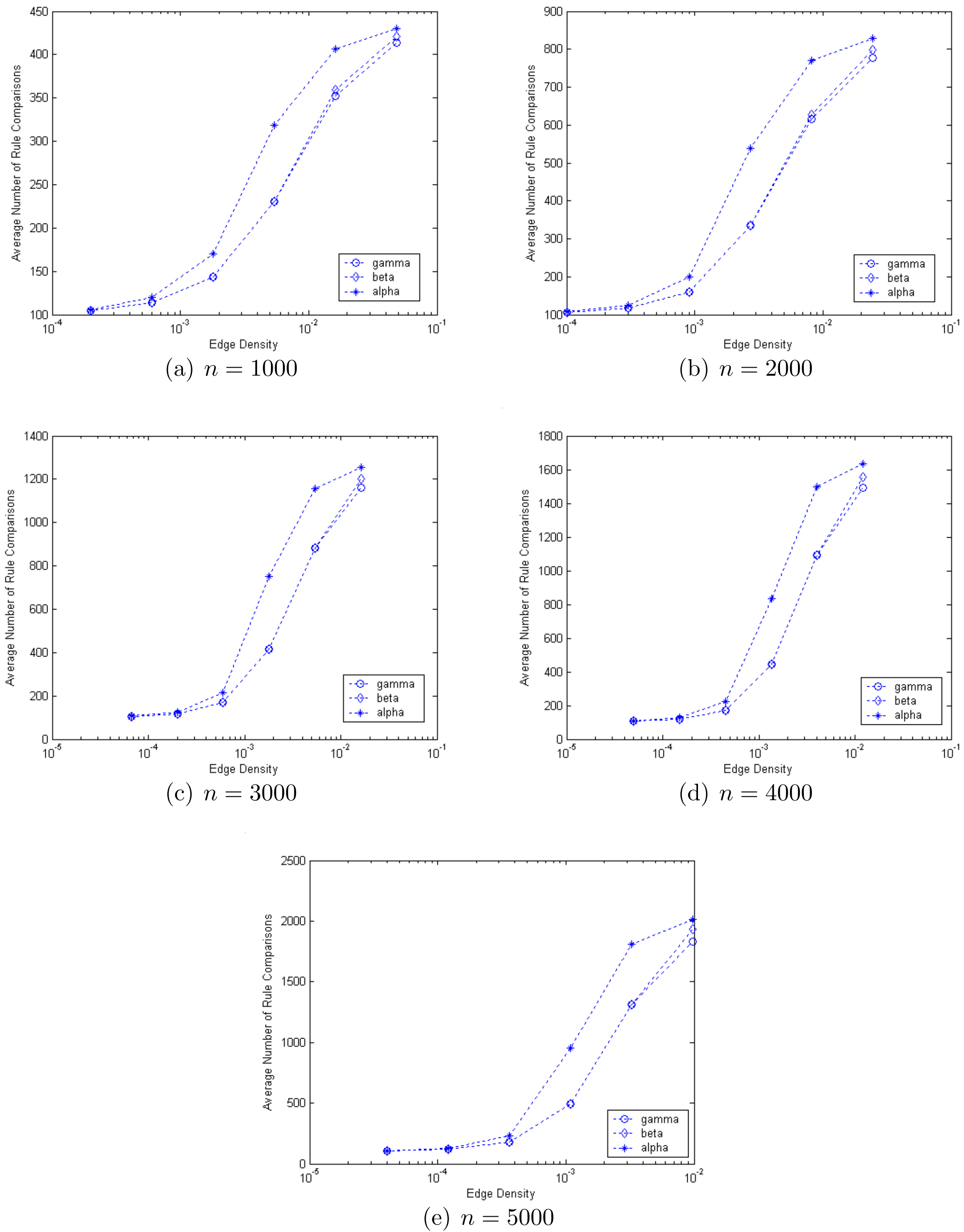


Figure 6.4: Zipf distribution.

Chapter 7: Conclusions and Future Work

This chapter provides with the summary of theoretical and empirical results established throughout the thesis work. It concludes with the potential areas for future work.

7.1 Conclusions

Firewalls enforce a security policy by inspecting packets arriving or departing a network. This is accomplished by comparing the packet header to a list of rules until the first match is found. This process can be improved if more popular rules appear earlier in the policy, unfortunately a simple sorting method is not possible due to precedence constraints (relative order of rules) must be maintained. As a result this problem is equivalent to job shop scheduling, a known $\mathcal{NP}$-hard problem.

This paper proves that the problem of determining optimal firewall rule set ordering is $\mathcal{NP}$-hard, by reducing it from the problem of single machine job scheduling with precedence conditions. Hence, for any policy, it is apparently infeasible to find the optimal rule order in reasonable time.

Previous work [10, 11] mentioned the impact of suboptimal placement of rules in a firewall policy in the context of increasing transmission speeds. The author also present a simple algorithm for sorting the rules which have no precedence constraints between them [10, 11]. However, in order to maximize the optimization and minimize the average delay experienced by the incoming packets, rules with precedence relationships between them must be reordered along with the independent rules.

This paper presents a novel heuristic sorting technique, called the γ algorithm,

which is statistically superior to previous approaches. In addition to that, this paper also compares the performance of three different heuristic sorting algorithms α, β and γ, based on the average number of cases where the algorithm is unable to find the optimal ordering and the average number of rule comparisons.

The α heuristic during each pass selects and inserts the rule with the highest average spanning tree probability from the set of rules that do not have any incoming edge to them. The β algorithm during each pass selects and inserts the rule's subgraph that has the highest average subgraph probability. The γ algorithm during each pass selects the rule with the highest average sub graph probability and inserts that rule if it has no dependents on it, otherwise it recursively sorts the sub graph of its dependents until it finds a rule that has no dependents on it.

Two set of experiments were performed, the first set of results exhibit the average number of breaks for different heuristics as the edge density increases. Since determining the optimal ordering for large policies utilizing brute force is not feasible. Hence, large number of experiments were performed with small policy size with n equal to 8, 9, 10 and 11. The second set of experimental results represent the variation in average number of rule comparisons for different heuristics as the edge density is increased for the policy size ranging between 100 and 5000.

In all the experiment with policy size of 8, 9, 10 and 11, on an average γ algorithm breaks fewer percentage of times as compared to the α and β algorithms. Also, for the policy size ranging between 100 and 5000, in all the experiments and for different probability distributions, ordering obtained from γ algorithm results in lower average delay as compared to the α and β algorithm ordering delays.

This paper also acknowledges the trade off between fewer rule comparisons and higher computational cost for the γ algorithm.

In the context of policy management, this paper presents an anomaly condition

which might exist in a policy or can be introduced during the addition of a new rule to a policy. In particular, a set of rules higher above a certain rule in the policy might shadow it. Hence a shadow detection algorithm is presented which will enable the policy to be free from anomaly and will also enhance the policy management.

7.2 Future Work

This thesis compared three different sorting techniques that used the same heuristic. In every approach, heuristic compares and selects the rule from the set of candidate rules, that has the highest arithmetic mean probability of rules in its subgraph or spanning tree. Other heuristics with a different mathematical function such as geometric mean or harmonic mean could also be attempted with the same approaches.

As seen in the experimental results, the γ algorithm is an improvement over β algorithm. This improvement was result of mathematical analysis of a set of cases for which β algorithm failed to find the optimal ordering. However, this set was not comprehensive of all the possible cases. Therefore, a more exhaustive analysis with the same heuristic might result in further improvement.

Also, in the experiments with policy size of 8, 9, 10 and 11, it was observed that the break cases for none of the algorithm is subset of the break cases for any other algorithm. Hence, further analysis of these cases where one algorithm fails but other doesn't can lead to a new approach which will be a blend of three approaches presented in this paper; and perform better than all existing approaches.

This thesis also presented an algorithm which detects whether a rule in question is shadowed by the set of rules that are above this rule in the policy and intersect with it. However, determining the smallest subset of these intersecting rules that can shadow the rule in question is a combinatorial problem. Hence, heuristics can be developed to narrow down the size of this set to the actual contributor rules.

In this paper, rules and their dependencies are represented utilizing DAG. An alternative approach to improving firewall performance involves geometrical representation and analysis of rules [21]. In this approach each rule with k parameters can be represented in a single k dimensional space. Hence, the surface obtained can then be split into non-overlapping k-oid's . In the new policy each k-oid will represent a rule. Also, since none of the k-oid's overlap with each other therefore, each rule will be independent of any other rule in the transformed policy. This will enable the rule in the transformed policy to be sorted in the decreasing order by their hit probability.

References

[1] RFC 791. http://www.rfc-editor.org/rfc/rfc791.txt.

[2] Subrata Acharya, Jia Wang, Zihui Ge, Taieb F. Znati, and Albert Greenberg. Traffic-aware firewall optimization strategies. In *Proceedings of IEEE International Conference on Communications (ICC)*, 2006.

[3] Ehab Al-Shaer and Hazem Hamed. Firewall policy advisor for anomaly detection and rule editing. *IEEE/IFIP Integrated Network Management (IM'03)*, 2003.

[4] Ehab Al-Shaer and Hazem Hamed. Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.

[5] Carstan Benecke. A parallel packet screen for high speed networks. In *Proceedings of the* 15th *Annual Computer Security Applications Conference*, 1999.

[6] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernard Plattner. Router plugins: A software architecture for next-generation routers. *IEEE/ACM Transactions on Networking*, 8(1), February 2000.

[7] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proceedings of ACM SIGCOMM*, pages 4 – 13, 1997.

[8] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings of IEEE INFOCOM*, pages 397 – 413, 2000.

[9] CRS Report for Congress, 2008. http://www.fas.org/sgp/crs/terror/RL32114.pdf.

[10] Errin W. Fulp. Optimization of network firewall policies using ordered sets and directed acyclical graphs. Technical report, Wake Forest University, Computer Science Department, 2004.

[11] Errin W. Fulp. Optimization of network firewalls policies using directed acyclical graphs. In *Proceedings of IEEE Internet Management Conference (IM'05)*, 2005.

[12] Errin W. Fulp and Ryan J. Farley. A function-parallel architecture for high-speed firewalls. In *Proceedings of IEEE International Conference on Communications*, 2006.

[13] P. L. Hammer, E. L. Johnson, and B. H. Korte. *Discrete Optimization II.* NORTH-HOLLAND, 1979.

[14] Michael R. Horvath. Managing rule policies across a function parallel firewall array. Master's thesis, Wake Forest University, Computer Science Department, 2007.

[15] E. L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[16] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self similar nature of ethernet traffic. *IEEE Transactions on Networking*, 2:1–15, 1994.

[17] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978A.

[18] Olivier Paul and Maryline Laurent. A full bandwidth atm firewall. In *Proceedings of the* 6th *European Symposium on Research in Computer Society ESORICS'2000*, 2000.

[19] Venkatesh Prasad Ranganath and Daniel Andresen. A set-based approach to packet classification. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 889 – 894, 2003.

[20] W. E. Smith. Various optimizers for single stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[21] Priyank Warkhede, Subhash Suri, and George Varghese. Fast packet classification for two-dimensional conflict-free filters. In *Proceedings of IEEE INFOCOM*, pages 1434 – 1443, 2001.

[22] Avishai Wool. A quantitative study of firewall configurations errors. *IEEE Computer*, 37(6):62–67, June 2004.

[23] Robert L. Ziegler. *Linux Firewalls.* New Riders, second edition, 2002.

[24] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls.* O'Reilly, 2000.

Vita

ASHISH TAPDIYA

- 1703 Aspen Way, Winston-Salem, NC 27106
  email: tapda7@wfu.edu
  phone: (336) 692-1883

EDUCATION

- Masters of Science, Computer Science
  Wake Forest University, Winston Salem, NC
  December 2008
  Thesis: "Firewall Policy Optimization and Management"
  3.952 GPA

- Bachelor of Engineering in Information Technology
  Shri Govindram Seksaria Institute of Technology and Sciences, Indore, INDIA
  April 2005

HONORS

- Upsilon Pi Epsilon (2008) International Honor Society for the Computing Sciences.

EXPERIENCE

- *Software Engineer*
  GreatWall Systems, Winston Salem, NC
  May - August 2008
  Researched and developed algorithm for removing shadows from firewall policy, resulting in enhanced policy management and anomaly free security policy.

- *Information Systems Assistant*
  Information Systems, Wake Forest University, Winston-Salem, NC
  January - May 2007
  Installed and configured Asterisk Server, a Linux based open source IP-PBX. Integrated Asterisk Server with Cisco Call Manager to enable interoperability while minimizing cost.

- *Assitant Systems Engineer*
  Tata Consultancy Services Ltd., Mumbai, INDIA
  September 2005 - October 2006
  Proposed, implemented and tested SMS forward feature resulting in enhancement of functionality provided by Nortel's MSC.