

# Optimization of Network Firewall Policies Using Ordered Sets and Directed Acyclical Graphs\*

Errin W. Fulp  
Department of Computer Science  
Wake Forest University  
Winston-Salem, NC, USA 27109  
nsg.cs.wfu.edu  
fulp@wfu.edu

## Abstract

Firewalls enforce a security policy by inspecting and filtering traffic arriving or departing from a secure network. This is typically done by comparing an arriving packet to a list of rules and performing the matching rule action, which is accept or deny. Unfortunately packet filtering can impose significant delays on traffic due to the complexity and size of rule sets. Therefore, improving firewall performance is important, given network Quality of Service (QoS) requirements and increasing network traffic loads.

One approach for improving traditional firewall performance is to reduce the number of rule comparisons required per packet. This can be done by placing the most commonly matched rules closer to the beginning of the rule list. Unfortunately a straightforward sort is not possible, since the relative order (precedence relationship) of certain rules must be maintained to preserve policy integrity. Utilizing Directed Acyclical Graphs (DAG) to represent the precedence of rules, this paper will prove that determining the optimal firewall rule order is  $\mathcal{NP}$ -hard. However, a simple and effective sorting heuristic is introduced for reducing the average number of comparisons. Simulation results will show the proposed rule list manipulation technique can significantly improve firewall performance (70% fewer rule comparisons) by providing lower delays while maintaining the original policy integrity.

**Keywords:** Security management, firewalls, policy representation, and performance optimization.

## 1 Introduction

Inspecting traffic sent between networks, a firewall provides access control, auditing, and traffic control based on a security policy [3, 28, 29]. A security policy is a list of ordered rules, as seen in table 1, that defines the action to perform on matching packets. For example, an accept action passes the packet into or from the secure network, while deny causes the packet to be discarded.

---

\*This work was supported by the U.S. Department of Energy MICS (grant DE-FG02-03ER25581). The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOE or the U.S. Government.

In many implementations, the rule set is stored internally as a linked list. A packet is sequentially compared to the rules, starting with the first, until a match is found; otherwise, a default action is performed [28, 29]. This is referred to as a *first-match* policy and is used in many firewall systems including the Linux firewall implementation `iptables` [24].

Traditional firewall implementations consist of a single, dedicated machine, similar to a router, that sequentially applies rules to each arriving packet. However, packet filtering represents a significantly higher processing load than routing decisions [23, 27, 29]. For example, a firewall that interconnects two 100 Mbps networks would have to process over 300,000 packets per second [28]. Successfully handling these traffic loads becomes more difficult as rule sets become more complex [4, 21, 29]. Furthermore, firewalls must be capable of processing even more packets as interface speeds increase. In a high-speed environment (e.g. Gigabit Ethernet), a single firewall can easily become a bottleneck and is susceptible to DoS attacks [4, 8, 12, 13]. An attacker could simply inundate the firewall with traffic, delaying or preventing legitimate packets from being processed.

One approach to increase firewall performance focuses on improving hardware design. Current research is investigating different distributed firewall designs to reduce processing delay [4, 8, 21], and possibly provide service differentiation [10]. Another approach focuses on improving performance via better firewall software [6, 11, 15, 16, 23]. Similar to approaches that address the longest matching prefix problem for packet classification [7, 9, 24], solutions typically represent the firewall rule set in different fashions (e.g. tree structures) to improve performance. While both approaches, together or separate, show great promise, each requires radical changes to the firewall system, and therefore are not amenable to current or legacy systems.

This paper introduces a new method to improve performance that can be applied to list oriented firewall systems. Specifically, the paper addresses reordering a firewall rule set to minimize the average number of rule comparisons to determine the action, while maintaining the integrity of the original policy. Integrity is preserved if the reordered and original rules always arrive at the same result. To maintain integrity, this paper will model the rule set as a Directed Acyclical Graph (DAG), where vertices are firewall rules and edges indicate precedence relationships. Given this representation, any linear arrangement of the policy DAG (which is a list of rules) is proven to maintain the original policy integrity. Unfortunately, determining the optimal rule order from all the possible linear arrangements is proven to be  $\mathcal{NP}$ -hard, since it is equivalent to sequencing jobs with precedence constraints for a single machine [14]. Although determining the optimal order is  $\mathcal{NP}$ -hard, this paper will introduce a simple heuristic to order firewall rules that reduces the average number of comparisons while maintaining integrity. Simulation results show the proposed reordering method yields rule orders that are comparable to optimal (11% difference); thus, provides a simple means to significantly improve firewall performance and lower packet delay.

The remainder of this paper is organized as follows. Section 2 describes the rule and policy models used in this paper. Policy DAG's are introduced to represent precedence relationships among rules, and their linear arrangements are utilized to determine possible rule orders. Optimization and a proof that ordering firewall rules is  $\mathcal{NP}$ -hard is given in section 3. Section 4 describes a simple approach for sorting firewall rules. The performance of these methods are investigated experimentally in section 5. Finally, section 6 summarizes this paper and discusses some areas of future research.

## 2 Modeling Firewall Security Policies

As described in the introduction, a firewall rule set, also known as a firewall policy, is traditionally an ordered list of firewall rules. Firewall policy models have been the subject of recent research [1, 2, 15]; however, the primary purpose is anomaly detection and policy verification. In contrast, the policy model introduced in this paper is designed for firewall performance optimization and integrity. Firewall performance refers to reducing the average number of comparisons required to determine an action, while integrity refers to maintaining the original policy intent. Although improving the worst case performance is important, it is not possible without changing the list-based representation [15, 23].

No.	Proto.	Source		Destination		Action	Prob.
		IP	Port	IP	Port		
1	UDP	1.1.*	*	*	80	deny	0.01
2	TCP	1.*	*	1.*	90	accept	0.02
3	TCP	2.*	*	2.*	20	accept	0.25
4	UDP	1.*	*	*	*	accept	0.22
5	*	*	*	*	*	deny	0.50

Table 1: Example security policy consisting of multiple ordered rules.

### 2.1 Firewall Rule and Policy Models

In this paper, a rule  $r$  is modeled as an ordered tuple of sets,  $r = (r[1], r[2], \dots, r[k])$ . Order is necessary among the tuples since comparing rules and packets requires the comparison of corresponding tuples. Each tuple  $r[l]$  is a set that can be fully specified or contain wildcards ‘\*’ in standard prefix format. For the Internet, security rules are commonly represented as a 5-tuple consisting of: protocol type, IP source address, source port number, IP destination address, and destination port number [28, 29]. For example, the prefix  $192.*$  would represent any IP address that has 192 as the first dotted-decimal number. Given this model, the ordered tuples can be supersets and subsets of each other, which forms the basis of precedence relationships. In addition to the prefixes, each filter rule has an action, which is to accept or deny. However, the action will not be considered when comparing packets and rules. Similar to a rule, a packet (IP datagram)  $d$  can be viewed as an ordered  $k$ -tuple  $d = (d[1], d[2], \dots, d[k])$ ; however, wildcards are not possible for any packet tuple.

Using the previous rule definition, a standard security policy can be modeled as an ordered set (list) of  $n$  rules, denoted as  $R = \{r_1, r_2, \dots, r_n\}$ . A packet  $d$  is sequentially compared against each rule  $r_i$  starting with the first, until a match is found ( $d \Rightarrow r_i$ ) then the associated action is performed. A match is found between a packet and rule when every tuple of the packet is a subset of the corresponding tuple in the rule.

**Definition** Packet  $d$  matches  $r_i$  if

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

The rule list  $R$  is comprehensive if for every possible legal packet  $d$  a match is found using  $R$ . Furthermore, two rule lists  $R$  and  $R'$  are equivalent if for every possible legal packet  $d$  the same action is performed by the two rule lists. If  $R$  and  $R'$  are different (e.g. a reorder) and the lists are equivalent, then the **policy integrity** is maintained.

As previously mentioned, a rule list has an implied precedence relationship where certain rules must appear before others if the integrity of the policy is to be maintained. For example consider the rule list in table 1. Rule  $r_1$  must appear before rule  $r_4$ , likewise rule  $r_5$  must be the last rule in the policy. If for example, rule  $r_4$  was moved to the beginning of the policy, then it will *shadow* [2] the original rule  $r_1$ . Shadowing is an anomaly that occurs when a rule  $r_j$  matches a preceding rule  $r_i$ , where  $i < j$ . As a result of the relative order  $r_j$  will never be utilized. However, there is no precedence relationship between rules  $r_1$ ,  $r_2$ , or  $r_3$  given in table 1. Therefore, the relative ordering of these three rules will not impact the policy integrity. This paper assumes the original policy is free from any anomalies. Likewise, when a policy is reordered to improve performance it should not introduce any anomalies, which will occur if precedence relationships are not maintained. As a result, a model is needed to effectively represent precedence relationships.

## 2.2 Modeling Precedence Relationships

The precedence relationship between rules in a policy will be modeled as a Directed Acyclical Graph (DAG) [22, 17]. Such graphs have been successfully used to represent the relative order of individual tasks that must take place to complete a job (referred to as a task graph model). Since certain rules must appear before others to maintain policy integrity, this structure is well suited for modeling the precedence of firewall rules.

Let  $G = (R, E)$  be a *policy DAG* for a rule list  $R$ , where vertices are rules and edges  $E$  are the precedence relationships (constraint). A precedence relationship, or edge, exists between rules  $r_i$  and  $r_j$ , if  $i < j$  and the rules intersect.

**Definition** The intersection of rule  $r_i$  and  $r_j$ , denoted as  $r_i \cap r_j$  is

$$r_i \cap r_j = (r_i[l] \cap r_j[l]), \quad l = 1, \dots, k$$

Therefore, the intersection of two rules results in an ordered set of tuples that collectively describes the packets that match both rules. The rules  $r_i$  and  $r_j$  intersect if every tuple of the resulting operation is non-empty. In contrast, the rules  $r_i$  and  $r_j$  do not intersect, denoted as  $r_i \not\cap r_j$ , if at least one tuple is the empty set. Note the intersection operation is symmetric; therefore, if  $r_i$  intersects  $r_j$ , then  $r_j$  will intersect  $r_i$ . The same is true for rules that do not intersect.

For example consider the rules given in table 1, the intersection of  $r_1$  and  $r_4$  yields (UDP, 1.1.\*, \*, \*, 80). Again, the rule actions are not considered in the intersection or match operation. Since these two rules intersect, a packet can match both rules for example  $d = (\text{UDP}, 1.1.1.1, 80, 2.2.2.2, 80)$ . The relative order must be maintained between these two rules and an edge drawn from  $r_1$  to  $r_4$  must be present in the DAG, as seen in figure 1. In contrast consider the intersection of rules  $r_2$  and  $r_3$  in the same policy. These two rules do not intersect due to the fifth tuple (destination port). A packet cannot match both rules indicating the relative order can change; therefore, an edge will not exist between them.

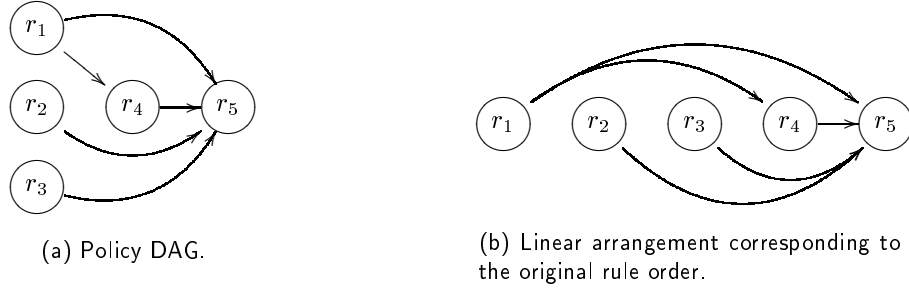


Figure 1: Policy DAG representations of the firewall rules given in table 1. Vertices are rules while edges indicate precedence requirements.

The match operation can be used to identify the precedence relationships in the previous examples, but it cannot do so in every case. Consider a *partial-match* example [1], where  $r_a = (\text{UDP}, *, 80, 10.*, 90, \text{accept})$  and  $r_b = (\text{UDP}, 10.*, 80, *, 90, \text{reject})$ . The intersection of  $r_a$  and  $r_b$  is  $(\text{UDP}, 10.*, 80, 10.*, 90)$ ; therefore a packet, such as  $d = (\text{UDP}, 10.10.10.10, 80, 10.10.10.10, 90)$ , can match both rules. If  $r_a$  appears before  $r_b$  then the packet  $d$  is accepted, but if  $r_b$  occurs before  $r_a$  then  $d$  is rejected. As a result, the order of  $r_a$  and  $r_b$  in the original policy must be maintained. However, the match operation is unable to identify the precedence in this example.

Using the policy DAG representation a linear arrangement is sought that improves the firewall performance. As depicted in figure 1(b), a linear arrangement (permutation or topological sort) is a list of DAG vertices where all the successors of a vertex appear in sequence after that vertex [22]. Therefore it follows that a linear arrangement of a policy DAG represents a rule order, if the vertices are read from left to right. Furthermore, it is proven in theorem 2.1 that any linear arrangement of a policy DAG maintains integrity.

**Theorem 2.1** *Any linear arrangement of a policy DAG maintains integrity.*

**Proof** Assume a policy DAG  $G$  is constructed from the security policy  $R$  that is free of anomalies. Consider any two rules  $r_i$  and  $r_j$  in the policy, where  $i < j$ . If an edge between  $r_i$  and  $r_j$  in  $G$  does not exist, then a linear arrangement of  $G$  can interchange the order of the two rules. A reorder will not effect integrity since a packet cannot match both rules, as indicated by the intersection operation. Shadowing is not introduced due to the reorder since the intersection operation is symmetric. If an edge does exist between the rules, then their relative order will be maintained in every linear arrangement of  $G$ ; thus maintaining precedence and integrity. ■

### 3 Rule List Optimization

As mentioned in the introduction, it is important to inspect packets as quickly as possible given increasing network speeds and QoS requirements. Using the policy DAG to maintain policy in-

tegrity, a linear arrangement is sought that minimizes the average number of comparisons required. However, this will require information not present in the firewall rule list. Certain firewall rules have a higher probability of matching a packet than others. As a result, it is possible to develop a *policy profile* over time that indicates frequency of rule matches (similar to cache hit ratio). Let  $P = \{p_1, p_2, \dots, p_n\}$  be the policy profile, where  $p_i$  is the probability that a packet will match rule  $i$  (first match in the policy). Furthermore, assume a packet will always find a match,  $\sum_{i=1}^n p_i = 1$ ; therefore  $R$  is comprehensive. Using this information, the average number of rule comparisons required is

$$E[n] = \sum_{i=1}^n i \cdot p_i \quad (1)$$

For example, the average number of comparisons required for the rule set in table 1 is 4.18.

Given a policy DAG  $G = (R, E)$  and policy profile  $P = \{p_1, p_2, \dots, p_n\}$  a linear arrangement  $\pi$  of  $G$  is sought that minimizes equation 1. In the absence of precedence relationships, the average number of comparisons is minimized if the rules are sorted in non-increasing order according to the probabilities [25], which is also referred to as Smith's algorithm [26]. Precedence constraints causes the problem to be more realistic; however, it also makes determining the optimal permutation more problematic.

Determining the optimal rule list permutation can be viewed as job scheduling problem for a single machine with precedence constraints [14, 20]. The notation for such scheduling problems is  $\alpha|\beta|\gamma|\delta$ , where  $\alpha$  is the number of machines,  $\beta$  is the precedence (or absence of) which can be represented as a DAG,  $\gamma$  is a restriction on processing time, and  $\delta$  is the optimality criterion [14]. Determining the optimal rule order is similar to the  $1|\beta|1|\sum w_i C_i$  scheduling problem, or optimality criterion, where  $w_i$  is a weight associated with a job (for example, importance) and  $C_i$  is the completion time. As previously noted, the  $1||\sum w_i C_i$  problem can be solved in linear time the using Smith's algorithm [26], which orders jobs according to non-decreasing  $\frac{t_i}{w_i}$  ratio, where  $t_i$  is the processing time of job  $i$ . In this case set  $t_i = 1$  and  $w_i = p_i \quad \forall i$ . However, Lawler [18] and Lenstra et. al. [20] proved  $1|\beta|1|\sum w_i C_i$  to be  $\mathcal{NP}$ -hard via the linear arrangement problem, which implies determining the optimal firewall rule order is also  $\mathcal{NP}$ -hard. Note, determining the number of possible permutations has been proven to be  $\#\mathcal{P}$ -hard [5].

**Theorem 3.1**  $1|\beta|1|\sum w_i C_i \propto$  *Determining the optimal order of a firewall rule list*

**Proof** Consider the  $1|\beta|1|\sum w_i C_i$  problem. Each of  $n$  jobs  $J_i, i \in I$ , has to be processed without preemption on a single machine that can handle at most one job at a time. For each  $i \in I$ , let  $w_i$  be the associated weight. Furthermore, let  $G = (V, E)$  be a DAG that represents the precedence order of the jobs  $J_i$ . Assume the processing time of each job equals 1 time unit, the weights to be  $0 \leq w_i \leq 1$  such that  $\sum w_i = 1$ , and  $\beta$ , which is  $G$ , to be a policy DAG. In this case, the optimization criterion  $\sum w_i \cdot C_i$  is the same as  $\sum p_i \cdot i$ , which is given in equation 1. Clearly, the optimal firewall rule ordering problem has a solution if and only if  $1|\beta|1|\sum w_i C_i$  has a solution. Therefore, determining the optimal permutation of firewall rules is  $\mathcal{NP}$ -hard. ■

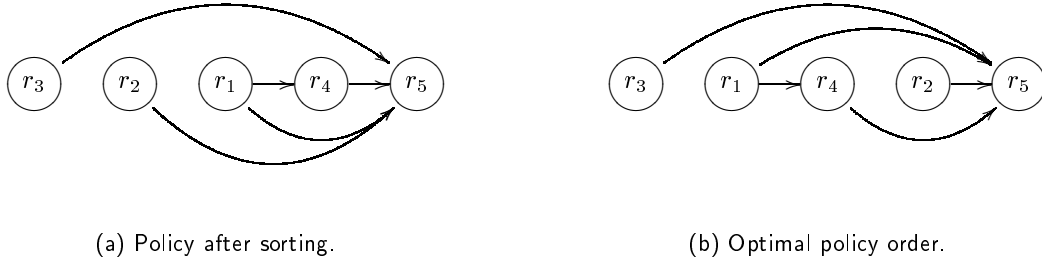


Figure 2: Example policy orders (linear arrangements) for the rule set given in table 1.

## 4 A Simple Rule Sorting Algorithm

Although determining the optimal rule permutation was proven to be  $\mathcal{NP}$ -hard, we are still interested in reducing the average number of comparisons required to process a packet. As previously discussed, a sorting algorithm must maintain the precedence relationships among rules. Of course an exhaustive search is possible if the number of rules is small (generate and test all possible topological orders); however as proven in the previous section, this is not feasible with a realistic rule list.

A simple algorithm starts with the original the rule set, then sorts neighboring rules based on non-increasing probabilities. However, an exchange of neighbors should never occur if the rules intersect. This test preserves any precedence relationships in the policy. For example, the following sorting algorithm uses such a comparison to determine if neighboring rules should be interchanged.

```

done = false
while(!done)
  done = true
  for(i = 1; i < n; i++)
    if( $p_i < p_{i+1}$  AND  $r_i \not\cap r_{i+1}$ )then
      interchange rules and probabilities
      done = false
    endif
  endfor
endwhile

```

Other sorting algorithms are possible if the completion time of the sort is an issue; however, the method presented is easy to implement and only requires a simple neighbor comparison. Assume the match probabilities for the rule list given in table 1. Applying the sorting algorithm to this rule list results in the ordering depicted in figure 2(a), which has 11% fewer comparisons on average. However when using the algorithm, it is possible that one rule can prevent another rule from being reordered. For example, rule  $r_1$  prevents rule  $r_4$  from being placed closer to the beginning of the rule set. However, if both rules  $r_1$  and  $r_4$  are placed closer to the beginning of the policy while

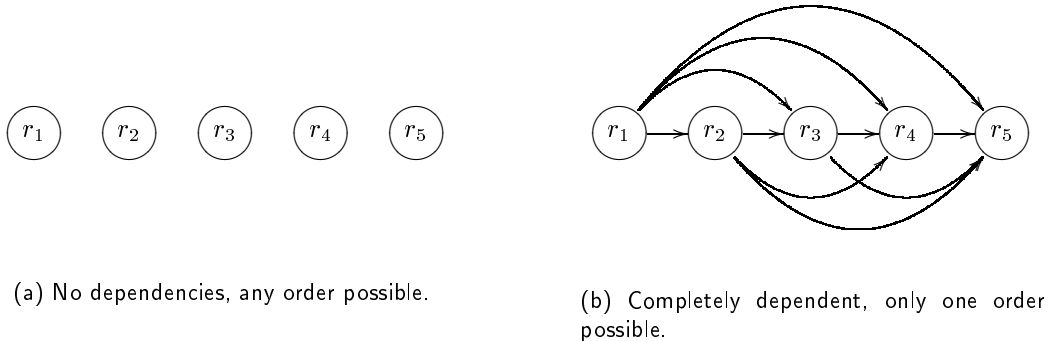


Figure 3: Example policy dependencies for a five rule list.

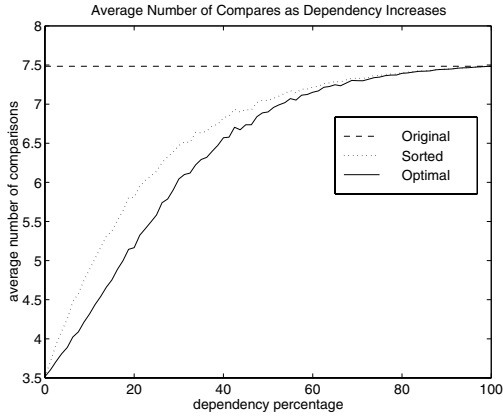
maintaining their relative order, the average number of comparison will be further reduced, 16% fewer. This is the optimal order for the 5 rule set, which is depicted in figure 2(b). Although this simple sorting algorithm is unable to move groups of rules, it can still improve the performance of the firewall system. Its effectiveness is measured experimentally in the next section.

## 5 Experimental Results

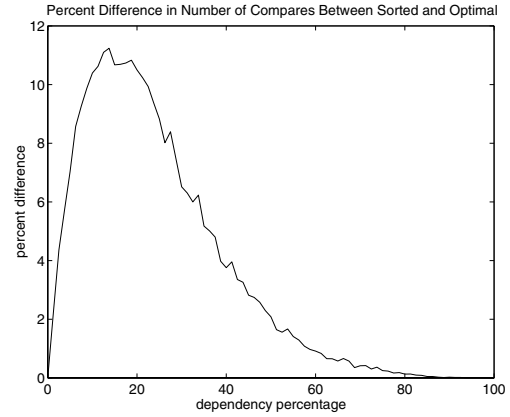
In this section, the average number of rule comparisons required after sorting the rules (using the algorithm described in section 4) is compared with average number of comparisons required using the original order and the optimal order. Note, the optimal rule ordering was determined via an exhaustive search. It was not feasible to determine the optimal ordering once the number of rules equaled 15 given the number of permutations to consider.

In the first experiment, lists of 10 firewall rules were generated with random precedence relationships. The match probability of each rule was given by a Zipf distribution [19], which assigns probabilities according to the rank of an item. For this simulation, the last rule had the highest probability which is consistent with most policies (last rules are more general). As a result, the original order yields the worst average number of comparisons. The precedence relationships ranged from no dependencies as seen in figure 3(a), to completely dependent depicted in figure 3(b). A rule list with no dependencies has an optimal order that can be obtained using Smith’s algorithm. In contrast, only one ordering (the original) is possible if the list is completely dependent. A list with no dependencies will have a zero dependency percentage, while a completely dependent list will have a dependency percentage of 100%. Dependency percentages between these two extremes were modeled by randomly adding dependency edges based on the percentage value. For example consider a 5 rule list, a dependency percentage of 50% would have approximately half of the edges depicted in figure 3(b). Edges were included using a uniform distribution. Results (average number of rule comparisons) for a particular dependency percentage were then averaged over 1000 simulations.

Results of the first experiment are given in figure 4. The average number of comparisons required was lower for the sorted and optimal lists when the dependency percentage was low, as



(a) Average number of comparisons required.



(b) Percent difference in average number of comparisons required between sorted and optimal.

Figure 4: Average number of comparisons required for a security policy with 10 rules as the dependency percentage increases.

seen in figure 4(a). The maximum percent difference observed was 53% as compared to the original rule order, which was expected since there is a large number of possible permutations. When the precedence percentage approached 1, the values converge to the number required for the original list. This is due to the limited number of possible rule orders, only one order in the extreme case. The percent difference between the sorted and optimal order is shown in figure 4(b). At zero dependency percentage, the sorted and optimal orders are equal, since any ordering is possible. Similarly at a dependency percentage of 100, the sorted and optimal orders are equal since only one order is possible. Between these two extremes, the sorting algorithm remains close to the optimal value with a maximum 11% difference. Although not optimal, the results indicate the proposed sorting algorithm can provide reductions in the average number of comparisons.

The benefit of rule sorting on larger policies is also of interest; however as previously described, it was not possible to determine the optimal ordering once the number of rules approached 15. Therefore, the second experiment used larger rule sets, but only compared the original order and the sorted order. The number of rules ranged from 10 to 2000 rules, while the matching probabilities and dependency percentages were the same as the previous experiment. The results of this experiment are depicted in figure 5. The sorted rule set always performed equal to or better than the original order. As noted in the previous experiment, the percent difference is very large given few dependencies (e.g. 80% decrease for 400 rules with 0% dependency as seen in figure 5(a)), but approaches zero as the rules are completely dependent. As the number of rules increases, sorting is increasingly beneficial; although only at low dependency percentages. For example consider the zero dependency case, sorting a policy of 10 rules yielded a 60% reduction in the number of compares, as compared to a 80% reduction when sorting a policy of 400 rules. The benefit of sorting drops more significantly as the dependency increases with larger rule sets, as seen in figure 5(b). This is primarily due to the low matching probabilities of each rule, which requires a complete

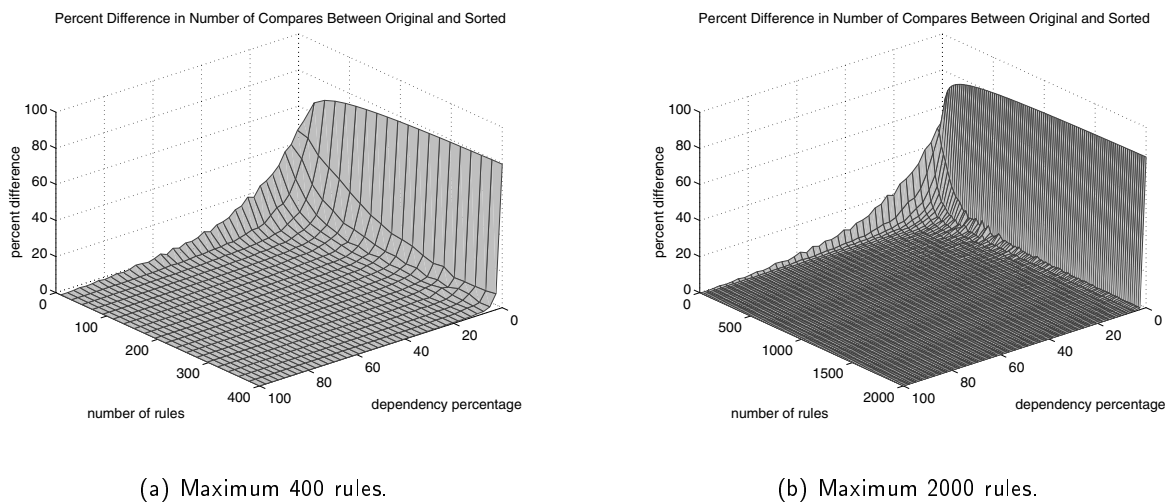


Figure 5: Percent difference in the average number of comparisons required between sorted and original rule orders.

reordering to have a significant impact on the average number of comparisons. Therefore, a large rule set can benefit from sorting if the dependency percentage is low.

## 6 Conclusions

Firewall systems inspect and filter traffic according to a certain security policy which traditionally consists of a list of ordered rules. Arriving packets are compared against the rules until a match is found and the corresponding action (accept or deny) is performed. Given increasing network speeds and traffic loads, it is important to inspect packets as quickly as possible.

This paper investigated rule ordering methods to improve the performance of network firewalls. Assuming each rule has a probability of a packet matching, firewall rules should be sorted such that the matching probabilities are non-increasing. This reduces the average number of comparisons and the delay across the firewall. However, a simple sort is not possible given precedence relations across rules. It is common in a security policy that two rules may match the same packet yet have different actions. It is this precedence relationship between rules that must be maintained to preserve integrity. This paper utilized Directed Acyclical Graphs (DAG) to represent the precedence order rules must maintain. Given this representation, a topological sort can be used to determine the optimal order (minimum average number of comparisons); however, this paper proved this is problem to be  $\mathcal{NP}$ -hard (similar to job scheduling for a single non-preemptive machine with precedence constraints). As an alternative, a simple sorting method was introduced that maintained the precedence order of the rules. Simulation results indicate this method can significantly reduce the average number of comparisons required and is comparable to optimal ordering.

Several areas exist for future research in optimizing firewall rule sets. The sorting method proposed in this paper is based on a simple algorithm. Although it can offer an improvement over the original rule order, algorithms that can move groups of rules could provide a larger reduction

in the average number of comparisons. The effect of stateful firewalls should also be addressed in future research. Security can be enhanced with connection state and packet audit information. For example, a table can be used to record the state of each connection, which is useful for preventing certain types of attacks (e.g., TCP SYN flood) [28, 29]. The impact of such rules on the firewall needs to be investigated and whether sorting can be done on-line to reflect temporal changes. In addition, more research is needed to determine more accurate probability distributions for packet matching and dependency percentages. Given this information, better algorithms can be designed and more realistic simulations can be performed.

## Acknowledgement

The author of this paper would like to thank Patrick Wheeler at The Department of Computer Science at U.C. Davis for his valuable review and input.

## References

- [1] E. Al-Shaer and H. Hamed. Firewall Policy Management Advisor for Anomaly Detection and Rule Editing. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, 2003.
- [2] E. Al-Shaer and H. Hamed. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.
- [3] S. M. Bellovin and W. Cheswick. Network Firewalls. *IEEE Communications Magazine*, pages 50–57, Sept. 1994.
- [4] C. Benecke. A Parallel Packet Screen for High Speed Networks. In *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999.
- [5] G. Brightwell and P. Winkler. Counting Linear Extensions is #P-Complete. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, 1991.
- [6] M. Christiansen and E. Fleury. Using Interval Decision Diagrams for Packet Filtering. Technical report, BRICS, 2002.
- [7] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of ACM SIGCOMM*, pages 4 – 13, 1997.
- [8] U. Ellermann and C. Benecke. Firewalls for ATM Networks. In *Proceedings of INFOSEC'COM*, 1998.
- [9] A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification. In *Proceedings of the IEEE INFOCOM*, pages 397 – 413, 2000.
- [10] E. W. Fulp. Firewall Architectures for High Speed Networks. Technical Report 20026, Wake Forest University Computer Science Department, 2002.
- [11] E. W. Fulp. Firewall Policy Models Using Ordered-Sets. Technical report, Wake Forest University Computer Science Department, 2004.
- [12] R. Funke, A. Grote, and H.-U. Heiss. Performance Evaluation of Firewalls in Gigabit-Networks. In *Proceedings of the Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 1999.

- [13] S. Goddard, R. Kieckhafer, and Y. Zhang. An Unavailability Analysis of Firewall Sandwich Configurations. In *Proceedings of the 6th IEEE Symposium on High Assurance Systems Engineering*, 2001.
- [14] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimizing and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287 – 326, 1979.
- [15] A. Hari, S. Suri, and G. Parulkar. Detecting and Resolving Packet Filter Conflicts. In *Proceedings of IEEE INFOCOM*, pages 1203–1212, 2000.
- [16] HiPAC. High Performance Packet Classification. <http://www.hipac.org>.
- [17] E. Horowitz, S. Sahni, and D. Mehta. *Fundamentals of Data Structures in C++*. Computer Science Press, 1995.
- [18] E. L. Lawler. Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints. *Annals of Discrete Mathematics*, 2:75 – 90, 1978.
- [19] 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.
- [20] J. K. Lenstra and A. H. G. R. Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22 – 35, 1978.
- [21] O. Paul and M. Laurent. A Full Bandwidth ATM Firewall. In *Proceedings of the 6th European Symposium on Research in Computer Security ESORICS'2000*, 2000.
- [22] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1999.
- [23] L. Qui, G. Varghese, and S. Suri. Fast Firewall Implementations for Software and Hardware-Based Routers. In *Proceedings of ACM SIGMETRICS*, June 2001.
- [24] V. P. Ranganath and D. 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.
- [25] R. Rivest. On Self-Organizing Sequential Search Heuristics. *Communications of the ACM*, 19(2), 1976.
- [26] W. E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, 3:59 – 66, 1956.
- [27] S. Suri and G. Varghese. Packet Filtering in High Speed Networks. In *Proceedings of the Symposium on Discrete Algorithms*, pages 969 – 970, 1999.
- [28] R. L. Ziegler. *Linux Firewalls*. New Riders, second edition, 2002.
- [29] E. D. Zwicky, S. Cooper, and D. B. Chapman. *Building Internet Firewalls*. O'Reilly, 2000.