Policy Distribution Methods for Function Parallel Firewalls

Michael R. Horvath
GreatWall Systems
Winston-Salem, NC 27101, USA

Errin W. Fulp
Department of Computer Science
Wake Forest University
Winston-Salem, NC 27109-7311, USA
fulp@wfu.edu

Patrick S. Wheeler
Department of Computer Science
University of California at Davis
Davis, CA 95616, USA

***Abstract*—Parallel firewalls offer a scalable low latency design for inspecting packets at high speeds. Typically consisting of an array of m firewalls, these systems filter arriving packets according to a security policy. Given the firewall array, the rules can be distributed in two fashions. Data parallel copies the entire policy to each firewall and distributes packets. In contrast, function parallel distributes the rules and duplicates packets.**

The function parallel design can provide significantly lower delays than an equivalent data parallel design, however performance is dependent on how the rules are distributed. Therefore, policy management is vital to the performance of the function parallel firewall system. This paper will describe the guidelines necessary to maintain policy integrity, which guarantees that a function parallel and a traditional firewall provide the same action for a packet. Based on these requirements, a policy can be divided into autonomous *chains* (sub-policies) that can be distributed across the firewall array. Although determining the optimal distribution will be shown to be $\mathcal{NP}$-hard, an effective algorithm will be described. Simulation results will indicate the distribution algorithm can provide an 86% reduction in the average processing delay as compared to previous distribution methods.

***Index Terms*—Security, firewalls, parallel, policy, management**

I. Introduction

Network firewalls must continually improve their performance to meet increasing network speeds, traffic volumes, and Quality of Service (QoS) demands. Unfortunately, firewalls often have more capabilities than standard networking devices, and as a result the performance of these security devices lags behind [1], [2], [3]. Furthermore, computer networks grow not only in speed, but also in size, resulting in convoluted security policies that take longer to apply to each packet [4], [5].

When a security solution cannot keep pace with the speed of incoming data, it either allows packets through without inspection or places incoming packets into a growing queue, thus becoming vulnerable to Denial of Service (DoS) attacks. With either of these possibilities, even a network with a *perfect* firewall policy (short in length and optimally ordered [6], [7]) is susceptible to attacks resulting in prolonged delays, data loss, or both, and it is for this reason that a new firewall architecture is necessary.

Parallel firewall designs provide a low latency solution, scalable to increasing network speeds [1], [8]. Unlike a traditional single firewall, the parallel design consists of an array of firewalls, each performing a portion of the work that a single firewall performed. As network speeds increase, the additional load is distributed across the array, providing a solution that can be implemented using standard hardware. There are two primary approaches to parallelization: data parallel and function parallel, as shown in figure 2.

In a data parallel design, the firewall policy is copied to each individual firewall in the array [1]. When a packet arrives, it is sent to one firewall for processing; *load balancing* is then necessary to prevent overloading one firewall in the array. This design offers increased throughput over a single firewall architecture, and it is also redundant, providing protection against device failure. Furthermore, the data parallel architecture has a relatively simple design since adding or removing a firewall does not require any changes to the other firewalls in the array. However, stateful inspection requires that all traffic from a certain connection or exchange to traverse the same firewall. Unfortunately, connection tracking is difficult to perform at high speeds [2], and the performance benefit only occurs under high traffic load when all firewalls are busy.

A function parallel design uses the same firewall array but distributes the rules, as seen in figure 2(b). Arriving packets are duplicated so that each firewall processes the same packet, but each firewall has fewer rules in its local policy [8], [9]. The function parallel design has several unique advantages. First, the function parallel design can result in faster processing since every firewall is utilized to process a single packet. Reducing the processing time, instead of the arrival rate (as with data parallel), yields better performance because each firewall in the array processes packets regardless of the traffic load. The processing delay can be reduced further with the addition of new firewalls. Second, unlike the data parallel design, the function parallel design can maintain state information about existing connections. The new state rule can be placed in any firewall since a packet will be processed by every firewall.

Of course the overall performance of a function parallel design is dependent on the policy and how it is distributed. For example, if a very small number of *popular* rules appear at the beginning of the policy, then a data parallel design is more effective. However, simulation results with realistic policies will show the function parallel design can provide lower processing delays. However as explained in this paper, the performance increase will depend on how rules are distributed.

An important constraint on any new firewall design is that the semantic integrity of the policy must be maintained; the packets that were accepted or rejected by the original policy must be acted upon in the same manner, without exception. When distributing rules from the original policy, their relative order must be preserved where there is an overlap in the regions covered. This can be accomplished using accept sets that describe the set of packets that will be accepted by a policy. The union of the accept set of each firewall must equal the accept set of the original policy (the array will accept the same packets as a single firewall), while the intersection must be empty (a legitimate packet is only accepted by one firewall). While maintaining integrity, the rules must be placed such that system performance is maximized. The sum of the expected number of rule comparisons is used to evaluate rule distributions, where a smaller value indicates lower latency.

To distribute rules across the array, the policy is first divided into rule chains (groups of intersecting rules) that can be distributed to any firewall in the array. Using the cost metric, rule chains are distributed to minimize the average number of comparisons. The problem of finding a distribution of rule chains with the minimum average number of comparisons is shown to be $\mathcal{NP}$-hard. However, a fast distribution method is described that orders the rule chains, merges the chains in each local policy, then sorts the local policies. Experimental results show this distribution method can provide better results than previous methods (over 86% improvement).

The remainder of this paper is structured as follows. Section II reviews firewall policies and attributes that are used for rule distribution for the function parallel firewall. The function parallel firewall design, rule distribution, and management are described in section III. Section IV will demonstrate the performance of the function parallel rule distribution method. Section V reviews the parallel firewall design and discusses some open questions.

II. Firewall Security Policies

As seen in figure 1(a), a security policy can be described as an ordered set (list) of n rules, denoted as $R = \{r_1, r_2, ..., r_n\}$. When a packet arrives, it is compared with the rules to determine the appropriate match. A packet matches a rule when every tuple of the packet is a subset of the corresponding rule tuple.

In terms of the Internet, a firewall rule commonly consists of 5-tuples: protocol type, source IP address, source port number, destination IP address, and destination port number [5]. Tuple values can be fully specified, given as a range, or contain wildcards `'*'` in standard prefix format. As a result, precedence constraints can exist between rules, which indicates the relative order between certain rules must be maintained [6]. In addition to the prefixes, each filter rule has an action, which is to accept or deny.

Typically, rules are sequentially compared starting with first rule until a match is found. This is referred to as a *first-match* policy and is generally the default behavior for the majority of firewall systems including the Linux firewall implementation `iptables` [10].

A. Policy Accept and Deny Sets

Given a firewall security policy, it is important to determine the packets that will be accepted, denied, or not match any rule. Assume a policy R exists, let A be the set of packets that will be accepted, let D be the set of packets that will be denied, and let U be the set of packets that do not match any rule. If the set of all possible packets is C, then a policy R is comprehensive if $U = \emptyset$ (i.e. $A \cup D = C$). Therefore, policy R is comprehensive if for every possible packet a match is found, which is an important objective. Furthermore, assume R does not necessarily equal R' in terms of the rules that comprise the policies.

There are many different ways to implement a given policy (e.g. using a single or parallel firewall) or even modify it (e.g. reorder, combine, add, or remove rules); therefore, it is important to determine equivalence and policy integrity. Consider two comprehensive policies R and R' that have accept sets A and A' respectively. The two policies are considered equivalent if $A = A'$. Therefore, if policy R is replaced by an equivalent policy R' then the **integrity** of R is maintained. Therefore, it is important to maintain the precedence constraints when implementing a firewall security policy.

B. Policy Optimization

Given multiple equivalent policies, it is important to determine which equivalent policy will provide the best firewall performance. Since the number of rule comparisons directly impact system performance, a metric that can be used is the average number of comparisons required to determine the first match [6]. Policies that have a lower average number of comparisons should have lower processing delay.

Minimizing the number of rule comparisons requires information not associated with a security policy. Given a policy, certain firewall rules have a higher probability of matching a packet than others. Using this information, it is possible to create a policy profile that indicates the probability of a first match for each rule in the policy. Let $P = \{p_1, p_2, ..., p_n\}$ be the policy profile, where p_i is the probability that a packet will first match rule i. Furthermore, assume a packet will always find a match; therefore R is comprehensive, $\sum_{i=1}^{n} p_i = 1$. Using this information, the average number of rule comparisons required for a single firewall implementing R is

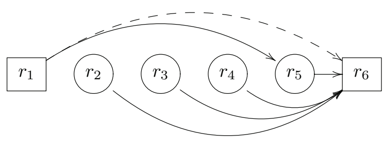
$$E(R) = \sum_{i=1}^{n} i \cdot p_i \qquad (1)$$

III. Policy Management for Function Parallel Firewalls

As described in the introduction, a function parallel system consists of an array of m firewalls connected to a packet duplicator [8], [9]. Each firewall j has a local policy R^j that is some portion of the rules of the original policy R. Therefore, each firewall has a local accept set A^j and deny set D^j. As a

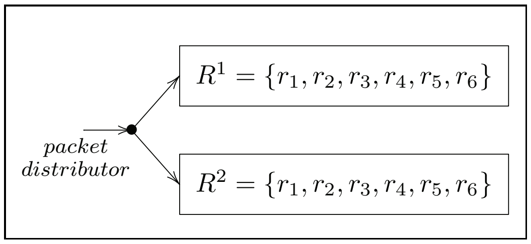
No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	UDP	190.1.1.*	*	*	80	deny	0.05
2	UDP	210.1.*	*	*	90	accept	0.10
3	TCP	180.*	*	180.*	90	accept	0.15
4	TCP	210.*	*	220.*	80	accept	0.20
5	UDP	190.*	*	*	*	accept	0.20
6	*	*	*	*	*	deny	0.30

(a) Example security policy consisting of multiple ordered rules.

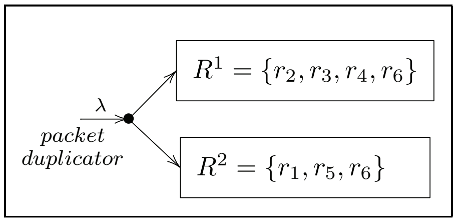


(b) Policy DAG, vertices are rules and edges are precedence requirements.

Fig. 1. Example firewall policy (ACL) and policy DAG.


$R^1 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$
packet distributor
$R^2 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$

(a) Data parallel design, packets distributed across two processors.



(b) Function parallel design, rules distributed across two processors.

Fig. 2. Two parallel designs for network firewalls. The original security policy consists of six rules $R = \{r_1, ..., r_6\}$ and each design consists of two firewalls (depicted as solid rectangles, where local policies are given within each rectangle).

packet arrives to the firewall system, it is duplicated and sent to the queue of each firewall in the array. Thus, each firewall will verify its own local policy against every packet that arrives to the network. A firewall performs the action associated with the rule in its local policy which matches the packet and then proceeds to the next packet in its queue. Each firewall in the array operates independently, therefore intercommunication is not required [8].

Given this design, the local policies of each firewall must be both comprehensive and disparate in terms of accept ranges. To elaborate, the sum of the ranges of every rule in a local policy must cover the entire range of possible packet header configurations. Also, there may be no overlap of accept ranges, those ranges which correspond to entrance to the network, between any two firewalls. Together, these two requirements assure that each firewall contains a rule which will match any given packet, and that no two firewalls will ever duplicate the same packet into the network. This guarantees integrity and can be more formally stated in the following theorem. [8].

Theorem 1: An array of m firewalls arranged in a function parallel fashion enforcing a comprehensive policy R can operate independently and maintain integrity if policy rules are distributed such that: each local policy is comprehensive, $\bigcup_{j=1}^{m} A^j = A$, and $\bigcap_{j=1}^{m} A^j = \emptyset$.

Proof: The first requirement, comprehensiveness, ensures each local policy will either accept or deny a packet ($\bigcup_{j=1}^{m} U^j = \emptyset$). The second requirement $\bigcup_{j=1}^{m} A^j = A$ indicates that collectively the system will accept only the packets accepted by the policy R. The last requirement, $\bigcap_{j=1}^{m} A^j = \emptyset$, ensures multiple firewalls will never accept the same packet (no overlaps in the local accept sets), therefore only one copy of a packet will be accepted. As such, the integrity of the policy R is maintained by the parallel firewall.

■

Several possible distributions may exist that meet requirements defined in Theorem 1. Consider the distribution of the policy given in Figure 1(a) across an array of two independent firewalls shown in Figure 2(b). In this case, the local policy of the upper firewall will only accept packets from the `210` and `180` address ranges, while the lower firewall will only accept packets from the `190` address range. Duplicating the deny all rule, r_6, is required to make the local policies comprehensive. Other distributions are possible, such as distributing rules based on the protocol ($R^1 = \{r_1, r_2, r_5, r_6\}$ and $R^2 = \{r_3, r_4, r_6\}$) or destination ports ($R^1 = \{r_1, r_4, r_5, r_6\}$ and $R^2 = \{r_2, r_3, r_6\}$). Given the number of possible distributions, it is important to determine which provides the best performance. In addition, the rule distribution must be done quickly since the policy profile (rule first match probabilities) will change over time.

A. Firewall Policy Distribution

The processor will inspect the arriving packet using the local policy in a top-down fashion, starting with the first rule, to find the first match. The local policy must maintain integrity as described by the original policy-DAG. For example, if a precedence edge exists between two rules, and both are assigned to the same processor, then their relative order in the original policy must be maintained. This order prevents shadowing in the local policy. In addition, rules must be distributed so that only one processor will accept a legal packet, and the remaining processors will deny it.

Given a firewall policy, several rule distributions may be able to maintain integrity. Identifying the sets of rules that form independent accept sets, called *rule chains*, can help maintain integrity when rules are distributed. A *rule chain* is the smallest ordered list of intersecting rules that forms an accept set which does not intersect with another rule chain.

A rule chain can be found by starting with a rule in R that does not have any preceding constraints (no incoming preceding edges in the policy DAG) and then following the precedence edges until a rule is encountered that has no successive precedence constraints (no outgoing precedence edges). All the rules along this path belong to a rule chain. Note that an accept rule can only belong to one rule chain. Therefore, if two chains share an accept rule, then they are considered one chain. In contrast, deny rules can be duplicated across multiple chains. For example, rule r_6, given in figure 1(a), would be the last rule in each rule chain. Once all the rules have been associated with a chain, all possible rule chains have been found for the policy. For example, the rule chains for the policy given in figure 1(a) are $c_1 = \{r_1, r_5, r_6\}$, $c_2 = \{r_2, r_6\}$, $c_3 = \{r_3, r_6\}$, and $c_4 = \{r_4, r_6\}$.

Once the rule chains have been determined, they can be distributed to the firewall nodes in the array. When multiple chains are given to a node, the rules that belong to the chains must be merged to form the local policy. Merging requires rules to adhere to the precedence constraints specified by the original policy DAG (as in rule sorting) [6]. For example, merging c_2, c_3, and c_4 requires placing r_6 at the end. Once the chains have been assigned to a local policy, the rules can be sorted in accordance to precedence constraints to reduce the number of comparisons.

Distributing chains and merging rules will maintain policy integrity since the accept sets for the chains are independent (first condition), and every accept rule in the original policy exists in only one chain (second condition). Furthermore, merging the rules assures shadowing will not occur in the local policies (assuming shadowing does not occur in the original policy). This approach is more formally stated in the following theorem.

Theorem 2: Distributing and merging the rule chains of a policy R across function parallel firewall will maintain policy integrity.

Proof: Assume d chains are found in the policy R and let A^k represent the accept set of the k^{th} chain. Each rule chain represents an independent accept set since all intersecting rules will always belong to the same chain and accept rules only belong to one chain, therefore the intersection requirement is met ($\bigcap_{k=1}^{d} A^k = \emptyset$). For the second requirement, $\bigcup_{k=1}^{d} A^k = A$, consider a packet that is accepted by rule r_i in the policy R, therefore r_i is the first match. The rule r_i would also be the first match in the chain that contains r_i regardless which processor r_i resides since all intersecting rules will belong to the same chain (any subsequent matching rules would appear after r_i) and merging prevents shadowing. In addition, since every rule must belong to a chain, the second condition is satisfied.

■

B. Rule Distribution Performance

A rule distribution is sought that minimizes the number of comparisons required to determine an accept.Given a comprehensive policy R and an array of m firewalls, each firewall j in the array will implement a local-policy R^j which consists of n^j rules where r_i^j is the i^{th} rule in the local-policy. In addition, let p_i^j is the probability of the i^{th} rule in local-policy j being the first match.

To determine the average number of comparisons for a given packet, assume each firewall in the array requires one *time-unit* to compare the packet to a rule. Then assume the firewalls are initially empty and synchronized so that each starts processing a packet at the same time. When the first packet arrives, it is compared to the first rule in each of the m local policies. Therefore, after the first time-unit, the packet has been compared to m rules. The probability that the *original* first match (as defined by the R) is found in the first time-unit is the sum of the probabilities of the first rule in each local policy. Similarly, the probability the first-match occurs in two time-units is equal to the sum of the probabilities of the second rule in each local policy. The expected number of rule comparisons required to find the original first match in a function parallel firewall can be computed as

$$\sum_{i=1}^{\max(n^j)} i \cdot \sum_{j=1}^{m} p_i^j = \sum_{j=1}^{m} \sum_{i=1}^{n^j} i \cdot p_i^j, = \sum_{j=1}^{m} E(R^j) \qquad (2)$$

However, each rule in R is considered only once in the calculation, since only the average number of comparisons required for a first-match is considered. If a rule is duplicated (such as r_6 in the distribution given in figure 2(b)), then it is only considered once in the calculation at its earliest occurrence within the local policies. As a result of only considering each rule once, the sum of the probabilities across the local policies should equal one, $\sum_{j=1}^{m} \sum_{i=1}^{n^j} p_i^j = 1$.

For example, the expected number of comparisons required to find the original first match for the system given in Figure 2(b) is

$$1 \cdot (p_1^1 + p_1^2) + 2 \cdot (p_2^1 + p_2^2) + 3 \cdot (p_3^1 + p_3^2) =$$
$$1 \cdot (p_2 + p_1) + 2 \cdot (p_3 + p_5) + 3 \cdot (p_4 + p_6) = 2.35$$

Note that rule r_6 is duplicated in the distribution to maintain integrity. It is the third rule in R^1 and the fourth rule in R^2; however, the rule is only considered once in the calculation above.

The overall performance of the firewall is improved by balancing the number of rules on each node as well as placing rule chains so that high probability rules are located near the beginning of each local policy. Balancing the number of rules prevents firewalls in the array from becoming a bottleneck, specifically those with longer local policies. For example, consider two firewalls configured in a function parallel fashion.

Let L be the set of rule chains determined from a policy with n rules. Furthermore assume the first match probability for each rule p_i in the policy is equal to $\frac{1}{n}$. In this case equation 2 becomes $\frac{1}{n}\sum_{j=1}^{2}\sum_{i=1}^{n^j} i = \frac{1}{n}\sum_{j=1}^{2}\frac{n^j(n^j+1)}{n^j}$. The optimal number of rules per local policy is given by taking the derivative of this equation which yields $n^j = \frac{n}{2}$.

Therefore given a set of rule chains our objective is to find a partitioning of the rule chains into m subsets (local policies) such that $\max(n^i, n^j)$ is minimized for all $i, j <= m$, where n^i is the total number of rules in local policy R^i. Unfortunately this objective is equivalent to the k-partitioning problem, which is known to be $\mathcal{NP}$-complete [11].

Definition: (k-partitioning). Partition a given set of positive integer numbers into k subsets such that the sums of the elements in each subset are equal.

An algorithm which can find an optimal distribution of rules when all probabilities are the same can then find a solution to the k-partitioning problem. Therefore determining the optimal rule distribution for a function parallel firewall is $\mathcal{NP}$-hard.

Theorem 3: k-partitioning $\propto$ Determining the optimal policy distribution for function parallel firewall

Proof: Recall that for k-partition the problem is to decide whether a given multiset of integers S can be partitioned into k subsets that have the same sum. Given input $\langle S, k\rangle$ for the k-partition problem, perform the following polynomial-time transformation. For each integer l in set S create a distinct rule chain of length l. We now have a set of rule chains (call it L) where each individual rule chain represents an integer in the original multiset S. Now simply solve the function parallel rule distribution on input $\langle L, k\rangle$, where $k = m$ the number of firewalls in the array. The solution to function parallel rule distribution returns k subsets where $max(n^i, n^j)$ is minimized for all $i, j \leq k$ – call this value $maxDifference$. To determine if there is a solution to the k-partition problem we must simply determine if the value of $maxDifference$ is zero. We may perform this check in polynomial time in the following manner:

```
Let $high = 0$ and $low = \infty$ // highest and lowest positive sum
**for** $(1 \leq i \leq k)$
    **if** $n^i < low$, **then** $low = n^i$
    **if** $n^i > high$, **then** $high = n^i$
**end**
**if**$(high - low == 0)$ // all subsets sum to same value
    convert each rule chain back to its corresponding integer value
    return the $k$ subsets of $S$ that comprise the solution to
    $k$-partition
**else**
    No Solution
**end**
```

■

C. Sorted Horizontal Rule Distribution Algorithm

A fast and efficient distribution algorithm is needed since determining the optimal rule distribution may be problematic. The rule distribution algorithm proposed in this paper first determines the set of rule chains L for the policy R. Once L has been determined, the algorithm sorts the rule chains according to the average number of comparisons per rule chain. Note the chains can appear in any order since they are independent (non-overlapping accept sets). For example consider the policy in figure 1(a) the ordered set of chains would be $\{c_2, c_3, c_4, c_1\}$ since $\{E(c_2) < E(c_3) < E(c_4) < E(c_1)\}$. Using the sorted list, the chains are distributed and merged across the processors in a *horizontal* fashion. For example if the firewalls in the array are sequentially numbered starting with 0, then the k^{th} ordered chain, c_k, is assigned to the $k \bmod m$ processor. The horizontal distribution attempts to evenly distribute the rules across the local policies (primary objective). In addition sorting ensures chains with the smallest average number of comparisons are distributed first (placed near the end of the local policies). Once the local policies are determined, the rules are sorted again (while maintaining dependencies) to further reduce the number of comparisons.

For the function parallel system shown in figure 2(b) and the policy given in figure 1(a), the distribution using this method would be $R^1 = \{c_3, c_1\}$ and $R^2 = \{c_2, c_4\}$ which is equivalent to $R^1 = \{r_1, r_5, r_3, r_6\}$ and $R^2 = \{r_4, r_2, r_6\}$. The average number of comparisons for the first match is 2.25, using this distribution.

IV. Experimental Results

The performance of the proposed distribution method for function parallel firewalls (section III-C) was measured under realistic conditions using simulation. A comparison is made between the *sorted horizontal rule distribution algorithm* presented in this paper and a simple distribution method where rule chains are distributed horizontally to firewalls in a round robin fashion. In addition, simulation results will show the performance of a comparable data parallel firewall.

For each experiment the parallel designs always consisted of four firewalls. Packets lengths were uniformly distributed between 40 and 1500 bytes, while all legal IP addresses were equally probable. 1024 firewall rules were generated such that accepted packets are 6 times more likely than denied packets, which has been shown to occur in actual firewall policies [7].

Figure 3 shows the average packet delay of the parallel firewalls and the rule distribution methods. As seen in the graph, if the arrival rate is low the function parallel designs typically perform better than the equivalent data parallel configuration. This is because in the function parallel system all the firewalls are used to process packets regardless of arrival rate (none are idle). When arrival rate is low it is possible to have firewalls idle in the data parallel design.

At high arrival rates, the data parallel design has a lower delay than a function parallel system using a horizontal distribution. This is expected since data parallel design is more capable of managing high data rates due to dividing the arriving stream of packets. If the chains are sorted based on the average number of comparisons prior to distribution, the function parallel design performs better than the data parallel system. The percent improvement ranges from 25% to 86% better than a simple horizontal distribution. This is because sorting chains moves rules (or chains of rules) that more more likely to match a packet closer to the beginning of the local

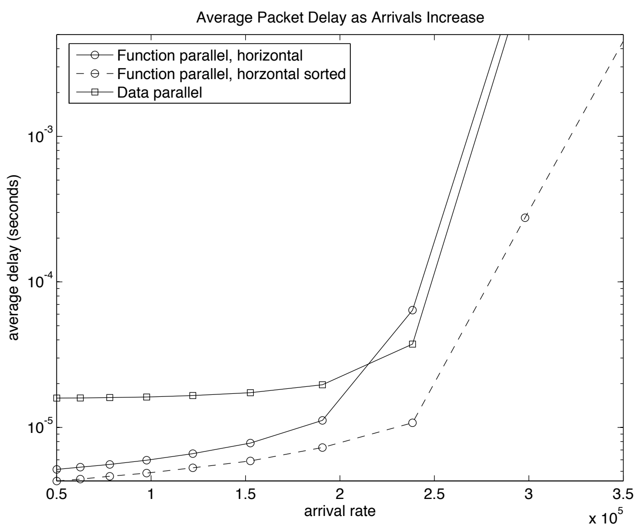


Fig. 3. Average packet delay as the arrival rate increases for different parallel firewall designs and rule distributions. Parallel array consisted of four firewalls. The proposed horizontal sorted rule distribution for the function parallel design consistently performs the best.

policies. Therefore the distribution algorithm presented in this paper is a effective method for improving function parallel performance.

V. Conclusions

Functional parallelism is a scalable solution for inspecting packets in a high-speed environment. However, the system performance is dependent on how the rules are distributed.

This paper described guidelines for function parallel policy management. Integrity is maintained if firewall rules are distributed based on the security policy *accept set*, which describes the set of packets that will be accepted. Distribution must be performed such that the union of each local accept set equals the original accept set (original and distributed policies accept the same packets), while the intersection of the local accept sets is the empty set (a packet will be accepted by only one firewall). Given a policy, many different distributions that maintain integrity are possible. The expected number of comparisons for a first-match was developed to compare possible distributions. A distribution algorithm was then described that ordered rule chains (intersecting sets of rules). Experimental results show the proposed sorted horizontal distribution method can provide significant performance improvements (upwards of 86%) as compared to a simple round robin distribution.

While the function parallel firewall architecture is very promising, several open questions exist with regards to policy distribution. For example, given the need for QoS in future networks, it is important to develop methods for distributing rules such that traffic flows are isolated. In this case a certain type of traffic would be processed by a certain firewall. Another open question is the optimization of local firewall policies, including redundant policies. However, optimization can only occur if the integrity of the policy is maintained. Finally the ability to dynamically manage changing policies, in terms of match probabilities and rule additions/deletions, should be considered. Given these policy dynamics fast algorithms that provide simple updates, instead of complete redistributions, are needed.

Acknowledgment

The authors would like to thank Wake Forest University and GreatWall Systems, Inc. for their support. This research was funded by GreatWall Systems, Inc. via United States Department of Energy STTR grant DE-FG02-06ER86274. The views and conclusions contained herein are those of the author(s) 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.

References

[1] C. Benecke, “A parallel packet screen for high speed networks,” in *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999.
[2] 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.
[3] E. D. Zwicky, S. Cooper, and D. B. Chapman, *Building Internet Firewalls*. O'Reilly, 2000.
[4] A. Wool, “A quantitative study of firewall configuration errors,” *IEEE Computer*, vol. 37, no. 6, pp. 62 –67, June 2004.
[5] R. L. Ziegler, *Linux Firewalls*, 2nd ed. New Riders, 2002.
[6] E. W. Fulp, “Optimization of network firewall policies using directed acyclical graphs,” in *Proceedings of the IEEE Internet Management Conference (IM'05)*, 2005.
[7] S. Acharya, J. Wang, Z. Ge, and T. F. Znati, “Traffic-aware firewall optimization strategies,” in *Proceedings of the IEEE International Conference on Communications*, 2006.
[8] E. W. Fulp and R. J. Farley, “A function-parallel architecture for high-speed firewalls,” in *Proceedings of the IEEE International Conference on Communications*, 2006.
[9] E. W. Fulp, “An independent function-parallel firewall architecture for high-speed networks (short paper),” in *Proceedings of the International Conference on Information and Communications Security*, 2006.
[10] 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*, 2003, pp. 889–894.
[11] M. R. Garrey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP completeness*. W.H.Freeman and Company, 1979.