DOE/ER/25581-1

Firewall Architectures for High-Speed Networks

Final Report

August 2007

Errin W. Fulp

Wake Forest University
Department of Computer Science
Winston-Salem, NC 27109

Contents

DOE/ER/25581-1

Firewall Architectures for High-Speed Networks

Final Report

Date Issued/Published August 2007

Errin W. Fulp

Prepared for the United States Department of Energy Office of Science

Work Performed Under Grant Number DE-FG02-03ER25581

Project Summary

Firewalls are a key component for securing networks that are vital to government agencies and private industry. They enforce a security policy by inspecting and filtering traffic arriving or departing from a secure network [5, 40, 41]. While performing these critical security operations, firewalls must act transparent to legitimate users, with little or no effect on the perceived network performance (QoS). Packets must be inspected and compared against increasingly complex rule sets and tables, which is a time-consuming process. As a result, current firewall systems can introduce significant delays and are unable to maintain QoS guarantees. Furthermore, firewalls are susceptible to Denial of Service (DoS) attacks that merely overload/saturate the firewall with illegitimate traffic [20, 27, 35, 39, 40]. Current firewall technology only offers a short-term solution that is not scalable; therefore, the **objective of this DOE project was to develop new firewall optimization techniques and architectures** that meet these important challenges.

Firewall optimization concerns decreasing the number of comparisons required per packet, which reduces processing time and delay. This is done by reorganizing policy rules via special sorting techniques that maintain the original policy integrity. This research is important since it applies to current and future firewall systems. Another method for increasing firewall performance is with new firewall designs. The architectures under investigation consist of multiple firewalls that collectively enforce a security policy. Our innovative distributed systems quickly divide traffic across different levels based on perceived threat, allowing traffic to be processed in parallel (beyond current *firewall sandwich* technology). Traffic deemed safe is transmitted to the secure network, while remaining traffic is forwarded to lower levels for further examination. The result of this divide-and-conquer strategy is lower delays for legitimate traffic, higher throughput, and traffic differentiation (a key component for maintaining QoS). Furthermore, the distributed design is scalable to traffic loads and is less susceptible to DoS attacks. Simulation and analytical results show these new architectures out-perform any current firewall system, providing higher throughput, lower delays, and predictable traffic differentiation.

1 List of Accomplishments

This project investigated several new and important research questions in network security. Areas of interest included firewall policy optimization, parallel firewall system design, and policy distribution methods. Given these diverse areas, the results of this research project can benefit current and future firewall systems. Relevant publications from this project are cited after each area.

- **Security Policy Models** - Firewall policy representations are an integral part of this project. Models must represent the policy integrity, while providing a means for optimization. New models were developed by this project that utilize ordered sets, tries, and directed graphs to achieve these objectives [18, 13].

- **Security Policy Optimization** - The order of firewall rules significantly impacts the security (integrity) and performance (processing time) of a security policy. Using the policy models developed by this project, we have developed algorithms that can optimize firewall security policies (reduce the number of comparisons required) [36, 13].

- **Parallel Firewall Designs** - Parallel firewall systems consist of an array of firewalls that provide a scalable solution for securing high-speed networks. Two new *function parallel* designs were developed by this project and have shown significant performance improvements. For example, simulation results and analytical models show an m-times reduction in processing delay is possible using the new function parallel designs as compared to current data parallel designs [12, 14, 15].

- **Security Policy Distribution** - Given an array of firewalls, rules must be distributed such that the integrity of the original policy is maintained (parallel firewall and single firewall arrive at the same decision for the same packet). Using the policy Directed Acyclical Graph (DAG) model developed by this project, we have established certain criteria that must be followed to maintain integrity in parallel-firewall architectures [16].

- **Parallel Intrusion Detection Systems** - Many of parallel designs can be applied to Intrusion Dection Systems (IDS), thereby reducing the latency associated with payload packet inspections. This project conducted an initial survey of parallel IDS techniques [37].

- **Open-Source Implementation and Synergistic Activities** - We have implemented the function parallel firewall architecture using Linux PC's that can provide a low-cost, scalable, high-speed firewall system. This work has also resulted in two provisional patents for high-speed security devices, the creation of a company GreatWall Systems (co-founded by the PI), and the funding of Phase I and II STTR proposals (GreatWall Systems and Wake Forest University). In addition a collaborative effort has started between this project and DOE Pacific Northwest National Laboratories (PNNL) that will integrate high-speed security techniques into their infrastructure.

1.1 Students Supported

This research project supported the following students, each associated with the Computer Science Department at Wake Forest University.

- Stephen Tarsa, undergraduate researcher, results published in the 2005 IEEE IM [13] and IEEE ISCC [18, 36]. Research has resulted in a patent pending.
- Ryan J. Farley, MS thesis: "Function-Parallel Firewalls for High-Speed Networks," completed in 2006. Results published in IEEE ICC 2006 [15], IASTED Parallel Computing and Networking Conference [12] completed in 2005. Research has resulted in a patent pending.
- Micheal Horvath, MS thesis "Policy Management Methods for Function Parallel Firewalls," completed in 2007. Results published in the SPIE High Capacity Optical Networks and Enabling Technologies [16].

1.2 Publications and Invited Presentations

This research project has yielded multiple publications and presentations at various ACM/IEEE conferences and workshops. The focus of these publications include: policy optimization/managment (a new area discovered by this project), parallel firewalls, and parallel IDS.

Refereed publications

- Errin Fulp and Patrick Wheeler. A Taxonomy of Parallel Techniques for Intrusion Detection. In *Proceedings of the ACMSE, Special Session on Computer and Network Security*, 2007.
- Errin 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.
- Errin W. Fulp, Micheal R. Horvath, and Christopher C. Kopek. Managing Security Policies for High-Speed Function Parallel Firewalls. In *Proceedings of the SPIE International Symposium on High Capacity Optical Networks and Enabling Technology*, 2006.
- Errin W. Fulp. Parallel Firewall Designs for High-Speed Firewalls. In *Proceedings of the IEEE INFOCOM, High-Speed Networking Workshop*, 2006.
- Stephen J. Tarsa and Errin W. Fulp. Balancing Trie-Based Policy Representations for Network Firewalls. In *Proceedings of the IEEE International Symposium on Computer Communications (ISCC'06)*, 2006.
- Errin W. Fulp and Ryan J. Farley. A Function-Parallel Architecture for High-Speed Firewalls. In *Proceedings of the IEEE International Conference on Communications*, 2006.

- Ryan J. Farley and Errin W. Fulp. Effects of Processing Delay on Function-Parallel Network Firewalls. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, 2006.

- Errin W. Fulp and Stephen J. Tarsa. Trie-Based Policy Representations for Network Firewalls. In *Proceedings of the IEEE International Symposium on Computer Communications (ISCC'05)*, 2005.

- Errin W. Fulp. Optimization of Network Firewall Policies Using Directed Acyclical Graphs. In *Proceedings of the IEEE Internet Management Conference (IM'05)*, 2005.

Invited presentations

- Errin W. Fulp. Improving the Performance of Firewalls and Intrusion Protection Systems for High-Speed Networks, 2006. Department of Computer Science Seminar, North Carolina State University.

- Errin W. Fulp. Techniques for Improving the Performance of Signature-Based Network IDS, 2006. DOE Pacific Northwest National Laboratory, Cyber Security Group Seminar.

- Errin W. Fulp. Security Issues for the Next Generation of Networks, 2005. Department of Computer Science Seminar, Old Dominion University.

- Errin W. Fulp. Advances in Firewall Architectures for High-Speed Networks, 2005. DOE High-Performance Network Research Meeting, DOE Brookhaven National Laboratory.

- Errin W. Fulp. Firewall and Intrusion Detection Systems for High-Speed Networks, 2005. DOE Pacific Northwest National Laboratory, Cyber Security Group Seminar.

- Errin W. Fulp. Firewall Optimization Techniques and Architectures for High-Speed Networks, 2004. DOE Pacific Northwest National Laboratory, Cyber Security Group Seminar.

- Errin W. Fulp. Firewall Architectures for High-Speed Networks, 2004. DOE High-Performance Network Research Meeting, DOE Fermi National Accelerator Laboratory.

1.3 Synergistic Activities

This project has started several important synergistic activities, these includes a new company and collaborations with several national research laboratories. In collaboration with the Wake Forest University Office of Technology Management, the following two preliminary patents filled for policy optimization and parallel firewall architectures.

- "Computer Network with Function-Parallel Firewall," Errin W. Fulp and Ryan J. Farley. U.S. Patent Pending No. 60/638,436 May 2005

- "Methods and Systems for Firewall Policy Optimization," Errin W. Fulp and Stephen J. Tarsa. U.S. Patent Pending No. 60/655,664 May 2005

Based on promising experimental results, the Office of Technology Asset Management has started a business, GreatWall Systems located in Winston-Salem NC. The business will offer computer/network security solutions that are based on this research. To date, GreatWall Systems has receive both public (DOE STTR Phase I and II) and private funding.

Other synergistic activities started during this research project include the following.

- Founded the Network Security Group(`nsg.cs.wfu.edu`) at Wake Forest University, an inter-campus collaboration to discuss current security issues.
- Program Committee for the IEEE INFOCOM High-Speed Networks Workshop, 2007.
- Established the *Research and Education Collaboration Initiative* between Wake Forest University and PNNL Cyber Security Group that supports security research and education.
- Program Committee and Local Arrangements Chair for the 2006 Eighth International Conference on Information and Communications Security.
- Co-Chair for the Special Session on Computer Security, ACMSE, 2007.

2 Project Motivation

Network firewalls remain the forefront defense for most computer systems. Guided by a security policy, these devices provide access control, auditing, and traffic control [5, 40, 41]. As seen in table 1, a security policy is a set of ordered rules that define the action to perform on matching packets. Given the packet and/or connection information, rules indicate the action to take place for each packet, such as discard, forward, or redirect. Security can be further enhanced with connection state 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) [41].

Traditional firewall implementations consist of a single dedicated machine, similar to a router, that sequentially applies the rule set to each arriving packet. However, packet filtering can represent a significantly higher processing load than routing decisions [29, 35, 41]. For example, a firewall that interconnects two 100 Mbps networks would have to process over 300,000 packets per second [40]. Successfully handling this high traffic becomes more difficult as rule sets become more complex [6, 27, 41]. 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 [6, 11, 19, 20]. An attacker could simply inundate the firewall with traffic, delaying or preventing legitimate packets from being processed. Therefore, new network firewall solutions are necessary to manage these security threats.

This project investigated two methods for improving the performance of network firewalls. The first method involves the representation and rule reorganization of the firewall policy.

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
1	TCP	140.*	*	130.*	80	deny	0.1
2	TCP	140.*	*	*	80	accept	0.05
3	TCP	150.*	*	120.*	90	accept	0.15
4	UDP	150.*	*	*	3030	accept	0.3
5	*	*	*	*	*	deny	0.4

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

These techniques seek to reduce the number of operations required to determine the correct action for an arriving packet. Although these techniques can improve the performance and is applicable to current firewalls, the performance improvement is limited. The second approach uses parallel firewalls to provide scalable performance improvements. Both approaches and the associated research findings are described in the following sections.

3 Security Policy Models

A firewall rule set, also known as a firewall policy or Access Control List (ACL), is traditionally an ordered list of firewall rules. Firewall policy models have been the subject of recent research [2, 3, 22]; however, the primary purpose is anomaly detection and policy verification. In contrast, the policy model developed by this project was 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.

3.1 Firewall Rule and Policy Models

A firewall rule r can be modeled as an ordered tuple of sets, $r = (r[1], r[2], ..., r[k])$. 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, specify a range, 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 [40, 41]. 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 $d = (d[1], d[2], ..., d[k])$; however, ranges and 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 $R = \{r_1, r_2, ..., r_n\}$. 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. 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` [30]. For this project we assume a first-match evaluation is always used. 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

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

3.1.1 Policy Accept and Deny Sets

Given a firewall security policy and a first-match evaluation, it is important to determine the packets that will be accepted, denied, or not match any rule. Assume a policy R exists and first-match evaluation is used. 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.

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.

3.2 Modeling Rule List Precedence Relationships

Because of the first-match evaluation, 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_2, likewise rule r_5 must be the last rule in the policy. If for example, rule r_2 was moved to the beginning of the policy, then it will *shadow* [3] the original rule r_1. However, there is no precedence relationship between rules r_2 and r_3 given in table 1. Therefore, the relative ordering of these two rules will not impact the policy integrity and can be changed to improve performance.

Performance refers to the number of rule comparisons required to find the first match for a given policy and packet. We will assume the original policy is free from any anomalies. 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.

This project developed a new model for the rule precedence relationships using a Directed Acyclical Graph (DAG) [28, 23]. 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 policy 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 $i < j$, the actions for each rule are different, and the rules intersect.

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

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

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.

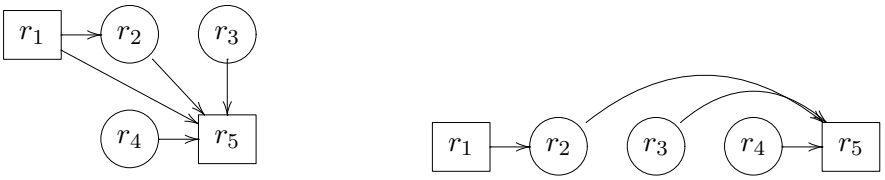
For example consider the rules given in table 1, the intersection of r_1 and r_2 yields `(TCP, 140.*, *, 130.*, 80)`. Again, the rule actions are not considered in the intersection or match operation. Since these two rules intersect, a packet can match both rules for example $d =$ `(TCP, 140.1.1.1, 80, 130.1.1.1, 80)`. Furthermore, the actions of the two rules are different. Therefore, the relative order must be maintained between these two rules and an edge drawn from r_1 to r_2 must be present in the policy DAG, as seen in figure 1(a). In contrast consider the intersection of rules r_1 and r_5. These two rules intersect, indicating packets belonging to the set `(TCP, 140.*, *, 130.*, 80)` would match both rules. However, it does not matter which of the two rules a packet matches first, since the action is the same for both rules. Therefore, an edge does not exist between rules r_1 and r_5 in the diagram. Similarly, rules r_2 and r_3 do not intersect due to the second tuple (source IP address). A packet cannot match both rules indicating the relative order can change; therefore, an edge will not exist between them.

The match operation can be used to identify precedence relationships, but it cannot do so in every case. Consider a *partial-match* example [2], where $r_a =$ `(UDP, *, 80, 10.*, 90, accept)` and $r_b =$ `(UDP, 10.*, 80, *, 90, deny)`. The intersection of r_a and r_b is `(UDP, 10.*, 80, 10.*, 90)`; therefore a packet, such as $d =$ `(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 [28]. 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 3.1 that any linear arrangement of a policy DAG maintains integrity.

Theorem 3.1 *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



(a) Policy DAG.

(b) Linear arrangement corresponding to the original rule order.

Figure 1: Rule list DAG representations of the firewall rules given in table 1. Vertices are rules (circle is an accept rule and square is a deny rule) while edges indicate precedence requirements.

r_i and r_j in G does not exist, then a linear arrangement of G can interchange the order of the two rules. An edge will not exist if the rules do not intersect; however, a reorder will not effect integrity since a packet cannot match both rules. Shadowing is not introduced due to the reorder since the intersection operation is symmetric. An edge will not exist if the two rules intersect but have the same action; however, a reorder will not effect integrity since the same action will occur regardless of which rule is matched first. 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. ∎

3.3 Trie-Based Policy Representation

The previous sections described a model that represents the firewall policy in a list form, which can be implemented using a simple list data structure. However, non-linear data structures can also be used to represent the firewall policy that can provide several performance benefits.

This project developed a new security policy representation called the *policy trie*, which provides faster processing of packets while maintaining the integrity of the original policy. The policy trie T is a n-ary trie structure consisting of k levels that stores a security policy. Each level $T[l]$ corresponds to a rule tuple (except for the root), while nodes on a certain level store the tuple values $T[l, v]$. Unlike the standard binary trie structure [1], the policy trie is unique since a node can have multiple children, similar to an n-ary tree. This is required since a node will store a rule tuple (multibit field), not just a single bit as done in [34]. Tuples at each level are organized from specific to general (reading left to right). For reference, levels will be numbered sequentially starting with zero for the root node. Likewise, nodes of a particular level will be numbered sequentially starting with zero for the left-most node. Since each level stores a tuple, a path from the root node to a leaf represents a firewall rule, as seen in figure 2.

To create a policy trie T, rules are added in the order they appear in R. A rule r is added to T by starting with the root node on the first level and comparing the values of its children with the corresponding tuple of r. If one of the children is equal (not just a subset)

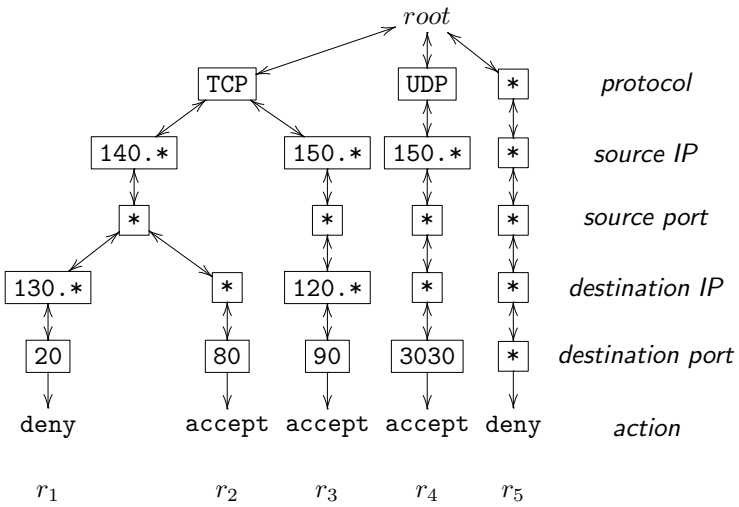


Figure 2: Policy trie representation of the firewall rules given in table 1.

to the corresponding rule tuple, then r will share this node and a new node is not added for this level. The trie is traversed to that node and the process of comparing children to the rule is repeated using the next tuple in r. If an exact match does not exist, a new child node is created that contains the value of the corresponding packet tuple. In order to maintain the specific-to-general organization of the trie, the new node is inserted in the rightmost available position such that it is before (to the left of) any sibling that is a superset of the new node. The new node forms a chain of nodes that stores the remaining tuple values of the rule.

Consider the events that occur when rule r_2 is added to the trie given in figure 2. Rule r_2 has the same protocol, IP source, and source port values as r_1; therefore, r_2 will share these nodes. Since the destination IP is different, this forms a chain consisting of this tuple, the destination port, and action. This chain connects to the source port node, which adds r_2 to the trie. It is this structure that allows the elimination of multiple rules simultaneously. For example if a UDP packet is compared using the trie given in figure 2, then rules r_1, r_2, and r_3 are eliminated after the protocol tuple comparison.

Once the new rule is added, if any nodes exist to the right of the new rule, then this represents a rule re-order and may result in a shadowing. The intersection of the new rule and each of these right-most rules is taken and the action of right rule, the rule that appears first in the ordered policy, is applied.

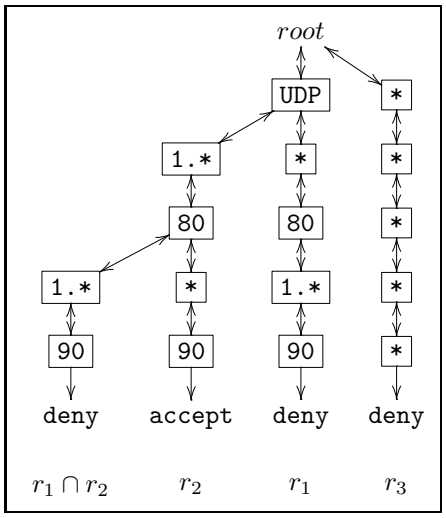
Definition The intersection of rule r_i and r_j, denoted as $r_i \cap r_j$ is

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

For all intersections that yield valid rules, the results form a subtree of the newly added rule. The same method is applied to this subtree. For example consider the rules given in figure 3(a). Note that the relative order of the rules must be preserved; otherwise integrity is not maintained. When r_2 is added to the policy trie, the source IP address will cause the rule to be placed before r_1 and the intersection must be taken. The intersection of r_1 and r_2 is `(UDP, 1.*, 80, 1.*, 90)`. The result of the intersection indicates a packet can

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action
1	UDP	*	80	1.*	90	deny
2	UDP	1.*	80	*	90	accept
3	*	*	*	*	*	deny

(a) Example rule list, where the rules must maintain their relative order.



(b) Policy trie that requires intersection of r_1 and r_2.

Figure 3: List and trie representation of a security policy where the policy trie requires the intersection operation to maintain the integrity of the list.

match both rules, for example d = (UDP, 1.1.1.1, 80, 1.1.1.1, 90). Therefore this intersection rule, with the action of r_1, must be added to the trie. The final policy trie is given in figure 3(b), which maintains the integrity of the policy given in 3(a). When r_3 is added, it is located to the right of r_1 and r_2, thus intersections are not performed. In this example rules r_1 and r_2 are considered a *partial-match* [3]. The DAG policy representation described in [10] will not correctly represent partial-match rules, because subsets (as done with the match operation) are used to create the structure instead of intersections. As a result, the DAG structure described in [10] is **not** suitable for firewall policies since integrity cannot be maintained.

To process a packet d using the policy trie T (also referred to as searching T), the corresponding tuple of the packet is compared with the children of the root node. Comparisons of nodes are always performed from left and right, or specific to general. Once a match is found, the current node is marked and the trie is traversed to the matching child. The procedure is repeated with the remaining rule tuples. If no match is found, the search backtracks to the parent node and finds the next matching node that has not been visited, continuing the process of left to right comparison. Once a path has been found from the root node to a leaf where all the rule tuples match $(p[l] \subseteq T[l,i], l = 1, ..., k)$ the associated action is performed.

3.3.1 Policy Trie Integrity

As previously stated, a necessary objective of any policy representation is its ability to maintain the policy integrity. This occurs if the new representation is equivalent to the original list-based policy. A policy trie T is equivalent to the original security policy R for

any legal packet d, if searches of T and R result in the same action being performed. Unlike other representations, this is proven true in the following theorem.

Theorem 3.2 *A policy trie T is equivalent to the original security policy R.*

Proof Assume a trie T is constructed with k levels using the process previously described from an n rule security policy (ordered list). Furthermore assume the completed trie is searched using the previously described method. One of the following three cases will occur during the creation of the trie.
Case 1 *Rule reorder does not occur during creation.* If rule reorders do not occur during the creation of T, then rules will appear from left to right in T as they appear in R (an in-order traversal of T yields R). As a result, the policy representations are equivalent because nodes are tested from left to right.
Case 2 *Rule reorders occur without intersections.* Consider a trie T consisting of $n-1$ rules from R, which are added using the process previously described. In addition, let the $n-1$ rules be ordered in T as they are in R. Assume a new rule r_n is added to T and is located to the left of an existing rule r_m. Let S be the set of rules that appear to the right of r_n in T, $S = \{r_i, m \leq i \leq n\}$. If the rules in S do not intersect with r_n, then a packet cannot match both r_n and any rule in S. As a result, testing r_n before any rule in S is not significant since shadowing is not introduced; thus, the reorder does not effect policy integrity.
Case 3 *Reorder and intersections occur during creation.* Consider a trie T consisting of $n-1$ rules from R, which are added using the process previously described. In addition, let the $n-1$ rules be ordered in T as they are in R. Assume a new rule r_n is added to T and is located to the left of the existing rule r_m. Let S be the set of rules that appear to right of r_n in T, $S = \{r_i, m \leq i \leq n\}$. Assume r_n does intersect with rule r_i in S. The intersection represents the set of packets that match r_n and r_i, where the tuples of the intersection are the more specific of the two rules. There must be at least one tuple in r_i more specific than the corresponding tuple in r_n, otherwise r_n is shadowed in the original rule list. The intersection rule will be located to the left of r_n and have the action of the r_i. Therefore, the intersection rule will always be tested first, and if true the action of the rule r_i is applied. This is the correct response; therefore, the reorder will not affect integrity. ∎

3.3.2 Push-Down Policy Tries

The policy trie as described thus far may require *backtracking* when a packet is processed (search is performed). Backtracking searches can have a worst-case performance that is equal to a list representation [29]. Although the penalty for backtracking in an n-ary trie is not as severe as a standard binary version, the conversion to a non-backtracking trie can reduce the number of tuple-compares, which is the objective of the representation.

A non-backtracking policy trie, referred to as a push-down policy trie, is created by replicating, or *pushing-down* general rules in the original policy. A general rule is a superset of at least one other rule in the policy, and is defined as a range of values, containing at least one wild-card in the standard prefix notation. The push-down procedure replicates more general rules in subset subtries, that would match the same packets as the general rule. As a result, the union of the push-down rules is a proper subset of the original rule.

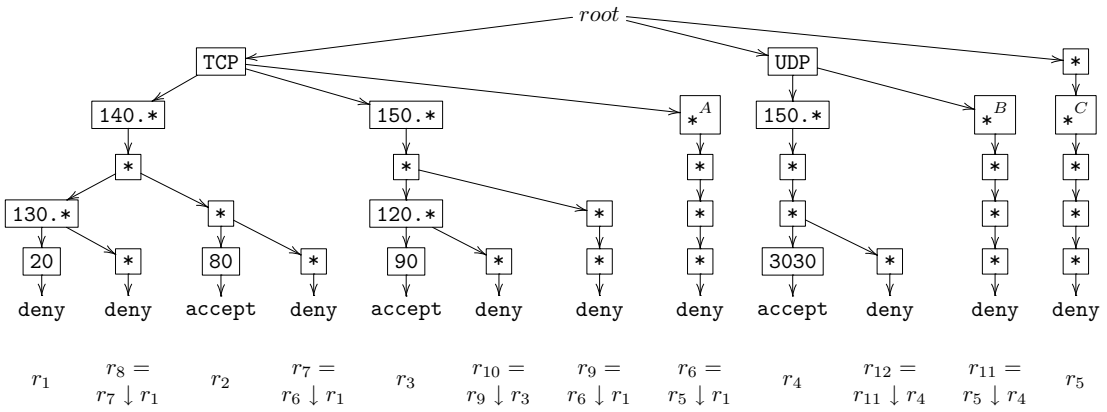


Figure 4: Push-down policy trie representation of the firewall rules given in table 1.

Definition The push-down of rule r_g to r_s, denoted as $r_g \downarrow r_s$ is

$$r_g \downarrow r_s = (r_s[1,..,l], r_g[(l+1),...,k], r_g[action])$$

where the $r_g[i] = r_s[i], i = 1,...,(l-1)$ and l is the index of the first tuple of r_s that is a subset of the corresponding tuple in r_g.

A general rule can only be pushed-down to rules that appear to left of it in the trie. Furthermore, a non-backtracking policy trie is created when **all** general rules are pushed-down; therefore, $r_g \downarrow r_s, \forall s < g$. This can be easily implemented using a post-order traversal of the policy trie. Note that while push-down always creates a rule that is more specific than the general rule being pushed, the resulting rule may still be a superset of other rules in the policy trie, and thus must be pushed down. For example, consider the push-down policy trie given in figure 4. When rule r_5 is pushed-down to rule r_1 it creates rule r_6. This is a general rule that is pushed-down again to rule r_1 yielding r_7. This process repeats yielding r_8, which cannot be pushed-down further.

As done with the original (backtracking) policy trie, we must be certain that the integrity of the original policy is maintained when using a push-down policy trie. Theorem 3.3 proves that push-down and original policy tries are equivalent. Therefore it can be stated that the push-down policy trie maintains integrity since, the original policy trie was proven to do so in theorem 3.2.

Theorem 3.3 *A push-down policy trie T_p is equivalent to the original policy trie T, which is equivalent to the original security policy R.*

Proof Assume a push-down trie T_p is constructed with k levels from an n rule ordered list. Consider node i on level l, $T_p[l,i]$, that is part of rule r_s. The children of node i include the children of siblings (of node i) that appear to the right, if node i matches the sibling. Recall the push-down procedure is recursive. If a tuple value is not present among the children of node i, the associated rule(s) are not a possible alternative since node i is not a match. Furthermore, given the procedure for constructing the push-down trie, the children are always ordered as they appear on the right. The rules will be tested in T_p in the same order as T which is equivalent to R. ∎

3.3.3 Worst Case Analysis

As described in the previous section, the push-down policy trie offers a performance increase by eliminating the need to traverse backwards. In this section, the worst case performance and storage requirement of the push-down trie is analyzed theoretically. A primary objective of the policy trie representation is to reduce the number of tuple-comparisons required per packet. Before this can be done, two important lemmas about push-down policy tries are required. The associated proofs have been omitted due to space constraints [17].

Lemma 3.4 *A node in a push-down policy trie cannot have more than n (the number of rules) children.*

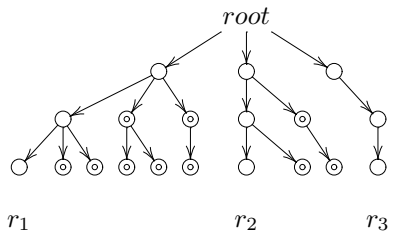
Lemma 3.5 *Each node traversal at a particular level in a push-down policy trie eliminates at least one rule from consideration.*

The previous two lemmas provide important bounds on the structure of any push-down policy trie. As a result, the worst case number of tuple-comparisons is $O(n + k)$, which is proven in theorem 3.6. Comparing this bound with the worst case for a list-based representation, the push-down policy trie requires a fraction $(1/k)$ of the processing.

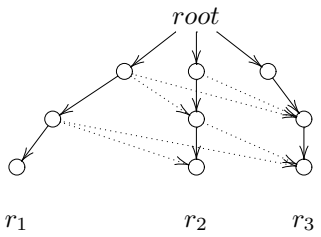
Theorem 3.6 *A comprehensive push-down trie consisting of k levels and constructed from n rules requires $O(n + k)$ number of tuple-comparisons to match a packet in the worst case.*

Proof We must always traverse every level of the trie (k tuple-compares) to determine the action. Following from lemmas 3.4 and 3.5, we know that traversing a node eliminates at least one rule, which would require n traversals in the worst case. As a result, the worst case number of tuple-compares is $k + n$. ∎

It is important to note that intersection and push-down operations do increase the number of nodes in the push-down trie, which increases the storage requirement. This is evident in the number of nodes required for the push-down trie depicted in figure 4 as compared to the original trie given in figure 2. Given an n rule firewall policy, push-down causes the worst case storage requirement to occur under two specific conditions. The first condition is $r_i \Rightarrow r_j, \forall i < j < n$, a rule matches all the rules that appear to the right, maximizing the number of push-downs that occur. The second condition is $r_i[l] \neq r_j[l], \forall i < j < n, 1 \leq l \leq k$; none of the tuples are equal and nodes are never shared. This worst case is depicted in figure 5(a), where a 3 rule list requires 20 nodes in the push-down policy trie. However, the number of nodes required by the trie can be greatly reduced by converting it into a Directed Acycical Graph (DAG) [1, 29]. In the context of a DAG, the push-down operation directly references nodes instead of replicating them as in the policy trie. For example, consider the push-down trie given in figure 4. The parents of the nodes labeled A and B could point to the node labeled C, which eliminates the need for new nodes for rules r_6 and r_{11}. The other push-down rules can be replaced in a similar fashion. Similarly, the DAG equivalent of the push-down policy trie given in figure 5(a) is depicted in figure 5(b) and requires significantly fewer nodes. The DAG conversion causes the worst case push-down trie to only require $k \cdot n$ tuples, which equals the storage requirement for a list representation.



(a) Worst case push-down policy trie. Double circle nodes are the result of push-down operations.



(b) DAG representation. The dotted arrows replace the push-down nodes.

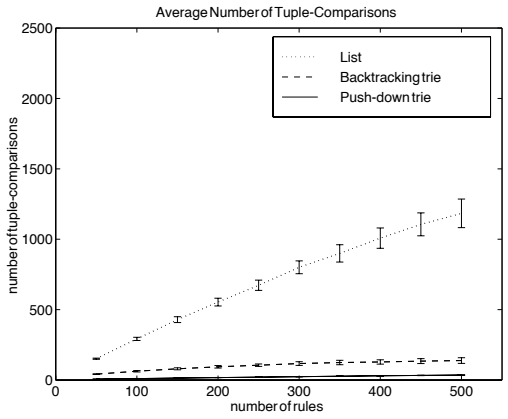
Figure 5: Worst case push-down policy trie ($r_1 \Rightarrow r_2$, $r_1 \Rightarrow r_3$, and $r_2 \Rightarrow r_3$) and the DAG equivalent. Assume the original security policy has three rules, where a rule has only three tuples.

The intersection of two rules, required when rule reordering occurs, may also result in a *new rule* (a new combination of existing tuples). The worst case policy would require the intersection among all rules to result in a valid rule, where tuples of the new rule alternate between the two rules. In addition, the rules would have to be listed according to the first tuple from most general to most specific. In this situation, intersection operations result in chains. Therefore, the worst case number of nodes required to store the policy trie would be $O(n^2)$. Although this is a higher space requirement than the standard list representation, it only occurs under very specific circumstances. Furthermore, the significance of the additional space requirement is relative to the frequency of the worst case packet(s).

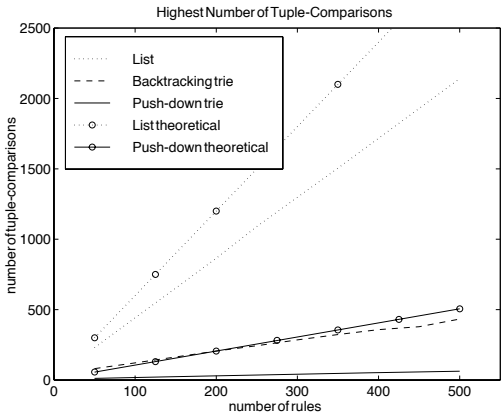
3.3.4 Experimental Results

The previous section described a new network security policy representation called a policy trie that was shown to provide theoretically better performance than the standard list-based representation. Simulation results presented in this section will confirm the worst case number of tuple-comparisons and show that similar performance gains are achieved in the average case. In addition, the average and worst case storage requirements for the different policy representations are presented and will be shown to also remain within their theoretical bounds.

Simulations were conducted using list, backtracking trie (original trie), and push-down trie representations of firewall policies. A random rule generator was used to create valid rule sets with a realistic degree of rule intersection. The generator was set to allow a slightly lower number of tuple permutations at high levels (source, source port, etc.) so that the shape of resulting policy tries would mirror those of real-world firewall rule sets. Policy sizes ranged from 50 rules to 500 rules, where 50 different policies were generated per policy size. Sets of 10,000 packets were passed through representations of each policy and the resulting decisions made were validated against the original rule set. Statistics concerning the average and worst case number of tuple-comparisons were recorded as well as the amount of storage required for each policy representation



(a) Average number of tuple-comparisons. Error bars represent one standard deviation.



(b) Highest number of tuple-comparisons.

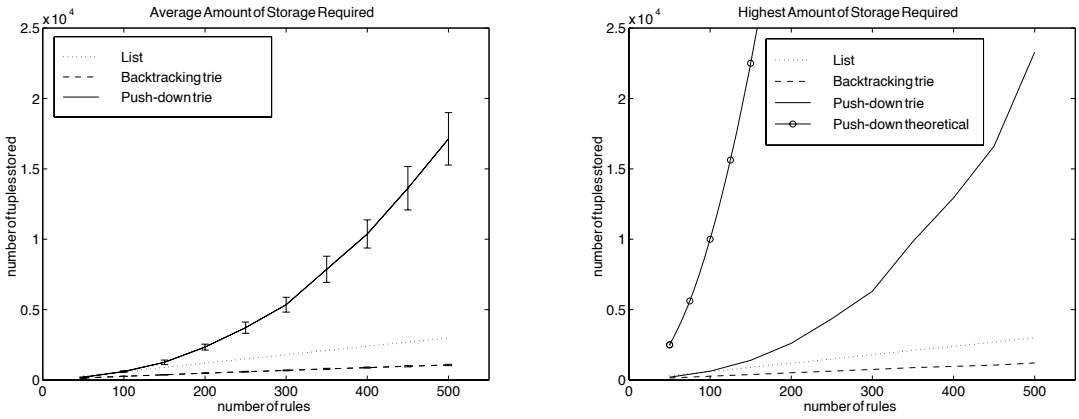
Figure 6: The average and worst case number of tuple-comparisons required for the firewall experiments. Push-down trie provided the best performance in both cases.

3.3.5 Tuple-Comparisons Results

Results for the tuple-comparisons are given in figure 6. As expected, when the policy sizes increased, both trie representations always performed considerably better than the linear rule set. In all cases, each representation reached the same decision, indicating that they were equivalent; thus maintaining integrity. As seen in figure 6(a), the average performance for backtracking tries appears to be similar to that of push-down tries. However, the backtracking trie required 5 times as many comparisons on average than the push-down trie, while the original list required 34 times as many tuple-comparisons on average.

The variance for the average number of comparisons in push-down tries was slightly lower than that of backtracking tries. As a result, push-down tries sometimes performed significantly better than their backtracking counterparts. Compared to linear implementations, the variance of trie-based implementations was very low and relatively constant. The standard deviation of the average number of comparisons in either trie implementation never rose above 20 per packet over 10,000 packets. Though this number may seem significant, the variance of linear policies averaged more than 46 comparisons and sometimes ranged as high as 100 comparisons.

The worst case number of tuple-comparisons required by each representation is given in figure 6(b). Similar to the average case results, both trie representations out-performed the list representation. Compared to the push-down trie the backtracking trie required 7 times as many of tuple-comparisons to reach a decision in the worst case, while the list representation required 31 times as many. In addition, the performance of push-down tries and list representations were within the theoretical bounds.



(a) Average number of tuples stored. Error bars represent one standard deviation.

(b) Highest number of tuples stored.

Figure 7: The average and worst case number of tuple stored for the firewall experiments. Backtracking trie required the least amount of storage in both cases.

3.4 Storage Results

The amount of storage required, measured in the number of tuples, is depicted in figure 7. As predicted, when policy sizes increased the backtracking tries consistently used less (a third for these experiments) of the storage required by the list representation. In contrast, the push-down tries required the most storage. The push-down trie storage was nonlinear with respect to the number of rules and on average required 10 times as much storage as the backtracking trie. Furthermore, the variance of push-down tries averaged over 1,000 nodes, while the variance of backtracking tries averaged less than 50 nodes. Although the push-down trie representation required the most storage, the observed worst case requirement was well below the theoretical upper bound of n^2, requiring on average 92% fewer tuples.

The amount of storage required by both trie representations is directly related to the degree of rule overlap. Backtracking tries benefit from rule overlap because overlap results in a greater number of shared nodes. In contrast, the storage requirement for push-down tries improves when there are fewer subset relations in a policy, since fewer push-down operations occur.

4 Security Policy Optimization Techniques

An important research area during the first year was the development of the firewall policy models: policy Directed Acyclical Graphs (DAG) and policy tries. These models were designed to maintain policy integrity and have been key for directing firewall design (such as topology). *Security policy optimization* is the reorganization of firewall rules to reduce the search time required to find the appropriate match (average or worst case). How optimization is performed depends on the data structure used to represent the policy.

4.1 List-Based Policy 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 integrity, 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 $P = \{p_1, p_2, ..., p_n\}$ 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

$$E[n] = \sum_{i=1}^{n} i \cdot p_i \tag{1}$$

For example, the average number of comparisons required for the rule set in table 1 is 3.85.

Given a policy DAG $G = (R, E)$ and policy profile $P = \{p_1, p_2, ..., p_n\}$ a linear arrangement π 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 [31], which is also referred to as Smith's algorithm [33]. 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 for a single machine with precedence constraints [21, 26]. The notation for such scheduling problems is $\alpha|\beta|\gamma|\delta$, where α is the number of machines, β is the precedence (or absence of) which can be represented as a DAG, γ is a restriction on processing time, and δ is the optimality criterion [21]. 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 [33], 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 [24] and Lenstra et. al. [26] 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 [8].

Theorem 4.1 $1|\beta|1|\sum w_i C_i \propto$ *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 β, which is G, to be a rule list 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. ∎

The sorting algorithm developed during the first year of the project used the following comparison to determine if neighboring (appear consecutively) rules should be interchanged.

$$\textbf{if}(p_i < p_{i+1} \textbf{ AND } r_i \not\parallel r_{i+1})\textbf{then}$$

The sorting method compared neighboring rules in this fashion until further exchanges were not possible. It is important to note the match probabilities will not change after sorting, since only non-intersecting rules may be exchanged. Sorting rules in this fashion can have positive impact on the average number of comparisons required. The algorithm reduced the average number of comparisons upwards of 70% for small policies, which is a significant improvement. However, larger more realistic rule-sets were not able to have the same performance increase.

The augmented algorithm developed this year also considered neighboring rules; however an exchange never occurs if the rules intersect and have different actions. This test preserves any precedence relationships in the policy. For example, the new sorting algorithm used such a comparison to determine if neighboring rules should be exchanged. Note, a_i denotes the action associated with the i^{th} rule.

$$\textbf{if}(p_i < p_{i+1} \textbf{ AND } (r_i \not\parallel r_{i+1} \textbf{ OR } a_i == a_{i+1}))\textbf{then}$$

Unlike the previous method, sorting in this fashion may impact the match probabilities. If two rules do not intersect, then a relative reorder will not effect the match probability since the appropriate first match is maintained for any packet, which is given in X. If two rules intersect and have the same action, then a relative reorder will change the match probability, since the rule that appears first in the relative reorder will have a higher probability (at the expense of the other rule, which may need to be reordered). This simple modification can result in further reductions in the average number of rule comparisons. The performance increase is depicted in figure 8. As seen in the figure, the augmented sorting method consistently performs better. This is because exchanges can still occur even if the dependency percentage is high. If an exchange occurs between two rules with the same action, a portion of the probability will be shifted towards the beginning of the list.

4.2 Trie-Based Policy Optimization

The policy trie and PDT representations described in section 3.3 dramatically reduce the number of comparisons required to process a packet. Similar to policy list sorting, the average number of comparisons can be further improved by organizing the trie such that the more popular rules appear towards the left. Since constructing the trie may reorder rules, building a trie from a sorted list may **not** result in a sorted trie. Furthermore, exchanging sub-tries of different specificities runs the risk of violating the integrity of the associated policy. As a result, sorting tries and PDTs is limited to comparisons between rules of the same specificity (nodes that share the same parent node). In firewall policies controlling of a wide variety of services and hosts, this severely limits the effectiveness of sorting.

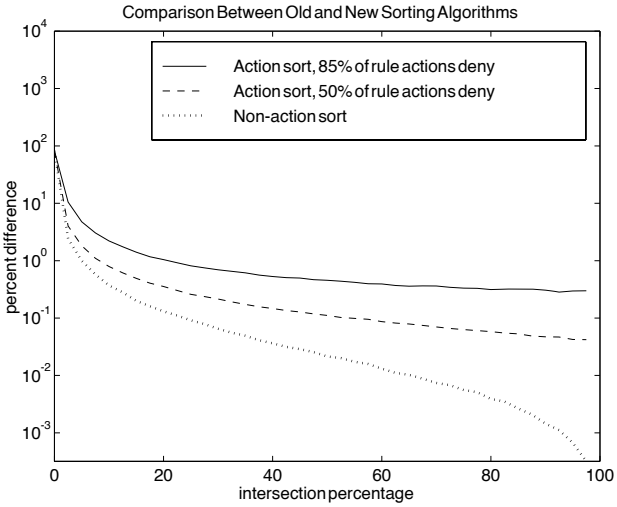
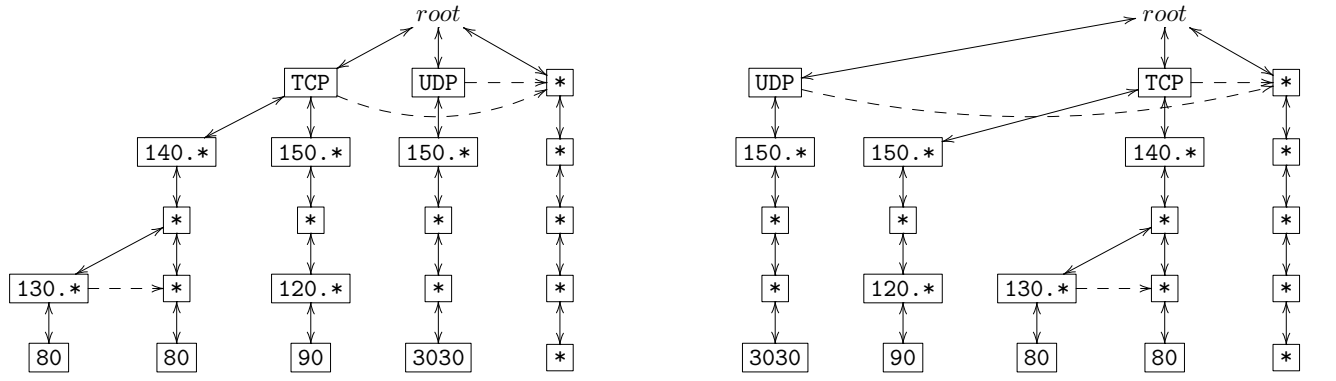


Figure 8: Percent decrease in the number of rule comparisons as the dependency percentage increases for a 1000 rule policy. The new sorting algorithm (action sort) performs better.



(a) Original policy trie.

(b) Sorted policy trie.

Figure 9: Original and sorted trie representations of the firewall rules given in figure 1. Dashed edges represent precedence constraints between nodes that share the same parent.

As with list-based policies, policy-DAGs can be used to model the precedence relationships between sub-tries. Sorting based on the constraints of a DAG allows the potential reordering of rules of different specificities. Given a parent node in the trie, a DAG is created with nodes representing each of the children's sub-tries. These nodes are comprised of the most general root to leaf path beginning with each respective child. Nodes are then compared to determine where subset relationships exist and edges are drawn to represent the subsequent precedence relationships. In figure 9(a) the children of the *root* node result in two edges being drawn. Three rules in the `TCP` sub-trie and the one rule in the `UDP` sub-trie intersect with the default rule in the `*` sub-trie. Edges are drawn between `TCP` and `*`, as well as between `UDP` and `*`. In contrast, consider the children of the `TCP` sub-trie. Rules in the `140.*` sub-trie do not intersect with rules in the `150.*` sub-trie, so an edge is not drawn between these siblings. Based on the DAG, the sub-tries can be reordered to improve performance without violating policy integrity, which is the basis of the sorting technique described in the next section.

4.2.1 Ordering Policy Sub-Tries

Consider a policy trie T that contains sibling nodes i and j, where T_i is the sub-trie with i as the root node and T_j is a sub-trie with j as the root node. Let $P(i)$ be the sum of the probabilities of the rules contained in sub-trie of root i, while $C(i)$ denotes the number of comparisons required to completely traverse sub-trie that has i as the root node, equal to the number of branches. In order to reduce the average number of tuple comparisons, sub-tries that share the same parent node should be ordered such that the $P(i)$ values are non-ascending and higher match probabilities occur first, from left to right. If sub-tries that share a parent node have the same probability, $P(i)$ equals $P(j)$, then they should be arranged such that their $C(\cdot)$ values appear in non-descending order.

Though reordering two rules may be beneficial, the constraints of a policy DAG must be considered to ensure integrity. For example, take two neighboring sub-tries T_i and T_{i+1} that share the same parent node and are out of order. Let $T_i \not\cap T_j$ indicate the rules in T_i do not intersect with the rules in T_j ($r_s \not\cap r_t, \forall r_s \in T_i, \forall r_t \in T_j$). The two sub-tries can be rotated about the parent if and only if $T_i \not\cap T_{i+1}$ or if all the rules in sub-tries T_i and T_{i+1} have the same action. Similar to finding the linear sequence of a policy DAG's, these conditions maintain the policy trie integrity, which is proven in the following theorem.

Theorem 4.2 *Given a policy trie T, exchanging sub-tries that have the same parent node based on the associated policy-DAG maintains integrity.*

Proof Consider two sub-tries, T_i and T_j, in a policy trie T ($i < j$, without loss of generality). T_i and T_j may be exchanged if there is no edge from any node in T_i to T_j. By construction, the nodes in each set represent the most general root to leaf path in their respective sub-trie and by definition each node is a subset of its respective power set, T_i or T_j.

Suppose that a set of packets d overlaps with rules in both T_i and T_j and the associated rule actions are different. The reordering of these two rules would result in a shadowing. It follows that, by definition of the DAG, there exists an edge between these two rules. This violates the choice of T_i and T_j. By contradiction, no such rules will exist and therefore, no anomaly will be introduced into a sorted trie.

```
function sortTrie(trieNode m)
    if(m is leaf node) return
    done = false
    while(!done)
        done = true
        for each child i of m that has a right neighboring sibling
            if(P(i) < P(i+1) AND (T_i ⋔ T_{i+1} OR
            action(T_i)==action(T_{i+1}))then
                exchange T_i and T_{i+1}
                done = false
            elseif(P(i) == P(i+1) AND C(i) > C(i+1) AND
            (T_i ⋔ T_{i+1} OR action(T_i)==action(T_{i+1}))then
                exchange T_i and T_{i+1}
                done = false
            endif
        endfor
    endwhile
    for each child i of m
        sortTrie(i)
    endfor
endfunction
```

Figure 10: Trie sorting algorithm.

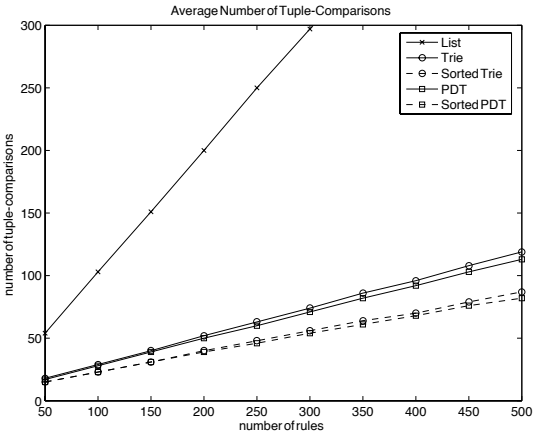
4.2.2 A Trie Sorting Algorithm

Using the guidelines for maintaining integrity described in the previous section, figure 4 presents a simple sorting technique for policy tries or PDTs. The algorithm starts with a call at the *root* node, sorting child sub-tries based on match-probability and the number of branches contained in each sub-trie. As previously described, an exchange of sub-tries does not occur if the sub-tries intersect and have different actions. The function is then called on each child node, and the process repeats until the leaf nodes are reached.

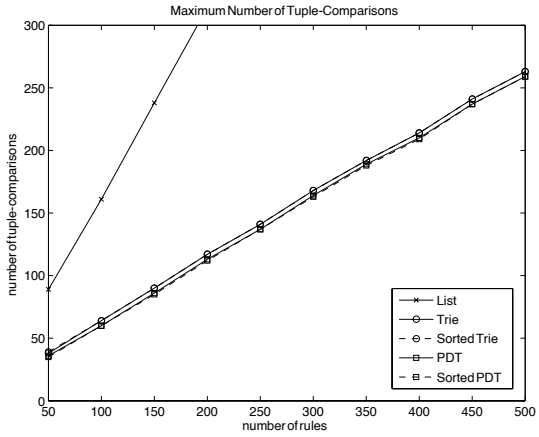
At each step, probabilities are calculated for sub-tries whose constraints permit comparison. Probability values for nodes created as the result of the intersection procedure are given a probability less than either of their two original rules, as they will match a smaller set of packets than their ancestors. For all other sub-tries, their cumulative probability is the summation of the probabilities of their leaves.

In the event that two probabilities are equal, the tie break is decided by the number of branches in each trie, $C(\cdot)$. In this case, we want to make sure that tie break always results in a better ordering, satisfying the inequality $P(j) \cdot C(j) + P(i) \cdot (C(i) + C(j)) < P(i) \cdot C(i) + P(j) \cdot (C(i) + C(j))$, where $(P(j), C(j))$ and $(P(i), C(i))$ are the cumulative probabilities and branches for two sub-tries T_i and T_j. By construction, we know that $P(i) = P(j)$ and $C(i) > C(j)$. Simplification yields the true statement, $C(j) < C(i)$, confirming that the inequality is satisfied.

Figure 9(b) depicts the sorted version of the policy trie given in 9(a). The first function call attempts to sort the sub-tries `TCP`, `UDP`, and `*` (protocol) about their parent node. Although the probability for the sub-tries `TCP` and `UDP` equal ($P($`TCP`$) = P($`UDP`$) = 0.3$) the sub-tries are exchanged since $C($`TCP`$) < C($`UDP`$)$. Although $P($`*`$)$ has the highest probability, it cannot be moved towards the left because of precedence edges. The children of `TCP` node are then sorted (IP source), where the `140.*` and `150.*` sub-tries are exchanged based on the number of branches. Sorting continues with the grandchildren of the `140.*` node, the `130.*` and `*` sub-tries (IP destination), which are not exchanged since they intersect.



(a) Average number of tuple-comparisons.



(b) Highest number of tuple-comparisons.

Figure 11: The average and worst case number of tuple-comparisons required.

This algorithm can be implemented by keeping *buckets* attached to rule action tuples to measure the relative frequency of matches for individual rules. Then, based on this information, the tries can be rebalanced offline and implemented in the firewall. In simulations, sorted tries provided significantly better processing time to all traffic by using DAG sorting to favor high frequency rules.

4.2.3 Trie-Based Policy Optimization Experimental Results

The previous sections described how policy tries and PDTs can be sorted to reduce the average number of tuple comparisons. Simulation results in this section will measure the impact of the sorting method under realistic conditions. Firewall policies of sizes ranging from 50 to 500 rules were generated using a random-rule generator that ensured anomaly free policies and imitated the shape of common security policies. Sets of 10,000 packets were then generated and skewed to favor random subsets of rules over others to simulate realistic traffic distributions. Of note, our traffic generation algorithm mimics high flow to a small set of services, with no regard for their placement in the policy. This is in contrast to the models used in [4], which sought to produce DoS traffic that exploited the structure of linear and backtracking trie implementations. Linear, back-tracking trie, and PDT implementations were created and evaluated in their original form. Then, based on frequency analysis of the traffic set, they were rebalanced using DAG sorting and evaluated for comparison.

The results for the average and higher number of tuple-comparisons are given in figure 11. As reported in [4], the trie implementations performed significantly better than the list, yielding a 81% reduction in the number of tuple-comparisons required. In the case of back-tracking tries, balancing resulted in 25% fewer comparisons than unbalanced tries. This is a result of the reduction of cumulative delay by processing most packets faster. Not only does the targeted traffic stream benefit, but those packets now requiring more comparisons also benefit. Though the average number of comparisons decreased, the worst case performance

No.	Proto.	Source IP	Source Port	Destination IP	Destination Port	Action	Prob.
4	UDP	150.*	*	*	3030	accept	0.3
5a	UDP	*	*	*	*	deny	0.3
3	TCP	150.*	*	120.*	90	accept	0.15
1	TCP	140.*	*	130.*	80	deny	0.1
2	TCP	140.*	*	*	80	accept	0.05
5b	TCP	*	*	*	*	deny	0.1

Table 2: Security policy given in figure 2 sorted with r_5 (*, *, *, *, deny) into rules r_{5a} and r_{5b}.

for single packet evaluation for sorted tries remained the same. This is due to the nature of back-tracking search in which all paths must be traversed for some packets no matter what the order.

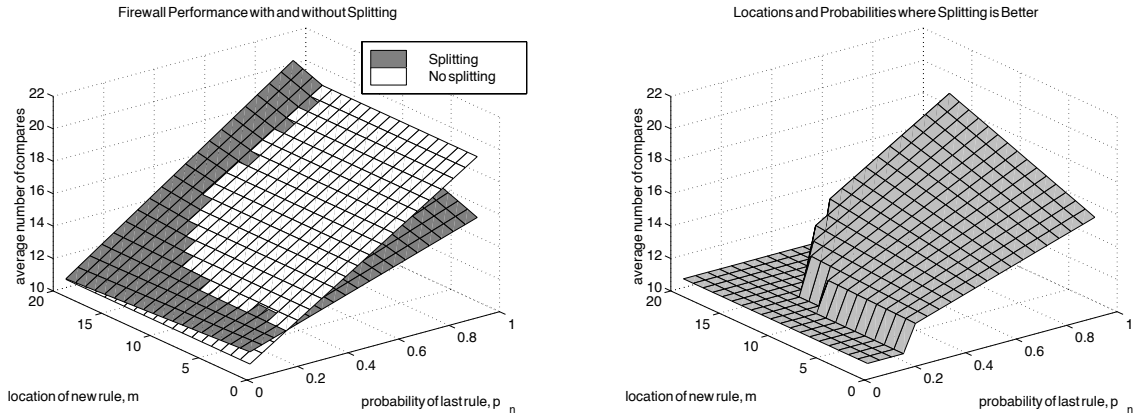
Push-Down Tries performed the best of all when sorted. The sorted PDT required 83% fewer comparison on average than a list and 27% fewer comparisons on average than an unsorted PDT. In addition, the maximum number of comparisons in worst case situations decreased slightly, a function of their structural replication rules. As unpopular rules are shuffled to the back of the PDT, they are not replicated elsewhere in the PDT unless needed. This effectively allows certain rules to be excluded from most evaluations even in worst-case situations.

4.3 Rule Splitting

Rule splitting takes a general rule and creates more specific rules that collectively perform the same action over the same set of packets. Here, we are utilizing rule splitting to reduce the average number of comparisons. For example in table 2, rule r_5 is split into two separate rules, r_{5a} for UDP and r_{5b} for TCP. Once the rules are positioned based on their probabilities and their relation to other rules, the average number of rule comparisons is reduced to 2.6 (after sorting and splitting) which is a 16% less.

It many not be advantageous to split a general rule since it adds another rule to the policy. For example, assume a policy contains 20 rules where the first 19 rules have the same probability. Assume the last rule can be split and the new specific rule has a probability that is $\frac{3}{4}$ of the last rule. The impact of the probability of the last rule and the location of the new split rule, m, is depicted in figure 12. The average number of comparisons is reduced as the split rule is located closer to the first rule (surface decreases as m approaches one). Furthermore, splitting yields better results when the general rule has a high probability. However as seen in figure 12(b), the closer the specific rule is to the location of the original rule the average number of comparisons increases, which is the penalty of adding one more rule to the policy.

The effect of splitting a single rule can be described mathematically as follows. Consider n rules where rule r_n can be split into rules $r_{n,l}$ and $r_{n,r}$ (whose union is the original rule). In addition, assume the split rule $r_{n,l}$ will be located at the m^{th} position ($1 \leq m < n$), while rule $r_{n,r}$ will remain at the n^{th} location. We want to determine the best location m, which



(a) Original and splitting.

(b) Locations and probabilities where splitting is best.

Figure 12: Example of splitting the last rule in a policy. Surfaces depict the impact of split rule location and the probability of the last rule (p_n). The probability of the new split rule is $\frac{3}{4} \cdot p_n$.

yields a lower average number of comparisons as compared to the original rule set. This can be defined mathematically in the following formula.

$$\sum_{i=1}^{n} i \cdot p_i > \sum_{i=1}^{m-1} i \cdot p_i + m \cdot p_{n,l} + \sum_{i=m}^{n-1} (i+1) \cdot p_i + (n+1) \cdot p_{n,r}$$

The left side of the inequality is the average number of comparisons for the original rule set. The right hand side of the inequality is the average number of comparisons with the specific rule at location m. If we assume the rules located between m and n have an equal probability (denoted as p) we can solve the previous equation for m.

$$m < \frac{n \cdot p - n \cdot p_n + (n+1) \cdot p_{n,r}}{p - p_{n,l}}$$

The new rule must be located between the first and m^{th}; however, its final location will depend on the relationship with the other rules (cannot be placed before any rule for which it is a superset). This result can be applied iteratively to multiple rules and repeatedly to the same rule.

5 Parallel Firewalls Designs

As described in the introduction, parallelization offers a scalable technique for improving the performance of network firewalls. Using this approach an array of m firewalls processes packets in parallel, as seen in figure 13. However, the designs depicted in the figure differ based on what is distributed: packets or rules. Using terminology from parallel computing, distributing packets can be considered *data parallel* since the data (packets) is distributed

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 3: Example firewall policy used for the parallel firewall examples.

across the firewall [9]. In contrast, *function parallel* designs distribute the policy rules across the firewalls.

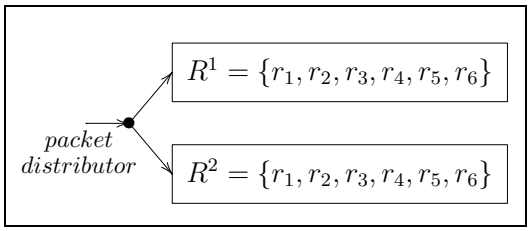
5.1 Data-Parallel Architecture

As shown in figure 13(a), data parallel firewall architecture consists of an array of identically configured firewalls [6]. Each firewall j in the system implements a local policy R^j, where $R^j = R$. Arriving packets are distributed across the firewalls for processing (one packet is sent to one firewall), allowing different packets to be processed in parallel. Since the accept set for each firewall j equals the accept set of the original policy, $A^j = A$, policy integrity is maintained.

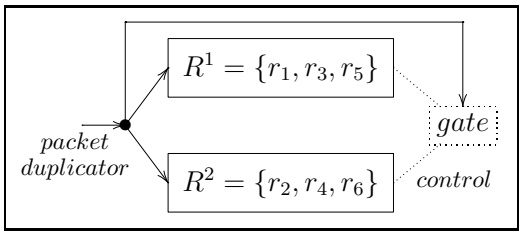
Distributing packets across the array allows a data parallel firewall to increase system throughput (number of packets processed per unit time) as compared to a traditional (single machine) firewall [6]. Furthermore, increased throughput is easily achieved with the addition of firewalls; therefore, this approach is very scalable. However the data parallel approach has three major disadvantages. First, stateful inspection requires all traffic from a certain connection or exchange to traverse the same firewall (where the stateful rule resides) or the constant distribution and management of stateful rules. As a result, successful connection tracking is difficult to perform at high speeds using the data parallel approach [6, 27]. Second, distributing packets is only beneficial when each firewall in the array has a significant amount of traffic to process (firewalls are never idle). The performance benefit (higher throughput) only occurs under high traffic loads. Finally, the design does not differentiate between traffic classes only load balancing. Therefore efficiently maintaining different QoS requirements is not possible.

5.2 Function Parallel Architecture with Gate

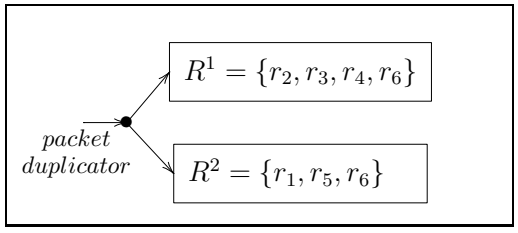
Unlike the data parallel model which distributes packets, the function parallel design distributes policy rules across the firewall array [15]. The function parallel design consists of multiple firewalls connected in parallel and a *gate* device. As seen in figure 13(b), when a packet arrives to the function parallel system it is forwarded to every firewall and the gate. Each firewall processes the packet using its local policy, including any state information. Since the local policies are smaller than the original, the processing delay is reduced as com-



(a) Data-parallel, packets distributed across equal firewalls.



(b) Function parallel with gate, rules distributed across firewalls.



(c) Function parallel, rules distributed across independent firewalls.

Figure 13: Various 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).

pared to a traditional firewall. Once the firewall finishes processing a packet, it then signals the gate indicating either no match was found, or provides the rule number and action if a match was found. The gate stores the results for the packet and determines the final action to perform using the policy DAG.

Since firewalls only implement a portion of the original policy, it is critical that rule distribution is done to maintain integrity. The integrity of a policy R is maintained if the rules are distributed such that every rule in R exists in the system and if the precedence constraints of R are observed in each local policy R^j. As a result, the accept set of the gate equals the accept set of the original policy [15]. This is more formally stated in the following theorem.

Theorem 5.1 *An array of m firewalls and a gate device arranged in a function parallel fashion enforcing a comprehensive policy R will maintain integrity if policy rules are distributed to each firewall j such that: every rule in R is assigned to at least one firewall and the precedence constraints of R are observed in* R^j.

Proof Let R' be the ordered subset of the rules in policy R that matches a packet d. Given a first match policy, the first rule in R' is the correct result. The first condition of the theorem ensures that these rules will exist in the system. The second condition ensures shadowing will never occur if multiple rules from R' exist in the same local policy, therefore then local first match is produced for d is the correct result. If these rules exist in different local policies then the gate will be given multiple matches from the set R' for d; however, the gate will determine the appropriate rule since it always applies the lowest numbered rule of the local first matches, thus policy integrity is maintained.

Therefore, a traditional single firewall and the function parallel firewall will always give the same result for the same packet. Several different distributions are possible that adhere the described guidelines. Essentially the rule numbers (indexes from the original policy) in each local policy must be in ascending order, as seen in figure 13(b). Note the policy used is given in table 3.

The function parallel design has several significant advantages over traditional and data parallel firewalls. First, the function parallel design results in faster processing since every firewall is utilized to process a single packet. Reducing the processing time, instead of the arrival rate (as done in the data parallel design), yields better performance since each firewall in the array processes packets regardless of the traffic load. The processing delay can be further reduced 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.

There are three disadvantages of the function parallel design. First, there is a possible limitation on scalability, since the system cannot have more firewalls than rules. However, given the size of most firewall policies range in the thousands of rules [38], the scalability limit is not an important concern. Second, the system is unable to differentiate traffic, therefore specific QoS constraints may not be provided. Thirdly, the gate requires specialized hardware and introduces an additional delay. Therefore, to achieve the full potential of the function parallel design it is preferable to eliminate the gate device and allow the firewalls to operate independently.

5.3 Independent Function Parallel Architecture

As described in the previous subsection, a function parallel system consists of an array of firewalls where arriving packets are duplicated and policy rules are distributed. Each firewall processes an arriving packet using its local policy and a gate device is required to ensure integrity is maintained. However, it is possible to allow the firewalls to operate independently, thus eliminating the gate device and any need for inter-firewall communications.

Consider a function parallel system consisting of m firewalls that enforces a comprehensive security policy R. Each firewall j in the array has a local comprehensive policy R^j that is a portion of the security policy R. Therefore, each firewall has a local accept set A^j and a deny set D^j. Integrity will maintained without a gate device if rules are distributed such that a packet $d \in D$ is dropped by all firewalls, while a packet $a \in A$ is accepted by only one firewall. This is more formally stated in the following theorem.

Theorem 5.2 *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.

An example distribution of the policy given in table 3 across an array of two independent firewalls is shown in figure 13(c). In this case, the local policy of the upper firewall will accept only packets from the `210` and `180` address range, 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 $R^2 = \{r_2, r_3, r_6\}$). In each example a precedence edge will never connect two rules in two different firewalls (no inter-firewall edges), and as a result, integrity is maintained. Policy distribution can be done to balance the packet load (distribute popular rules across the array) or to achieve a certain QoS objective. Of course the number of distributions will depend on the original security policy, where fewer precedence edges allow more distributions.

Like the function parallel system that relies on a gate device, the independent function parallel system can manage state information since a packet is sent to every firewall. However, allowing the firewalls to operate independently has several important unique advantages. First, the elimination of the gate device causes the function parallel design to be compatible with a variety of firewall devices since specialized equipment is not needed. Second, the independent function parallel system will have lower processing delays than an equivalent data parallel system or a function parallel system with a gate device. Third, local-policies can be designed to process certain types of traffic on certain firewalls, yielding the ability

to provide service differentiation which is an important component for maintaining QoS requirements.

Although the system has many significant advantages, it is not redundant. Integrity will be lost if a firewall fails since a portion of the policy (local accept set) will not be available. Fortunately, loss of a firewall will only result in a more conservative policy (fewer packets accepted), which is better than the previous function parallel design with gate device. Redundancy can be provided by duplicating the local policy to another firewall. As done in [6], firewalls can be interconnected to determine if redundant rules should be processed.

5.3.1 Policy Distribution

Given a function parallel firewall array and 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 table 3, 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 table 3 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 be the original policy DAG (as in rule sorting). For example, merging c_2, c_3, and c_6 requires placing r_6 at the end.

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 5.3 *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.

5.3.2 Policy Distribution Performance

A rule distribution is sought that minimizes the number of comparisons required to determine an accept [16]. 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

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

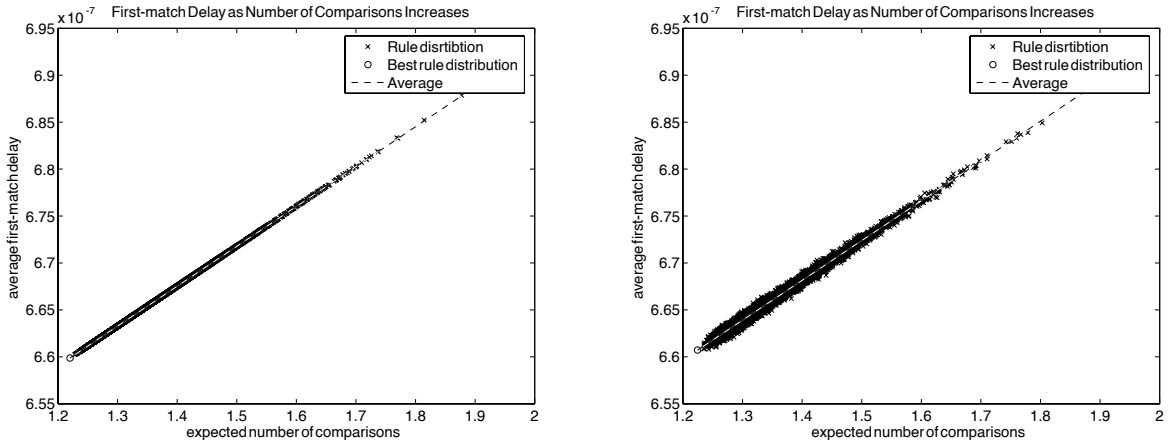
However, each rule in R is considered only once in the calculation, since only the average number of comparisons required for a first-match is considered. If a rule is duplicated (such as r_6 in the distribution given in figure 13(b)), then it is only considered once in the calculation at its earliest 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 13(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. Using the same two firewall function parallel system, if the rules were distributed based on protocol then the expected number of comparisons required to find the original first match would be

$$1 \cdot (p_1^1 + p_1^2) + 2 \cdot (p_2^1 + p_2^2) + 3 \cdot (p_3^1 + p_3^2) = 1 \cdot (p_2 + p_3) + 2 \cdot (p_2 + p_4) + 3 \cdot (p_5 + p_6) = 2.30$$



(a) Performance given same traffic.

(b) Performance using random traffic.

Figure 14: Packet delay as compared to the expected number of rule comparison for a function parallel system (four firewall array) and a 48 rule policy. Each point represents a equivalent distribution across the array.

The second distribution can be considered better since it requires fewer compares to determine the first-match. To achieve optimal performance, it is necessary to find a rule distribution that minimizes the average number of comparisons before the original first match is found.

The validity of the expected number of comparisons (equation 2) as a performance metric was evaluated under realistic conditions using simulation. Each experiment simulated a function parallel system consisting of a four firewall array implementing a 48 rule policy. The rule match probability followed an inverse Zipf distribution, which is commensurate with actual firewall policies [25, 38]. In addition, rules did not overlap (except for a default deny all rule) which allows for a large number of possible distributions.

Given the 48 rule security policy, the performance of the distribution that minimized average number of rule compares was compared against 10,000 random distributions that each maintained integrity of the original policy. For each simulation the firewall system packets arrived at a rate of 1 Gbps, and the average first-match delay and the sum of the average number of comparisons were collected.

Figure 14(a) shows the delay and expected number of comparisons for different distributions and the same traffic trace. The traffic trace was generated such that packet lengths were uniformly distributed between 40 and 1500 bytes and all legal IP addresses were equally probable. As seen in the graph, the expected number of comparisons is linearly proportional to the first-match delay. In addition, the best performance is obtained when the expected number of comparisons is minimized. Therefore the average number of comparisons is a very good metric given known traffic.

Performance of the metric under randomly generated traffic is depicted in figure 14(b). In these experiments packets were randomly generated using the same parameters as the first

experiment. As seen in this graph, the performance metric is not able to provide an exact performance prediction due to the variability in the traffic. However, the best performance still occurs when the average number of compares is minimized. Therefore, the metric is still able to provide insight into performance and can be used to compare different rule distributions in a function parallel firewall.

5.3.3 Policy Distribution Algorithm

Improving the overall performance of the firewall is achieved 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. However, the goal of balancing the number of rules across nodes when distributing rule chains is difficult. Examine the simple case of rule distribution where the probability of each rule in the original policy is the same. An algorithm to find the optimal solution must then be able to find the distribution with the minimum average difference in rule counts between the nodes. This case is equivalent to the Equal-Subset-Sum problem, described as follows:

Definition: (EQUAL-SUBSET-SUM). Given a set $S = \{s_1, ..., s_n\}$ of positive integers, are there two disjoint non-empty subsets $X, Y \subseteq S$ such that the sum of the integers of X is equal to the sum of the integers of Y.

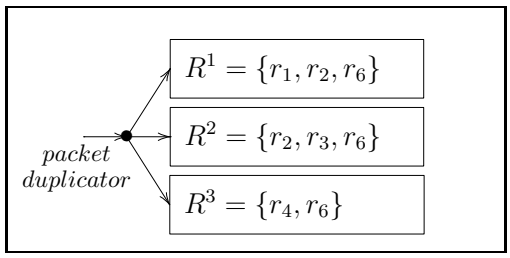
An algorithm which can find an optimal distribution of rules when all probabilities are the same, can then find a solution to the Equal-Subset-Sum problem, where the set S of positive integers is equivalent to the number of rules in each chain, and the subsets X and Y are nodes. The Equal-Subset-Sum problem, however, has been shown to exist in the class of problems known as NP-Complete.

A simple rule distribution algorithm first sorts the rule chains according to the average number of comparisons per rule chain ; for example, $L = \{c_2, c_3, c_4, c_1\}$ 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. As a result, chains with the smallest average number of comparisons are distributed first (placed near the end of the local policies). This simple, greedy distribution technique ensures the most popular rules appear near the beginning of the local policies. For the function parallel system shown in figure 13(b) and the policy given in table 3, 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.

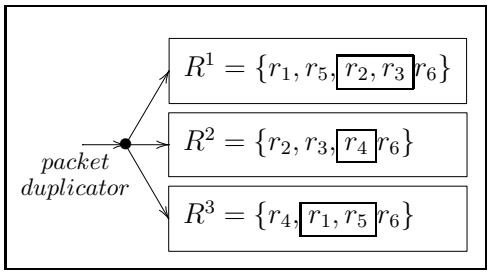
5.3.4 System Redundancy

Similar to the traditional and data parallel firewalls, a disadvantage of the current function parallel design is its inability to withstand a single firewall failure. If a single firewall fails in the function parallel system, then part of the policy is lost and integrity is not maintained. A simple solution commonly used for traditional firewalls is to duplicate the entire system; however, this solution is cost prohibitive, not efficient, and difficult to manage.

Redundancy can be provided in the function parallel design by replicating rules (including state generated), instead of firewalls. Consider the function parallel system depicted in figure 15(a), that consists of three firewalls and an six rule policy. If the local rule list of the neighbor



(a) Function parallel non-redundant rule distribution.



(b) Function parallel with redundant rule distribution. Boxed rules within local policies are only used if neighboring firewall fails.

Figure 15: Function parallel rule distributions, each system consisting of three firewalls and six rules.

below is added to the end of every firewall, as seen in figure 15(b), then any non-continuous firewall nodes can fail and integrity will be maintained.

The criteria for maintaining integrity still applies and may be problematic. Since the last rule in many policies is a deny (or possibly accept) all, the redundant rules given to the right-most firewall are shadowed (r_8 shadows r_1 and r_2 in figure 15(b)). This can be addressed by exchanging the redundant rules with the original rule in the right-most firewall if the left-most firewall fails. The redundant design also requires short-circuit evaluations to prevent multiple firewalls from evaluating the same rule. However unlike current solutions, the function parallel design does not require the duplication of firewalls for redundancy.

5.4 Theoretical Models

We utilized theoretical models to guide system design and simulation to predict performance. If arrivals and service times are assumed to be exponentially distributed, then a firewall system can be considered an open network of M/M/1 queues (Jackson network) [7, 32]. Probabilities can be assigned to each link to indicate the likelihood of moving to the next firewall, which are given from the policy profile (hit ratio) described in section 4. The average end-to-end delay for q cascading firewalls (traversal path) interconnected by a graph T can be computed as

$$E(T) = \sum_{i=1}^{q} \frac{1}{\mu_i - \lambda_i}$$

where $1/\mu_i$ is the service time (processing and transmission) and λ_i is the arrival rate to firewall i. As a result, we have a theoretical model for the average delay across a firewall system. Consider the parallel, function parallel, and hierarchical firewall designs given in figure 13. Assume each firewall system consists of m firewalls and implements the same n rule security policy. Let the total arrival rate to each system be λ packets per unit time and assume each firewall can perform x rules per unit time.

For the parallel firewall, traffic arrives at the packet distributor, which evenly distributes

the traffic. As a result the arrival rate to each firewall is $\frac{\lambda}{m}$. The service rate is $\frac{x}{n}$, since each firewall implements the complete rule set. The end-to-end delay across any firewall in the data parallel firewall is

$$E_d(T) = \frac{1}{\frac{x}{n} - \frac{\lambda}{m}}$$

In contrast, rules are distributed across each firewall in the function parallel firewall. Furthermore, all traffic arriving to the system is forwarded to each firewall. If we assume the rules in each firewall are independent (rules distributed using a non-backtracking trie), then the end-to-end delay across any firewall in the functional-parallel firewall is

$$E_f(T) = \frac{1}{\frac{m \cdot x}{n} - \lambda}$$

The relative speedup compared to a parallel firewall system is $\frac{1}{m}$; therefore, the functional-parallel system has the potential to be m times faster than a (data) parallel system. However, the scalability of the functional-parallel system is dependent on the rule set.

The impact of increase arrival rate, number of rules, and number of firewalls can be compared theoretically using the equation given above. Figure 16 show the average packet delay as the arrival rate increases. The parallel firewall performs better than the single firewall since the system can process multiple packets simultaneously. The function parallel system performs better than the parallel system which indicates reducing the processing requirement is more significant than reducing the arrival rate. This is also shown in figure 16(b) where the number of rules in the policy also increases.

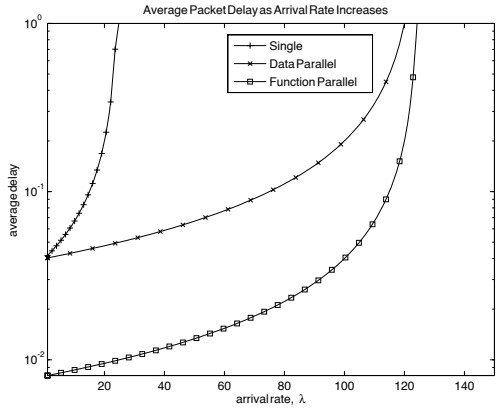
The impact on increasing the number of rules on average packet delay is given in figure 17(a). The data and function parallel systems consisted of five firewalls, however the function parallel system performs better than the data parallel firewall. The performance improvement is constant as the number of rules increases indicating the function parallel design is better.

Figure 17(b) shows the average delay as the number of firewalls in the array increases. As firewalls are added to the two parallel designs, the function parallel system continues to reduce the average delay. The adding more firewalls to the function parallel design reduces the number of rules per local policy, therefore reducing the average processing time. As firewalls are added to the data parallel system the average delay remains constant. More firewalls does not improve the performance since additional firewalls do not have packets to process.

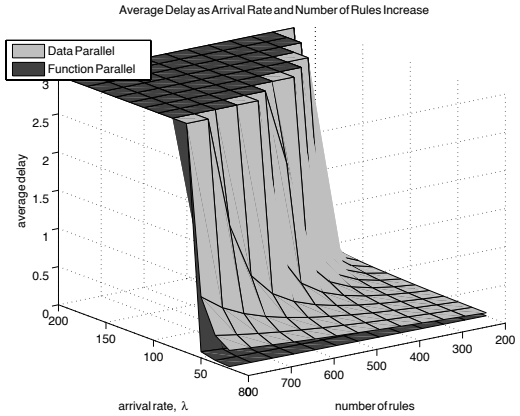
5.5 Parallel Firewall Experimental Results

The performance of a traditional single firewall, the data parallel firewall, and the function parallel firewall (with gate device and independent) was measured under realistic conditions using simulation. Firewalls were simulated to process 6×10^7 rules per second, which is comparable to current technology.

For each experiment the parallel designs always consisted of the same number of firewalls. The gate device delay was equivalent to processing three firewall rules. Short-circuit evaluation was simulated for the gated design, where the firewalls in the array are notified to

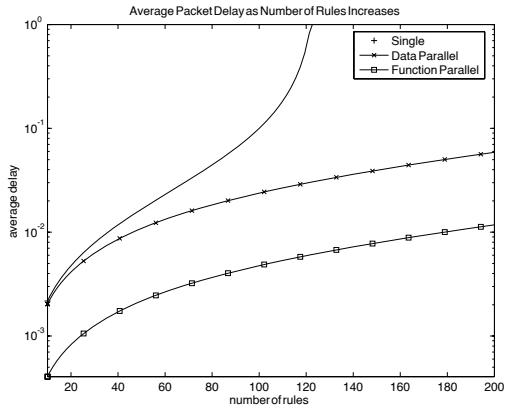


(a) Average delay as arrival rate increases.

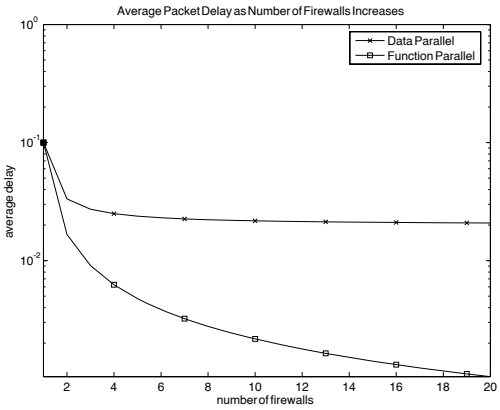


(b) Average delay as arrival rate and the number of rules increases.

Figure 16: Theoretical comparison of different firewall designs.

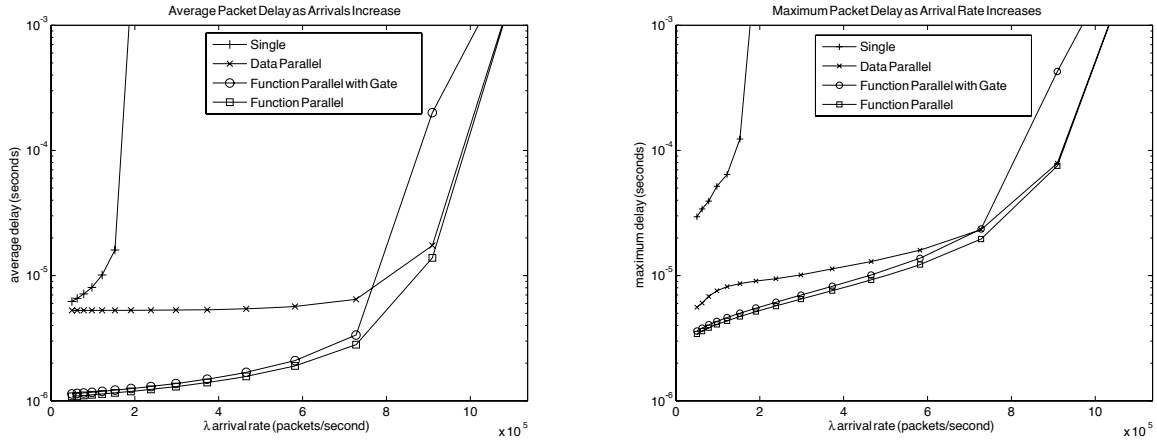


(a) Average delay as the number of rules increases.



(b) Average delay as the number of firewalls increases.

Figure 17: Theoretical comparison of different firewall designs.



(a) Average delay.

(b) Maximum delay.

Figure 18: Packet delay as the packet arrival rate increases. Parallel designs consisted of five firewalls.

stop processing a packet once the appropriate match was determined. No additional delay was added to the data parallel system for packet distribution (load balancing); therefore, the results observed for the data parallel design are better than what should be expected.

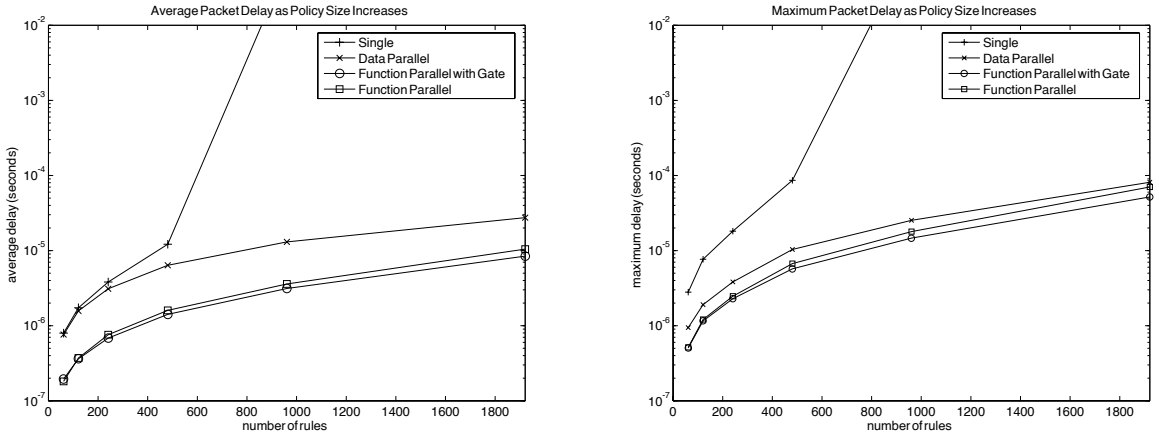
Packets lengths were uniformly distributed between 40 and 1500 bytes, while all legal IP addresses were equally probable. Firewall rules were generated such that the rule match probability was given by an inverse Zipf distribution. As a result, rules near the beginning of the policy were specific (matched a limited range of packets) while rules near the end were more general, which is commensurate with actual firewall policies [25, 38]. Rules were distributed for the function parallel design such that no inter-firewall dependency edges existed, and if possible, more popular rules were located at the top of the local-policies. This distribution provides load balancing and ensures integrity is maintained.

Three sets of experiments were performed to determine the effect of increasing arrival rates, increasing policy size, and increasing number of firewalls. For each experiment 1000 simulations were performed, then the average and maximum packet delay were recorded.

5.5.1 Increasing Arrival Rates

The impact of increasing arrival rates on the four firewall designs is shown in figure 18. In this experiment, each system implemented the same 1024 rule firewall policy [38] and both parallel designs consisted of five firewalls. The arrival rate ranged from 5×10^3 to 1×10^6 packets per second (the highest arrival rate resulted in more than 6 Gbps of traffic on average).

As seen in figure 18, the parallel designs performed considerable better than the traditional single firewall. As the arrival rate increased, the parallel designs were able to handle larger volumes to traffic due to the distributed design. As seen in figure 18(a), the function parallel firewall had an average delay that was consistently 4.0 times lower than the data



(a) Average delay.

(b) Maximum delay.

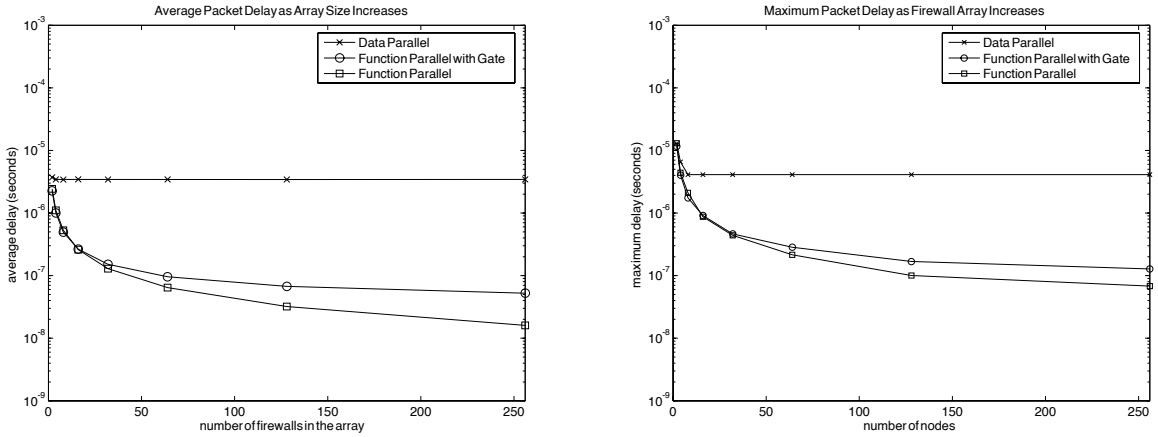
Figure 19: Packet delay as the number of rules increases. Parallel designs consisted of five firewalls.

parallel design, while the independent function parallel design average delay was 4.3 times lower. This is expected because each firewall in the function parallel design is used to inspect arriving packets regardless of the arrival rate. The impact of the gate delay is more prominent as the arrival rate increases. Similar to the average delay results, the function parallel design had a maximum delay 34% lower than the data parallel design, while the independent function parallel design was 38% lower.

5.5.2 Increasing Policy Size

The effect of increasing the policy size (number of rules) for the four firewall designs is given in figure 19. In this experiment, the arrival rate was again 1×10^5 packets per second (yielding more than 0.5 Gbps of traffic on average) and both parallel designs consisted of five firewalls. Policies ranged from 60 to 3840 rules which is considered within the bounds of typical policies [38].

When the policies consisted of relatively few rules, the single and data parallel firewalls observed similar delays. However as seen in figure 19(a), the parallel designs performed considerable better than the traditional single firewall once the policy contained more than 1000 rules. The function parallel firewall had an average delay that was 4.12 times lower than the data parallel design, while the independent firewall was 3.79 times lower. This slight difference is primarily due to short-circuit evaluation, where the gate informs firewalls to stop processing a packet once the appropriate match is found. However this is only a marginal gain given the inter-firewall communication and specialized hardware required for short-circuit evaluation.



(a) Average delay.

(b) Maximum delay.

Figure 20: Packet delay as the number of firewalls increases. Policies consisted of 1024 rules.

5.5.3 Increasing Firewall Array Size

Figure 20 shows the effect of increasing number of firewalls for the two parallel firewall designs. The number of firewalls ranged from 2 to 256, the number of rules was 1024, and arrival rate was 2×10^5 packets per second (again, yielding more than 1 Gbps of traffic).

As seen in figure 20(a), the function parallel design consistently performed better than the data parallel firewall design. As firewalls were added, the function parallel system always observed a reduction in the delay. This delay reduction trend is expected until the number of firewalls equals the number of rules. In contrast, the delay for data parallel design quickly stabilizes and the addition of more firewalls has no impact. This is because after a certain point any additional firewalls will remain idle, thus these additional firewalls are unable to reduce the delay. As additional firewalls are added the performance difference between the function parallel firewall and theoretical limit becomes larger. The local policy delay becomes smaller as more firewalls are added; however, the gate delay remains constant, thus more prominent in the total delay experienced.

5.6 Firewall Grids

The function parallel design described in the previous section seeks to provide lower packet processing delay by distributing rules across an array of firewalls. In many cases, the function parallel design provides lower delays than an equivalent data parallel design. This was shown theoretically and experimentally using simulations in this report. However, the function parallel design does not provide the best performance in all situations.

Given extremely high traffic loads, the data parallel design can perform better than an equivalent function parallel system. Furthermore, the policy profile (hit rate distribution) has a significant on performance, as seen in figure 21. Experiments performed assumed the policy profile followed an inverted Zipf distribution, where rules near the end of the policy had

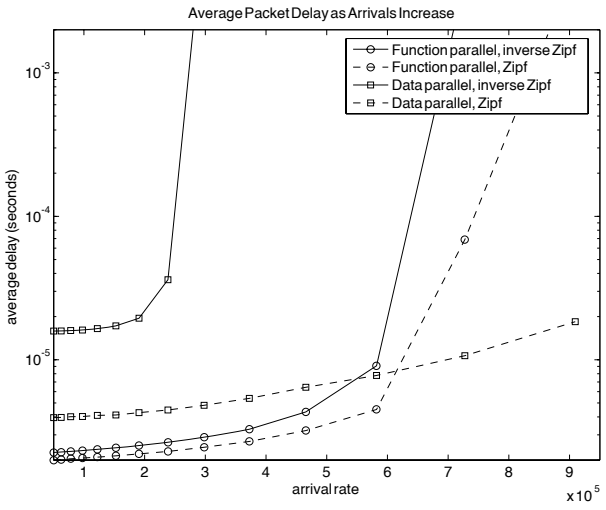


Figure 21: Average packet delay for the parallel design, each consisting of four firewalls, as the arrival rate increases. Firewall policies consisted of 4096 rules and had either a Zipf or inverse Zipf profile.

higher probabilities of being the first match. The function parallel design has an advantage in this case because lower rules (the popular rules located near the end of the policy) are evaluated earlier as compared to a data parallel design. However if more popular rules appear near the beginning of the policy, the data parallel design performs better. Similarly, the rule distribution effects the performance of the function parallel system. As previously described function parallel rule distribution (without gate) requires duplicating certain deny rules. As the number of duplicate rules increases the performance decreases, as seen in figure 22. However, it is important to note function parallel still performs better when the number of rules is doubled. Finally, function parallel does not perform better if rule processing can be done in sub-linear time, since reducing the number of rules does not significantly reduce the processing time. This type of processing is possible with hardware firewalls, where static rules can be effectively represented using Ternary Content Addressable Memory (TCAM) or a Field Programmable Gate Array (FPGA) [29, 35].

Since neither parallel design always performs best, the best design uses a highly configurable array of firewalls. The array of firewalls can then change based on current traffic and policy profiles. Function parallel design would be used when under low traffic loads or when the policy profile follows a Zipfs distribution. In addition, it is possible to provide a hybrid function and data parallel design. The firewalls can be divided into a data parallel array of function parallel groups. This design would combine high throughput and redundancy of from the data parallel design and low processing delay from the function parallel design.

5.7 Parallel Intrusion Detection Systems

Although firewalls and Intrusion Detection Systems (IDS) have some differences, they essentially perform very similar operations. Both inspect and apply a policy to packets traversing the system. Actions applied to packets include accept, deny, and, in the case of IDS, log-

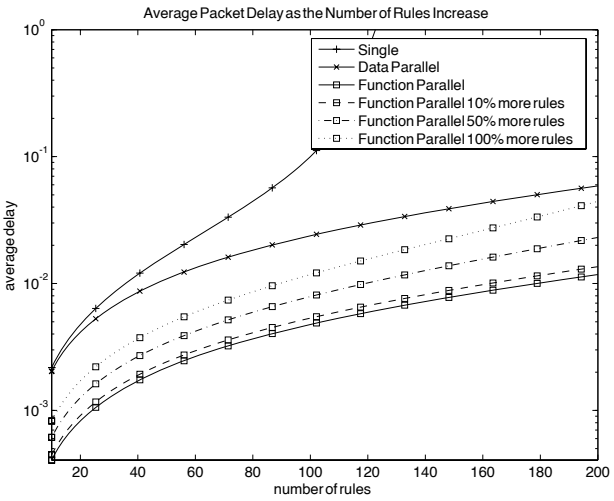


Figure 22: Average packet delay for the parallel designs, each consisting of four firewalls, as the number of rules increases.

ging and generating alerts. When an alert is generated, it may request a rule change in the firewall system to completely block the associated traffic. This connection between IDS and the firewall creates an IPS. We will describe Snort IDS to better understand how IDS works and to identify possible areas for parallelization.

Snort is a widely used, signature-based IDS, which consists of a packet decoding module, preprocessing model, detection engine, and alert engine. The decoding module associates a packet with a particular protocol. The preprocessor performs different functions, such as flow detection, reassembly, and capturing the packet state for the IDS. These results are given to the detection engine, which, like a firewall, applies a series of rules against the packet stream.

An IDS rule expresses the action to perform on matching packets/streams and has a format similar to that of a firewall rule. For example, consider the following Snort rule.

```
alert udp any any -> 10.1.1.0/24 222  (content:"|00 11 22 33 aa|"; msg:"rpcd request")
```

Each Snort rule consists of three components. The first identifies the action that must be taken if there is a match. The second denotes the primary match criterion (similar to the 5 tuples from a firewall rule). In this example, the match criterion identifies any TCP packet destined for the `10.1.1.0/24` address space and port 222. The third component contains rule options and describes any additional match criteria (for example, patterns in the payload) and parameters for executing the action. In the example above, the Snort would search for the hexadecimal pattern `"00 11 22 33 aa"` in the payload. If it is also a match, the message `"rpcd request"` is generated. Therefore, Snort allows the specification of packet header and payload match criteria.

The processing time associated with intrusion detection can be significant [4]. As both the number of signatures and network line speeds continue to increase, it becomes even more important to develop efficient scalable approaches for IDS. In June 2003 there were just over 1500 signatures in the default Snort ruleset, in late 2006 there were more than 5000 default

signatures. Even with current network speeds, an IDS processing traffic between two 100 Mbps networks must be able to process over 300,000 packets per second [40]. As High Speed Networks (HSNs) such as the United States Department of Energy's UltraScienceNet and the Experimental Science Network that connect sites at 5 Gbps become more prevalent, the need for more efficient intrusion detection systems becomes more pressing.

Parallelism is one technique that may be used to help reduce IDS processing time. Parallelization can occur at different *levels* of the intrusion detection system. For example the entire IDS can be duplicated and arriving packets can be distributed to the various systems. Likewise certain components comprising the system can be duplicated, such as the content matching function. At each level either function parallelism (in which work is distributed) or data parallelism (in which data is distributed) may be employed.

5.8 Implementation, Testing, and Collaborations

The project has also started network security collaboration with Deborah Frincke and Jon McCoy from the DOE Pacific Northwest National Laboratories (PNNL). During this summer the PI will work at PNNL to integrate the optimization techniques and parallel designs into the security infrastructure (PNNL currently utilizes a data parallel design). This is a unique opportunity to apply this parallel firewall research to an actual high-speed network environment.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[2] 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.

[3] E. Al-Shaer and H. Hamed. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.

[4] K. Anagnostakis, S. Antonatos, M. Polychronakis, and E. Markatos. E^2xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of IFIP International Information Security Conference (SEC'03)*, 2003.

[5] S. M. Bellovin and W. Cheswick. Network Firewalls. *IEEE Communications Magazine*, pages 50–57, Sept. 1994.

[6] C. Benecke. A Parallel Packet Screen for High Speed Networks. In *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999.

[7] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. John Wiley and Sons, Inc., 1998.

[8] 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.

[9] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1999.

[10] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next-Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1), February 2000.

[11] U. Ellermann and C. Benecke. Firewalls for ATM Networks. In *Proceedings of INFOSEC'COM*, 1998.

[12] R. J. Farley and E. W. Fulp. Effects of Processing Delay on Function-Parallel Architecture Network Firewalls. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, 2006.

[13] E. W. Fulp. Optimization of Network Firewall Policies Using Directed Acyclical Graphs. In *Proceedings of the IEEE Internet Management Conference (IM'05)*, 2005.

[14] 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.

[15] 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.

[16] E. W. Fulp, M. R. Horvath, and C. Kopek. Managing Security Policies for High-Speed Function Parallel Firewalls. In *Proceedings of the SPIE International Symposium on High Capacity Optical Networks and Enabling Technology*, 2006.

[17] E. W. Fulp and S. J. Tarsa. Network Firewall Policy Tries. Technical Report 20049, Wake Forest University Computer Science Department, 2004.

[18] E. W. Fulp and S. J. Tarsa. Network Firewall Policy Representation Using Ordered Sets and Tries. In *Proceedings of the IEEE International Symposium on Computer Communications (ISCC'05)*, 2005.

[19] 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.

[20] 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.

[21] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimizing and Approximation in Determinstic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287 – 326, 1979.

[22] A. Hari, S. Suri, and G. Parulkar. Detecting and Resolving Packet Filter Conflicts. In *Proceedings of IEEE INFOCOM*, pages 1203–1212, 2000.

[23] E. Horowitz, S. Sahni, and D. Mehta. *Fundamentals of Data Structures in C++*. Compuer Science Press, 1995.

[24] E. L. Lawler. Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints. *Annals of Discrete Mathematics*, 2:75 – 90, 1978.

[25] 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.

[26] J. K. Lenstra and A. H. G. R. Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22 – 35, 1978.

[27] 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.

[28] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1999.

[29] L. Qui, G. Varghese, and S. Suri. Fast Firewall Implementations for Software and Hardware-Based Routers. In *Proceedings of ACM SIGMETRICS*, June 2001.

[30] 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.

[31] R. Rivest. On Self-Organizing Sequential Search Heuristics. *Communications of the ACM*, 19(2), 1976.

[32] M. Schwartz. *Telecommunication Networks: Protocols, Modeling, and Analysis*. Addison-Wesley, 1987.

[33] W. E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, 3:59 – 66, 1956.

[34] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proceedings of ACM SIGCOMM*, pages 191–202, 1998.

[35] S. Suri and G. Varghese. Packet Filtering in High Speed Networks. In *Proceedings of the Symposium on Discrete Algorithms*, pages 969 – 970, 1999.

[36] S. J. Tarsa and E. W. Fulp. Balancing Trie-Based Policy Representations for Network Firewalls. In *Proceedings of the IEEE International Symposium on Computer Communications (ISCC'06)*, 2006.

[37] P. S. Wheeler and E. W. Fulp. A Taxonomy of Parallel Techniques for Intrusion Detection. In *Proceedings of the ACMSE, Special Session on Computer and Network Security*, 2007.

[38] A. Wool. A Quantitative Study of Firewall Configuration Errors. *IEEE Computer*, 37(6):62 –67, June 2004.

[39] J. Xu and M. Singhal. Design and Evaluation of a High-Performance ATM Firewall Switch and Its Applications. *IEEE Journal on Selected Areas in Communications*, 17(6):1190–1200, June 1999.

[40] R. L. Ziegler. *Linux Firewalls*. New Riders, second edition, 2002.

[41] E. D. Zwicky, S. Cooper, and D. B. Chapman. *Building Internet Firewalls*. O'Reilly, 2000.