Balancing Trie-Based Policy Representatons for Network Firewalls*

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

Abstract

Firewalls inspect arriving packets according to a security policy. The complexity of these policies can cause significant delays in the processing of packets, resulting in degraded performance, traffic bottlenecks, and ultimately violating Quality of Service (QoS) constraints. As network capacities continue to increase, the improvement of firewall performance is a main concern.

One technique that dramatically reduces required processing is the representation of security policies in software with n-ary tries. This paper describes trie balancing methods that further improve performance by placing more frequently used rules in high precedence positions which require fewer tuple comparisons. A proof of sorted trie integrity is presented along with experimental results showing that on average, sorting reduces the number of comparisons by 27% as compared to the original trie and by 83% as compared to a list representation. Sorting methods are described in detail and their benefits are demonstrated empirically.

1. Introduction

Firewalls provide security by applying a security policy to arriving packets. A policy is a list of rules which define an action to perform on matching packets, such as accept or deny [11]. Determining the appropriate action is typically done in a first-match fashion, dictated by the first matching rule appearing in the policy. Unfortunately, the time required to process packets increases as policies grow larger and more complex. In high-speed environments firewalls quickly become bottlenecks and are susceptible to DoS attacks [1, 5, 6]. Under these conditions, attackers simply inundate the firewall with packets, delaying or preventing legitimate traffic from being processed.

Given these threats, research is focused on reducing packet processing delay by improving either firewall hardware or software. Hardware-based solutions include the application of specialized systems and parallelization techniques [1, 3, 8]. Though beneficial, hardware improvements require costly upgrades and are not applicable to legacy systems. Optimizing firewall software yields performance improvements without such costs.

Firewall software improvements include policy optimization and efficient policy implementations. Both methods reduce the number of comparisons per packet that are required to determine an appropriate match. List reordering is one such method that places more popular rules near the beginning of the policy and maintains policy integrity [2]. Although determining the optimal rule order is $\mathcal{NP}$-hard, reordering can be accomplished using rule match probabilities and a policy-DAG that ensures integrity [2]. This has the effect of lowering the average number of comparisons made per packet by making it more likely that a packet will match rules near the beginning of the list. Reordering the policy rules cannot improve the worst case number of comparisons, which is always $k \cdot n$ where k is the number of rule tuples and n is the number of rules.

Another software improvement is the use of non-list policy representations such as search or retrieval trees (tries) that have the advantage of improving both the average and worst case number of comparisons [9, 10]. In [4], a new n-ary trie representation was developed for firewall policies in which rules are grouped together based on similar tuples. Grouping rules in this fashion eliminates redundant comparisons. Unlike traditional list implementations, it also allows consideration of multiple rules at once. The worst case performance of these structures was proven and experimentally demonstrated to be significantly lower than an equivalent

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

list-representation, with a bound of $1/k$ number of comparisons in the worst case. In contrast to other non-linear representations, the policy trie maintains policy integrity, which indicates the policy trie and list always arrive at the same result for any given packet.

Although the policy trie has several advantages, it does not take into account traffic analysis. While rule sorting provides some benefit, a trie's natural specific-to-general construction prohibits reordering rules of different specificities, limiting the overall effectiveness of sorting. This paper incorporates the use of policy-DAGs to guarantee the security integrity of policy tries. We show that the specific-to-general specification can by altered with no compromise of integrity given certain conditions determined by the DAG. This allows us to sort a far greater number of rules resulting in significant performance gains given varying traffic conditions. Simulation results show the trie sorting method can reduce the number of comparisons by over 27% as compared to the original trie and by 83% as compared to an equivalent list representation.

Section 2 discusses the formal modeling of security policies. Section 3 describes sorting list-based policies. Trie-based representations are discussed in section 4, while trie integrity and sorting methods are described in section 5. Section 6 presents experimental results and section 7 considers future research directions.

2. Firewall Policy Models

A firewall policy consists of an ordered list of rules, often implemented with a linked list. Based on packet header information, each rule defines a subset of network traffic and an action to be carried out on packets in this set. Rules are modeled as an ordered tuple, $r = (r[1], r[2], ..., r[k])$. Each tuple represents possible values of packet header field. Internet traffic is commonly defined by five fields, Protocol, Source Address, Source Port, Destination Address, and Destination Port. These fields can be fully specified or contain wildcards '$*$' in standard prefix format. Each tuple represents a finite set of values; therefore, the set of all possible packets is finite. Packets are also viewed as k-tuples, $d = (d[1], d[2], ..., d[k])$, however each tuple must be fully specified.

As packets pass through a firewall, their header information is sequentially compared to the fields of a rule. If the packet is a subset of a rule, it is said to be a match and the associated action, to accept or reject, is performed. Otherwise, the packet is compared to the next sequential rule. A default rule, or catch-all, is often placed at the end of a policy with action reject. The addition of a default rule makes a policy comprehensive, indicating that every packet will match at least one rule. In the event that a packet matches multiple rules, the action of the first matching rule is taken. Policies that employ this form of short-circuit evaluation are called first-match policies and account for the majority of firewall implementations, including Netfilter's `iptables`.

Using the rule model, a policy R is represented by an ordered list of n rules: $R = \{r_1, r_2, ..., r_n\}$. Two policies, R and R', are considered equal if the policies define the same action for every packet d. In this case, policy R' is said to maintain the policy integrity of R. Sets of packets may match multiple rules, resulting in subset-superset relationships. In first-match policies, this forms the basis of precedence relations that must be preserved in order to maintain policy integrity.

2.1. Modeling Precedence using Policy-DAGs

The precedence relationships between rules in a policy can be modeled with a Directed Acyclical Graph (DAG) [7]. Let $G = (R, E)$ be a policy-DAG for a policy R, where vertices are rules and edges E are the precedence relationships. Each edge introduces a constraint into the graph. 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. 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 from the intesection operation is the empty set. Note the intersection operation is symmetric: if r_i intersects r_j, then r_j will intersect r_i. The same is true for rules that do not intersect.

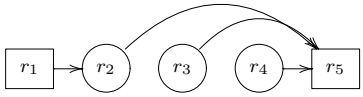
Using the policy-DAG to maintain integrity, a linear arrangement is sought that improves firewall performance. As depicted in figure 1(b), a linear arrangement is a list of DAG vertices where all the successors of a vertex appear in sequence after that vertex. It follows that these arrangements of a policy-DAG represent a rule order, obtained by reading the vertices from left to right. This method of generating rule orders with a DAG has been proven to maintain integrity in [2].

3. List-Based Policy Sorting

Given increasing network speeds and QoS requirements, it is critical that a firewall quickly inspects packets. Reordering policies to minimize the number of comparisons required to filter a packet requires information not present in the policy itself. Since some firewall rules have a higher probability of matching packets than others, it is possible to develop a distribution of the expected number of comparisons per rule, called a *policy profile*. Over time, the frequency of rule matches is collected, similar to a cache

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

(a) Security policy.



(b) Policy-DAG.

Figure 1. Example security policy and associated policy-DAG.

hit ratio. We define $\{p_1, p_2, ..., p_n\}$ to be the policy profile, where p_i is the probability that a packet will match rule r_i first. The average number of rule comparisons required is $E[n] = \sum_{i=1}^{n} i \cdot p_i$ and can be used to compare different rule orderings.

Determining the optimal rule order is analogous to job-shop scheduling and is thus $\mathcal{NP}$-hard. However, a simple rule sorting heuristic was introduced in [2]. This algorithm starts with the original policy and sorts neighboring rules based on non-increasing probabilities. The exchange of neighbors is governed by the constraints of a policy DAG, preserving precedence relationships. This technique reduces the average number of comparisons but does not improve worst case performance, which is $k \cdot n$ tuple comparisons. To address worst case performance, an important benchmark of a firewall's efficiency, new policy representations are needed.

4. Policy Tries

In [4], a non-linear structure called a policy trie was developed to represent firewall security policies. This hierarchical structure has the ability to improve both the average and worst case number of comparisons required to process a packet. The policy trie T is a n-ary trie consisting of k levels that store a security policy. Each level corresponds to a rule tuple, except for the root. Nodes on each level store tuple values from each rule. Unlike the standard binary trie structure [7], a node must be able to have multiple children in order to store a rule tuple rather than a single bit field as in [10]. Tuples at each level are organized from specific to general. The trie construction takes into account precedence relationships, introducing rule intersections to ensure that the trie is equivalent to the original policy [4].

To process a packet d using the policy trie T (also referred to as searching T), each tuple of the packet is compared with the children at the corresponding level, starting with the root node. The search is conducted with a preorder backtracking traversal. Consequently, the policy is evaluated from specific to general. Once a complete root to leaf path is found, the associated action is performed.

4.1. Push-Down Policy Tries

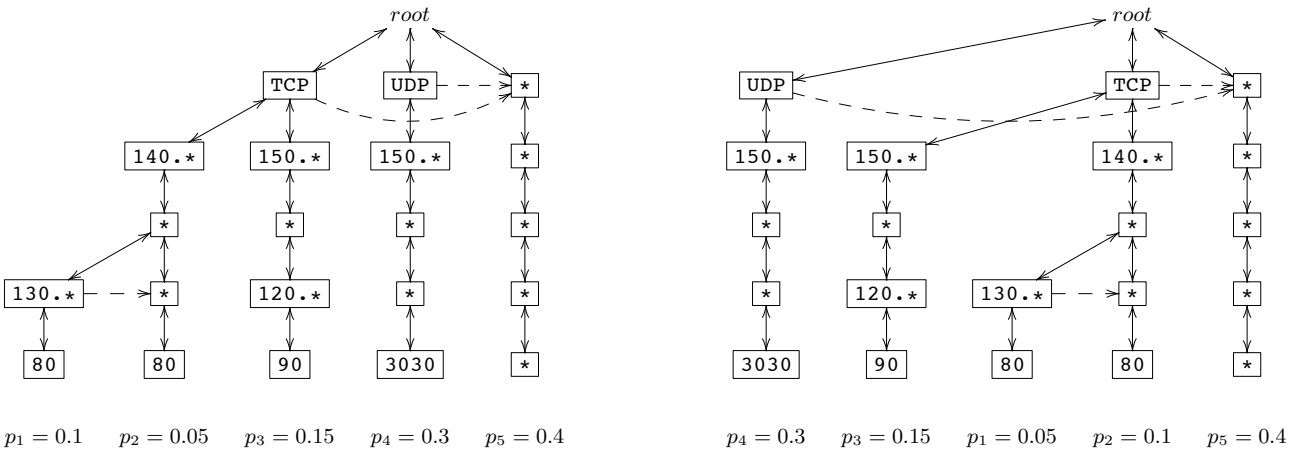
Since policy tries require *backtracking* they have a worst-case performance equal to that of a list representation [9]. Although the penalty for backtracking in an n-ary trie is not as severe as a standard binary version, the conversion to a non-backtracking trie reduces the number of tuple-compares, and significantly improves the worst case bound, an objective of this representation.

A non-backtracking policy trie, referred to as a Push-Down policy Trie (PDT), 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. The push-down procedure replicates more general rules in subset sub-tries 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. This process maintains the integrity of the trie while reducing the number of comparisons to $1/k$ of a list representation in the worst case. The disadvantage of a PDT is the increased storage required, which was proven to be n^2 in the worst case [4].

5. Maintaining Policy Trie Integrity

The policy trie and PDT representations described in the previous sections 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 a wide variety of services and hosts, this severely limits the effectiveness of sorting.

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



(a) Original policy trie.

(b) Sorted policy trie.

Figure 2. Original and sorted trie representations of the firewall rules given in figure 1(a). Dashed edges represent precedence constraints between nodes that share the same parent.

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 2(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 section 5.2.

5.1. 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 the intersecting 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 5.1 *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.

As in the problem of sorting a list-based policy, sorting a policy trie can be viewed as job-shop scheduling [2]. Thus, determining the optimal trie order can be considered $\mathcal{NP}$-hard.

5.2. A Trie Sorting Algorithm

Using the guidelines for maintaining integrity described in the previous section, figure 3 presents a simple sorting technique for policy tries or PDTs. The algorithm is invoked at the root and recursively sorts subtries by exchanging neighbors based on match-probability and the number of branches. Probabilities for nodes that are the result of an intersection operation are assigned a value less than either of the two original rules, as intersections define a smaller set of packets than either parent. For all other sub-tries the 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 2(b) depicts the same policy trie before and after sorting. A trace of this example case has been omitted due to space limitations. 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.

6. 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

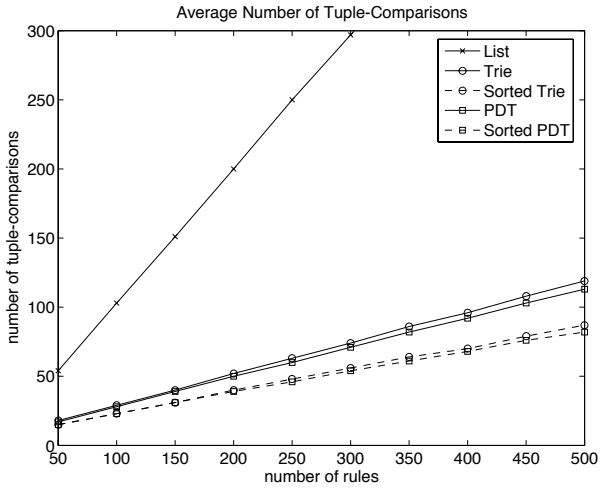
```
function sortTrie(trieNode m)
    if(m is leaf node) return
    done = false
    while(!done)
        done = true
        for each child i of m that has a right neighboring sibling
            if(P(i) < P(i+1) AND (T_i ⋔ T_{i+1} OR
            action(T_i)==action(T_{i+1}))then
                exchange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

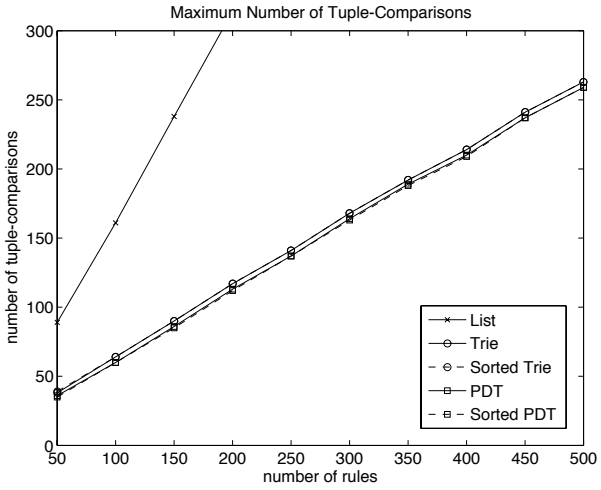
and skewed to favor random subsets of rules over others. Of note, our traffic generation algorithm mimics high flow to a small set of services, with no regard for their placement in the policy. This is in contrast to the models used in [4], which sought to produce DoS traffic that exploited the structure 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 highest number of tuple-comparisons are given in figure 4. 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 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



(a) Average number of tuple-comparisons.



(b) Highest number of tuple-comparisons.

Figure 4. The average and worst case number of tuple-comparisons required.

elsewhere in the PDT unless needed. This effectively allows certain rules to be excluded from most evaluations even in worst-case situations.

7. Conclusions

A firewall's fundamental operation is a single tuple comparison. The delay associated with processing a packet is directly proportional to the number of comparisons made by the firewall. A policy trie is a non-linear policy representation that reduces the number of comparisons required to determine the appropriate rule match [4]. Uniquely, policy tries are also proven to maintain policy integrity.

This paper improved the average case performance of policy tries using sorting techniques. By organizing the trie such that more popular rules appear to the left in positions reached with fewer tuple comparisons, processing overhead was reduced significantly. Since precedence relationships prevent arbitrary branch reordering, a policy-DAG was used to guarantee that any such operations did not violate policy integrity. Simulations indicate that this technique reduces the average number of comparisons by 27% and 83% over unsorted trie and list implementations, respectively. Therefore, sorting is a simple method for improving performance in trie-based firewalls.

Areas for furture research include faster sorting algorithms for on-line application. Security can also be enhanced with connection state and packet audit information. More research is needed to determine more accurate probability distributions for packet matching and dependency percentages. Given this information, better algorithms can be designed and tested with more realistic simulations.

References

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

[2] E. W. Fulp. Optimization of network firewall policies using directed acyclical graphs. In *Proceedings of the IEEE Internet Management Conference (IM'05)*, 2005.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] L. Qui, G. Varghese, and S. Suri. Fast firewall implementations for software and hardware-based routers. In *Proceedings of ACM SIGMETRICS*, June 2001.

[10] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM*, pages 191–202, 1998.

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