# Trie-Based Policy Representations for Network Firewalls*

Errin W. Fulp and Stephen J. Tarsa
Wake Forest University
Department of Computer Science
Winston-Salem, NC, USA
nsg.cs.wfu.edu
{fulp|tarssj2}@wfu.edu

## Abstract

*Network firewalls remain the forefront defense for most computer systems. These critical devices filter traffic by comparing arriving packets to a list of rules, or security policy, in a sequential manner. Unfortunately packet filtering in this fashion can result in significant traffic delays, which is problematic for applications that require strict Quality of Service (QoS) guarantees. Given this demanding environment, new methods are needed to increase network firewall performance.*

*This paper introduces a new technique for representing a security policy that maintains policy integrity and provides more efficient processing. The policy is represented as an n-ary retrieval tree, also referred to as a trie. The worst case processing requirement for the policy trie is a fraction compared a list representation, which only considers rules individually (1/5 the processing for TCP/IP networks). Furthermore unlike other representations, the n-ary trie developed in this paper can be proven to maintain policy integrity. The creation of policy trie structures is discussed in detail and their performance benefits are described theoretically and validated empirically.*

## 1. Introduction

The benefits of highly interconnected computer networks have been accompanied by an increase in network-based security attacks. As a result, firewalls, also referred to as packet filters, have become critical components of network security systems. Firewalls provide access control, auditing, and traffic control based on a security policy by inspecting packets sent between networks [3, 23]. The security policy is an ordered list of rules that defines an action to perform on arriving or departing packets. When a packet arrives at the firewall it is sequentially compared against the rules until a match is found [23]. This is referred to as a *first-match* policy and is used in the majority of firewall systems including the Linux firewall implementation `iptables` [19]. Once the match is determined the associated action is performed and the packet is either accepted or denied.

Firewalls are often implemented as a dedicated machine, similar to a router. Unfortunately, packet filtering requires a significantly higher amount of processing time than routing [18, 21]. Processing time increases as rule sets increase in length and complexity [4, 17]. As a result, a firewall in a high-speed environment (e.g. Gigabit Ethernet) can easily become a bottleneck and is susceptible to DoS attacks [4, 8, 13, 14]. These attacks merely inundate firewalls with traffic, delaying or preventing legitimate packets from being processed. Methods of improving firewall performance are needed for the next generation of high-speed networks and security threats.

Improving hardware is one method for increasing the amount of traffic a firewall can process. Current research is investigating different distributed firewall designs to reduce processing delay [4, 17] and possibly provide service differentiation [10]. While performance can increase using these methods, it requires multiple machines and/or specialized hardware. As a result these improvements are not amenable to legacy systems and thus do not provide a solution to many systems.

Improving software is another method to increase firewall performance that is applicable to a larger set of

*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 authors 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.

systems [5, 16, 18]. Similar to approaches that address the longest matching prefix problem for packet classification [6, 7, 9, 19, 20], these solutions employ better policy representations and searching algorithms. For example, retrieval trees (tries) offer quick search times and have been utilized to decrease the packet processing time [18, 20]. These policy models use the classical definition of a trie structure, which is a variation of a binary tree [1]. While this method groups rules in an efficient manner, the firewall tuples are stored in a binary format (one bit per branch) that increases the processing overhead and is difficult to implement (ultimately requiring a grid of binary tries) [7]. Furthermore, these binary trie structures are designed to determine the longest matching prefix, which results in the *best-match* rule (not typically used in network firewalls). As described in [20] first-match is possible, however it requires additional information and comparisons to rank possible rules.

Another representation used Directed Acyclical Graphs (DAG's) to store packet header fields (multibit field) in [6]. This structure was shown to efficiently store filter rules for layer four switching. However, the DAG structure may have difficulty maintaining integrity if partial-matching rules exist. Trees have also been successfully used to model firewall policies in [2, 15]; however, the primary purpose of this research was locating rule conflicts and anomalies, not improving processing time.

This paper introduces a new security policy representation called a policy trie that is readily implemented and significantly reduces the packet processing time. Rules are represented as an ordered set of tuples, maintaining precedence relationships among rules and ensuring policy integrity (policy trie and list always arrive at the same result). The policy is modeled as an $n$-ary retrieval tree (trie), uniquely combining the retrieval efficiency of a trie and the flexibility of an $n$-ary tree.

When the policy trie is created rules are grouped by tuples (parts of the rule), allowing the elimination of multiple rules as a packet is processed and the trie is traversed. This is in contrast to the traditional list representation that can only consider one rule at a time. As a result, it will be proven that the policy trie has a worst case performance that is a fraction of a list representation ($1/k$, where $k$ is the number of tuples). Furthermore unlike other representations, the policy trie maintains policy integrity; therefore this structure can easily and effectively represent current security policies. These theoretical results are verified via simulation under realistic conditions.

The remainder of this paper is organized as follows:

| | | Source | | Destination | | |
|---|---|---|---|---|---|---|
| **No.** | **Proto.** | **IP** | **Port** | **IP** | **Port** | **Action** |
| 1 | TCP | 140.* | * | 130.* | 20 | accept |
| 2 | TCP | 140.* | * | * | 80 | accept |
| 3 | TCP | 150.* | * | 120.* | 90 | accept |
| 4 | UDP | 150.* | * | * | 3030 | accept |
| 5 | * | * | * | * | * | deny |

**Table 1. Example TCP/IP security policy consisting of multiple ordered rules.**

Section 2 describes the models for firewall rules and a standard (list-based) security policy. The new policy representation called a policy trie is introduced in section 3. Methods for creating and searching the policy trie, and proofs for policy integrity and performance are described. Detailed proofs are provided in [12]. The performance of the policy trie is investigated experimentally under realistic conditions in section 4. Finally, section 5 summarizes the policy trie representation and discusses some areas of future research.

## 2. Firewall security policies

As previously described, a firewall security policy has been traditionally defined as an ordered list of firewall rules [23], as seen in table 1. A rule $r$ can be viewed as an ordered tuple of sets [22], for example rule $r = (r[1], r[2], ..., r[k])$. Each tuple $r[l]$ can be fully specified or contain wildcards '*' in standard prefix format. For example the prefix 192.* would represent any IP address that has 192 as the first dotted-decimal number. For TCP/IP networks, rules are represented as a 5-tuple as seen in table 1. The tuples for TCP/IP are: protocol, IP source address, source port number, IP destination address, and destination port number. Order is necessary among the tuples since comparing rules and packets requires the comparison of corresponding tuples. In addition to the prefixes, each firewall rule has an action, which is to accept or deny. An accept action passes the packet into or from the secure network, while deny causes the packet to be discarded. Using the rule definition, a standard **security policy** can be modeled as an ordered set (list) of $n$ rules, denoted as $R = \{r_1, r_2, ..., r_n\}$.

Similar to a firewall rule, a packet (IP datagram) $d$ can be viewed as an ordered $k$-tuple $d = (d[1], d[2], ..., d[k])$; however, wildcards are not possible for any packet tuple. An arriving 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 utilized by the majority of firewall systems [19]. A match is found between a packet and rule when every tuple of the packet is a proper subset of the corresponding tuple in the rule.

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

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

For example the packet $d =$ `(TCP, 140.1.1.1, 90, 130.1.1.1, 20)` would match the first and the fifth rules in table 1. However using the first match policy, the packet would be accepted since the first rule is the first match. Although the two rules match the same packet and have different actions, it is important to note that this is **not** considered an anomaly.

*Shadowing* is a type of anomaly, which occurs when a rule $r_{i+k}$ matches a preceding rule $r_i$, thus rendering $r_{i+k}$ obsolete. For example, assume the rule `(UDP, 130.*, *, *, *, accept)` was added to the end of the policy given table 1. This new rule is shadowed by the fifth rule ($r_6 \Rightarrow r_5$); thus, the new rule will never be utilized. Security policy anomaly detection and correction is the subject of continued research [2, 15, 23] and is not the focus of this paper. Therefore, this paper will assume such filter conflicts are not present in the security policies.

A security policy $R$ is considered comprehensive if for every possible legal packet $d$ a match is found using $R$. The policy given in table 1 is comprehensive due to the last rule. Furthermore, we will say two rule lists $R$ and $R'$ are equivalent if for every possible legal packet $d$ the same action is performed by the two rule lists [11]. This definition will be extended to include different policy representations. As described in the introduction, this paper is interested in improving network firewall performance. Firewall performance will be measured using the number of *tuple-comparisons* required to find the first match. The worst case performance for a list-based representation is $k \cdot n$ tuple-compares, which occurs when the last rule (default rule) is the first match.

## 3. A new security policy representation

In this section a new security policy representation called the *policy trie* is introduced, 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

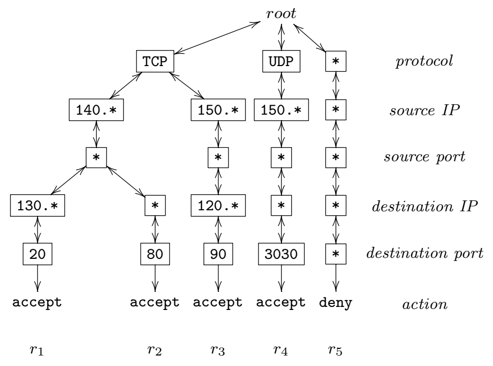


**Figure 1. Policy trie representation of the firewall rules given in table 1.**

tree. This is required since a node will store a rule tuple (multibit field), not just a single bit as done in [20]. 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 1.

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) 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 1. 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 1, 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 reorder and may result in a shadowing. The intersection of the new rule and each of these right-most rules is taken [11] 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 2(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 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 2(b), which maintains the integrity of the policy given in 2(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* [2].

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.

## 3.1. 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 [12].

**Theorem 3.1** *A policy trie $T$ is equivalent to the original security policy $R$.*

## 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 [18]. 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 [18]. 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.

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

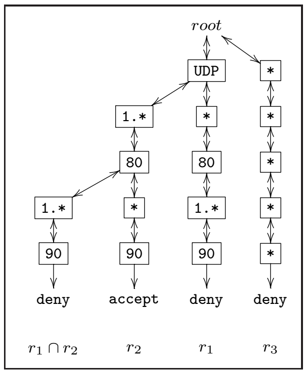
$$r_g \downarrow r_s = (r_s[1, .., l], r_g[(l+1), ..., k], r_g[action])$$

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 3. 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.

| 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 2. List and trie representation of a security policy where the policy trie requires the intersection operation to maintain the integrity of the list.**

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 3.2 states 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 stated to do so in theorem 3.1 [12].

**Theorem 3.2** *A push-down policy trie $T_p$ is equivalent to the original policy trie $T$, which is equivalent to the original security policy $R$.*

## 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 [12].

**Lemma 3.3** *A node in a push-down policy trie cannot have more than $n$ (the number of rules) children.*

**Lemma 3.4** *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 stated in theorem 3.5. 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 [12].

**Theorem 3.5** *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.*

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 3 as compared to the original trie given in figure 1. 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. However, the number of nodes required by the trie can be greatly reduced by converting it into a Directed Acyclical Graph (DAG) [1, 18]. 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 3. 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. The

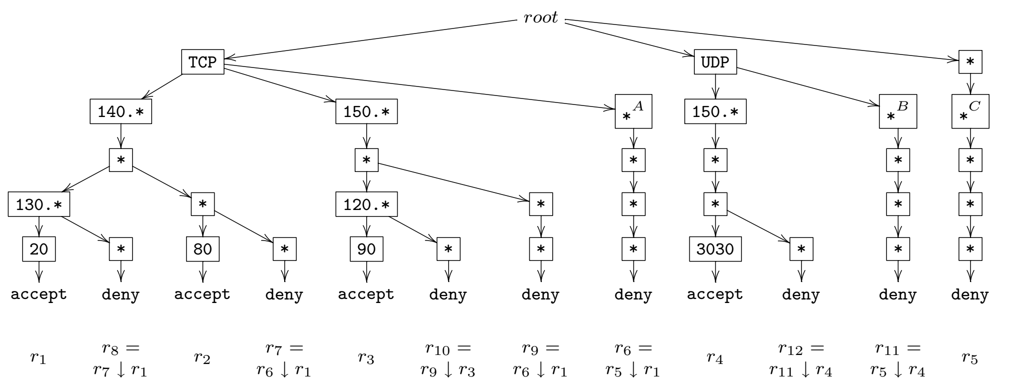


**Figure 3. Push-down policy trie representation of the firewall rules given in table 1.**

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. The proof has been omitted due to space limitations [12].

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 $O(n^2)$. Again, the proof has been omitted due to space limitations. 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).

## 4. Experimental results

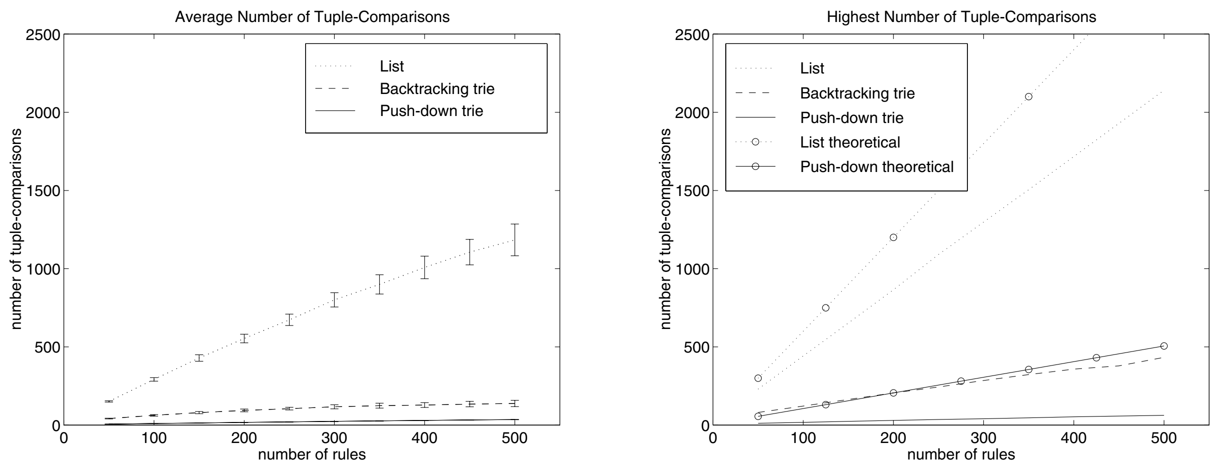
The previous section described a new network security policy representation called a policy trie that was shown to provide theoretically better performance than the standard list-based representation. Simulation results presented in this section will confirm the worst case number of tuple-comparisons 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 remain well below 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

### 4.1. Tuple-comparisons results

Results for the tuple-comparisons are given in figure 4. 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 4(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



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

(b) Highest number of tuple-comparisons.

**Figure 4. The average and worst case number of tuple-comparisons required for the firewall experiments. Push-down trie provided the best performance in both cases.**

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 4(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.

### 4.2. Storage results

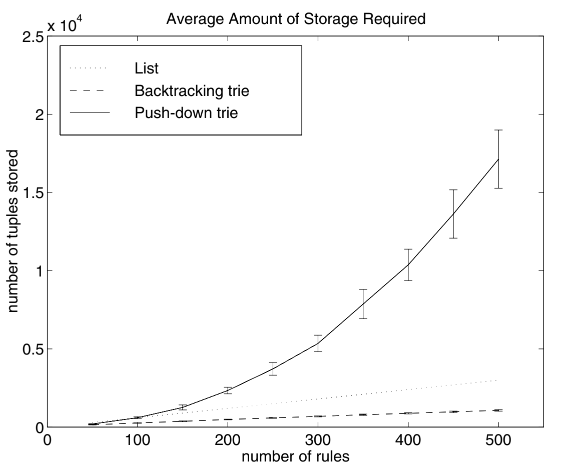
The amount of storage required, measured in the number of tuples, is depicted in figure 5. As predicted, when policy sizes increased the backtracking tries consistently used less (a third for these experiments) of the storage required by the list representation. In contrast, the push-down tries required 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.

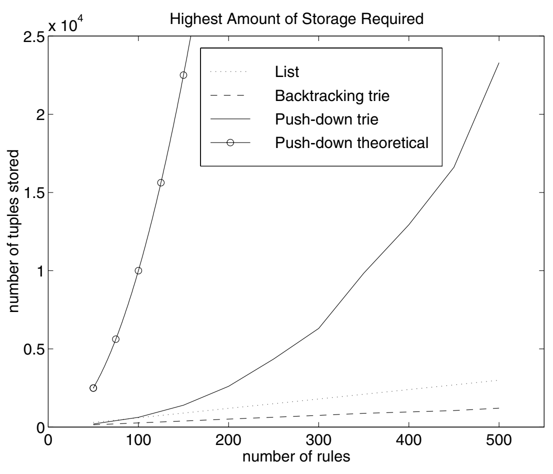
## 5. Conclusions

Network firewalls must continue to match, or exceed, the ever increasing speed and volume of network traffic if they are to remain effective. Unfortunately, the traditional single machine firewall that utilizes a list-based security policy can easily become overwhelmed in this environment. Therefore, new methods to increase firewall performance are needed to meet the demands of the next generation of networks and applications.

This paper introduced a new firewall security policy representation called the policy trie that maintains policy integrity while requiring significantly less processing time. Rules are modeled as an ordered set of tuples, allowing the precedence relationship between rules to be maintained. The security policy is represented as



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



(b) Highest number of tuples stored.

**Figure 5. The average and worst case number of tuple stored for the firewall experiments. Backtracking trie required the least amount of storage in both cases.**

an $n$-ary retrieval tree (trie), which combines the fast search properties of a trie with the flexibility of a $n$-ary tree. The policy trie stores similar rules together, which the allows the elimination of multiples rules as a packet is processed. This is in contrast to a standard list-based representation, which can only consider rules individually. This yields a worst case performance for the policy trie that is only $1/k$ of a list representation, where $k$ is the number of tuples in a rule. Furthermore unlike other representations, the policy trie maintains the integrity of the original policy. The integrity and performance improvement of the policy trie was proven theoretically and demonstrated using simulation under realistic conditions.

While the policy trie has shown great promise, more research is needed to evaluate its ability to manage a dynamic policy. In many cases, rules must be added and removed over time to reflect the current security status (for example, stateful firewalls used for connection tracking [23]). A simple solution would just recreate the trie from the corresponding list-based policy once a rule is removed. Another approach would tag rules if they are the result of intersection or push-down operations, which allows their quick removal if the corresponding original rule is removed. Another area of research would investigate the average performance of the firewall system. Possible methods for improving the average case would involve sorting the trie nodes based on match probabilities or reorganizing the tuple order of the trie. This must be done while maintaining the integrity of the policy, which is not a trivial problem.

## 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. Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.

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

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

[5] M. Christiansen and E. Fleury. Using interval decision diagrams for packet filtering. Technical report, BRICS, 2002.

[6] 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.

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

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

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

[10] E. W. Fulp. Firewall architectures for high speed networks. Technical Report 20026, Wake Forest University Computer Science Department, 2002.

[11] E. W. Fulp. Optimization of firewall policies using directed acyclical graphs. In *Proceedings of the IEEE Internet Management Conference*, 2005.

[12] E. W. Fulp and S. J. Tarsa. Network firewall policy tries. Technical Report 20049, Wake Forest University Computer Science Department, 2004.

[13] 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.

[14] 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.

[15] A. Hari, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE INFOCOM*, pages 1203–1212, 2000.

[16] HiPAC. High performance packet classification. `http://www.hipac.org`.

[17] 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.

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

[19] 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.

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

[21] S. Suri and G. Varghese. Packet filtering in high speed networks. In *Proceedings of the Symposium on Discrete Algorithms*, pages 969 – 970, 1999.

[22] P. W. Zehna. *Sets with Applications*. Allyn and Bacon, 1966.

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