OPTWALL: A Hierarchical Traffic-Aware Firewall

Subrata Acharya[†], Mehmud Abliz[†], Bryan Mills[†], Taieb F. Znati[†,‖]
[†]Department of Computer Science,
[‖]Telecommunications Program
University of Pittsburgh,
Pittsburgh, PA 15260
(sacharya, mehmud, bmills, znati)@cs.pitt.edu

Jia Wang[§], Zihui Ge[§], Albert Greenberg[§]
[§]AT&T Labs Research,
Florham Park, NJ 07932
(jiawang, gezihui, albert)@research.att.com

***Abstract*— *The overall efficiency, reliability, and availability of a firewall is crucial in enforcing and administrating security, especially when the network is under attack. The continuous growth of the Internet, coupled with the increasing sophistication of the attacks, is placing stringent demands on firewall performance. These challenges require new designs, architecture and algorithms to optimize firewalls. In this paper, we propose OPTWALL, an adaptive hierarchical firewall optimization framework aimed at reducing operational cost of firewalls. The main features of the proposed approach are the hierarchical design, splitting techniques, an online traffic adaptation mechanism, and a strong reactive scheme to counter malicious attacks (e.g. Denial-of-Service (DoS) attacks). To the best of our knowledge, this work is the first of its kind to use traffic characteristics in the design of an adaptive hierarchical firewall optimization framework. To study the performance of OPTWALL, a set of experiments are conducted on Linux ipchains. The performance evaluation study uses a large set of firewall policies and traffic traces managed by a Tier-1 ISP and provides security access for the ISP network from/to its business partners. Results show the high potential of OPTWALL to reduce the operational cost of firewalls. In particular, the results show that a performance improvement of nearly 35% can been achieved in a heavily loaded network environment.*

I. Introduction

The constantly changing nature, scale and scope of information technology environments, coupled with the increasing number and complexity of security threats, is forcing Tier-1 ISPs to resort to increasingly complex security policies and mechanisms. Firewalls constitute the cornerstone of most network defense systems and have proven to be an effective solution to monitor and regulate traffic. The efficiency of firewalls in protecting the infrastructure, however, depends not only on the integrity and coherence of the security policies they are configured to implement, but equally importantly on the speed at which these policies are enforced.

With the dynamic change in the network load, topology, and bandwidth demand, firewalls are becoming a bottleneck. All these factors create a demand for more efficient, highly available, and reliable firewalls. Optimizing firewalls, however, remains a challenge for network designers and administrators.

A typical present day firewall enforces its security policies *via* a set of multi-dimensional packet filters (rules). Optimization of this multi-dimensional structure has been proven to be a NP hard [1], [2] problem. This has motivated the research community to focus on various approaches to provide reliable and dependable firewall optimization methods. In-spite of a strong focus towards an efficient design, the techniques used thus far are static, and fail to adapt to the dynamic traffic changes of the network. In particular, current techniques have failed to include the traffic characteristics in the design and optimization of firewalls. Current firewall designs do not support adaptive mechanism to detect and counter attacks in a network environment characterized by heavy traffic fluctuations. Hence, they fail to operate efficiently under adverse conditions.

The main objective of this paper is to address the shortcomings of the current firewalls and increase their ability to deal with dynamic changes in network load and topology, particularly when the network is under attack. To achieve this goal, the paper proposes a hierarchical framework for traffic-aware firewall optimization. The basic tenet of this framework is that the design of next generation firewalls must leverage their packet inspection capabilities with traffic awareness in order to optimize the operational cost they incur in defending against intrusions and denial of service attacks.

Traffic-aware firewall optimization is challenging as the number of security policies a firewall has to enforce for enterprise networks is large. This is further compounded by the limited resources of firewalls relative to the increased ability of the network to process and forward traffic at extremely high speed. In this paper, the focus is on optimizing the most widely used *'list based'* firewalls. To achieve this goal we propose a hierarchical firewall optimization approach, to create a load-balanced policy subset. The main challenge in the construction of these subsets stems from the need to maintain semantic integrity of the policy set at each level of the hierarchy.

The major contributions of the paper are:

- The design of OPTWALL, an adaptive hierarchical firewall optimization framework. In this framework we propose an optimal solution to construct the hierarchy based on rule-splitting, while maintaining the integrity of the original firewall rule set.

- A set of heuristics, based on a trade-off between optimality, time complexity, and resource requirements to convert the list based firewall rule sets into integrity preserving hierarchical rule subsets.
- An adaptive, traffic-aware protocol to detect and defend against traffic anomalies.
- An experimental study to assess the performance of the proposed solutions and measure the impact of dynamically exploiting the traffic characteristics on the performance of firewalls.

The rest of the paper is organized as follows: Section II describes the background on list based firewalls. The rule cost metric is described in Section III and we introduce the OPTWALL framework in Section IV. Section V depicts the OPTWALL splitting design approaches. We present the evaluation and results in Section VI. Section VII presents the related work. Finally, we conclude the paper in Section VIII.

II. List Based Firewalls

A security rule is a multi-dimensional structure, where each dimension is either a set of network fields or an action field. The rule set defines the security policies which must be enforced by the firewall.

In an Internet environment, a rule is defined by a set of *source ip addresses*, a set of *destination ip addresses*, a set of *service types* and an action field. The service type typically includes both the underlying *protocol type* and a *port number*. An action field can be either *accept*, *deny*, or *forward*. An accept action allows the packet access into the protected domain. A deny action causes a packet, in violation of the security policy, to be rejected. Finally, a forward action leads to further inspection of the packet. Formally, a rule R can be represented as: $R = [\Phi^1, \Phi^2, \cdots, \Phi^k; \Sigma]$, where Φ^j represents network fields and Σ is an action field. An instance of a typical rule in an Internet environment can be of the following structure:

$$< src = \{s_1, s_2, \cdots, s_n\};\ dst = \{d_1, d_2, \cdots, d_m\};$$
$$srv = \{\sigma_1, \sigma_2, \cdots, \sigma_l\};$$
$$action = \{drop|accept|forward\} >$$

where s_i represents a source IP address, d_i a destination IP address, and σ_i a service type.

In list-based firewalls, rules describing the network security policies form a "priority" list. The priority of a rule, also referred to as the *rule rank*, is based upon its position within the list. Earlier occurring rules have higher rank than later ones. A rule-set can be converted to a tuple-set by creating all possible permutations of the fields in a rule. A formal representation of a tuple is:

$$< src = s_i;\ dst = d_i;\ srv = \sigma_i;$$
$$action = \{drop|accept|forward\}$$

where s_i represents a source IP address, d_i a destination IP address, and σ_i a service type.

List-based firewalls work by examining the tuples in sequential order. For each packet, the first matching tuple determines the action taken by the firewall. List based firewalls preform well if the list size is small. As the list size scales up to millions of tuples, managing and optimizing the firewall policies is a challenge. Moreover highly dynamic traffic changes introduces the challenge of dynamic rule optimization.

III. Rule Cost Metric

The main factor that affects the performance of a firewall is the processing overhead due to packet inspection. The metric calculation is done for a rule and can easily be applied to tuples within a rule.

To capture the overhead cost incurred by a firewall to process a rule and enforce the security policy, two metrics are defined. The first metric, denoted as *rule_size()*, measures the size of a given rule in terms of the number of bits necessary to determine unambiguously a match between the rule definition and the corresponding fields of a packet under inspection. The assumption underlying the *rule_size()* metric stems from the fact that the complexity of a matching operation is proportional to the size of the rule. Formally, given a rule r, rule_size(r) can be defined as:

$$rule\_size(r) = \begin{cases} \sum_{\mathcal{S}_p, \mathcal{D}_p} \{\alpha_1 \times \|s_p\| + \alpha_2 \times \|d_p\|\} \\ +\beta \times N_s \times (\|Pr_r\| + \|Po_r\|), \end{cases}$$

where, α_1, α_2, and β are weight parameters, $\mathcal{S}_p$ and $\mathcal{D}_p$ are respectively the set of source and destination prefixes which occur within the definition of the rule, s_p and d_p are the bit representation of the source and destination prefixes, respectively, N_s is the number of services defined within the rule, and Pr_r and Po_r are the bit representation of the protocol and port identifiers, respectively.

The second metric used in our experimentation is the cost of operating on a given rule set. This cost depends on the rule's rank and size, and on how often the rule is invoked by the firewall. Formally, given a set of rules $r_1, r_2, \ldots, r_k$, the cost of a given rule, r_i, $cost(r_i)$, is defined as follows:

$$cost(r_i) = \ hit\_count(r_i) \ \times \ \sum_{\forall r_k \in \mathcal{P}r_i} \|r_k\|$$

where, $\mathcal{P}r_i$ is the set of r_i's predecessors in the list-based set of rules.

Using the above metrics, the aim of optimization is to reduce the rule set size and consequently the processing time of the rule set. This in turn reduces the overall firewall operational

cost. The resources that affect are the CPU utilization and the memory usage of the firewall machine.[1]

IV. OPTWALL

Contrary to a list-based structure, a hierarchical design leads to efficient organization of rule sets, thereby increasing significantly the performance of the firewall. OPTWALL uses a hierarchical approach to partition the original ruleset into mutually exclusive subsets of rules to reduce the overhead of packet filtering.

In OPTWALL, the processing of a packet at a firewall starts at the root of the hierarchical structure. The packet is subsequently forwarded to the remaining levels of the hierarchy for further processing. Packet processing completes if a match between the attributes of the packet, as defined by the firewall security policy, occurs. In this case, the *action*, defined by the corresponding firewall rule, is enforced. Alternatively, on a non-match, a default action is invoked. The default action can either be *accept*, in which case the packet is forwarded to destination, or *reject*, in which case the packet is dropped. In the following, a formal specification of the objective and basic operation of OPTWALL are discussed.

A. OPTWALL Design Goals

Given a large rule set, the objective of OPTWALL is to partition this set into *'K'* mutually exclusive subsets. Each subset is associated with a unique filter which represents a superset of the associated policy subset.

The hierarchical approach of the OPTWALL architecture is driven by three main design goals:

1) Reduce the cost of processing the firewall rule set, defined at the average processing time a packet incurs before an action is enforced by the firewall,
2) Preserve the semantics of the original rule set, and
3) Maintain the optimality of the rule set as traffic patterns and rule sets change.

It is to be noted that in its general form the *'K-partition'* problem is NP hard, as it can be reduced to the *'Clustering'* [4] or the *'K-median'* problem [5]. Figure 1 depicts the process of partitioning N rules into K subsets.

To address the complexity of the partitioning problem, OPTWALL uses an iterative approach to partition the original set of rules and produce a multi-level hierarchy of mutually exclusive, cost-balanced rule subsets. Initially, the rule set is divided into two subsets and filters, which covers the rules contained in each subset, are developed. The resulting subsets, along with their corresponding filters, form the first level of the hierarchy. This iterative process continues until further division of the subsets at the current level of the hierarchy is no longer cost effective. Note that this cost also includes the cost of determining the filters. The OPTWALL partitioning process is described in Figure 2.

In the following sections we will present the processes used to achieve each of OPTWALL design goals. We first describe

[1]The metric used in this paper follow the same guideline as in [3].

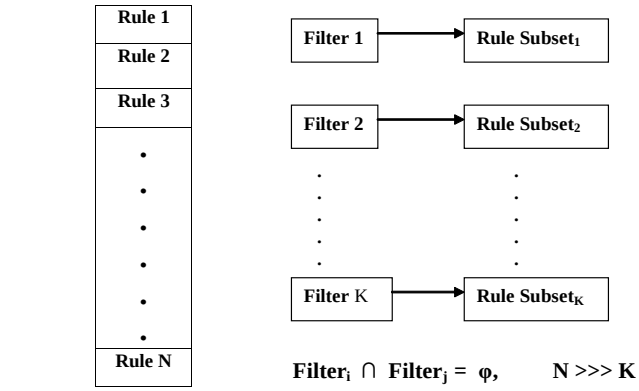

Fig. 1: N rules into K partition problem

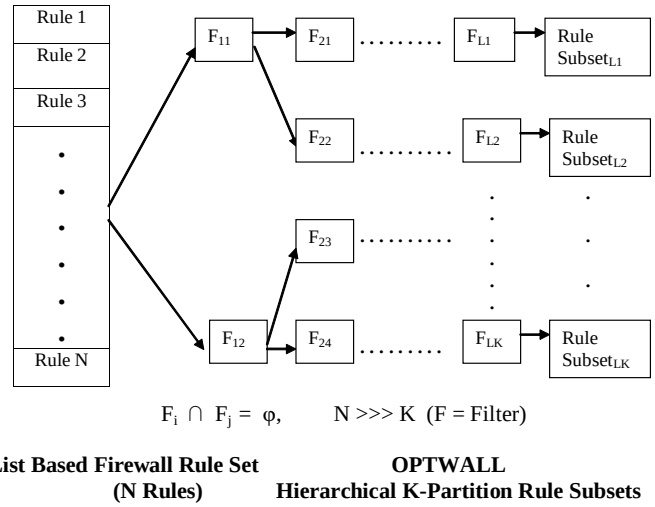

Fig. 2: Basic operation of OPTWALL

the multi-level data structure composed of rule subsets and their corresponding filters. We then discuss the procedure used to build the OPTWALL hierarchical structure and the actions required to maintain this structure.

B. Data Structure

In order to process the rules, OPTWALL uses a hierarchical data structure in which the deepest level of the hierarchy contains the rule subsets and the intermediate levels contain filters which cover the rules included in those subsets.

The design of the data structure must ensure that the operational cost is reduced. The design must also ensure that the semantic integrity of the original rule set is preserved. It is to be noted that the operational cost is determined by the deepest rule subset. Balancing the hierarchical structure in order to reduce the length of the deepest rule subset

is, therefore, vital if the desire is to achieve the maximum reduction in processing cost. Furthermore, the data structure must be designed in such a way that the re-balancing process, in response to traffic changes, can be achieved with minimal overhead.

Semantic integrity of the original rule set can be achieved, during the rule set partitioning process, by computing filters that represent accurately and completely the rule subsets. Furthermore, packet processing must follow the same semantics specified by the filters resulting from the partitioning process. If the rules are split and re-ordered, in order to optimize operational cost, the process of re-enforcing the original rule semantics must be achieved with reduced overheard.

C. Hierarchical Structure Building

The process of building the hierarchical structure described previously is accomplished using three basic stages: *pre-processing*, *ordering*, and *splitting*. In the following, we discuss the basic operations carried out at each of these design stages.

The pre-processing stage takes as its input the original list based rule set and produces an optimized rule set. This optimized rule set consists of fully disjoint and concise rules, where all rule redundancies and dependencies are removed [3]. The fact that the rules in the rule set are mutually disjoint provides OPTWALL with full flexibility to re-order the rules and divide them into rule subsets, without violating the semantics of the original rule set.

In the pre-ordering stage, rules are re-ordered such that the highest cost rules are moved to the top of the rule set. As stated previously, the cost of a rule is based upon the size of the rule and the amount of traffic processed by that rule, as indicated by its hit count. By re-ordering rules the overall cost of processing traffic is reduced.

The goal of the splitting stage is to produce a partition of the original rule set into a set of mutually disjoint rule subsets. This process involves taking the pre-processed rule set and dividing it into rule subsets, each of which is defined by a filter. Each filter is a series of disjoint tuples that fully cover its corresponding rule subset.

To partition the original rule set, OPTWALL uses a multi-step process, whereby it initially splits the original rule set into two subsets. It then recursively runs this splitting process on the subsets produced by the previous stage to generate the next level of the hierarchical structure. This splitting process continues until the overall processing cost overshadows the benefit gained by further splitting the current subsets. When this occurs, the splitting process terminates and the previous level is selected as the feasible optimal depth of the hierarchical structure.

The efficiency of the partitioning process strongly depends on the way the rule subsets are produced at different levels of the hierarchy. Several strategies to produce feasible rule set splitting can be used. These strategies are discussed in SectionV.

The produced hierarchical structure is then converted to a series of IP-table rule subsets. It is to be noted that most list based firewalls, such as *Linux ipchains*, support the ability to forward packets from one list to another for further processing. Consequently, the OPTWALL hierarchical structure can be used to augment the filtering capabilities of list based firewalls.

D. Hierarchical Structure Maintenance

The hierarchical structure is built to reflect the current traffic pattern and rule sets. As the traffic pattern and rule sets change, the hierarchical structure must be updated to maintain its balance. To detect changes, OPTWALL monitors the traffic logs in real-time and adjusts the hit counts. OPTWALL asserts that changes have occurred if the difference between the old and updated hit counts of any rule exceeds a predetermined threshold. This threshold, a tunable parameter, is determined based on the traffic characteristics and the policy set under consideration.

If the need to balance the hierarchical structure rises, OPTWALL uses the existing traffic logs to update the cost of rules in the rule subsets, including rules which have been added to reflect a new security policy. OPTWALL then uses *re-ordering*, *re-splitting*, and *promoting* to re-establish the balance of a hierarchical structure.

Re-ordering consists of re-prioritizing the rule subsets at the deepest level of the hierarchical structure. This process is necessary to take into consideration the impact of traffic variations on the hit count of rules in a given rule set. Re-ordering is triggered when a the different between the current and previous hit counts of a given rule exceeds the threshold.

Re-splitting is invoked when a sub-hierarchical structure becomes out of balance, due to traffic variations. A sub-hierarchical structure is considered to be out-of-balance if the average packet processing cost exceeds a predefined threshold. This process can occur at any level, including the root of the hierarchical structure. When sub-hierarchical structure is out of balance, splitting is applied to the original rule subset that generated this sub-hierarchical structure. In some cases, it is not possible to produce a more balanced hierarchical structure, in which case the level is marked as currently optimal and the threshold for the intermediate levels are increased.

Promoting aims at reducing the overhead of packet processing at different levels of the hierarchy. The need for rule promotion occurs when a single rule hit count increases dramatically and exceeds its predefined threshold. This scenario is likely to occur during anomalous traffic behavior, typically observed during Denial-of-Service (DoS) attacks. To mitigate the impact of DoS attacks and drastically reduce the cost of processing traffic generated by these attacks, the rule is promoted to a level above the filters. Depending on the rule's priority, promotion may continue recursively until it reaches its appropriate priority level. In the extreme case, the rule may be moved all the way up to the root of the hierarchical structure. This promotion is temporary and the rule, as it never removed from the rule subsets. The reason behind the temporary promotion stems from the transitory nature of DoS attacks. Once the traffic has returned to its normal levels, the

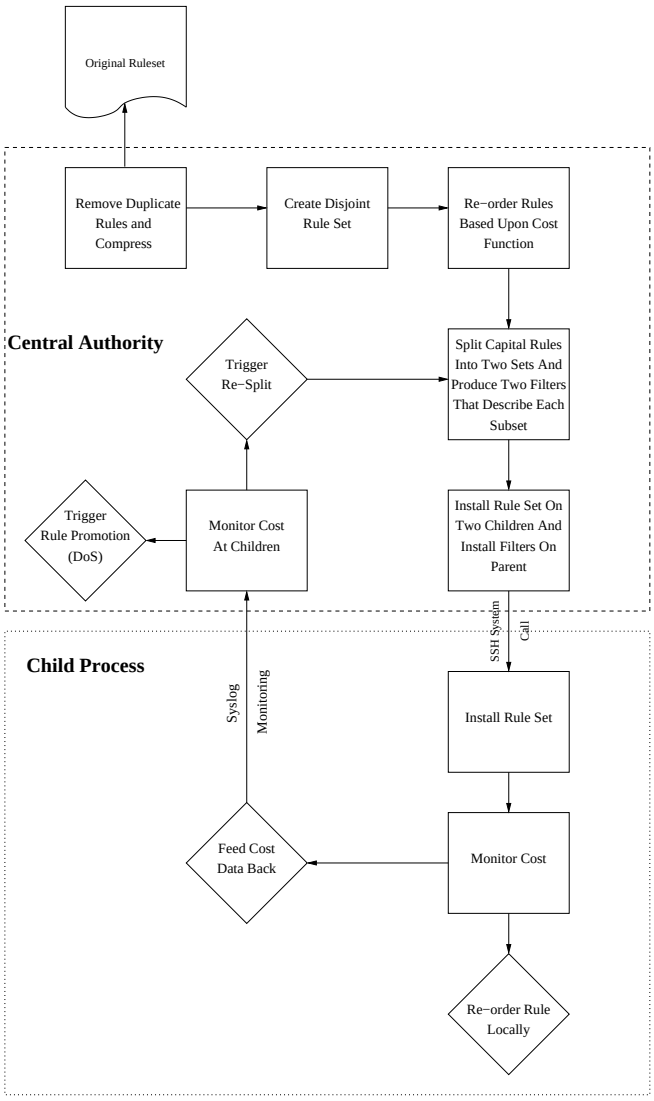

Fig. 3: OPTWALL: Architecture

promoted rule can be removed from the higher levels.

The automatized interaction between the levels (parent-child modules) of OPTWALL is illustrated in Figure 3. Each level, starting from the root, acts as a central authority to a lower level in the hierarchy.

V. OPTWALL Splitting Design Approaches

The efficiency of the splitting process, in terms of packet processing overhead, strongly impacts the performance of the firewall. In this section, we first describe the splitting process and discuss various solutions proposed for splitting the rule set. In this paper we describe a rule with single attribute value as a tuple. We will use the tuple set as the input to our splitting process.

The output of the splitting operation are two filters and their corresponding tuple subsets. The filters and tuple subsets are semantically similar to that of a single list based tuple set. The process of splitting relies upon three basic functions *MATCH()*, *DISTANCE()*, and *WIDEN()*. All three function are available on the filter object and all accept a single argument of a tuple.

The MATCH() function checks to see if a tuple is covered by the filter. The source and destination IP addresses are compared to the range specified in the filter. Similarly the port number is compared to the port range specified in the filter. The protocol type is matched to a list of protocol types the filter evaluates upon. This function returns true if the tuple matches the tuple and false otherwise.

The DISTANCE() function calculates the distance between a given tuple and the filter. If the filter matches the tuple then the value returned by this function is 0. Otherwise, this function returns a positive number between 0 and 1, not inclusive. The distance is based on the entire tuple.

To calculate the distance between two IP addresses we look at the numerical distance between them is considered. If the IP addresses represent ranges, the distance function based on the distance between the two furthest points within the ranges is calculated. A similar procedure is used to calculate the distance between ports or port ranges. The protocol distance is set to 0 if the protocol already exists in the protocol list for the filter. Otherwise the distance is set to 1. All the distances are then normalized to the maximum values of their respective fields. The summation of this normalized values are then weighted and re-normalized to produce a value between 0 or 1.

The WIDEN() function is used to expand a filter such that it matches the given tuple. This is achieved by expanding the IP range, port range, and protocols. A function calculates the cost of the tuple based on traffic characteristics and other tuple properties.

The driver of the splitting process is the search for a set of filters, which covers the the hierarchical structure without violating the semantic integrity of the original ruleset, in order to improve the operational cost of the firewall. Ideally, optimal splitting ensures that, at the end of the partitioning process, all subsets has equal cost. Consequently, when an optimal split is achieved, the average processing cost of each packet is reduced by half of its original cost. An optimal strategy for performing a cost-balanced split of the original set of rules is to use two sub-lists and alternatively place the rules in each list, starting with the highest cost rule, until the set of rules is exhausted. While this strategy is optimal, it is not always feasible. This due to the fact fact that each rule subset produced at each stage of the splitting process must have a mutually disjoint set of filters. Computing such filters may not be always achievable.

The next subsection focuses on the issues related to the design of splitting the tuple set into hierarchical tuple subsets. First, an optimal solution is presented and its applicability in a real firewall setting is discussed. A variety of heuristics, which achieve near optimal solutions with reduced overhead, are then presented.

A. Optimal Approach

The optimal splitting approach is based on an A^* search strategy. Achieving an optimal partition is possible since the

cost can be calculated cumulatively for any partition as it is fixed and does not vary with the tuple priority. The basic steps of the Optimal Solution are depicted in Algorithm 1.

The function $g(n)$ determines the cost of the configuration in the current state. The function $h(n)$, on the other hand, computes the optimal cost of the remaining unassigned tuples if placed in either of the subsets. The function $h_{max}(n)$ calculates the maximum cost of the remaining tuples. This can be used as a guideline to terminate the computation of the filters if the cost benefit resulting from this new filters does not improve on the gains of the previous configuration.

Another mechanism, which is used to reduce the overhead incurred by the search of the feasible optimal solution, is to prune the search space. This is triggered when the difference between $h_{max}(n)$ and $h_{min}(n)$ is lower than a specified error percentage. This enables the search to converge to filters of a nearly optimal solution at a much faster rate.

Even though a feasible optimal solution can be obtained, the worst case time complexity is of the order of 2^N, where N is the number of tuples. As the number of tuples becomes large searching for such a solution leads to a firewall bottleneck. Another shortcoming of the optimal solution is that the memory requirement can also become prohibitive as the number of tuples becomes very large. To address these drawbacks a set of heuristics are proposed. These heuristics converge to a nearly optimal solution, while maintaining a time complexity linear in the number of tuples.

B. Heuristic Solution

The heuristic solutions proposed are local greedy search solutions aimed at determining a set of filters and splitting the list based tuple set into two tuple subsets. Each tuple of the list based set is disjoint from the other. This aids the performance and effectiveness of the approach to split the tuples into smaller tuple subsets. As mentioned in [6] application of greedy scheme works best when the tuples are all disjoint from one another. In other words, making tuples disjoint from each other enables full flexibility for tuple splitting and re-ordering based on traffic characteristics.

Depending on the choice of the initial filters, five different variations of the Greedy Heuristic are proposed. The first variation of the Greed Heuristic is to deterministically assign the highest priority tuples as the initial filters. This heuristic is referred to as **Hit count-Hit count Heuristic**. The idea behind choosing the highest ranked tuples as the initial filters is to assign the highest costing tuples into different tuple subsets in order to arrive at a cost balanced solution. The main steps of the algorithm is described in Algorithm 2.

The next variation of the Greedy Heuristic is to assign one initial filter as the highest costing tuple and the next initial filter as one amongst the rest of the tuples which is at a maximum distance from the highest cost tuple. The distance is calculated using the *DISTANCE* function as stated previously. This variation of the Greedy Heuristic is referred to as the **Hit count-Max distance Heuristic**.

The third variant of the Greedy Heuristic uses a randomly

Algorithm 1 Optimal Solution

$g(i,n)$ = cost of $list_a$ and $list_b$ after adding tuple n to list i
$h(n)$ = current cost of optimally placing the remaining tuples
$cost(i,n)$ = $g(i,n)$ + $h(n)$
$filter_a$, $list_a$ = filter and tuples for list A
$filter_b$, $list_b$ = filter and tuples for list B
$stack$ = stack ordered with least cost on top
INPUT:
$tuples[]$ - List of tuples sorted by cost
ALGORITHM
$counter$ = 0, $list_a$ = $\emptyset$, $list_b$ = $\emptyset$
$currentTuple$ = $tuples[counter]$
while $counter < tuples.size()$ **do**
  if $cost(A, currentTuple) < cost(B, currentTuple)$ **then**
    if $filter_a \cap filter_b$.widen($currentTuple$) $\neq$ $< any, any, any, any >$ **then**
      $stack.add(< list_a$,$list_b \cup currentTuple$, $filter_a$,$filter_b.widen(currentTuple)$, $counter >)$
    end if
    $filter_a$.widen(currentTuple), $list_a$.add(currentTuple)
    if $filter_a \cap filter_b = < any, any, any, any >$ **then**
      $< list_a, list_b, filter_a, filter_b, counter >$ = stack.pop()
    end if
  else
    if $filter_a$.widen($currentTuple$) $\cap$ $filter_b \neq$ $< any, any, any, any >$ **then**
      $stack.add(< list_a \cup currentTuple$,$list_b$, $filter_a.widen(currentTuple)$,$filter_b$, $counter >)$
    end if
    $filter_b$.widen(currentTuple), $list_b$.add(currentTuple)
    if $filter_a \cap filter_b = < any, any, any, any >$ **then**
      $< list_a, list_b, filter_a, filter_b, counter >$ = stack.pop()
    end if
  end if
  $counter++$
  $currentTuple$ = $tuples[counter]$
end while
OUTPUT:
$< filter_a, list_a, filter_b, list_b >$

Algorithm 2 Hit count-Hit count Heuristic

INPUT:
tuples[] - List of tuples sorted by cost
$filter_a = tuples[0]$
$filter_b = tuples[1]$
for $i = 2$ to tuples.length() **do**
  if $filter_a.matches(tuples[i])$ **then**
    add $tuple[i]$ to $list_a$
  else if $filter_b.matches(tuples[i])$ **then**
    add $tuple[i]$ to $list_b$
  else
    $distance_a = filter_a.distance(tuples[i])$
    $distance_b = filter_b.distance(tuples[i])$
    if $distance_a < distance_b$ **then**
      $filter_a.widen(tuples[i])$
      add $tuple[i]$ to $list_a$
    else
      $filter_b.widen(tuples[i])$
      add $tuple[i]$ to $list_b$
    end if
  end if
end for
OUTPUT:
$filter_a$ - filter tuple for list A
$filter_b$ - filter tuple for list B
$list_a$ - list of tuples for child A
$list_b$ - list of tuples for child B

selected initial filter assignment. This heuristic is referred to as **Random-Random Heuristic**. A randomized algorithm is used to determine initial filters from a set of possible filters. The selected set is then used to build the hierarchical structure.

The fourth variant of the Greedy Heuristic is to consider the distance between all possible pairs of filters. The pair which contains the filters with maximum distance from each other is selected. This strategy has potential to split the tuples into well balanced sets. This heuristic is referred to the **Max distance-Max distance Heuristic**. The complexity for all the above approaches is proportional to the number of tuples in the initial tuple set.

The fifth variant of the Greedy Heuristic is the **All Pair Heuristic**. This variant considers all possible pairs of tuples as initial filters. Using the method depicted in Algorithm 2 we determine a split for each possible pair and then pick the split with the least cost.

The results for All Pair Heuristic are not included as the heuristic never converged to a solution due to the excessive overhead required to obtain the most cost efficient configuration among all possible pairs of tuples. The time complexity of this heuristic is of the order of N^3, where N is the number of tuples. For large values of N, the computational cost of the heuristic becomes prohibitive.

```
: rule (
    : src (                    : srv (
        : 10.10.10.2               : ospf
        : 10.10.10.3               : traceroute
        : 10.10.10.4               : echo-requests
        : 10.10.10.5               : ping-replies
        )                          )
    : dst (                    : action(
        : 10.20.10.1               : accept
        : 10.20.10.4               )
        : 10.20.10.5       )
        )
```

Fig. 4: Rule Structure

```
num;date;time;orig;type;action;alert;i/f_name;i/f_dir;
product;src;dst;s_port;service;proto;……..

1;27Jul2005;23:59:04;10.10.10.1;log;accept;;qfe1;
inbound;X;10.30.10.1;10.20.10.1;53480;161;udp;;;
```

Fig. 5: Traffic Log Instance

VI. Performance Evaluation

In this section we describe the experiments and evaluations to validate the proposed OPTWALL scheme. First, we briefly described the synthetic data used in this evaluation study. The experimental set up, traffic generation and the evaluation results are discussed.

A. Experimental Data

The data set used in the experimental study emulates, in terms of size and number of rules, the data of of a typical list based firewall managed by large ISPs supporting a variety of customers. In this performance evaluation study, it is assumed that the ISP provides secure access to and from a group of customers and business partners. The data set consists of firewall rule sets and traffic logs.

Each rule set consists of several thousand rules. Each rule is a multi-dimensional structure of tuples. A rule set contains on average one million tuples. Figure 4 shows an instance of a rule structure. The dimensions of the rule include the source address, src, the destination address, dst, the service types, srv, and the action. Each dimension contains multiple values. An instance of a tuple of this rule is $< src : 10.10.10.2;\ dst : 10.20.10.1;\ srv : ospf; action : accept >$. Figure 5 depicts an entry of the firewall traffic log. The firewall logs one entry per session.

B. Experimental Setup

The experimental set up for the evaluation of the proposed OPTWALL approach consists of a machine acting as a firewall and another generating traffic and collecting logs. The machines used for our evaluation are $AMDAthlon^{tm}$ 64 bit Processors 3000+ running $Ubuntu\ Linux$ operating system. The machines are isolated for testing to ensure that there are

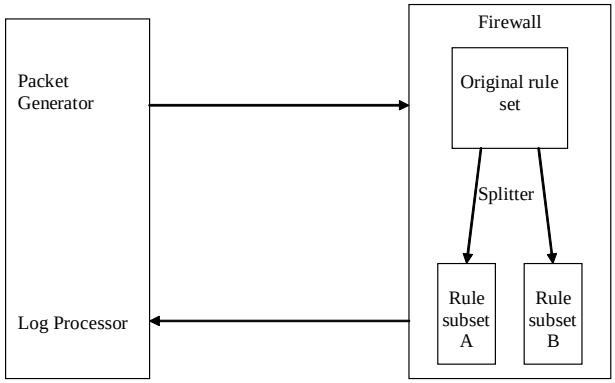

Fig. 6: Experimental Setup

no other variants. Figure 6 shows the block diagram of the experimental setup.

C. Traffic Generation

There were two types of traffic characterizations used to evaluate OPTWALL, the worst case and emulated case behavior. In worst case scenario, traffic is composed of a single packet type that does not match any of the tuples. This assures that the packet will be caught only by the default action tuple. The emulated traffic is generated by creating packets that match each tuple and proportionally sending them to a traffic trace similar to a large ISP's firewall operation. The worst case traces were used to study the worst case performance of OPTWALL in comparison to the baseline case, a list based firewall. Performance at worst case was determined by using constant traffic rates and measuring the overall CPU utilization. Traffic rates were determined by loading the firewall from 25% to 100% utilization with the installed list based rule set. A similar approach was used to determine the load for the emulated traffic evaluations.

D. Evaluation results

The following subsection discusses the various results highlighting the potential of the proposed OPTWALL approach.

1) Hierarchical model evaluation: This study was performed to evaluate the potential of the hierarchical design and its effect on efficient firewall optimization *w.r.t.* a list based design. The extent of the hierarchy depends on the tuple set size, the traffic characteristics and the variability in traffic. For our evaluation we fixed the tuple size, load applied and the splitting approach used to determine the benefit from the proposed hierarchical design. The experiments were conducted on a heavily loaded system and using the best performing heuristic amongst all the solutions proposed earlier in the paper. We use a tuple set of nearly 5,000 tuples, load of 1,440 packets/sec and the Max Distance-Max Distance Heuristic for our evaluations. Results as in Figure 7 shows the potential of the proposed OPTWALL framework. It is to be noted that after a point re-splits cause more harm than good. The results

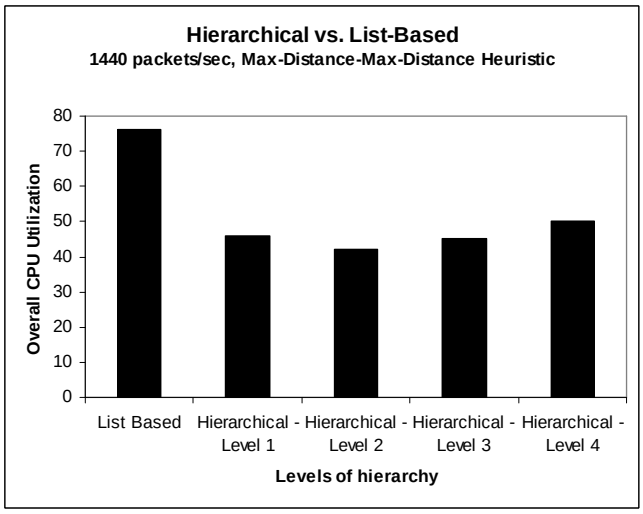

Fig. 7: Hierarchical vs. List-Based

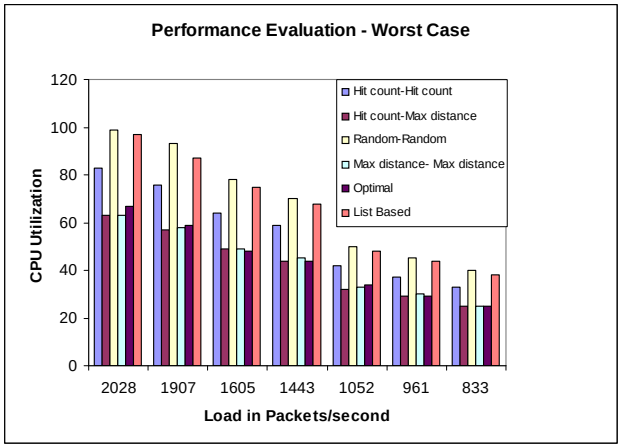

Fig. 8: Performance Evaluation (Worst-Case - 60,000 tuples)

depict a way to arrive at a sweet spot between the number of re-splits and the gain to due the hierarchical design.

2) Worst case performance evaluation: The next study performed is to determine the worst case packet processing cost of the firewall. A worst case packet processing occurs when very packet entering the system requires processing of the entire tuple subset. This helps to determine the maximum packet rate for worst case traffic processed by the firewall. Various tuple sizes are used for the evaluations. The results are for a typical large tuple set, consisting of 60,000 tuples. From Figure 8 it can be concluded that the Optimal Approach and Max distance-Max distance Heuristic perform the best in comparison to the baseline list based approach. It is to be noted that filters determined by the Optimal Approach shows better traffic filtering than the heuristics approaches.

3) Emulated traffic performance evaluation: The next study is to determine the CPU consumption of the firewall when the traffic applied follows the normal traffic trace. Results as in Figure 9 show the benefit of the proposed scheme.

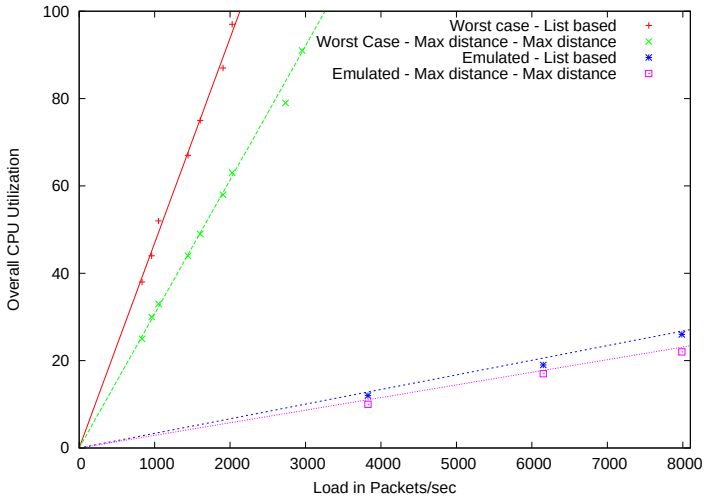

Fig. 9: Emulated Traffic Performance Evaluation

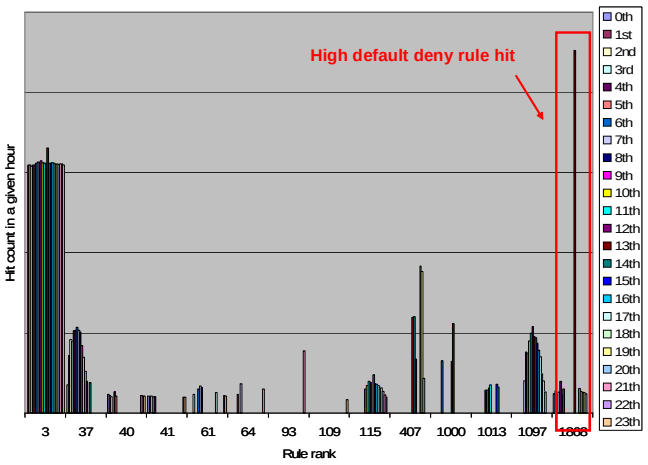

Fig. 10: Default Deny Hit Count for a typical day

The CPU improvement in the worst case is about 35% and in the emulated case is about 14%. Since, the CPU consumption is additive, any gain on the emulated case can be translated as a capacity for dealing with more anomalous traffic that can be handled by the firewall. In other words, OPTWALL can deal with a larger predicted traffic volume and also a much larger anomalous traffic.

4) Handling attacks evaluation: The aim of this study is to test the strength of OPTWALL in handling attacks and traffic fluctuations. Since the hit counts for default action tuples are large and unpredictable, it can cause a huge bottleneck to the entire firewall operation. Figure 10 illustrates an instance of a large hit count for a default action tuple. To test the performance of OPTWALL in handling such attacks we emulated the attack and increasing the hit count of a certain default action tuple from $0 \sim 100,000$. Figure 11 shows the competence of OPTWALL in countering dynamic traffic changes and hence aiding the steady maintenance of firewall operation.

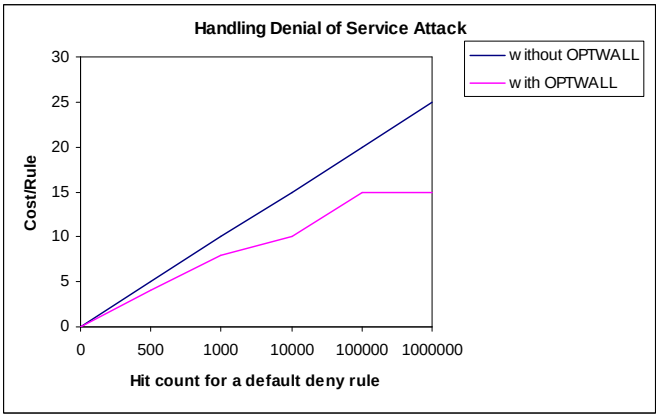

Fig. 11: Countering DoS Attacks

5) Sensitivity analysis evaluation: The final study is aimed at sensitivity analysis of the proposed OPTWALL approaches. The analysis was performed for tuple sizes varying from $0 - 1000$ tuples. Figure 12 details a comparative study between the baseline list based, the best heuristic and the optimal solution. Results depicted are for a heavily loaded firewall operation. From the results it can be inferred that the heuristic solutions are best suited for a hierarchical firewall optimization framework.

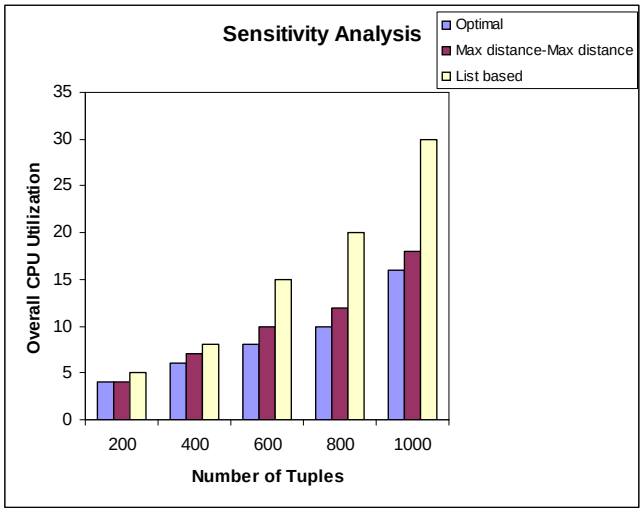

Fig. 12: Sensitivity Analysis

VII. Related Work

Due to the enormous impact of firewalls on network security, there has been a significant amount of research work on how to optimize firewalls. Much of this work, however, has been in the area of firewall policy modeling and optimization [7], [8], [9], [10], [11], [12], [13], [14], [15], [16].

Very few attempts have been made to achieve multi-dimensional firewall optimization. In [17], a tool to model firewall policies and detect conflicts is described. In this work, the authors focus mainly on single attribute rules. Similarly, in [13] a constraint logic programming (CLP) framework to analyze rule sets is discussed. These research work offer a good insight in how to model and analyze rule sets. Neither of these works, however, consider optimizing a multi-dimensional rule set. The approach proposed in [7] optimizes the firewall rule set using Directed Acyclic Graphs (DAGs) to describe rule dependencies. However, it does not provide a methodology to build the DAG. Furthermore, for complex graphs this scheme is ineffective. In [18], a framework to analyze and optimize rule sets is described. However, the authors do not provide specific details on how optimization can be achieved within the proposed framework. Furthermore, this work does not consider the traffic characteristics in its optimization approach.

Recently there has been great attention to address traffic-aware firewall optimization. Some efforts in rule reordering using traffic specifications as in [6] have been proposed. But they consider a very small firewall policy set (~ 200) and there is absence of complete rule reordering due to dependencies in the policy set. Furthermore, all traffic characterizations are not considered for firewall optimization. [3] presents a tool geared towards adaptive optimization of list based firewalls. However, the work falls short of addressing non-linear policy optimization. In other words, it does not consider any traffic-aware design improvements in the firewall structure. The work presented in this paper builds on [3] and achieves complete non-linear policy reordering by removal of all rule dependencies. The proposed firewall optimization approach, 'OPTWALL' is a novel adaptive hierarchical design geared towards very large policy optimization.

VIII. Conclusions and Future Work

A firewall is a combination of hardware and software used to implement a security policy governing the flow of network traffic between two or more networks. In its simplest form, a firewall acts as a security barrier to control traffic and manage connections between internal and external network hosts.

Firewalls have proven to be useful in dealing with a large number of threats that originate from outside a network. They are becoming ubiquitous and indispensable to the operation of the network. The continuous growth of the Internet, coupled with the increasing sophistication of attacks, however, is placing further demands and complexity on firewalls design and management.

This paper focuses on the problem of firewall optimization. To this end, the paper proposes a hierarchical framework, OPTWALL, for traffic-aware firewall optimization. The basic tenet of this framework is that the design of next generation firewalls must leverage their packet inspection capabilities with traffic awareness in order to optimize the operational cost they incur in defending against intrusions and denial of service attacks. To the best of our knowledge this is the first effort towards using firewall traffic log information and hierarchy to design and optimize firewalls. The performance of the OPTWALL approach both for worst case and normal operation of the firewall is studied. The results show that OPTWALL leads to reduced operational cost of firewalls.

OPTWALL presents a novel method to use hierarchy in optimizing list based firewalls. It helps to achieve the maximum benefit *via* various splitting processes to arrive at feasible optimal and near optimal solutions. We are presently working on extending the hierarchical design concept onto physically distributed firewalls. This would imply that the rule subsets could be run on different machines or in parallel on the same machine.

IX. Acknowledgments

We would like to thank Alexandre P. Ferreira for his valuable insights and feedback.

Appendix

The following segment describes the intuition behind the proposed research. First the optimal solution for a list based firewall policy set is presented and then the solution for K-partitions is discussed.

A. Optimal solution - List based

A list based firewall is a sequence of tuples that are composed of filter fields and an action to be executed for packets that match the filter profile. Each tuple has a counter that counts every time the tuple has been fired and has a rank that determines its position in the list based sequence.

Each test of a filter for a tuple consumes certain CPU processing time. Assuming that the cost of testing is the most expensive operation, the total CPU cost of a sequence of tuples is the sum of the costs of the number of times the tuple is tested. For a tuple i the number of times it is tested is a summation of the tuple's hit count plus the hit count of all tuples that succeeds it.

$$C = \sum_{i=1}^{N} \sum_{j=i}^{N} H_j \Rightarrow C = \sum_{i=1}^{N} i * H_i \quad (1)$$

where, C represents the total cost of the list based tuple processing, and H_i represents the hit count of tuple i.

With this result we can define the weighted cost of the tuple $i * H_i$.

The lowest cost of the sequence is achieved by keeping the list in an inverse sorted order by hit count. The proof is by switching tuple k with tuple l in the formula above and since the only terms that change are
k * H_k is changed to l * H_k
and l * H_l is changed to k * H_l

Hence,

$$\Delta C = k*(H_k - H_l) - l*(H_k - H_l) = (k-l)*(H_k - H_l) \quad (2)$$

If $k < l$, this implies that the cost will decrease,
stated differently, ($\Delta C > 0$) only if $H_k - H_l < 0 \Rightarrow H_k < H_l$. Hence, the lowest cost is achieved when the tuples are ordered using their hit counts with the highest count as the first tuple.

B. Optimal solution - K partitions

Assuming a distribution of the tuples such that all tuples can appear only in one list and there is a function f(i) and g(i) that maps a tuple with rank i in list A or list B to the rank it occupied in the single list. In this case the following equation holds:

$$\forall i, j, f(i) \neq g(i) \tag{3}$$

$$\forall i \exists j, i = f(j) \vee i = g(j) \tag{4}$$

Implies that, no tuples are duplicated and all tuples appear in the new configuration.

$$C = \sum_{i=1}^{N} H_i + \sum_{i=l}^{N/K} i * H_{f(i)} + \sum_{i=l}^{N/M} i * H_{g(i)}, N = K + M \tag{5}$$

Cost of the new tuple plus the cost of each partition.

Each partition has to be sorted similar to the reorder discussion above. The tuples in the original sequence are to be as low as possible in the new partitions to reduce the cost. Exchanging tuples in the same row between lists does not alter the final cost of the firewall. Hence, the optimal solutions are as follows:
$f(i)$ = $2i$ or $2i$ + 1 and $g(i)$ = $2i$ or $2i$ + 1.

REFERENCES

[1] T. V. Lakshman and D. Stidialis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *In Proceedings of SIGCOMM*. ACM Press, 1998.

[2] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *In Proceedings of SIGCOMM*. ACM Press, 1999.

[3] S. Acharya, J. Wang, Z. Ge, T. Znati, and A. Greenberg, "Traffic-aware firewall optimization strategies," in *IEEE International Conference on Communications*, Istanbul, Turkey, June 2006.

[4] P. Brucker, "On the complexity of clustering problems," in *in Optimization and Operations Research*. Springer-Verlag, pp. 45-54, 1977, 1997.

[5] S. Singh, F. Baboesu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *SIGCOMM*, 2003.

[6] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *ASIACCS*, 2006.

[7] E. W. Fulp, "Optimization of network firewalls policies using directed acyclic graphs," in *Proceedings of the IEEE Internet Management Conference*, 2005.

[8] ——, "Parallel firewall designs for high-speed networks," in *INFOCOM*, 2006.

[9] L. Qiu, G. Varghese, and S. Suri, "Fast firewall implementations for software-based and hardware-based routers," in *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 2001, pp. 344–345.

[10] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *in Proceedings of Hot Interconnects*, 1999.

[11] S. Singh, F. Baboesu, G. Varghese, and J. Wang, "Packet classification on multiple fields," in *SIGCOMM*, 1999.

[12] ——, "Packet classification using multidimensional cutting," in *SIGCOMM*, 2003.

[13] P. Eronen and J. Zitting, "An expert system for analyzing firewall rules," in *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, Copenhagen, Denmark, Nov. 2001, pp. 100–107.

[14] S. Hinrichs, "Integrating changes to a hierarchical policy model," in *Proceedings of 9th IFIP/IEEE International Symposium on Integrated Network Management*. Nice, France: IEEE, 2005.

[15] E. Al-Shaer and H. Hamed, "Modeling and management of firewall policies," *IEEE Trans. Network and Service Management*, vol. 1, no. 1, Apr 2004.

[16] S. J. Tarsa and E. W. Fulp, "Trie-based policy representations for network firewalls," in *Proceedings of the IEEE International Symposium on Computer Communications*, 2006.

[17] E. Al-Shaer and H. Hamed, "Modeling and management of firewall policies," *IEEE Trans. Network and Service Management*, vol. 1, no. 1, Apr 2004.

[18] J. Qian, S. Hinrichs, and K. Nahrstedt, "ACLA: A framework for access control list (acl) analysis and optimization," in *Communications and Multimedia Security*, 2001.