Accelerating Firewalls: Tools, Techniques and Metrics for Optimizing Distributed Enterprise Firewalls

Subrata Acharya

The overall efficiency, reliability, and availability of firewalls are crucial in enforcing and administering security, especially when the network is under attack. These challenges require new designs, architecture and algorithms to optimize firewalls. Contrary to a list-based structure, a de-centralized (hierarchical) design leads to efficient organization of rule-sets, thereby significantly increasing the performance of the firewall. The objective is to transform the original list-based rule-set into more efficient and manageable structures, in order to improve the performance of firewalls. The main features of this approach are the hierarchical design, rule-set transformation approaches, online traffic adaptation mechanisms, and a strong reactive scheme to counter malicious attacks (e.g. *Denial-of-Service (DoS)* attacks [1]).

1 Introduction

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 [2, 3] 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 in current literature are static, and fail to adapt to the dynamic traffic changes of the network. The large number of security policies in enterprise networks poses the most important challenge to traffic-aware firewall optimization. Furthermore, with

Subrata Acharya
Towson University, Towson, MD, USA

I. Zelinkaet al. (Eds.): Handbook of Optimization, ISRL 38, pp. 529–551.
springerlink.com

the increased ability of current networks to process and forward traffic at extremely high speed, firewalls are becoming highly resource constrained. Thus, the main objective of this chapter is to address the shortcomings of the current firewalls and increase their ability to deal with dynamic changes in network load and topology, particularly under attack conditions. In this chapter, the focus is on optimizing the most widely used 'list-based' firewalls. To achieve this goal we propose a firewall transformation framework aimed at creating hierarchical firewall optimization rule subsets, to improve the operation and manageability of firewalls. The main challenge in the construction of these rule subsets is the need to maintain semantic integrity of the policy set at each level of the hierarchy. The overall goal is to improve the performance and manageability of such network systems.

The rest of the chapter is organized as follows: Section 1.2 introduces the *Firewall Transformation Framework*. We introduce the theory of the transformation approach in Section 1.3. We present details of the splitting approaches in Section 1.4. In Section 1.5 presents the architecture and implementation methodology details of the hierarchical firewall. We present the evaluation and results in Section 1.6. Finally, we conclude the chapter in Section 1.7.

2 Firewall Transformation Framework

In this section we introduce the *Firewall Transformation Framework* aimed at improving the performance and manageability of firewalls. A software firewall defines its security policies via a set of security policies or rules. These security policies govern the filtering of network packets to and from the autonomous system. In this chapter our aim is to improve the availability and good-put of firewalls by proposing transformative algorithms to the "*list-based*'' firewall representation into more manageable and performance efficient representations. Any proposed transformation should preserve the semantic integrity of the original firewall rule-set, in order that the *Tier-1 ISP* network administrator accepts and registers to replace the original firewall rule-set with the transformation. Additionally, the transformed firewall rule-set should reduce the operational cost of packet filtering, in turn improving the efficiency and manageability of the firewall. The Framework is defined formally as follows:

Let F represent the original "*list-based*" firewall rule-set. Let $T(F)$ represent the transformed firewall that preserves the properties and rules of the original firewall rule-set F. We define the cost function, $C(f)$, that represents the average operational cost of operation of firewall f.

$T(F)$ is an acceptable transformation of F *iff*:

- $T(F)$ preserves the properties and rules of F (*Semantic Integrity property*)
- $C(T(F)) \leq C(F)$ (*Cost property*)

We discuss details of these properties and prove the property for the *Firewall Transformation Approach* in Section 1.3.

3 Firewall Transformation Approach

In this section we detail the theory behind the above-discussed *Firewall Transformation Framework*. As stated in the previous section, the objective of this transformation is to obtain an "*acceptable*", *S (F)*, such that the transformation preserves the "*Semantic Integrity*" and "*Reduced Cost*" properties as discussed in the following.

The firewall transformation is achieved by the process of "*splitting*" or dividing the original "*list-based*" firewall rule-set into groups of rule subsets. The input firewall rule-set *F* is first sorted based on traffic characteristics (hit-count) before the transformation is initiated. The resulting rule subsets preserve the "*semantic integrity*" properties and rules of the original "*list-based*" rule-set. To state formally:

Let *S (F)* represent the Firewall Transformation approach of the original list-based firewall rule-set *F*.

Theorem 1.1: S (F) is an acceptable transformation of F iff:

- $S(F)$ *preserves the properties and rules of* F *(Semantic Integrity property)*
- $Cost\ (S(F)) \leq Cost\ (F)$ *(Cost property)*

Proof: We prove the above theorem via "*proof by contradiction*".

Semantic integrity property:

Let there be at-least one rule r in *S (F)* such that the action of *r* on a network packet *p* is different than that of *F* on *p*. This implies that the action of the firewall rule-set *F* on a given network traffic is different from the action taken by the new transformed representation *S (F)*. Thus, the semantics of the original firewall rule-set is not equivalent to that of the transformed rule-set.

The rules in the list-based rule-set are scanned in a sequential manner and divided into rule subsets based on traffic characteristics. Hence, the rules in the rule subsets in *S (F)* originate from the rule-set F. The manner, in which the rules are sub-divided into rule sub-sets, does not add any new rule or cause any rule deletions. This implies that no new rules are created or deleted due to the transformation process. This proves that the rule *r* in *S (F)* must belong to some priority level in the original rule-set *F*, which implies we arrive at a contradiction.

Hence, there are no rules in the transformed firewall rule-sets *S (F)* that have an action different than one that is in the original rule-set *F*. This proves that the semantic properties and the rules of the original rule-set are preserved after transformation.

Cost property:

Let the operational cost of the original firewall rule-set and the transformed rule-set be *C (F)* and *C (S (F)),* respectively. The focus of this research is on the most widely used list-based firewalls. For our evaluation we have assumed that rule matching is the most expensive operation. The operational cost of rule matching is proportional to the number of rules in the rule-set[1].

Let us assume for contradiction that *C (F) > C (S (F)).* For this assumption to be true, there exists at-least one packet *p* such that it matches a rule *r* in the firewall rule-set, where the cost of matching in the transformed firewall set is higher than the cost of matching in the original rule-set. Let the operational cost of processing the network packet *p* which matches rule *r* in the firewall rule-set *F* be represented as *x* and that matches rule *r* in transformed firewall *S (F)* be *y*, where *y* > *x*. Due to the list-based firewall operation, the only way *y* is greater than *x*, is if the rank of *r* in *S (F)* is lower than the rank of *r* in *F*. Since, both the firewall rule-sets are sorted by traffic characteristics (hit-count of incoming traffic), there exists rule *r′* in the firewall rule-subset *S (F)* that has higher hit-count than *r* and is lower rank in the original rule-set *F*. Since, all the rules in *F* are sorted according to hit-count information and there are no new rules created or deleted due to the transformation process, we arrive at a contradiction.

Hence, we prove that the operational cost of firewall rule-sets *S (F)* is <= the cost of the original rule-set *F*. In the worst case, the length of *S (F)* will be equal to the length of *F*, which implies *C (F) = S (F).*

In the next section 1.4 we discuss the details of two *Firewall transformation approaches*, namely, the *Optimal* and the *Heuristic* approach.

4 Firewall Splitting Approaches

4.1 Optimal Approach

The Optimal splitting approach is based on an *A** search [4] 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 Approach are depicted in Algorithm 1.

[1] The result has been verified for *Linux IPChains Firewall* [3]

```
1  g(i,n) = cost of list_a and list_b after adding tuple n to list i;
2  h(n) = current cost of optimally placing the remaining tuples;
3  cost(i,n) = g(i,n) + h(n);
4  filter_a, list_a filter and tuples for list A;
5  filter_b, list_b filter and tuples for list B;
6  stack = stack ordered with least cost on top;
   input  : tuples[ = list of tuples sorted by cost
7  counter = 0;
8  list_a = ∅;
9  list_b = ∅;
10 currentTuple = tuples[counter];
11 Compute filter_a, filter_b, list_a, list_b.
12 while counter < tuples.size() do
13     if cost(A,currentTuple) < cost(B,currentTuple) then
14         if filter_a ∩ filter_b.widen(currentTuple) ≠ < any,any,any,any > then
15             stack.add(< list_a,list_b ∪ currentTuple,filter_a,filter_b.widen(currentTuple),
16             counter >);
17         filter_a.widen(currentTuple), list_a.add(currentTuple);
18     if filter_a ∩ filter_b = < any,any,any,any > then
19         < list_a,list_b,filter_a,filter_b,counter > = stack.pop();
20     else
21         if filter_a.widen(currentTuple) ∩ filter_b ≠
22         < any,any,any,any > then
23             stack.add(< list_a ∪ currentTuple,list_b,filter_a.widen(currentTuple),filter_b,
24             counter >);
25         filter_b.widen(currentTuple), list_b.add(currentTuple);
26     if filter_a ∩ filter_b = < any,any,any,any > then
27         < list_a,list_b,filter_a,filter_b,counter > = stack.pop();
28 counter++;
29 currentTuple = tuples[counter];
   output : < filter_a,list_a,filter_b,list_b >
```

Algorithm 1: Optimal Approach for Rule-set Splitting

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 these new filters do 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.

4.2 Heuristic Approach

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 [5] 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 reordering based on traffic characteristics.

Depending on the choice of the initial filters, five different variations of the Greedy Heuristic are proposed. The first variation 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 are described in Algorithm 2.

```
   input  : tuples[] – list of tuples sorted by cost
 1 filter_a = tuples[0];
 2 filter_b = tuples[1];
 3 Compute filter_a, filter_b, list_a, list_b; for i = 2 to tuples.length() do
 4     if filter_a matches(tuples[i]) then
 5         add tuple[i] to list_a;
 6     else
 7         if filter_b.matches(tuples[i]) then
 8             add tuple[i] to list_b;
 9         else
10             distance_a = filter_a.distance(tuples[i]);
11             distance_b = filter_b.distance(tuples[i]);
12     if distance_a < distance_b then
13         filter_a.widen(tuples[i]);
14         add tuple[i] to list_a;
15     else
16         filter_b.widen(tuples[i]);
17         add tuple[i] to list_b;
   output : < filter_a, list_a, filter_b, list_b >
```

Algorithm 2: Hit count-Hit count Heuristic

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. 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 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 that 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.

4.3 Improvements to Rule-Set Splitting Approaches

As we discussed in the previous section, establishing near optimal rule splits directly affects the performance of the hierarchical filter in turn improving the operational cost of the firewall. The reason being the filter split governs the "*re-splits*", "*re-promotions*" and "*re-orders*" of the rules in the rule subsets, which helps to incorporate traffic dynamics in the filter operation for a given firewall. The better the split (*based on the traffic characteristics*) the more stable and balanced the hierarchical firewall.

In this section our goal is to exploit the traffic characteristics further in order to deter- mine the patterns in data which will aid the definition of better splits to enable hierarchical firewall optimization. In our analysis we conclude that the choice of initial filters should be accurate and is a defining factor in the design of stable rule split subsets. As discussed in Algorithm 2, consideration of a single criteria (e.g. *traffic volume/hit-count information*) is insufficient to determine accurate initial filters for the hierarchal rule split. We consider the patterns in the traffic in order to aid the choice of the initial filters. This will necessarily affect the computation of better or closer to optimal rule-set splits (*aiding better traffic filtering*). In the following subsection we present the clustering rule split approach to aid decentralized firewall optimization.

4.3.1 Clustering Rule Split

We determine the patterns in the traffic data via an exhaustive search algorithm to output a group of clusters. The firewall administrator specifies the criteria for the cluster organization. The criteria could be a single attribute (e.g. *protocol: TCP/UDP*) or a combination of attributes. Such a search results in sets of clusters based on the specified criteria. Since our problem deals with the design of two initial filters for the hierarchical split operation, we arrive at two cluster sets (*groups)* based on the self-similar criteria as defined earlier. These self-similar clusters are the precursors to the rule splitting approaches as discussed above.

We then pick the initial filters one from each cluster group. We then follow the rule splitting steps as discussed in Algorithm 2. The cluster groups act as a feedback to the rule splitting process in this step. Each rule during the splitting process tries to be retained in its self-similar cluster group during the process of hierarchical filter set determination. Since the clustering of the rules are run offline; we have the liberty to run the exhaustive search algorithm. The disjoint nature of the rules in the rule-sets enables the clustering and rule splitting process in determining traffic balanced rule split sets.

4.3.2 Parallel A* Approach

In the next improvement, we propose a change in the *Optimal splitting* approach presented above to aid faster splits and to enable the split of larger data sets. Previously we have proposed an variant of the *A** search strategy to search for the defining filters for each split set. The runtime for such an approach in worst case is 2^N, where N is the number of initial rules in the linear rule-set. For rules greater than *1000* Algorithm 1 did not terminate and hence did not produce the split sets we required. To overcome this problem we incorporated parallelism in the *A** search [6] to enhance the splitting performance on larger data sets. We extended the optimal search approach developed earlier to include parallelism. The challenge is to keep track of the various intermediate sub solutions. We used multiple hashing mechanisms to store the locally optimal solutions to be used later to arrive at the final optimal split subsets. The proposed parallel *A** approach perform much better and is able to arrive at rule splits for large data sets (*nearly 10,000 rules*).

4.3.3 Weighted Distance Function

The distance function as presented in Algorithm 2 is calculated by assigning equal weights to all the dimensions of a packet filter. Our analysis concludes that this assumption is not accurate. We propose that the distribution of the tuples amongst the dimensions should determine the weights that the dimensions take and the normalization should occur on them. We changed the present splitting approach of finding the defining filter of a tuple by including a weighted distance function to determine the weighted distance of a tuple from the defining filters. This approach leads to more balanced splits. Results show that the split is about the improve from a *30 : 70* split to a almost *45 : 55* split. This optimization aids to reduce the worst-case packet matching time for the hierarchical firewall we designed.

5 Design Architecture and Methodology

In this section we presents the architectures and algorithms for a de-centralized firewall optimization, *OPTWALL* [7]. As we have discussed earlier, 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 rule-set 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.

5.1 OPTWALL Design Goals

Given a large rule-set of size '*N*', 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:

- Reduce the *cost of processing* the firewall rule-set, defined as the average processing time a packet incurs before an action is enforced by the firewall,
- Preserve the *semantics* of the original rule-set, and

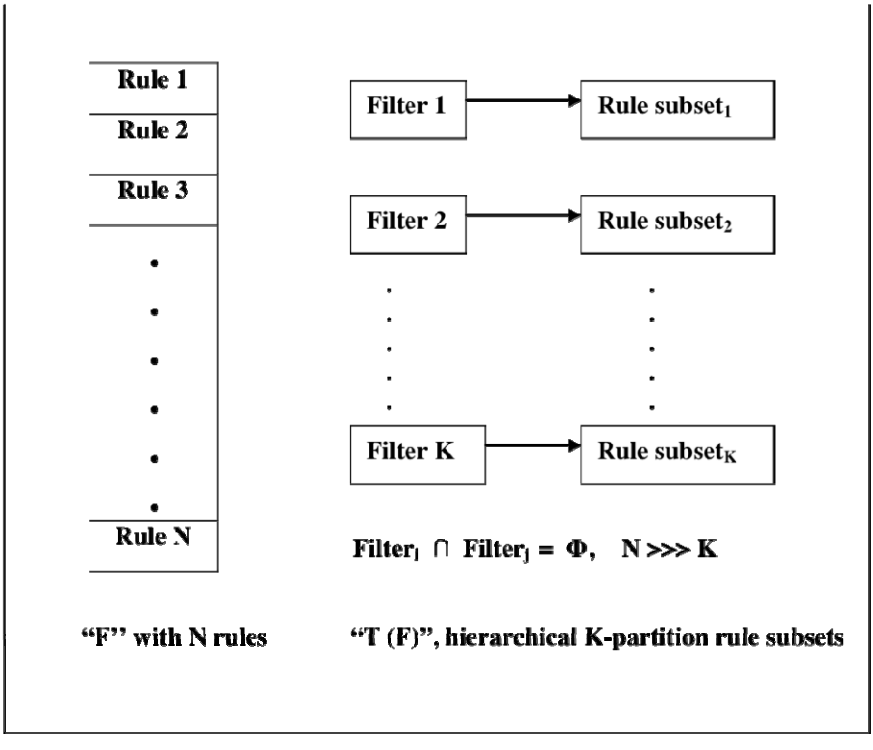

Fig. 1 N rules into K partition problem

– 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*' [8] or the '*K-median*' problem [9]. 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. 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. It is to be noted that this cost also includes the cost of determining the filters. The *OPTWALL* partitioning process is described in Figure 2.

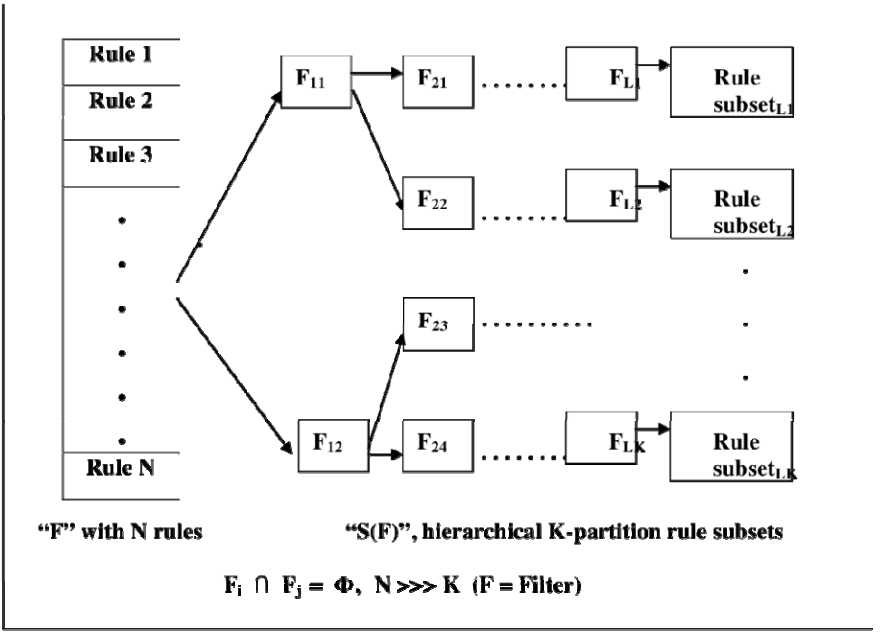

Fig. 2 Basic operation of OPTWALL

5.2 Hierarchical Firewall Optimization Model

In this section we will present the processes used to achieve each of *OPTWALL* design goals. We first describe 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.

5.2.1 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 reordered, in order to optimize operational cost, the process of reinforcing the original rule semantics must be achieved with reduced overheard.

5.2.2 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 the original list-based rule-set as its input 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 [10]. The fact that the rules in the rule set are mutually disjoint provides *OPTWALL* with full flexibility to reorder the rules and divide them into rule subsets, without violating the semantics of the original rule-set.

In the *reordering* stage, rules are reordered 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 reordering 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 rule subset is defined by

a tuple, which covers all rules in the subset. All such covering tuples are disjoint from one another. 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 later sections of this chapter.

The produced hierarchical structure is then converted to a series of *IP-table* rule sub- sets. It is to be noted that most list based firewalls, such as *Linux IPCHAINS* [4], 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.

5.2.3 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 *reordering*, *re-splitting*, and *promoting* to reestablish 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 the difference between the current and previous hit-counts of a given rule exceeds a preset 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 is not 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 promoted rule can be removed from the higher levels.

The automatic 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.

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

The outputs 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 functions 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.

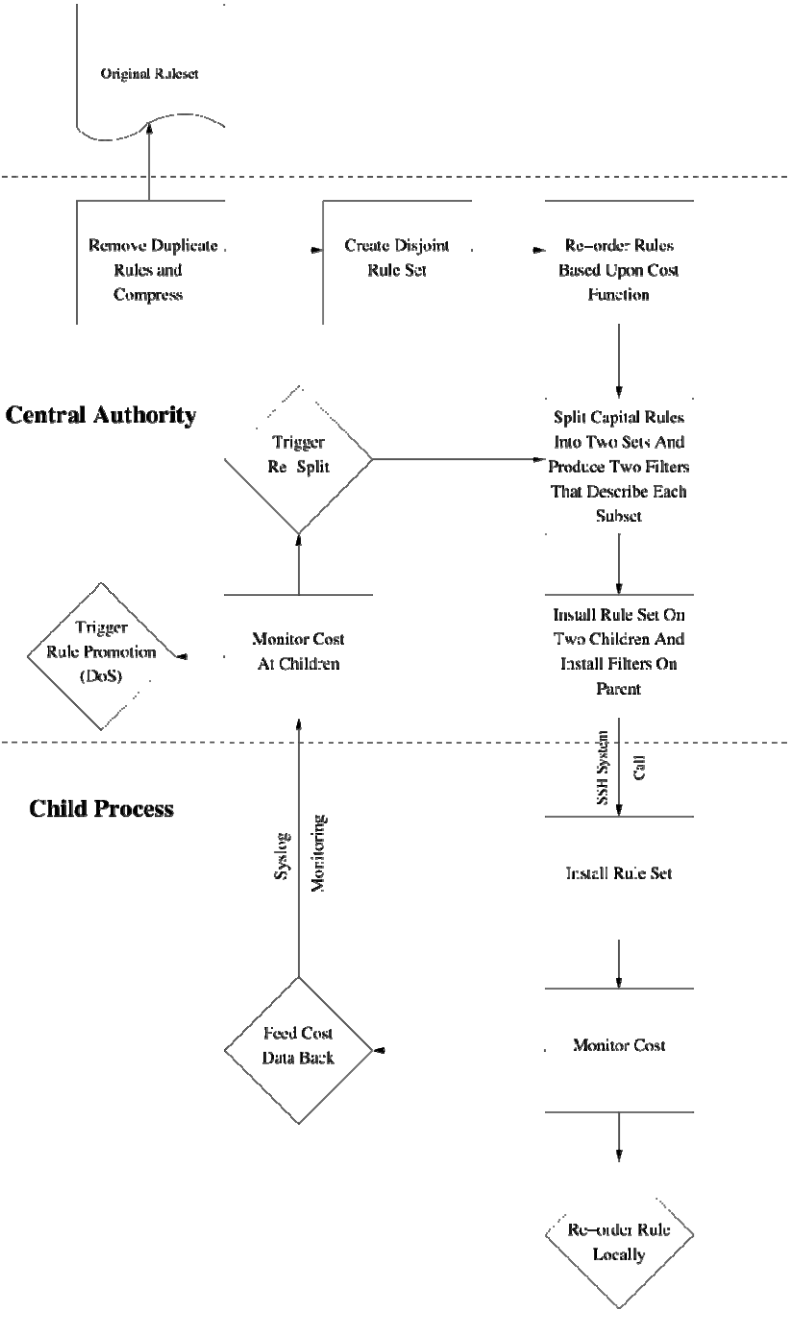

Fig. 3 OPTWALL Architecture

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*. The distance is based on the entire tuple.

The numerical value between two *IP* addresses represents the distance between them. If the *IP* addresses represent ranges, the distance function based on the distance between the two farthest 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 hierarchical structure without violating the semantic integrity of the original rule-set, aiming at improving the operational cost of the firewall. Ideally, optimal splitting ensures that, at the end of the partitioning process, all subsets have 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 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.

In the next section we present the detailed evaluation study of the proposed *OPTWALL* implementation.

6 Evaluation

In this section we describe the experiments and evaluations to validate the *Hierarchical Firewall Optimization* approach. We perform our validations by improving on the widely used open source firewall, *Linux IPCHAINS*. The data used for our experiments is emulated data from a large *Tier-1 ISP*. Our choice of firewall is representative of list-based firewall, which is the focus of this research.

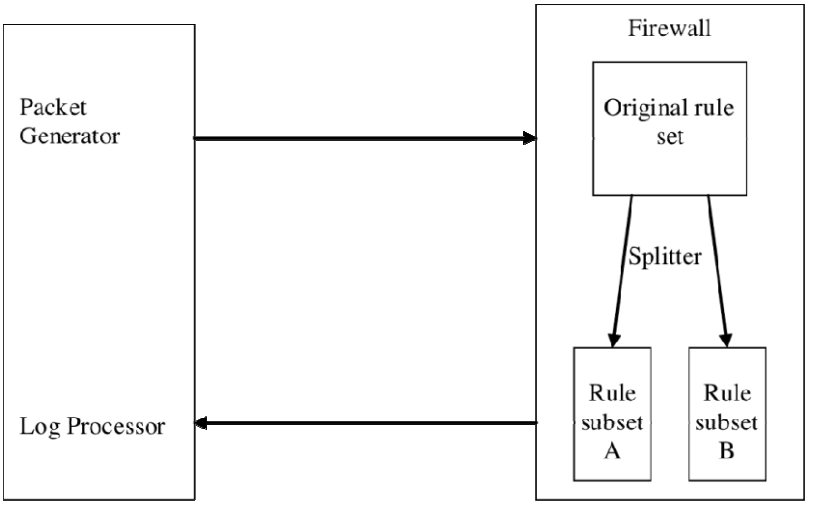

Fig. 4 Experimental Setup

The experimental setup for the evaluation of the proposed 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* Processor *3000+* running *Ubuntu Linux* operating system. The machines are isolated for testing to ensure that there are no other variants. Figure 4 shows the block diagram of the experimental setup.

There are two types of traffic characterizations used to evaluate *OPTWALL*, namely, the *worst case* and the *emulated case* behavior. In the *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 filtered only by the default action tuple. The emulated traffic is generated by creating packets that match each tuple and proportionally instantiating them to a traffic trace similar to that of a large *Tier-1 ISP*'s firewall operation. The worst case traces are used to study the worst case performance of *OPTWALL* in comparison to the baseline case, a *list-based firewall*. Performance at worst case is determined by using constant traffic rates and measuring the overall CPU utilization. Traffic rates are determined by loading the fire- wall from *25%* to *100%* utilization with the installed list-based rule-set. A similar approach is used to determine the load for the emulated traffic evaluations.

6.1 Evaluation Results

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

6.1.1 Hierarchical Model Evaluation

This study is 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 are conducted on a heavily loaded system and using the best performing heuristic amongst all the solutions proposed earlier in the chapter. 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 5 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 depict a way to arrive at a sweet spot between the number of re-splits and the gain to due the hierarchical design.

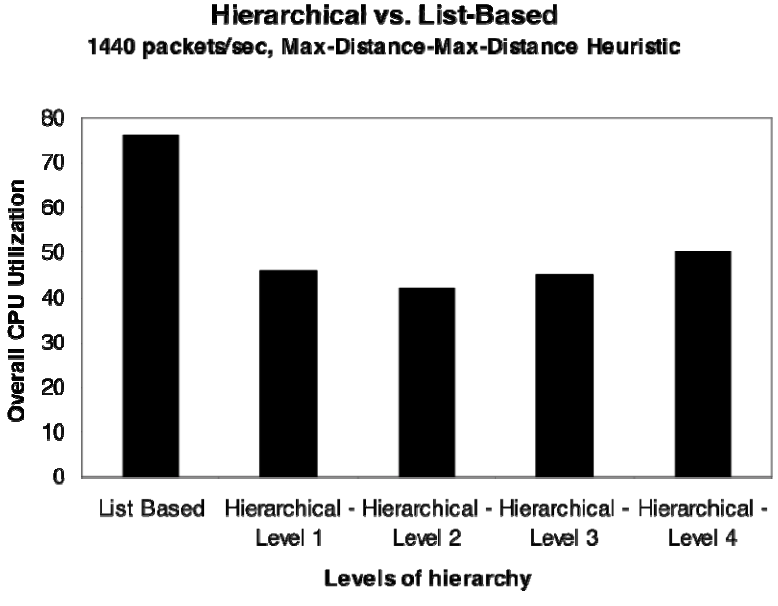

Fig. 5 Hierarchical vs. List-based performance

6.1.2 Worst case Performance Evaluation

The next study performed is to deter- mine the worst case packet processing cost of the firewall. A worst case packet processing occurs when every packet entering the system requires processing of the entire tuple subset. This means that it will match the last tuple, which is default deny. We used various tuple sizes for our evaluations. The results are for a typical large tuple set, consisting of 60,000 tuples. Due to the memory limitation of using the Optimal Approach, we use a

pruned approach, ~A* Approach for our evaluation. Results in Figure 6 demonstrate that the ~A* Approach and Max distance-Max distance Heuristic perform best in comparison to the base- line list-based approach. It is to be noted that filters output by the ~A* Approach perform better traffic filtering than the heuristics approaches.

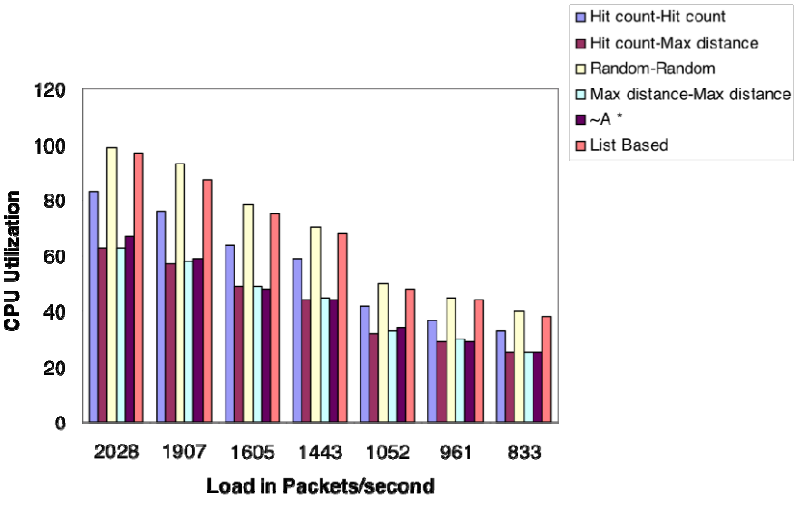

Fig. 6 Performance evaluation (Worst case – 60,000 tuples)

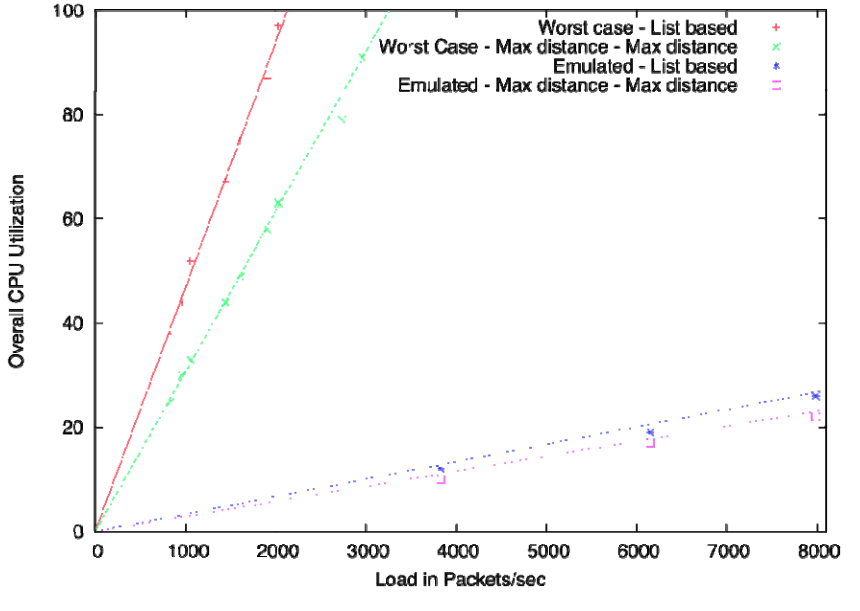

Fig. 7 Emulated traffic performance evaluation

6.1.3 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 7 show the benefit of the proposed scheme. 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.

6.1.4 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 8 illustrates an instance of a large hit-count for a default action tuple.

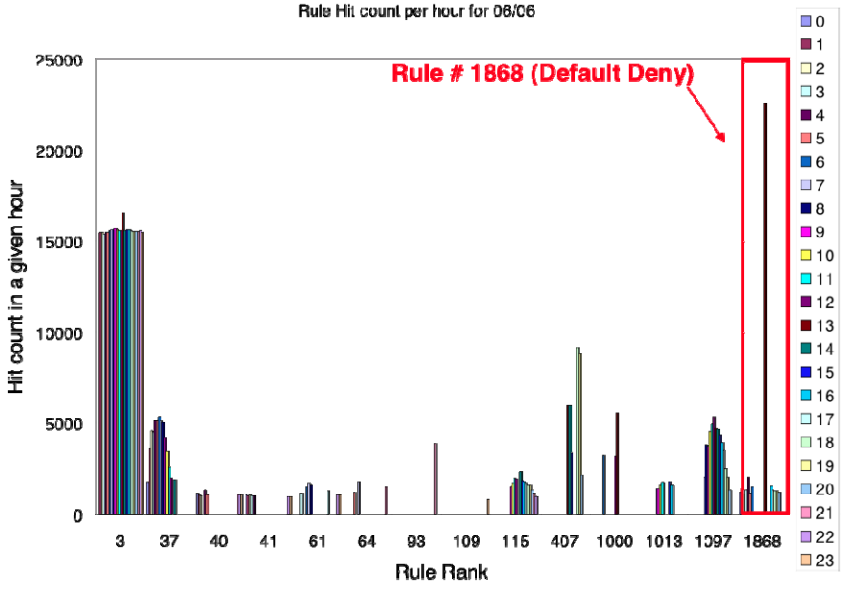

Fig. 8 Default deny rule hits

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 9 shows the competence of *OPTWALL* in countering dynamic traffic changes and hence aiding the steady maintenance of firewall operation.

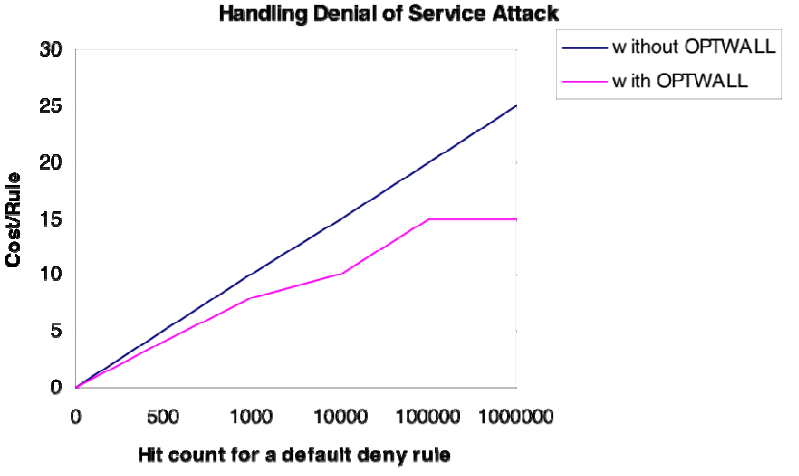

Fig. 9 Countering *DoS* attacks

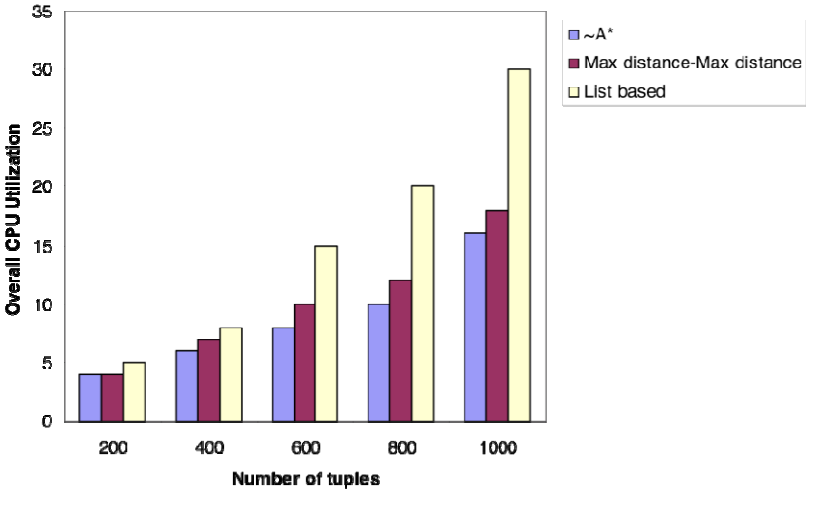

Fig. 10 Sensitivity analysis

6.1.5 Sensitivity Analysis Evaluation

The final study is aimed at *sensitivity analysis* of the proposed approaches. The analysis is performed for tuple sizes varying from *0 - 1000* tuples. Figure 10 details a comparative study between the baseline list-based, the best per- forming heuristic solution and the *~A* approach*. The evaluation is conducted for a heavily loaded firewall operation. From the results we conclude that the proposed heuristic solutions are best suited for hierarchical firewall optimization.

6.1.6 Improved Rule Splitting

In this study we evaluate the benefit of the improved traffic-aware splitting approach presented in Section 1.4.3. The evaluation is for the worst case and the emulated case operation of the firewall. The traffic load is *2000* packets/second and *7000* packets/second for worst case and emulated case, respectively. We invoke the best performing *Max-distance-Max-distance heuristic* to determine encompassing filters from the resulting cluster groups. The result is averaged over *20* runs of the experiment. The *Y-axis* represents the *CPU-Utilization* and the various approaches are represented in the *X-axis*. Results in Figure 11 and Figure 12 demonstrate the benefit of the proposed approach in improving the operational cost of firewall.

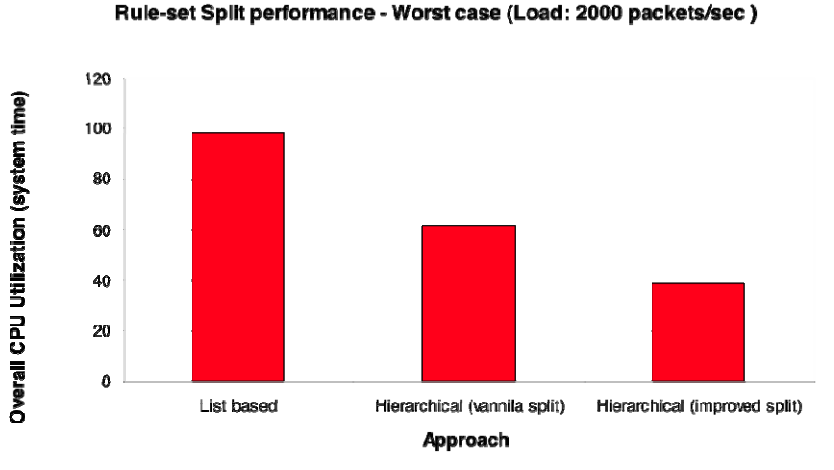

Fig. 11 Weighted split performance – Worst case

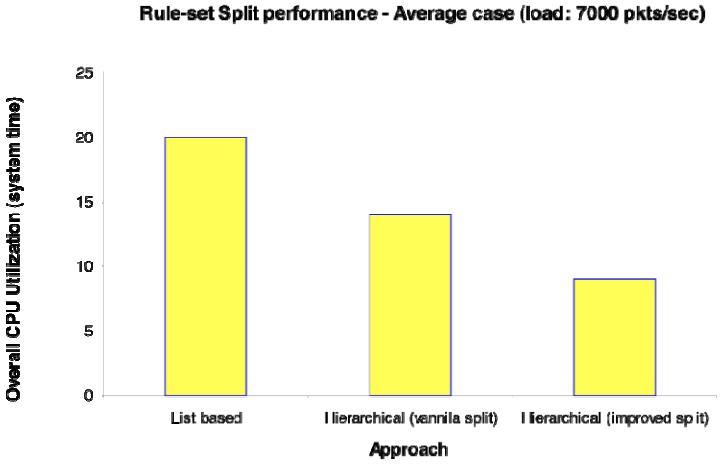

Fig. 12 Weighted split performance – Emulated case

7 Conclusion

This chapter introduces a novel firewall transformation framework and transformation approaches aimed at improving the performance and manageability of de-centralized firewall operation. We introduce a hierarchical splitting approach via the *OPTWALL* toolset. *OPTWALL* helps to achieve the maximum benefit via various splitting processes to arrive at feasible optimal and near optimal solutions. We study the performance of *OPTWALL* both for worst case and normal firewall operation. We also introduce a novel adaptive anomaly detection/countermeasure mechanism to deal with short term and long-term anomalies. Our proposed model and tool is flexible to be used in different firewall environments and data sets. We believe this research presents the design of a complete optimizing toolkit for firewall optimization. Results demonstrate that the proposed approach aids in improving the performance of de-centralized firewall operation.

References

[1] Denial of Service, `http://www.cert.org/homeusers/dos.html`
[2] Lakshman, T.V., Stidialis, D.: High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In: Proceedings of SIGCOMM. ACM Press (1998)
[3] Srinivasan, V., Suri, S., Varghese, G.: Packet classification using tuple space search. In: Proceedings of SIGCOMM. ACM Press (1999)
[4] Linux ipchains, `http://people.netfilter.org/rusty/ipchains`

[5] Hamed, H., Al-Shaer, E.: Dynamic rule-ordering optimization for high-speed firewall filtering. In: ASIACCS (2006)
[6] A* Search Algorithm, http://en.wikipedia.org/wiki/A*_algorithm
[7] Acharya, S., Abliz, M., Mills, B., Greenberg, A., Znati, T., Ge, Z., Wang, J.: Optwall: A hierarchical traffic-aware firewall. In: 14th Annual Network and Distributed System Security Symposium, San Diego, CA (February 2007)
[8] Brucker, P.: On the complexity of clustering problems. In: Optimization and Operations Research, pp. 45–54. Springer (1977, 1997)
[9] Charikar, M., Guha, S., Tardos, Shmoys, D.B.: A constant-factor approximation algorithm for the k-median problem. In: ACM Symposium on Theory of Computing (1999)
[10] Acharya, S., Wang, J., Ge, Z., Znati, T., Greenberg, A.: Traffic-aware firewall optimization strategies. In: IEEE International Conference on Communications, Istanbul, Turkey (June 2006)