A General Framework for Benchmarking Firewall Optimization Techniques

Ghassan Misherghi, Lihua Yuan, Zhendong Su, Chen-Nee Chuah, and Hao Chen

***Abstract*—Firewalls are among the most pervasive network security mechanisms, deployed extensively from the borders of networks to end systems. The complexity of modern firewall policies has raised the computational requirements for firewall implementations, potentially limiting the throughput of networks. Administrators currently rely on *ad hoc* solutions to firewall optimization. To address this problem, a few automatic firewall optimization techniques have been proposed, but there has been no general approach to evaluate the optimality of these techniques. In this paper we present a general framework for rule-based firewall optimization. We give a precise formulation of firewall optimization as an integer programming problem and show that our framework produces optimal reordered rule sets that are semantically equivalent to the original rule set. Our framework considers the complex interactions among the rules in firewall configurations and relies on a novel partitioning of the packet space defined by the rules themselves. For validation, we employ this framework on real firewall rule sets for a quantitative evaluation of existing heuristic approaches. Our results indicate that the framework is general and faithfully captures performance benefits of firewall optimization heuristics.**

***Index Terms*—Firewall optimization, ACL optimization, firewall management, ACL partitioning.**

I. Introduction

FIREWALLS are widely deployed in the current Internet as defense mechanisms to block unwanted traffic. With this ubiquity, many factors have led firewall configurations to become increasingly complex, including the introduction of new protocols and the regular discovery of worms. These factors reflect the increased functionality and utilization of the Internet. As one would expect, the computational requirements for enforcing these policies have risen over the years.

In the most abstract sense, firewalls are mechanisms that enforce network policies. Firewalls are best suited to network policies that involve defining the communication access privileges between networks or hosts. These access privileges most typically involve network, protocol, session, and host restrictions. Firewalls enforce these policies by mediating the communication among hosts in different networks. Upon receiving a packet, a firewall checks the packet's header against a set of user-specified rules (*inspection*) and forwards/drops the packet if it is desired/undesired (*filtering*). Through packet inspection and filtering, firewalls can intercept suspicious packets and prevent them from passing through. A firewall can enforce a complete network-wide access policy if all incoming/outgoing packets are configured to pass through the firewall.

Although packet inspection and filtering help improve network security, it is important to ensure that they do not encumber the availability and utility of the entire system. A firewall cannot forward a packet until inspection has finished. Therefore, inspection will incur additional latency to packets. With limited buffer space, prolonged packet inspection time may also cause the firewall to drop packets indiscriminately. The performance of a firewall should not be mitigated when under attack, otherwise its purpose would have been defeated. Although one could expect the rapid advancement of hardware to help alleviate this challenge, hardware upgrades may not always be practical. In addition, the increasingly popular *deep packet inspection* (DPI), which requires application-level interpretation of the packet data payloads, may incur additional performance overhead, outweighing the benefits provided through better hardware. We believe that firewall performance will remain a challenging issue.

The key component of a firewall configuration is the access control list (ACL). An ACL consists of an ordered list of rules, each with a predicate that describes which packets are matched by this rule and the action to be taken on matched packets. Contemporary firewalls provide numerous actions: a packet may be dropped, accepted, sanitized, transformed, logged, and nearly any combination thereof. A rule-based firewall maps the logic specified in the ACL to a list data structure. A packet is compared with each rule successively in the sequence until the *first matching* rule is found, and the action for this rule is taken on the packet. Many firewall implementations have slightly different semantics, such as *last matching*, *last with first matching*, and *conditional subsequences*. However, all these variations are equivalent to the first matching rule semantics, as can be shown through straightforward rule transformations. Rule-based firewalls, with popular models such as Cisco System's PIX firewall [6], Linux's Netfilter [15], and the BSD Packet Filter [14], are widely used in production networks.

Checking a packet against rules takes processing time, and thus to minimize firewall load and latency one can reduce the number of checks required for packet processing. Previous research has proposed a number of techniques to reorder the rules for better firewall performance [1, 7, 8, 12]. These

Manuscript received June 13, 2008. and accepted Feb. 26, 2009. The associate editor coordinating the review of this paper and approving it for publication was J. L. Hellerstein.

This research was supported in part by NSF NeTS-NBD Grant No. 0520320.

L. Yuan is with Microsoft (e-mail: lyuan@microsoft.com).

G. Misherghi is with Google (e-mail: ghassan@gmail.com).

Z. Su, C.-N. Chuah, and H. Chen are with the University of California, Davis (e-mail: {su, chuah, hchen}@ucdavis.edu).

This work was done while G. Misherghi and L. Yuan were with the University of California, Davis.

Digital Object Identifier 10.1109/TNSM.2009.041104

proposed techniques are heuristic and do not produce optimal rule reorderings. They either profile the rules to determine their importance (*e.g.*, by maintaining counters on rules as they match packets) or model dependencies among rules (*e.g.*, by checking whether the rules intersect but have different actions). This is imprecise because these techniques do not accurately capture the complex interactions among the rules (see Section V for a detailed discussion and examples). Without an approach that produces optimal rule reorderings, it is difficult to evaluate the relative benefits of firewall optimization techniques.

In this paper, we develop a general framework for rule-based firewall optimization. Our framework precisely captures the semantics of an ACL in terms of whether each packet is accepted or rejected. To achieve this, it divides the packet space into independent *partitions* to correctly consider the changed set of packets matched by rules as the packets are processed within an ACL. In addition, compared to existing approaches, we require only that the action taken for each packet remains the same, rather than the rules themselves. With such a precise model, our framework is able to find the optimal rule reordering. Thus, it can also be used to compare and evaluate other optimization approaches and understand their practical benefits or limitations. This paper chooses to focus more on the optimality of rule orders generated by the optimization because its direct impact on firewall performance. We put less focus on the running times of the optimization algorithms because it does not affect firewall performance and is an one-time offline process. We also limit our scope to optimization techniques for rule-based firewalls with stateless packet inspection. Optimizing application-based firewalls with stateful and deep packet inspection would require a significant extension to the existing techniques and is beyond the scope of this paper.

We summarize the main contributions of this paper:

- It provides the first algorithm that, given an ACL and a traffic profile, produces the *optimal* reordered rules. The algorithm is based on a novel rule-based partitioning of the packet space and a reduction to integer programming;
- It formally establishes the correctness of the algorithm. The proof is through a semantic formalization of firewalls and their equivalence, and an equivalence argument connecting this formalization with the reduction to integer programming; and
- It provides an evaluation framework for rule-based firewall optimization techniques because of the guarantee of optimal rule reordering. In particular, it has been used to empirically evaluate two representative heuristic algorithms [8, 12] and an additional one introduced in this paper on production firewall configurations. Results indicate that it is effective for understanding the trade-offs of firewall optimization techniques.

The rest of the paper is structured as follows. Section II shows an example network and policy that we will use throughout the paper to illustrate our approach. Section III presents details of our optimization algorithm, followed by a formal analysis of its correctness (Section IV). Then, in Section V, we discuss two representative existing approaches and introduce a new heuristic approach for rule-based firewall optimization. Finally, Section VI presents results of our empirical evaluation of these approaches within the framework, and Section VII concludes the paper.

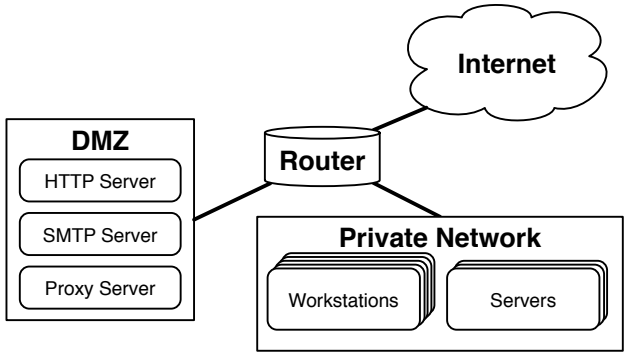


Fig. 1: An example network topology.

1:	**allow** dnet *DMZ* dport *http*
2:	**allow** dnet *DMZ* dport *mail*
3:	**deny** dport *sql*
4:	**allow** snet *DMZ* dnet *Internet*
5:	**allow** snet *Private* dnet *DMZ*
6:	**deny** all

ACL 1: An ACL to enforce the required policy for the network shown in Figure I.

II. Example Network and Policy

To illustrate the application of our framework, we will apply it to a simple, though realistic example. Figure 1 depicts the network topology for this example. The router shown in the figure mediates all communication among the private network, the demilitarized zone (DMZ), and the Internet. For this network, we assume the following required policy:

(R1) Communication sessions initiating from the Internet are only allowed for http and smtp connections to the DMZ servers;

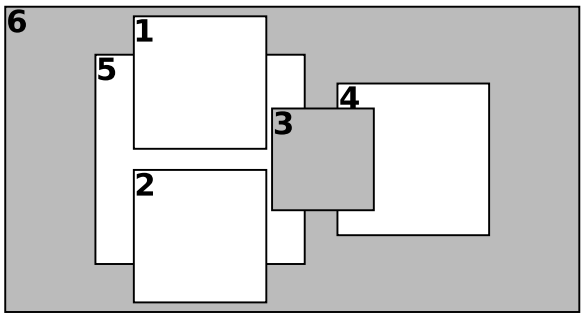
(R2) Communication sessions initiating from the private network are not allowed to the Internet. Instead, users must make external requests through the proxy server in DMZ;

(R3) Communication sessions initiating from the DMZ to the private network are not allowed; and

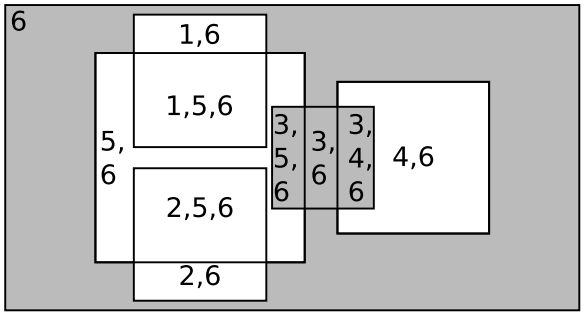
(R4) Due to the prevalence of worms, any inter-network communication to database servers is not allowed. This requirement takes precedence over the first three.

A network administrator may choose to use a firewall to enforce this policy. Conveniently, all inter-network communication must travel through the router shown in Figure **??**. We will assume that the administrator installs a firewall there.

For this example, we will use a simple ACL language, abstracting many protocol details. A single rule begins with the action (**allow** or **deny**), followed by the predicate against which to check packets. The predicate is simply the conjunction of several atoms, where a single atom is a requirement on the source/destination network of a packet (snet/dnet



(a) Overlapping rules



(b) The rule-based partition

Fig. 2: The packet space divided by ACL 1.

Algorithm 1 Partition the Packet Space

Require: $|r| = n$, $n \geq 0$
Ensure: Γ is the rule-based partition

```
1:  Γ ← P
2:  for i ← 1 to n do
3:      x ← packets(r_i)
4:      for all γ in Γ do
5:          if γ ∩ x = ∅ then
6:              continue
7:          else if x ⊂ γ then
8:              Γ.append(γ \ x)
9:              γ ← x
10:             break
11:         else if x ⊃ γ then
12:             x ← x \ γ
13:         else
14:             Γ.append(γ \ x)
15:             γ ← x ∩ γ
16:             x ← x \ γ
17:         end if
18:     end for
19:     Γ.append(x)
20: end for
```

NETWORK) or the source/destination port of the packet (sport/dport *PORT*). One can replace the predicate with the keyword *all*, denoting that every packet shall be matched by this rule.

ACL 1 illustrates how the above policy can be enforced. Requirement **R1** is achieved by the combined effect of Rule 1 (http) and Rule 2 (mail). Requirement **R2** is achieved by the combined effect of Rule 5, which allows the users in the private network to initiate connections to the proxy server in DMZ, and Rule 4, which allows the proxy server to initiate connections to the Internet. Rule 3 enforces **R4**, which has the highest priority in the policy. Rule 6 denies all other traffic, which also implicitly enforces **R3**. This seemingly simple configuration might be the result of thoughtful deliberation of an administrator. Because packets matched by Rules 1 and 2 can never be matched by Rule 3, the administrator can choose any ordering. One possible motivation to put Rules 1 and 2 earlier might be that the administrator expects more packets to be matched by these rules. However, if Rule 3 were placed behind Rules 4 and 5, database sessions from the DMZ to the Internet or from the private network to the DMZ would be incorrectly allowed. Therefore, even if Rules 4 and 5 might match more packets and putting them earlier would reduce the computation, the administrator cannot do so because of the defined policy. Rule 6 ensures ACL will not permit any traffic unintentionally and is the default behavior of most firewall products.

III. Optimization Framework

Our framework consists of several steps: *partitioning*, *profiling*, *dependency generation*, and *optimization*. The first step, partitioning, is used to divide the packet space into disjoint *blocks* according to the given ACL. The profiling step then measures the weights of blocks within the partition. The next step, dependency generation, examines the partition and rules to create a set of constraints on the positions of rules to admit only semantically equivalent rule reorderings. Finally, the optimization step uses information from previous steps to produce an integer program whose solutions yield semantically equivalent, *optimal* rule reorderings.

A. Rule-based Partitioning of Packet Space

Rule-based partitioning of the packet space is a key step in our optimization framework. The (disjoint) blocks of the partition are created such that for any two packets within a single block, the same set of rules from the ACL matches those two packets. This facilitates the correct optimization of firewall rule configurations in two key ways: (1) since all packets within a block will be matched by the same rule in any reordering of the ACL, checking for correct block action is sufficient; and (2) cost assignment can be attributed to blocks rather than rules, thus making cost calculation independent of the choice of rule ordering. As one shall see from Section V, the weight of a rule, as used in some optimization techniques, can actually vary on different Rule 5 order. Optimization techniques that failed to realize this depedency will not discover the optimal rule order.

To explain rule-based partitioning, we first demonstrate it on ACL 1. Figure 2a shows rectangles that represent the rules of the ACL. Light rectangles denote rules that have **allow** actions, while dark rectangles denote rules that have **deny** actions. Notice that the rectangle containing the entire figure represents Rule 6 (**deny *all***). Two rectangles overlap when the packets matched by the two corresponding rules intersect. For example, Rule 6 matches all packets, so it contains all the other rules. When two rules overlap, the rule that appears first

in the ACL is placed above the other. With some reorderings of the ACL, an intersecting region may change its action. This implies that rule reordering would not treat all packets identically. In essence, the problem of correct rule reordering requires that the final "picture" must look identical. These intersecting regions essentially represent sets of packets that are matched by multiple rules. By viewing these regions more precisely as disjoint blocks in a partition, we arrive at Figure 2b.

As a motivation for later phases of our framework, notice that we can infer several constraints on rule orderings from the relationship of rules associated with the partition. Since Rule 6 intersects with all the other rules, it must be placed after all the other rules with an **allow** action. Rule 5 has the most interesting relationships: it intersects with Rules 1, 2, 3, and 6. We can infer that Rules 1, 2, and 5 can be placed in any order relative to one another, while Rule 3 must be placed before Rules 4 and 5.

Algorithm 1 produces a partition of the packet space where packets in each block have the exact same set of matching rules. The algorithm works by iterating over the rules in the rule sequence, as seen in the outer loop on line 2. The loop has a simple invariant: Γ describes a partition of P, the packet space, where all packets in each block are matched by the same rules from the first to the i^{Eh} rule. With an empty rule sequence, the entire packet space is matched by no rules, hence Γ consists of one block. When a single rule is added, this partition will be split into two blocks: those packets that are not matched by the first rule, and those that are. The inductive step, from rule sequences of length i to $i+1$, is more difficult. Suppose the packets matched by the next rule have an intersection with some block γ in Γ. If the rule matches a subset of γ (line 7), then γ must be split into two: the block that also matches the new rule and the block that does not. If the rule matches a superset of γ (line 11), γ remains the same, but we must check for interactions of the remainder of the packets with other blocks. If the rule and γ intersect but neither is contained within the other (line 13), γ must be split into two new blocks: one that is matched by the rule and another that is not; while the remainder of the packets matched by the rule but not in γ must be checked against other blocks.

If a rule intersects with all blocks in the current partition, then the number of blocks doubles. Thus, in the worst case, the number of blocks may be exponential in the number of rules. While this raises complexity concerns, we expect in practice ACLs do not have such complex interactions among rules. Our empirical results in Section VI on production firewall configurations confirm this.

For efficient computation, Algorithm 1 uses Ordered Binary Decision Diagrams (BDDs) [5] to compactly represent blocks and to compute the various set operations performed, such as unions ($\cup$), intersections ($\cap$), and set differences ($\setminus$). Yuan *et al.* previously used BDDs to implement a static analysis tool for checking misconfigurations in firewall configurations [20]. Their implementation is very scalable. For example, it takes less than three seconds to check an ACL with more than 800 rules.

B. Partition Profiling and Rule Cost

A good metric for ACL cost is the expected time to process a single packet. Intuitively, with a lower packet processing time, the firewall can achieve higher throughput. To measure the expected time, we need some representative distributions, *i.e.*, probability mass functions over all packets in the packet space. For this discussion we denote the packet space as P; the ACL as r; the discrete random variable that assumes a packet as X; and the traffic profile as $\mathsf{profile}$, a mapping from packets to probabilities.

Assuming that cost is proportional to the number of checked rules, the expected cost to process a packet is the sum, over all packets in P, of the probability of the packet multiplied by the number of rules checked for that packet. Assuming a unit cost for all rule predicates, the expected cost can be stated as the following:

$$E[\mathsf{cost}(r, X)] = \sum_{s \in P} \mathsf{profile}(s) \times \mathsf{cost}(r, s)$$

where the cost to process a packet, $\mathsf{cost}(r, s)$, is the number of rules that are checked against s. Assuming that r_i is the first matching rule for packet s, we have $\mathsf{cost}(r, s) = i$.

Notice that with our rule-based partitions, the first matching rule for all packets in a block is identical. So for some block γ in Γ where the first matching rule is r_j, for each packet s in γ, $\mathsf{cost}(r, s) = j$. We will use a single value $\mathsf{c}_{r,\gamma}$ to denote the cost required for each packet in γ. This allows us to rewrite the expected cost as the following:

$$E[\mathsf{cost}(r, X)] = \sum_{\gamma \in \Gamma} \left(\mathsf{c}_{r,\gamma} \times \sum_{s \in \gamma} \mathsf{profile}(s) \right)$$

After continuing this factorization, we can sum the probabilities of all packets within the block to produce a single weight. Notice that these weight factors depend only on the traffic profile and are independent of any rule order. This leaves us with:

$$E[\mathsf{cost}(r, X)] = \sum_{\gamma \in \Gamma} \left(\mathsf{c}_{r,\gamma} \times \mathsf{weight}_{\gamma} \right) \tag{1}$$

To apply this definition of cost, we must address the measurement of traffic profiles for a given network and firewall configuration. We assume that a traffic profiling tool will provide the traffic statistics of each partition. Based on the traffic statics for each partition, the weights can be normalized so that the cost reflects the actual number of rules processed. Traffic profiles are also likely to change based on times of day and other factors. Therefore, it is necessary for the traffic profiling tool to dynamically monitor the traffic and update the traffic profile as needed. ProgME [19] is a highly scalable traffic measurement tool that can achieve the requirements here. ProgME allows administrators to write a small definition for the partitions of interest and collects traffic statistics for each partition directly.

We can apply this definition to our example, ACL 1. Assume that the administrator selects a distribution of the packet space such that the weights are determined to be as shown in Table I. Assume that r is the initial ACL. The cost for each block is shown in the third column of the table. The final column shows a different permutation of the ACL, r',

TABLE I: Weights for the blocks in ACL 1.

block	weight	cost(r)	cost(r')
$\{6\}$	0.02	0.12	0.12
$\{1,6\}$	0.05	0.05	0.20
$\{2,6\}$	0.05	0.10	0.25
$\{3,6\}$	0.02	0.06	0.02
$\{4,6\}$	0.30	1.20	0.90
$\{5,6\}$	0.10	0.50	0.20
$\{1,5,6\}$	0.20	0.20	0.40
$\{2,5,6\}$	0.20	0.40	0.40
$\{3,4,6\}$	0.03	0.09	0.03
$\{3,5,6\}$	0.03	0.09	0.03
total	1.00	2.81	2.55

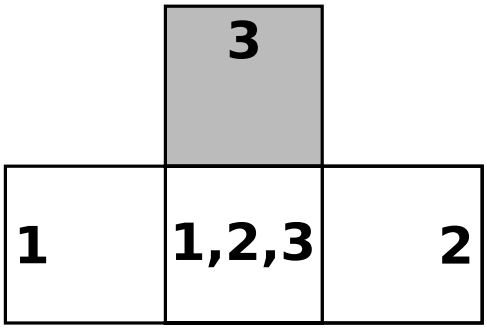


Fig. 3: An ACL demonstrating complex dependencies.

where the rules are reordered as $\langle 3,5,4,1,2,6\rangle$. This ordering retains the same meaning, but has a lower cost. In the later sections, we will show how our framework will select this order.

C. Dependency Generation

We have seen that one can infer dependencies between the relative positions of two rules that overlap, but with different actions. For example, we noted in ACL 1 that Rules 3 and 5 intersect and have different actions, so Rule 3 must be placed before Rule 5 in any valid reordering. While this reasoning suffices for most dependencies, it can be too conservative in general. Figure 3 shows an example where this form of constraint dependence is not sufficient. Notice that all three rules have a common intersection. Rule 3 has a different action from those for both Rules 1 and 2. Were we to use the above described dependencies, we would require that for any valid rule ordering, Rule 3 must be placed after Rules 1 and 2. This is too conservative: if Rule 3 is placed after either Rule 1 or Rule 2, the overlapping block will assume the correct action and thus the rule ordering will be correct. Thus, a dependency must not be between a rule pair. Instead, it should be between a rule i and a block's matching rules that have a different action from that of rule i. We denote these dependencies using the following format: $i \sqsupset \{j,k,\ldots,l\}$. Such a dependency requires that rule i must follow the *earliest rule* of $\{j,k,\ldots,l\}$.

Algorithm 2 shows how these constraints can be generated. It generates a dependency whenever there is a block with conflicting rules (lines 2–4). For these conflicting blocks, a dependency is generated for each rule with a different action from the first matching rule (lines 3–4). The constraint requires

Algorithm 2 Dependency Generator

Require: r is an ACL, with Γ as the rule-based partition
Ensure: D contains a set of required dependencies
1: $D \leftarrow \emptyset$
2: **for all** γ in Γ **do**
3: **for all** $x \in \mathsf{matchers}(r,\gamma)$ **do**
4: **if** $\mathsf{action}(x) \neq \mathsf{action}(\mathsf{firstMatch}(r,\gamma))$ **then**
5: $D \leftarrow D \cup \{x \sqsupset \mathsf{matchersCorrectAction}(r,\gamma)\}$
6: **end if**
7: **end for**
8: **end for**

that in any reordered ACL, the rule must be positioned after at least one of the matching rules with the same action (line 5). This flexible dependency relation of allowing any of the rules with correct actions to be processed first requires exactly that for all blocks and does not change the taken action. Intuitively, this requirement is all that is necessary to ensure that all packets are classified correctly.

For ACL 1, the dependencies are given by:

$$D = \left\{ \begin{array}{l} 4 \sqsupset \{3,6\}, 5 \sqsupset \{3,6\}, 6 \sqsupset \{1\}, 6 \sqsupset \{2\}, \\ 6 \sqsupset \{4\}, 6 \sqsupset \{5\}, 6 \sqsupset \{1,5\}, 6 \sqsupset \{2,5\} \end{array} \right\}$$

The permutations that satisfy the dependencies D include:

$$\begin{array}{lll} \langle 1,2,3,4,5,6\rangle & \langle 2,1,3,4,5,6\rangle & \langle 1,3,2,4,5,6\rangle \\ \langle 2,3,1,4,5,6\rangle & \langle 3,1,2,4,5,6\rangle & \langle 3,2,1,4,5,6\rangle \\ \langle 1,2,3,5,4,6\rangle & \langle 2,1,3,5,4,6\rangle & \langle 1,3,2,5,4,6\rangle \\ \langle 2,3,1,5,4,6\rangle & \langle 3,1,2,5,4,6\rangle & \langle 3,2,1,5,4,6\rangle \end{array}$$

After checking the cost of all permutations given the weights listed in Table I, the permutation $r' = \langle 3,5,4,1,2,6\rangle$ is found to be an optimal solution, with cost as shown in the table. Of course, exhaustively listing all such permutations is computationally expensive. To alleviate this, we formulate the constraints and cost as an integer program in order to take advantage of the modern advances in solving integer programs.

D. Integer Program Formulation

To formulate our problem as an integer program, several high-level concepts must be modeled. The most obvious of these is that the encoded solutions must describe some permutations of the given ACL. We introduce Boolean variables (over 0 and 1) to indicate where rules have been moved in a particular solution's ordering. These Boolean variables are written as $\mathbf{r}_{i,j}$, and take the value 1 only when the i^{th} rule in the original ACL is moved to position j. Naturally, any rule can be placed in any position, so i and j both range from 1 to the length of the ACL, n. The cost function described in Equation 1 requires terms for each block in the rule-based partition. Although the weight is constant, the cost function requires a factor set to the least position of some set of rules (the first matching rule for the block). Our dependency constraints also require that rules be placed after the least position of the same set of rules. This is difficult to model, as there is no such minimum selection that can be inserted directly into our linear equations. Our solution to this problem is to introduce additional Boolean variables $\mathbf{g}_{i,j}$, set to 1 only when the i^{th} block in Γ is matched by the j^{TM} rule in the

encoded permutation. Let m be the number of blocks in Γ. Consider the following integer program:

Minimize

$$\sum_{i=1}^{m} w_i \sum_{j=1}^{n} j\mathbf{g}_{i,j}$$

Subject to

(A)	$\sum_{j=1}^{n} \mathbf{r}_{i,j} = 1$	$\forall i \in [1,n]$
(B)	$\sum_{i=1}^{n} \mathbf{r}_{i,j} = 1$	$\forall j \in [1,n]$
(C)	$\sum_{j=1}^{n} \mathbf{g}_{i,j} = 1$	$\forall i \in [1,m]$
(D)	$\sum_{j=1}^{n} j\mathbf{r}_{k,j} - j\mathbf{g}_{i,j} \geq 1$	$\forall i,k \,.\, k \in \mathsf{after}(i)$
(E)	$\mathbf{g}_{i,j} - \sum_{k \in \mathsf{matches}(i)} \mathbf{r}_{k,j} \leq 0$	$\forall i \in [1,m], \forall j \in [1,n]$

where the auxiliary functions matches and after are defined precisely in Section IV-B. Intuitively, matches maps a block to its matching rules with the same action as its first matching rule, while after maps a block to its matching rules with a different action from its first matching rule.

Let us first address the choice for the optimization function. As shown in Equation 1, the cost should be the sum of block positions times their weights. Were the variables $\mathbf{g}_{i,j}$ set correctly so that for block i, $\mathbf{g}_{i,j}$ is 1 only for the rule with the least matching position at j, the inner sum would be equal to that position j. This yields that the cost term for each block is correctly equal to the weight of the block times the position of the block. Our optimization function thus faithfully computes the cost described by Equation 1. With a valid cost function, we must now ensure the Boolean variables take the correct values and describe a permutation.

The constraints labeled **A** ensure that a rule is moved to exactly one position in the permutation. The constraints labeled **B** ensure that there is no more than one rule in each position of the permutation. Together they require that solutions describe permutations of the ACL.

Constraint **C** ensures that a block has a matching rule. For this formulation, we will assume that the ACLs have explicit default policies placed at the end of the rulesets, so that this is indeed always the case. We can explicitly place the implicit rule at the end of the rule set for ACLs.

Constraint **D** ensures that the rules that must be placed after the first matching rule of some block are placed so. This is so because the summation requires that the term where $\mathbf{r}_{i,j}$ is set to one has a factor of j, while the first matching rule where $\mathbf{g}_{i,k}$ is set, has a factor k where $k < j$. All other terms are zero, hence the total sum must be greater than zero.

Constraint **E** ensures that the $\mathbf{g}_{i,j}$ is set to one of the matching rules for block i that has the correct action. We can see that for any equation where $\mathbf{g}_{i,j}$ is zero, the equation holds trivially. When $\mathbf{g}_{i,j}$ is nonzero, one of the matching rules must be set for the equation to hold true.

Notice that the Boolean variables, meant to be the "first matching" positions for blocks, may in fact be set to any matching rule for a block. However, if the variable was not set to the first matching position, the cost function could be further optimized by setting it so. Also, a higher position would incur more restriction on rules that have different actions for the block, so it would not necessarily yield the optimal order. In an optimal ordering, the position would be set correctly, hence the integer program formulation is sufficient.

IV. Formal Analysis of Algorithm

In this section, we argue the correctness of our integer program formulation, *i.e.*, that it produces a correct optimal rule reordering.

A. Semantics of Firewalls

We begin our analysis by formally defining the semantics of firewall rule sequences. Our semantic definition uses a set-theoretic model of the problem to facilitate the correctness proof. We first let P be the set of all distinct packets. We make no assumptions about the information contained in the packets.

For our semantic definition, we need to abstractly differentiate between the actions that a firewall can take on a packet. For generality, we are not concerned with the purpose of these actions.

A The set of all packet actions. Without loss of generality, we assume $A = \{\mathsf{accept}, \mathsf{reject}\}$.

R The set of all firewall rule sequences, *i.e.*, $R =: \{r | r : [1,n] \to \mathcal{P}(P) \times A \wedge n \in \mathbf{N}\}$. An individual rule is defined as an ordered pair of matched packets (a subset of P) and the taken action (an element of A). Note that we use $[i,j]$ to denote the inclusive range of integers $\{v \mid i \leq v \leq j \wedge v \in \mathbf{N}\}$. We use the term *rule sequence* rather than ACL to mirror the definition more closely.

$\mathsf{packets} : \mathcal{P}(P) \times A \to \mathcal{P}(P)$ A function that projects a rule to packets that the rule matches: $\mathsf{packets}(S,a) = S$.

$\mathsf{action} : \mathcal{P}(P) \times A \to A$ A function that projects a rule $\langle S, a \rangle$ to its action: $\mathsf{action}(S,a) = a$.

Notice that our definition of a firewall rule abstracts the notion of packet predicates by instead considering the set of packets matched. This not only simplifies the analysis, it also ensures generality with respect to predicate languages.

We now have the necessary background to introduce a semantic definition of rule sequences.

$\sigma : R \times P \to A$ A function that maps rule sequences and packets to actions. This function in effect defines that the action taken on a packet for a given rule sequence is the action of the least matching rule, *i.e.*,

$$\sigma(r,s) = a \quad \text{if} \quad \left(\begin{array}{ll} \exists i \in [1,|r|] \,.\, s \in \mathsf{packets}(r_i) \\ \wedge \quad a = \mathsf{action}(r_i) \\ \wedge \quad \forall j < i \,.\, s \notin \mathsf{packets}(r_j) \end{array} \right)$$

Σ: $R \to P \times A$ The semantic function, Σ, maps rule sequences to a complete classification of the entire packet space. This function models the core semantics of rule sequences:

$$\Sigma(r) = \{(s,a) | s \in P \wedge \sigma(r,s) = a\}$$

We now focus on concepts relevant to rule permutations and rule-based partitions of the packet space. The following definitions are provided for this purpose.

$\mathsf{perm} : R \to \mathcal{P}(R)$ The function mapping rule sequences to all permutations:

$$r' \in \mathsf{perm}(r) \quad \text{off} \quad \left(\begin{array}{l} |r| = |r'| \wedge \\ \exists \delta \,.\, \delta : [1,|r|] \leftrightarrow [1,|r|] \wedge \\ (\forall i \in [1,|r|] \,.\, r_i = r'_{\delta(i)}) \end{array} \right)$$

$\approx_r$ The relation $\approx_r$ relates packets that are matched by the exact same rules in rule sequence r:

$$s_1 \approx_r s_2 \quad \text{Giff} \quad \left(\begin{array}{c} \forall i \in [1, |r|] \,.\, s_1 \in \mathsf{packets}(r_i) \Leftrightarrow \\ s_2 \in \mathsf{packets}(r_i) \end{array} \right)$$

$\Gamma : R \to \mathcal{P}(\mathcal{P}(P))$ The function Γ maps rule sequences to a partition of P where any two packets of a single block are matched by the exact same rules: $\Gamma(r) = P/\approx_r$.

$\Phi : R \times \mathcal{P}(P) \dashrightarrow A$: A partial function mapping subsets of P to actions, providing all packets in the subset will have the same action for the rule sequence:

$$\Phi(r, S) = c \quad \text{eff} \quad \forall s \in S \,.\, \sigma(r, s) = c$$

B. Correctness of Algorithm

We now have the suitable definitions in place to begin our analysis. We first show that regardless of the rule permutation, if two packets are related by $\approx_r$, the same action will be taken on both of them.

***Lemma** 1 ($\approx_r$ implies identical action):* Let r be some rule sequence in R. For any two packets $s_1, s_2 \in P$ such that $s_1 \approx_r s_2$, it follows that $\sigma(r, s_1) = \sigma(r, s_2)$.

Proof: Consider any two packets $s_1, s_2 \in P$ such that $s_1 \approx_r s_2$. By the definition of $\approx_r$, we have $\forall i \in [1, |r|] \,.\, s_1 \in \mathsf{packets}(r_i) \Leftrightarrow s_2 \in \mathsf{packets}(r_i)$. Now assume that $\sigma(r, s_1) = a$ for some action $a \in A$. Then there must exist some i such that $s_1 \in \mathsf{packets}(r_i) \wedge \forall j < i \,.\, s_1 \notin \mathsf{packets}(r_j)$. Furthermore, $\mathsf{action}(r_i) = a$. Since $s_1 \approx_r s_2$, it follows that $s_2 \in \mathsf{packets}(r_i) \wedge \forall j < i \,.\, s_2 \notin \mathsf{packets}(r_j)$. Thus, by the definition of σ, $\sigma(r, s_2) = \mathsf{action}(r_i) = a$. Therefore, $\sigma(r, s_1) = \sigma(r, s_2)$. ∎

We can also show that permuting a rule sequence has no effect on the partition. This is intuitive, as the definition of Γ does not depend on the order of rules, but rather the presence of rules in a rule sequence.

***Lemma** 2 (perm preserves $\approx_r$):* Let $r \in R$ and $r' \in \mathsf{perm}(r)$. For any two packets $s_1, s_2 \in P$, $s_1 \approx_r s_2$ ff $s_1 \approx_{r'} s_2$. Stated alternatively, $\Gamma(r) = \Gamma(r')$.

Proof: Suppose that there are two packets $s_1, s_2 \in P$ such that $s_1 \approx_r s_2$. By the definition of $\approx_r$, $\forall i \in [1 : |r|] \,.\, s_1 \in \mathsf{packets}(r_i) \Leftrightarrow s_2 \in \mathsf{packets}(r_i)$. Since r' is a permutation of r, there exists a remapping function, δ, such that $\forall i \in [1 : |r|] \,.\, r'_{\delta(i)} = r_i$ and therefore that $\forall i \in [1 : |r|] \,.\, s_1 \in \mathsf{packets}(r'_{\delta(i)}) \Leftrightarrow s_2 \in \mathsf{packets}(r'_{\delta(i)})$. Because δ is a bijection from $[1 : |r|]$ to $[1 : |r|]$, so it follows that $\forall i \in [1 : |r|] \,.\, s_1 \in \mathsf{packets}(r'_i) \Leftrightarrow s_2 \in \mathsf{packets}(r'_i)$. Thus $s_1 \approx_{r'} s_2$. Symmetrically, we can show that for any two packets $s_1, s_2 \in P$, $s_1 \approx_{r'} s_2$ implies that $s_1 \approx_r s_2$, thus concluding the proof. ∎

***Theorem** 3:* Let $r, r' \in R$ such that $r' \in \mathsf{perm}(r)$. The two rule sequences r and r' are semantically equivalent, *i.e.*, $\Sigma(r) = \Sigma(r')$, if $\forall \gamma \in \Gamma(r) \,.\, \Phi(r, \gamma) = \Phi(r', \gamma)$.

Proof: First, by Lemma 2, $\Gamma(r) = \Gamma(r')$. Now suppose that $\Sigma(r) = \Sigma(r')$. By Lemma 1, for all $\gamma \in \Gamma(r)$, $\Phi(r, \gamma)$ and $\Phi(r', \gamma)$ are defined. Suppose that for some $\gamma \in \Gamma(r)$, $\Phi(r, \gamma) \neq \Phi(r', \gamma)$. Then there exists some $s \in \gamma$ such that $\sigma(r, s) \neq \sigma(r', s)$. However, this would imply that $\Sigma(r) \neq \Sigma(r')$, which contradicts the assumption that $\Sigma(r) = \Sigma(r')$. Thus $\Phi(r, \gamma) = \Phi(r', \gamma)$.

Conversely, suppose that for all $\gamma \in \Gamma(r)$, $\Phi(r, \gamma) = \Phi(r', \gamma)$. Let S and S' be two sets such that $S = \{(s, a) \mid s \in \gamma \wedge \Phi(r, \gamma) = a \wedge \gamma \in \Gamma(r)\}$ and $S' = \{(s, a) \mid p \in \gamma \wedge \Phi(r', \gamma) = a \wedge \gamma \in \Gamma(r)\}$. By the hypothesis, $S = S'$. Also, by the definition of Φ, $S = \{(s, a) \mid s \in \gamma \wedge \sigma(r, s) = a \wedge \gamma \in \Gamma(r)\}$ and $S' = \{(s, a) \mid s \in \gamma \wedge \sigma(r', s) = a \wedge \gamma \in \Gamma(r')\}$. Now since $\Gamma(r) = \Gamma(r')$ defines a partition of P, we have that $S = \{(s, a) \mid s \in P \wedge \sigma(r, s) = a\}$ and $S' = \{(s, a) \mid s \in P \wedge \sigma(r', s) = a\}$. Finally, by the definition of Σ, $\Sigma(r) = S = S' = \Sigma(r')$. ∎

Stated more simply, Theorem 3 asserts that a permutation of rule sequence r is only semantically equivalent to r when for each block in $\Gamma(r)$ the action of the first matching rule is the same as that of the first matching rule for r.

By reducing the semantic equality of permutations to checking for identical quality on large blocks of the packet space, we can simplify our firewall optimization algorithm so that it needs not consider the action of all packets. The blocks of $\Gamma(r)$ define a very coarse abstraction of the packet space that will always yield a correct analysis for permuted rules.

We now return to the integer program formulation of Section III. We will begin with more precise definitions used in the integer program.

n, m The value n is the length of the rule sequence r, while m is the number of blocks in $\Gamma(r)$, *i.e.*, $n = |r|$ and $m = |\Gamma(r)|$.

$\gamma : [1 : m] \to \mathcal{P}(P)$ For indexing purposes, we will assume an arbitrary ordering of the blocks in $\Gamma(r)$. We will denote this as a sequence γ.

$\mathbf{r}_{i,j}$ A Boolean variable (over 0 and 1) that is set to 1 only when rule r_i of the initial sequence is moved to position j in the current permuted sequence (where $i, j \in [1 : n]$).

$\mathbf{g}_{i,j}$ A Boolean variable (over 0 and 1) that is set to 1 only when the block γ_i has a matching rule with the same action as $\Phi(r, \gamma_i)$ at position j, without any matching rule with a different action in a position less than j (where $i \in [1 : m], j \in [1 : n]$).

w_i: The weight of the block γ_i.

$\mathsf{matches} : [1 : m] \to [1 : n]$ A function that maps a given block γ_i to the set of rules that match packets of γ_i and have the *same action* as $\Phi(r, \gamma_i)$. For showing correctness, we will assume that each partition has at least one matching rule. This requires that the default policy must be made explicit by placing a general rule with the same action at the end of the ACL.

$\mathsf{after} : [1 : m] \to [1 : n]$ A function that maps a given block γ_i to the set of rules that match γ_i but have a different action from $\Phi(r, \gamma_i)$ (and hence must not be the first matching rule for the block).

***Theorem** 4 (Main):* Let r be some rule sequence in R. The solutions to the integer program describe exactly the set of rule permutations that are semantically equivalent to r.

Proof: First we show that a solution to the integer program are permutations of the original rule. Consider a set δ containing (i, j) whenever $\mathbf{r}_{i,j} = 1$. Constraints **A** ensure that for all i in $[1 : n]$, there exists a unique j such that $\mathbf{r}_{i,j}$ is 1. It follows that δ is a function mapping $[1 : n] \to [1 : n]$. Constraints **B** ensure that for all j in $[1 : n]$, there exists a

unique i such that $\mathbf{r}_{i,j}$ is 1. This yields that δ is a bijection, and thus that the solution describes a permutation of r.

Suppose that for some constraint satisfying assignment describing permutation r', there exists some block, γ_i, such that $\Phi(r, \gamma_i) \neq \Phi(r', \gamma_i)$. For r', γ_i's first matching rule, j, must have a different action than the first matching rule for r, so it must be in $\mathsf{after}(i)$. However, it would contradict the constraints **D** for j to be the first matching rule of i, because the block γ_i must have a rule that matches it before j. It therefore cannot be the case that $\Phi(r, \gamma_i) \neq \Phi(r', \gamma_i)$, so $\Phi(r, \gamma_i) = \Phi(r', \gamma_i)$. By Theorem 3, $\Sigma(r') = \Sigma(r)$.

Suppose that there is a permutation of r' such that $\Sigma(r') = \Sigma(r)$. Since r' is a permutation of r, Constraints **A** and **B** can be satisfied by setting the $\mathbf{r}_{i,j}$ variables appropriately. We will also set $\mathbf{g}_{i,j} = 1$ for all γ_i that have first matching rule j in r'. Since there is only one first matching rule for all blocks, Constraints C are satisfied. By Theorem 3, for all γ_i, $\Phi(r, \gamma_i) = \Phi(r', \gamma_i)$. It must follow that all rules in $\mathsf{after}(\gamma_i)$ must occur after the matching rule of γ_i, so constraints **D** are satisfied. If constraints **E** were not satisfied, then there would have been some γ_i and block position j such that $\mathbf{g}_{i,j} = 1$, while there is no rule matching γ_i at position j. However, we have assigned $\mathbf{g}_{i,j} = 1$ only when the first matching rule is at position j, so this cannot be. Thus constraints **E** must be satisfied, and the permutation has a valid assignment.

We conclude that the satisfying solutions of the integer program describe precisely the set of semantics-preserving permutations of r. ■

Theorem 4 guarantees that all and only valid permutations will be captured by the integer program. Because the integer program selects the minimum permutation with respect to cost, we produce rule reorderings with the minimum cost. It is worth noting that although previous work [12] has claimed to produce optimal rule orderings, but because of the issue with position-dependent rule weight, it in fact cannot guarantee optimal rule reordering (see Section V).

C. Time Complexity Analysis

We now discuss the time complexity of the firewall optimization problem and the worst-case time complexity of our algorithm.

Hamed and Al-Shaer show that producing optimal rule orderings of ACLs is NP-complete [12] by a reduction from job scheduling, a well-known NP-complete problem. Thus, in theory, this problem is intractable, unless $P = NP$.

We now analyze the complexity of our algorithm. The partition of the packet space for an ACL may be exponential in the size of the ACL in the worst case. However, it is worth noting that typical ACLs do not exhibit this behavior and they often have blocks linear in number in the ACL size from our experience in analyzing a few production firewall configurations (see Section VI for more details). Next, the integer programs can be generated in polynomial time and are quadratic in size, both in the size of partitions. Finding valid solutions to integer programs is NP-complete. Though our algorithm has a worst-case high complexity, the algorithm is feasible for many practical scenarios as will be shown in Section VI.

V. Heuristic Approaches

In this section, we discuss a few heuristic algorithms for firewall rule optimization: two existing ones and a new algorithm that we introduce in this paper. These will be compared empirically within our framework in the next section. We will also survey additional related work on fast packet classification.

A. Rule-based Optimization

Cisco, in its ACL Manager [7], provides an ACL Optimizer that sorts rules according to the number of hits each rule receives. Acharya *et al.* [1] estimate traffic profile based on simple counters of rule hits and perform optimization based on this. A rule receiving more matching packets is considered "hot" and moved earlier. Cohen and Lund [8] use a greedy algorithm to iteratively move the rule that matches the most packets as early as possible. Hamed and Al-Shaer [12] propose the use of binary integer programming to optimize rule order based on "rule weights," which are evaluated based on both hits and resent.

Essentially, all these rule-based optimization techniques [1, 7, 8, 12] perform traffic-aware rule reordering based on the weights computed once using the original rule set. However, the number of hits received by each rule is dependent on the rule position in the firewall configuration and hence the "weight" will change as the rule is moved up or down the list. None of these prior efforts takes into account this subtlety, and hence the resulting rule orders are sub-optimal.

The following are three major drawbacks of these rule-based optimization techniques:

Position-dependent rule weight: Rule-based optimization cannot find the global optimal for two reasons. First, rule-based optimization measures the weights of a rule directly, using a hit-based traffic profile that counts the number of packets hit each rule. Hit-based traffic profile does not account for the fact that the preceding rules may shield the "hits" for later ones. Consider rules R_1 and R_2 in Figure 4 (ignore R_3 for now). The current traffic profile may find that R_1 and R_2 have weights of 7 and 5 respectively and therefore determine that the current order is optimal with a total cost of $7 * 1 + 5 * 2 = 17$. On the other hand, our approach will discover that the intersection of R_1 and R_2 has a weight of 5. The swapped order will have a total cost of $10 * 1 + 2 * 2 = 14$ instead.

Overly-conservative rule precedence constraints: Another cause of sub-optimality for rule-based optimization is that the rule-precedence constraints are unnecessarily conservative. Consider the case illustrated in Table II. If one generates a rule precedence for every pair of rules that have non-empty intersection but different actions, the current order will be the only feasible solution. However, among the six possible permutations of R_1, R_2, and R_3, only $\langle R_2, R_1, R_3 \rangle$ and $\langle R_2, R_3, R_1 \rangle$ are infeasible. Any of the other four are feasible and should be considered, as in our approach.

Instability due to position-dependent weight: Rule-based optimization not only cannot find optimal rule orderings, but also may require multiple steps of change, even if the actual traffic characteristics remains the same. Consider Figure 4

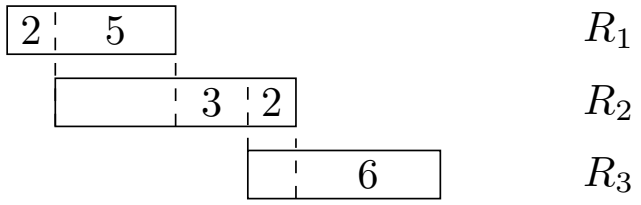


Fig. 4: Hit-based profile.

TABLE II: Example ACL where P_1, P_2, P_3, and P_4 denote disjoint sets of packets.

R_1:	accept	P_1, P_2		
R_2:	deny	P_1,	P_3	
R_3:	accept	P_1,		P_4

again, but now focus on R_3. On the first attempt, rule-based optimization will only move R_3 before R_2. However, once that happens, the weight of R_3 will increase from 6 to 8. The optimization algorithm will then move R_3 before R_1. Instead, our optimization framework attempts to capture the subtle interactions among firewall rules and their relative positions in order to compute the optimal rule ordering.

B. ACL Splitting

We now introduce a three-staged algorithm that: (1) divides a given ACL into cerebellums; (2) solves the cerebellums; and (3) merges the results into a solution for the original ACL. This algorithm is more scalable than our algorithm in Section III because of the smaller problems used for the computationally expensive integer program instances. Extracting the subprograms requires quadratic time in the size of the ACL, as does the merging stage. The main insight for this approach is that a globally aware algorithm needs only to focus on the rules that interact with one another while the outputs can be combined with a locally guided algorithm.

In the first phase, an ACL is split into rule groups by extracting the connected components of the *rule intersection graph*, an undirected graph with rules as nodes and edges between pairs of overlapping rules. In the case illustrated in Figure 5, four groups are created: $G_1 = \{a, b, c\}$, $G_2 = \{e, f\}$, $G_3 = \{d\}$, and $G_4 = \{g\}$. Since all groups are independent of each another, regardless of how groups are interleaved, the weights of individual rules do not change since the relative order of any rule in a single group does not. Using Tarn's algorithm this can be performed in linear time with respect to the size of the graph, and thus quadrille time with respect to the number of rules in the ACL [18].

We optimize each of the groups independently by using the integer program formulation in Section III. The ACLs we use for evaluation in Section VI show that the individual rule groups are often small. This is consistent with the low rule interactions we expect from ACLs used in practice.

Following the subprogram solutions, we interleave the resulting subproblem ACLs into one ACL. With the final ACL, we apply a greedy algorithm [8] that attempts to swap any two consecutive rules as long as the swap yields a lower cost and does not violate the semantic meaning of the original ACL.

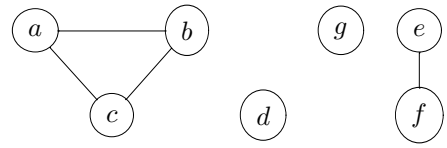


Fig. 5: A rule intersection graph.

TABLE III: ACLs used in our evaluation.

ACL	Source	Rules (#)	Blocks (#)	Groups (#)
A	Major ISP	854	946	11
B	A's core	54	146	10
C	Major ISP	24	49	4
D	Major ISP	141	149	2
E	D's core	13	21	1
F	Major ISP	23	27	2

Note that when any rule is inserted between rules from a single subproblem, the relative order of the rules in that subproblem may no longer be optimal, as the cost for rule placement is no longer consistent. Thus, this algorithm does not necessarily yield optimal ACL orders. In Section VI, we will evaluate how this algorithm performs in practice.

C. Fast Packet Classification

Much research [3, 9, 10, 16, 17] has investigated advanced data structures, such as decision trees, for packet classification. A survey by Gupta and Milken [11] offers a thorough summary of this area. These algorithms essentially trade memory space for faster lookup speed. While they offer significantly better upper-bounds for lookup time, the scalability of these algorithms remains a challenging issue. This is particularly true when more header fields or even data part of a packet need to be inspected. To exploit both the scalability of the list structure and the speed of the advanced data structures, Baboescu *et al.* [2] propose to use these advanced data structures to quickly identify a list of potential matching rules. Our algorithm can be extended to determine the optimal order of the list component and therefore further improve the performance of their work.

Baboescu and Varghese, in their work on Aggregated Bit Vector [4], propose to rearrange rules so that rules with the same value in certain fields are contiguous. This helps to localize the matches and to reduce false matches in the original Bit Vector algorithm. Their algorithm is based on the configuration alone and does not take traffic profile into account. Partition-based traffic profile can help further improve the performance of ABV.

VI. Empirical Evaluation

We have implemented our evaluation framework to measure the optimality of several algorithms: (1) the optimal partition-based algorithm proposed in this paper (**Partition**); (2) the heuristic ACL splitting algorithm proposed in this paper (**Split**); (3) Hamed and El-Shies rule-based algorithm (**Rule**) [12]; and (4) Cohen and Lund's Extended Greedy algorithm (**Greedy**) [8].

TABLE IV: Costs and running times (in seconds) of the four algorithms on the ACLs described in Table III.

ACL	Original	Partition		Split		Rule		Greedy	
	(cost)	(cost)	(time)	(cost)	(time)	(cost)	(time)	(cost)	(time)
A	331.893	∞	∞	119.710	1111.494	59.547	1582.857	192.230	0.799
B	12.771	6.078	13.754	12.640	31.908	7.102	5.649	7.830	0.056
C	7.607	3.769	3.136	3.857	12.513	4.585	3.105	4.948	0.024
D	50.769	17.238	1690.571	18.471	17.447	17.238	36.909	28.684	0.095
E	3.639	3.309	3.090	3.309	3.156	3.309	3.059	3.366	0.009
F	9.241	5.138	3.061	5.656	6.349	5.138	3.104	5.914	0.029

We implemented each algorithm, using shared code for generating traffic profiles, calculating costs, and generating integer programs. We employed an industrial-strength integer program solver, CPLEX [13], to solve our integer programs. For integer programs, the solver uses a branch and bounding algorithm with several heuristics to find optimal solutions. We ran our experiments on a system with a Xeon 2.8Ghz processor, 2GB of RAM, and Linux kernel 2.6.16.

For our evaluation, we have used a number of ACLs. Table III shows descriptions of these ACLs. For each ACL, we show from where we obtained it (**Source**), the number of rules (**Rules**), the number of blocks (**Blocks**), and the number of independent sub-problems (**Groups**).

ACL A is an 865 rule ACL deployed by a major ISP. ACL B is derived from ACL A by removing the rules that do not intersect with any other rules. We do this to understand how the algorithms perform on problems with potentially complex rule interactions. Note that by removing these independent rules, the remaining block count is high with respect to the number of rules. ACL C is also from a major ISP, though it is small and intended for a specific network. ACL D is a reasonably sized ACL deployed by a major ISP. Notice that for this ACL, there are not many complex rule interactions, as the number of distinct blocks is nearly the same as the number of rules. ACL E is identical to ACL D, however we have removed the independent rules again. ACL F is another small ACL deployed by a major ISP though it is specific to a small network. Notice that we include data about the number of groups of each ACL. This number is essentially the number of distinct sub-problems for the splitting algorithm, though we count all independent rules as one group, rather than exacerbating the count by such rules.

In our evaluation, we attempted several traffic profiles with different distributions. Note that traffic profiles are jointly determined by the arriving packets and the ACL written by individual administrators. Our approach is general and can operate using any plausible traffic distributions since our optimization framework does not assume any. In this paper, we choose to present our results based on traffic profiles consistent with the Zipf distribution, which is the only measured distribution we are aware of. The weights of the blocks were chosen randomly in conformance with the Zipf distribution with a parameter of 1.2. This is based on Cohen and Lund's observation of a strong Zipf-like pattern where few rules in each firewall are responsible for resolving most of the traffic [8].

The results are summarized in Table IV. We present the cost of the ACLS before and after each optimization techniques and they running time of each algorithm on every ACL. The cost affects directly the performance of a firewall and is the focus of this paper. The running time of optimization algorithms does not impact the firewall performance, but it is included to reflect the Royce's requirement of the optimization system.

For ACL A, we were not able to run **Partition**, as it exhausted the available memory. Notice that **Rule** performed the best in terms of cost, though with a slightly worse running time than **Split**. **Split** produced an ordering with cost in between that of **Greedy** and **Rule**.

For ACL B (*i.e.*, ACL A's core), **Partition** obtains an ordering with cost smaller than all the others, though also with a significantly longer running time. **Split**'s cost is unusually high, more so than **Greedy** and only slightly better than the original cost. **Split**'s behavior seems to have the most variance among the four algorithms.

For ACL C, **Partition** and **Split** lead in cost attribution, while also performing well in running time. The difference in cost is significant when considering the small cost of the ACL itself and **Greedy**'s cost as a base value. We notice that for this ACL, there is a very high ratio of blocks to rules, and we postulate that a more involved ACL with an intricate partition leaves more room for optimization by partition-based algorithms. Surprisingly, **Split** takes more time than **Partition** to process this ACL. We attribute this to high startup cost for each sub-problem (there are four sub-problems). Although the total time reported by CPLEX for solving the sub-problems was less than one second, each sub-problem invoked a new CPLEX instance, incurring the cost of program loading and initialization, and ultimately adding an additional 30 seconds to the running time. The other integer program-based solutions invoked only one CPLEX instance. This factor also had a noticeable effect on the times for ACL C and ACL F.

We notice that for ACL D, a partition based algorithm does not yield any cost benefit in comparison with a rule based algorithm. Looking at the count of blocks, we see that it is relatively small in comparison with the number of rules. It is very likely that for this partition, a rule-based algorithm is suitable for optimization. We see that **Partition** required significantly more processing time than the other algorithms. This is expected as this ACL is the largest of those it processed.

ACL E and ACL F seem to be particularly simple. Rule-based partitioning is sufficient for optimality, and greedy algorithms do nearly as good as optimal. None of the algorithms required an unusual amount of time.

We can draw a number of conclusions from our results. The first of these is that some ACLs are likely to require partition

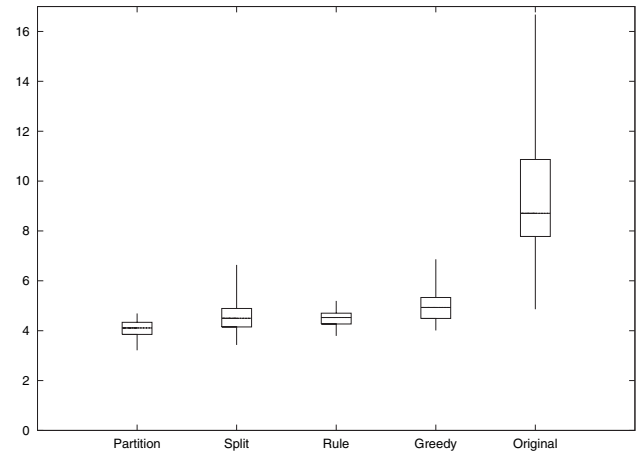


Fig. 6: Cost quartiles of the orderings of ACL C produced by the evaluated algorithms when run on 100 profiles.

aware algorithms for optimality. These ACLs are likely to be those that have a high number of blocks in comparison with the total number of rules. We see that locally aware algorithms, like **Greedy**, do yield an appreciable savings in cost, while incurring an almost negligible computational load. Rule-based algorithms yield considerable cost advantages beyond a local algorithm, and seem to scale well enough for real scenarios. We believe this algorithm may be of value for industry practitioners, despite the fact that it is not optimal. The partition-based algorithm is unlikely to scale to large ACLs without significant algorithm optimization and tuning, though it helps confirm how close other approaches are to the optimal bound. **Split** seems to have high variance in its results, though the cost advantage may be significant enough and the computational requirements small enough that it can be considered for deployment. We also notice that with our real ACLs, that partition sizes rarely doubles the number of rules. This suggests that partition-based algorithms, if optimized further, can be of practical use.

So far, the results in our evaluation are specific to one randomly chosen profile. We also attempted seeding 100 different profiles for ACL C to examine how sensitive these results are. Figure 6 summarizes the costs. The results seem consistent, and give us a higher level of confidence in the analysis of our previous results. Worth notice is that the global algorithms seemed insulated from the high variance of the original costs. More local algorithms, like **Split** and **Greedy**, are more variant, as the initial orderings may significantly change the final orderings. Despite this variance, **Split** had a slightly lower median value than **Rule**, giving some credence to its deployability.

VII. Conclusions

We have presented a general framework for evaluating optimization techniques for rule-based firewalls. This framework first divides the packet space into partitions where all the packets in any given partition match the same set of firewall rules. For each partition, the framework calculates the cost for the firewall to process all the packets in the partition based on traffic profile. Then, using these partitions, the framework generates the dependency of all the rules in the firewall. Finally, the framework formulates firewall optimization as an integer programming problem. Based on the framework, we developed a firewall optimization algorithm, and formally proved that our algorithm produces the *optimal* rule ordering. We applied our framework to empirically evaluate a few heuristic optimization algorithms: two existing ones and a new divide and conquer algorithm introduced in this paper. This framework allowed us to measure how close to the optimum these heuristic algorithms can produce rule orderings and the typical running time of these algorithm. Consequently, it enable us to make informed choice of optimization techniques.

References

[1] S. Acharya, J. Wang, Z. Ge, T. F. Zane, and A. Greenberg, "Traffic-aware firewall optimization strategies," in *Proc. International Conference on Communications*, 2006.

[2] F. Baboescu, S. Sin's, and G. Varghese, "Packet classification for core routers: is there an alternative to CAMs?" in *Proc. IEEE INFOCOM*, pp. 53–63, 2003.

[3] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, pp. 199–210, 2001.

[4] F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Trans. Networking*, vol. 13, no. 1, pp. 2–14, 2005.

[5] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992.

[6] Cisco Systems Inc., Cisco PIX 500 Series Security Appliances. http://www.cisco.com/en/US/products/hw/vpndevc/ps2030/index.html.

[7] Cisco Systems Inc., *User Guide for Access Control List Manager 1.6*, 2004.

[8] E. Cohen and C. Lund, "Packet classification in large ISPs: design and evaluation of decision tree classifiers," in *Proc. ACM SIGMETRICS*, pp. 73–84, 2005.

[9] P. Gupta and N. Mickey, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, pp. 147–160, 1999.

[10] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan./Feb. 2000.

[11] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, Mar. 2001.

[12] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *Proc. ACM Symposium on InformAtion, Computer and Communications Security*, pp. 332–342, 2006.

[13] ILOG Inc., ILOG CPLEX. http://www.ilog.com/products/cplex/.

[14] S. Joanne and V. Jacobson, "The BSD packet filter: a new architecture for user-level packet capture," in *USENIX Winter*, pp. 259–270, 1993.

[15] R. Russell, Linux 2.4 Packet Filtering Howto. http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html.

[16] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using Tully space search," in *Proc. ACM SIGCOMM*, pp. 135–146, 1999.

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. ACM SIGCOMM*, pp. 191–202, 1998.

[18] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[19] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: towards Programmable network MEasurement," in *Proc. ACM SIGCOMM*, 2007.

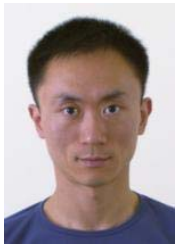
[20] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for FIREwall Modeling and ANalysis," in *Proc. IEEE Symposium on Security and Privacy*, pp. 199–213, 2006.

Ghassan Misherghi is a software engineer at Google working on Custom Search. He is focused on the performance and reliability concerns of running web services. Ghassan is also interested in a wide variety of software quality related topics, ranging from automated debugging to static partial verification.

Lihua Yuan is currently a Research Software Development Engineer at Microsoft. He received his Ph.D. in Electrical and Computer Engineering from the University of California, Davis in 2008. His research interests are in the area of computer networks and distributed systems, with a focus on network management, measurement, and security.

Zhendong Su is an Associate Professor in Computer Science at UC Davis, where he specializes in programming languages, software engineering, and computer security. He received both his M.S. and Ph.D. degrees in Computer Science from UC Berkeley, and both his B.S. degree in Computer Science and B.A. degree in Mathematics from UT Austin. He is the recipient of a Best Paper Award from the European Association for Programming Languages and Systems (1998), an ACM SIGMOD Distinguished Paper Award (2004), an NSF CAREER Award (2006), and the UC Davis College of Engineering Outstanding Junior Faculty Award (2007).

Chen-Nee Chuah is currently an Associate Professor in the Electrical and Computer Engineering Department at the University of California, Davis. She received her B.S. in Electrical Engineering from Rutgers University, and her M.S. and Ph.D. in Electrical Engineering and Computer Sciences from the University of California, Berkeley. Her research interests lie broadly in computer networks and wireless/mobile computing, with emphasis on Internet measurements, network anomaly detection, network management, multimedia, unaligned social networks, and vehicular ad hot networks. She received the NSF CAREER Award in 2003, and the Outstanding Junior Faculty Award from the UC Davis College of Engineering in 2004. In 2008, she was selected as a Chancellor's Fellow of UC Davis. She has served on the executive/technical program committee of several ACM and IEEE conferences and is currently an Associate Editor for IEEE/ACM TRANSACTIONS ON NETWORKING.



Hao Chen is an Assistant Professor in the Department of Computer Science at the University of California, Davis. He received his B.S. and M.S. in Biomedical Engineering from Southeast University in 1991 and 1994, respectively, and his Ph.D. in Computer Science from the University of California, Berkeley in 2004. His research interests are the area of computer security, including software security, network security, cellular network security, and Web security. He received the National Science Foundation CAREER Award in 2007. He has served on the technical program committees of several ACM and IEEE-sponsored conferences.