This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2008 proceedings.

Firewall Compressor: An Algorithm for Minimizing Firewall Policies

Alex X. Liu Eric Torng Chad R. Meiners
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, U.S.A.
{alexliu,torng,meinersc}@cse.msu.edu

***Abstract*—A firewall is a security guard placed between a private network and the outside Internet that monitors all incoming and outgoing packets. The function of a firewall is to examine every packet and decide whether to accept or discard it based upon the firewall's policy. This policy is specified as a sequence of (possibly conflicting) rules. When a packet comes to a firewall, the firewall searches for the first rule that the packet matches, and executes the decision of that rule.**

With the explosive growth of Internet-based applications and malicious attacks, the number of rules in firewalls have been increasing rapidly, which consequently degrades network performance and throughput. In this paper, we propose Firewall Compressor, a framework that can significantly reduce the number of rules in a firewall while keeping the semantics of the firewall unchanged.

We make three major contributions in this paper. First, we propose an optimal solution using dynamic programming techniques for compressing one-dimensional firewalls. Second, we present a systematic approach to compressing multi-dimensional firewalls. Last, we conducted extensive experiments to evaluate Firewall Compressor. In terms of effectiveness, Firewall Compressor achieves an average compression ratio of 52.3% on real-life rule sets. In terms of efficiency, Firewall Compressor runs in seconds even for a large firewall with thousands of rules. Moreover, the algorithms and techniques proposed in this paper are not limited to firewalls. Rather, they can be applied to other rule-based systems such as packet filters on Internet routers.

I. Introduction

Firewalls represent a critical component of network security. They are deployed at all points of entry between a private network and the outside internet to monitor all incoming and outgoing packets. A packet can be viewed as a tuple with a finite number of fields such as source/destination IP addresses, source/destination port numbers, and the protocol type. The function of a firewall is to examine every packet's field values and decide whether to accept or discard it based upon the firewall's policy. This policy is specified as a sequence of (possibly conflicting) rules. Each rule in a firewall has a predicate over some packet header fields and a decision to be performed upon the packets that match the predicate. A rule that examines d-dimensional fields can be viewed as a d-dimensional object. Real-life firewalls are typically 4-dimensional or 5-dimensional.

When a packet comes to a firewall, the firewall searches for the first (i.e., highest priority) rule that the packet matches, and executes the decision of that rule. Two firewalls are equivalent if and only if they have the same decision for every possible packet. Table I shows an example firewall of four rules. The format of these rules is based upon the format used in Access Control Lists on Cisco routers.

In this paper, we consider the Firewall Compression Problem: *given a firewall f, generate another firewall f' that is semantically equivalent to f but has the minimum possible number of rules.* We call this process "*firewall compression*".

Firewall compression is important for two major reasons. First, some firewall products have hard constraints on the number of rules that they support. For example, NetScreen-100 only allows firewalls with at most 733 rules. If your firewall has more than 733 rules, you must purchase a more expensive firewall product. Firewall compression may allow users to convert a large rule set into an equivalent small rule set bypassing the need for a more expensive firewall product. Second, many state-of-the-art systems still employ sequential search to identify the first firewall rule that matches a packet. Therefore, more rules means more per-packet processing time, and reducing the number of rules should improve firewall performance. This is especially important as the number of rules in firewalls increases dramatically due to more applications and services being deployed on the Internet and as more vulnerabilities, threats, and attacks are discovered.

Note that firewall compression does not interfere with firewall logging. When compressing firewall rules, decisions *accept*, *accept & log*, *discard* and *discard & log* are treated as four different decisions. Given a firewall f_1 and its compressed version f_2, a packet p is logged by f_1 if and only if p is logged by f_2. In some cases, firewall managers may monitor how many times each original rule in f_1 is the first rule to match a packet. Given a compressed firewall, this detailed monitoring can still be performed "offline" in software or hardware so as not to slow down packet processing speed. Full details will be given in the extended version of this paper.

A. A Motivating Example

We next give an intuitive example that shows the possibilities of generating an equivalent firewall with fewer rules. Our input firewall with 5 rules is depicted in Figure 1(A). For simplicity, we assume this firewall only examines one packet field F, and the domain of this field is $[1, 100]$. The geometric representation of these five rules is given in Figure 1(a). Geometrically, the predicate of a rule in a one-dimensional firewall can be represented as a segment, and the decision of

978-1-4244-2026-1/08/$25.00 ©2008 IEEE

Rule #	Source IP	Destination IP	Source Port	Destination Port	Protocol	Action
1	192.168.*.*	1.2.3.*	*	[4000, 5000]	TCP	discard
2	192.168.*.*	1.2.3.*	*	[0, 3999]	TCP	accept
3	192.168.*.*	1.2.3.*	*	[5001, 65535]	TCP	accept
4	*	*	*	*	*	discard

TABLE I
AN EXAMPLE FIREWALL

$F \in [41, 60] \rightarrow d_1$
$F \in [21, 55] \rightarrow d_2$
$F \in [45, 80] \rightarrow d_2$
$F \in [1, 65] \rightarrow d_3$
$F \in [75, 100] \rightarrow d_3$

$F \in [41, 60] \rightarrow d_1$
$F \in [21, 40] \rightarrow d_2$
$F \in [61, 80] \rightarrow d_2$
$F \in [1, 20] \rightarrow d_3$
$F \in [81, 100] \rightarrow d_3$

$F \in [41, 60] \rightarrow d_1$
$F \in [21, 80] \rightarrow d_2$
$F \in [1, 100] \rightarrow d_3$

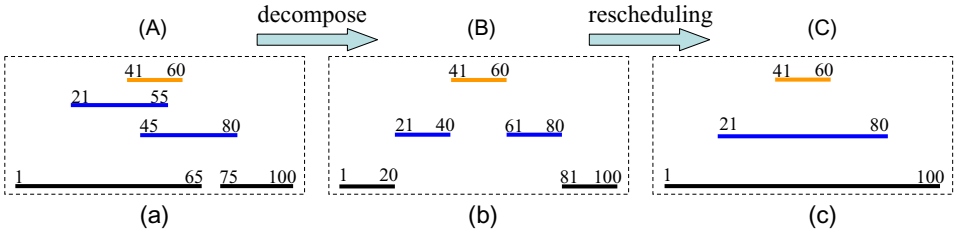

Fig. 1. minimizing firewall rules

the rule can be represented by the color of the segment. In Figure 1(a), we use five segments to represent the five rules in Figure 1(A), and we use three different colors to represent the three different decisions d_1, d_2, and d_3. Geometrically, a packet can be represented as a point, and the decision for the packet is the color of the first segment that contains the point.

To generate another sequence of rules that is equivalent to the firewall in Figure 1(A) but with the minimum number of rules, we first decompose the five rules into non-overlapping rules as shown in Figure 1(B). The geometric representation of these five non-overlapping rules is in Figure 1(b).

Based on the geometric representation of the five rules in Figure 1(B), we have the following observations. (1) If we schedule the interval $[41, 60]$ first, then we can schedule the two intervals $[21, 40]$ and $[61, 80]$ together using one interval $[21, 80]$ based on the first-match semantics. (2) Furthermore, we can use the intervals that have been scheduled, i.e., $[41, 60]$ and $[21, 80]$, to fill the gap between the two intervals $[1, 20]$ and $[81, 100]$. The three firewalls in Figure 1(A), 1(B) and 1(C) are equivalent, but with different numbers of rules.

B. Key Contributions

In this paper, we make the following three key contributions.

1) We propose an optimal algorithm for the one-dimensional firewall compression problem. This algorithm uses dynamic programming techniques.
2) We present a systematic approach to the multi-dimensional firewall compression problem. This algorithm achieves local optimality one dimension at a time using our optimal one-dimensional algorithm.
3) We conducted extensive experiments on both real-life and synthetic rule sets. The results show that our firewall compression algorithm achieves an average compression ratio of 52.3% on real-life rule sets.

Note that our firewall compression algorithm is designed to run off-line. Firewall operators do not need to read or manage the compressed firewall. That is, firewall operators can continue to design and maintain an intuitive and understandable firewall f while using our algorithms to generate and deploy a minimal, semantically equivalent firewall f'. Furthermore, the theory and algorithms presented in this paper are not limited to firewalls per se. Rather, they can be applied to the Access Control Lists (ACLs) on routers as well. Most routers on the Internet have ACLs in place for quality of service (QoS) filtering, traffic accounting, load balancing, etc.

The rest of the paper proceeds as follows. We first formally define our problem and notation in II. In Section III, we review related work. In Section IV, we present an optimal solution using dynamic programming techniques to a generalized one-dimensional firewall compression problem. In Section V, we give a solution to the multi-dimensional firewall compression problem. Experimental results are shown in Section VI. Finally, we give concluding remarks in Section VII.

II. FORMAL DEFINITIONS

We now formally define the concepts of fields, packets, firewalls, and the Firewall Compression Problem. A *field* F_i is a variable whose domain, denoted $D(F_i)$, is a finite interval of nonnegative integers. For example, the domain of the source address in an IP packet is $[0, 2^{32} - 1]$. A *packet* over the d fields $F_1, \cdots, F_d$ is a d-tuple $(p_1, \cdots, p_d)$ where each p_i $(1 \leq i \leq d)$ is an element of $D(F_i)$. We use Σ to denote the set of all packets over fields $F_1, \cdots, F_d$. It follows that Σ is a finite set and $|\Sigma| = |D(F_1)| \times \cdots \times |D(F_d)|$, where $|\Sigma|$ denotes the number of elements in set Σ and $|D(F_i)|$ denotes the number of elements in set $D(F_i)$ for each i.

A firewall rule has the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. A $\langle predicate \rangle$ defines a set of packets over the fields F_1 through F_d specified as $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ where each S_i is one nonempty interval that is a subset of $D(F_i)$. A packet $(p_1, \cdots, p_d)$ *matches* a predicate $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ and the corresponding rule if and only if the condition $p_1 \in$

$S_1 \wedge \cdots \wedge p_d \in S_d$ holds. We use α to denote the set of possible values that $\langle decision \rangle$ can be. Typical elements of α include accept, discard, accept with logging, and discard with logging.

Some existing firewall products, such as Linux's ipchains [1], represent source and destination IP addresses as prefixes in their rules. An example of a prefix is $192.168.0.0/16$ or $192.168.*.*$, both of which represent the set of IP addresses in the range from $192.168.0.0$ to $192.168.255.255$. Essentially, each prefix represents one integer interval (as we can treat an IP address as a 32-bit integer). In this paper, we uniformly represent firewall rules using intervals.

A *firewall* f over the d fields $F_1, \cdots, F_d$ is a sequence of firewall rules. The size of f, denoted $|f|$, is the number of rules in f. A sequence of rules $\langle r_1, \cdots, r_n \rangle$ is *complete* if and only if for any packet p, there is at least one rule in the sequence that p matches. A sequence of rules needs to be complete for it to serve as a firewall. To ensure that a firewall is complete, the predicate of the last rule in a firewall is usually specified as $F_1 \in D(F_1) \wedge \cdots F_d \in \wedge D(F_d)$, which every packet matches. Figure 2 shows an example of a firewall over the two fields F_1, F_2 where $D(F_1) = D(F_2) = [1, 100]$.

$$
\begin{array}{lllll}
r_1: & F_1 \in [20, & 40] & \wedge\ F_2 \in [30, & 50] \ \rightarrow accept \\
r_2: & F_1 \in [30, & 60] & \wedge\ F_2 \in [40, & 80] \ \rightarrow discard \\
r_3: & F_1 \in [1, & 100] & \wedge\ F_2 \in [1, & 100] \ \rightarrow accept
\end{array}
$$

Fig. 2. A firewall example

Two rules in a firewall may overlap; that is, a single packet may match both rules. Furthermore, two rules in a firewall may conflict; that is, the two rules not only overlap but also have different decisions. To resolve such conflicts, firewalls typically employ a first-match resolution strategy where the decision for a packet p is the decision of the first (i.e., highest priority) rule that p matches in f. The decision that firewall f makes for packet p is denoted $f(p)$.

We can think of a firewall f as defining a many-to-one mapping function from Σ to α. Two firewalls f_1 and f_2 are *equivalent*, denoted $f_1 \equiv f_2$, if and only if they define the same mapping function from Σ to α; that is, for any packet $p \in \Sigma$, we have $f_1(p) = f_2(p)$. For any firewall f, we use $\{f\}$ to denote the set of firewalls that are semantically equivalent to f.

We define the *Firewall Compression Problem* as follows. Given any firewall f_1, find a firewall that is semantically equivalent to f_1 with the minimum possible number of rules. More formally,

Definition 2.1 (Firewall Compression Problem): Given a firewall f_1, find a firewall $f_2 \in \{f_1\}$ such that $\forall f \in \{f_1\}$ the condition $|f_2| \leq |f|$ holds.

III. Related Work

There are two main types of firewalls: software-based and hardware-based. For software-based systems, packet classification is typically done using sequential search. Many studies have investigated faster software-based classification approaches [11]. However, these approaches rarely have been deployed due to their complexity and their potentially large space requirements. One such approach that is related to our work is the Geometric Efficient Matching (GEM) algorithm [9]. GEM searches for the first rule that matches a given packet using a similar data structure to our Firewall Decision Diagrams (FDD).

Hardware-based systems search all firewall entries in parallel for the first one that matches the given packet. However, these systems require that the firewall rules be in prefix format, which is more restrictive than interval format. This leads to the related but different problem of compressing firewalls that have prefix rules. Draves *et al.* proposed an optimal solution for one-dimensional prefix rules in the context of minimizing routing tables in [4]. Subsequently, in the same context of minimizing routing tables, Suri *et al.* proposed an optimal dynamic programming solution for one-dimensional prefix rules. They extended their dynamic program to optimally solve a special two-dimensional problem in which two rules either are non-overlapping or one contains the other geometrically [10]. Suri *et al.* noted that their dynamic program would not be optimal for rules with more than 2 dimensions. In [3], Dong *et al.* proposed four techniques of expanding rules, trimming rules, adding rules, and merging rules to minimize prefix rules. Meiners *et al.* proposed a systematic approach to minimizing prefix rules in [8]. Our method achieves more compression than these prefix approaches because interval rules allow more compression than prefix rules.

The problem of minimizing interval rules has been independently investigated by Applegate *et al.* in [2]. They prove that the two-dimensional problem with two decisions is NP-hard. They then give an optimal, polynomial time algorithm for the two-dimensional problem where there are only two decisions where all rules must be strip rules, which means that only one field can be a proper subset of its domain, and they use this to create $O(\min(n^{1/3}), OPT^{1/2})$-approximation algorithms for the general two-dimensional problem where n is the number of rules in the input firewall and OPT is the optimal firewall size. It is not obvious how to extend their ideas to more dimensions. Applegate *et al.* also cited a TopCoder programming contest named StripePainter that formulated and solved the one-dimensional problem and state the problem can be solved via dynamic programming with running time $O(Kn^3)$ where K is the number of distinct decisions. However, the StripePainter problem is a special case of our weighted one-dimensional firewall compression problem. Furthermore, our solution has a superior running time of $O(k^2 n)$ where k is the maximum number of rules that have a common decision.

IV. One-dimensional Firewall Compression

We first consider the *weighted one-dimensional firewall compression problem*, the solution of which will be used in the next section as a building block for the multi-dimensional firewall compression problem. We use dynamic programming to develop an optimal solution for this problem. Due to space limitations, we highlight the key ideas and omit the proofs of the lemmas and theorems.

A. Firewall Decomposition and Serialization

In a non-overlapping one-dimensional firewall, for any two rules, say $F \in [a,b] \rightarrow d_x$ and $F \in [c,d] \rightarrow d_y$, if they have the same decision (i.e., $d_x = d_y$) and the two intervals $[a,b]$ and $[c,d]$ are contiguous (i.e., $b+1 = c$ or $d+1 = a$), then the two rules can be merged into one rule (i.e., $F \in [a,d] \rightarrow d_x$ if $b+1 = c$, and $F \in [c,b] \rightarrow d_x$ if $d+1 = a$). A non-overlapping one-dimensional firewall is called *canonical* if and only if no two rules in the firewall can be merged into one rule. For example, Figure 1(B) shows a canonical firewall that is equivalent to the firewall in Figure 1(A).

Given a (possibly overlapping) one-dimensional firewall f, we first convert it to an equivalent canonical firewall f'. It is easy to prove that $|f'| \leq 2 \times |f| - 1$.

We then *serialize* the canonical firewall f' using the following two steps: (1) sort all the intervals in an increasing order, (2) replace the i-th interval with the integer i for every i. The resulting firewall f'' is called a *serialized firewall*. For any two non-overlapping intervals $[a,b]$ and $[c,d]$, if $b < c$, then we say the interval $[a,b]$ is *less* than the interval $[c,d]$. This serialization procedure creates a one-to-one mapping δ from the intervals in a canonical firewall to those in its serialized version while keeping the relations between intervals unchanged. In other words, two intervals S and S' are contiguous if and only if $\delta(S)$ and $\delta(S')$ are contiguous.

Next, we discuss how to compress the number of rules in the serialized firewall f''. Given the one-to-one mapping between f'' and f', an optimal solution for f'' can be directly mapped to an optimal solution for f'. We formulate the one-dimension firewall compression problem as the following firewall scheduling problem.

B. The Firewall Scheduling Problem

In the firewall scheduling problem, the input consists of a universe of tasks to be executed where each task has a color and a cost. More formally:

- Let $U = \{1, 2, \ldots, n\}$ be the universe of tasks to be executed. Each task i in U has a color. For any i ($1 \leq i \leq n-1$), task i and $i+1$ have different colors.
- Let $C = \{1, 2, \cdots, z\}$ be the set of z different colors that the n tasks in U exhibit, and for $1 \leq i \leq z$, let $|i|$ denote the number of tasks with color i.
- Let $X = \{x_1, \ldots x_z\}$ be the cost vector where it costs x_i to execute any task that has color i for $1 \leq i \leq z$.

Then an input instance to the firewall scheduling problem is $I = (U, C, X)$. We use $c(i)$ to denote the color of task i. It follows that the number of tasks with color $c(i)$ is $|c(i)|$.

Intuitively, U represents a serialized firewall where each task in U represents a rule in the firewall and the color of the task represents the decision of the rule. In the one-dimensional firewall compression problem, the cost of every task is 1; that is, we assign the value 1 to every x_i ($1 \leq i \leq z$). We consider the general weighted one-dimensional firewall compression problem because its solution can be used as a routine in solving the multi-dimensional firewall compression problem.

For any firewall scheduling input instance $I = (U, C, X)$, a firewall schedule $S(I) = \langle r_1, \ldots, r_m \rangle$ is an ordered list of m intervals. An interval $r_i = [p_i, q_i]$ where $1 \leq p_i \leq q_i \leq n$ is the set of consecutive tasks from p_i to q_i.

In a firewall schedule, a task is *fired* (i.e. executed) in the first interval that it appears in. More formally, the set of tasks fired in interval r_i of schedule $S(I)$ is $f(r_i, S(I)) = r_i - \bigcup_{j=1}^{i-1} r_j$. We call $f(r_i, S(I))$ the *core* of interval r_i in $S(I)$.

A schedule $S(I)$ of m intervals is a legal schedule for I if and only if the following two conditions are satisfied.

1) For each interval $1 \leq i \leq m$, all the tasks fired in interval i have the same color.
2) All tasks in U are fired by some interval in S; that is, $\bigcup_{i=1}^{m} f(r_i, S(I)) = U$.

The cost of interval r_i in legal schedule $S(I)$, denoted $x(r_i, S(I))$, is the cost x_j where j is the color that all the tasks in f_i exhibit. If $f_i = \emptyset$, we set $x(r_i, S(I)) = 0$. To simplify notation, we will often use f_i to denote $f(r_i, S(I))$ and $x(r_i)$ to denote $x(r_i, S(I))$ when there is no ambiguity.

The cost of a schedule $S(I)$, denoted $C(S(I))$, is the sum of the cost of every interval in $S(I)$, that is, $C(S(I)) = \sum_{i=1}^{m} x(r_i, S(I))$. The goal is to find a legal schedule $S(I)$ that minimizes $C(S(I))$.

C. An Optimal Solution

For any input instance I, we give an optimal solution using dynamic programming techniques. We start by making several basic observations to simplify the problem. The first is to define the notion of a canonical schedule.

Definition 4.1 (Canonical Schedule): For any input instance $I = (U, C, X)$, a legal schedule $S(I) = \{r_1, \ldots, r_m\}$ is a canonical schedule if for each interval $r_i = [p_i, q_i]$, $1 \leq i \leq m$, it holds that $p_i \in f_i$ and $q_i \in f_i$. □

We then observe that there exists an optimal canonical schedule for any input instance. This allows us to consider only canonical schedules for the remainder of this section.

Lemma 4.1: For any input instance I, for any legal schedule $S(I)$ with m intervals, there exists a canonical schedule $S'(I)$ with at most m intervals and with $C(S'(I)) = C(S(I))$. □

We next observe that in any canonical schedule S, swapping two adjacent intervals that do not overlap results in a canonical schedule with the same cost.

Lemma 4.2: For any input instance I, for any canonical schedule $S(I)$ containing two consecutive intervals $r_i = [p_i, q_i]$ and $r_{i+1} = [p_{i+1}, q_{i+1}]$ where $[p_i, q_i] \cap [p_{i+1}, q_{i+1}] = \emptyset$, the schedule $S'(I)$ that is identical to schedule $S(I)$ except interval $r'_i = r_{i+1} = [p_{i+1}, q_{i+1}]$ and interval $r'_{i+1} = r_i = [p_i, q_i]$ is also a canonical schedule. Furthermore, $C(S'(I)) = C(S(I))$. □

For any input instance I, we say that a schedule $S(I)$ is 1-canonical if it is canonical and task 1 is fired in the last interval of $S(I)$. A key insight is that there exists a 1-canonical optimal schedule $Opt(I)$ for any input instance I.

Lemma 4.3: For any input instance I and any canonical schedule $S(I)$ with m intervals, we can create a 1-canonical schedule $S'(I)$ with m intervals such that $C(S'(I)) \leq C(S(I))$. □

Let k be the number of tasks with the same color as task 1. Given Lemma 4.3 and the definition of canonical schedules,

there are k possibilities for the final interval $r_m = (1, q_m)$ in an optimal 1-canonical schedule $S(I)$. The right endpoint q_m must be one of the k tasks that has the same color as task 1.

We next observe that in any canonical schedule $S(I)$, each interval imposes some structure on all the previous intervals in $S(I)$. Specifically, the last interval r_m of any canonical schedule $S(I)$ partitions all previous intervals to have both endpoints lie strictly between consecutive elements of f_m, to the left of all elements of f_m, or to the right of all elements of f_m.

Lemma 4.4: For any input instance I, any canonical schedule $S(I)$, any interval $r_i = [p_i, q_i] \in S(I)$, consider any task $t \in f_i$. For any $1 \leq j \leq i-1$, let $r_j = [p_j, q_j]$. It must be the case that either $t < p_j$ or $q_j < t$. □

Given input instance $I = (U, C, X)$ with $|U| = n$, we define the following notations for $1 \leq i \leq j \leq n$:

- $I(i, j)$ denotes an input instance with a universe of tasks $\{i, \ldots, j\}$ and a set of colors that are updated to reflect having only these tasks and a set of costs that are updated to reflect having only these tasks.
- $Opt(I(i, j))$ denotes an optimal 1-canonical schedule for $I(i, j)$.
- $C(i, j)$ denotes the cost of $Opt(I(i, j))$.

Lemma 4.5: Given any input instance $I = (U, C, X)$ with $|U| = n$ and an optimal 1-canonical schedule $Opt(I(1, n))$.

1) If task 1 is the only task fired in the last interval of $Opt(I(1, n))$, then the schedule $Opt(I(2, n))$ concatenated with the interval $[1, 1]$ is also an optimal canonical schedule for $I(1, n)$, and $C(1, n) = x_{c(1)} + C(2, n)$.
2) If task 1 is not the only task fired in the last rule, letting t' be the smallest task larger than 1 fired in the last interval of $Opt(I(1, n))$, then the schedule $Opt(I(2, t'-1))$ concatenated with the schedule $Opt(I(t', n))$ where the last interval of $Opt(I(t', n))$ is extended to include task 1 is also an optimal canonical schedule for $I(1, n)$, and $C(1, n) = C(2, t'-1) + C(t', n)$. □

Based on the above observations, we formulate our dynamic programming solution to the firewall scheduling problem. For $1 \leq j \leq z$, we use G_j to denote the set of all the tasks that have color j. Recall that we use $c(i)$ to denote the color of task i ($1 \leq i \leq n$). Therefore, for $1 \leq i \leq n$, $G_{c(i)}$ denotes the set of all the tasks that have the same color as task i.

Theorem 4.1: $C(i, j)$ can be computed by the following recurrence relation.
For $1 \leq i \leq n$, $C(i, i) = x_{c(i)}$.
For $1 \leq i < j \leq n$, $C(i, j) = \min(x_{c(i)} + C(i+1, j), \min_{x \in G_{c(i)} \wedge i+2 \leq x \leq j]}(C(i+1, x-1) + C(x, j)))$. □

D. Firewall Scheduling Algorithm

Figure 3 shows the pseudocode of the firewall scheduling algorithm based on Theorem 4.1. This algorithm uses two $n \times n$ arrays C and M. In array C, a nonzero entry $C[i, j]$ stores the cost of an optimal schedule $Opt(I(i, j))$. In array M, for a nonzero entry $M[i, j]$, if $M[i, j] = i$, it means that i is the only task fired in the last interval of $Opt(I(i, j))$; if $M[i, j] \neq i$, it means that the smallest numbered task (other than i) that is also fired in the last interval of $Opt(I(i, j))$ is $M[i, j]$.

Firewall Scheduling Algorithm
Input : (1) array $color[1..n]$ where $color[i]$ is the color of task i;
(2) array $cost[1..z]$ where $cost[j]$ is the cost of executing task j;
(3) array $group[1..z]$ where $group[h]$ is the set of all tasks with color h;
Output : (1) an optimal schedule of the n tasks;
(2) the cost of the optimal schedule;
Variables: C, M: array [1..n][1..n] of integer; /*initial values of C and M are zeros*/
Steps:
1. FSA-Cost$(1, n)$; /*compute optimal cost, store trace info in M*/
2. Print-FSA$(1, 1, n)$; /*print an optimal schedule using array M*/
3. print the optimal cost $C[1, n]$;
End

FSA-Cost(i, j)
if $C[i, j] = 0$ then{
1. $min \leftarrow cost[color[i]]$ + FSA-Cost$(i+1, j)$;
2. $M[i, j] \leftarrow i$;
3. for every element x in $group[color[i]]$ do
if $i + 2 \leq x \leq j$ then
if FSA-Cost$(i+1, x-1)$ +
FSA-Cost$(x, j) < min$ then{
$min \leftarrow$ FSA-Cost$(i+1, x-1)$ +
FSA-Cost(x, j);
$M[i, j] \leftarrow x$;
}
4. $C[i, j] \leftarrow min$;
}
return $C[i, j]$;

Print-FSA(t, i, j)
if $i = j$ then print interval $[t, i]$;
else{
if $M[i, j] = i$ then{
Print-FSA$(i+1, i+1, j)$;
print interval $[t, i]$;
}else{
Print-FSA$(i+1, i+1, M[i, j]-1)$;
Print-FSA$(t, M[i, j], j)$;
}
}

Fig. 3. Firewall Scheduling Algorithm

The function FSA-Cost(i, j) computes the cost for $Opt(I(i, j))$. At the same time, this function also stores the trace information in array M. The information stored in M by FSA-Cost is used by the function Print-FSA. The function Print-FSA(t, i, j) basically prints out the optimal schedule $Opt(I(i, j))$, but in the last interval of $Opt(I(i, j))$, the left point i is replaced by t.

The complexity of this algorithm is $O(k^2 n)$ where n is the total number of tasks and $k = \max_{i \in P} |i|$ is the maximum number of tasks in U that exhibit the same color. Note that $\lceil n/z \rceil \leq k \leq \lceil n/2 \rceil$. The $O(k^2 n)$ running time follows from two observations. First, we need to compute $C(i, j)$ for at most kn pairs of (i, j). For every task $i \geq 1$, we need to compute $C(i, n)$. In addition, for any task $i+1$ where $i \geq 1$, we only need to compute $C(i+1, j-1)$ where task j has the same color as task i and $j > i$, and there are at most $k-1$ such values of j. Second, we need to compare at most k values when computing $C(i, j)$.

V. Multi-dimensional Firewall Compression

In this section, we present Firewall Compressor, a framework for compressing multi-dimensional firewalls. Similar to the algorithm for minimizing multi-dimensional prefix rules in [8], we process one dimension at a time using the optimal one-dimensional solution. Given a firewall f_1, our Firewall Compressor algorithm consists of the following four steps:

1) Convert f_1 to a firewall decision diagram f_2.

2) Reduce f_2 to a reduced firewall decision diagram f_3.
3) Compute a firewall f_4 from f_3.
4) Remove redundant rules from f_4. The resulting firewall f_5 is the final output.

A. Step 1: Conversion to Firewall Decision Diagrams

To facilitate processing a firewall one dimension at a time, we first convert firewall f_1 to an equivalent firewall decision diagram f_2 [5] using the FDD construction algorithm in [6].

A *Firewall Decision Diagram* (FDD) with a decision set DS and over fields $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following five properties:

1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes.
2) Each node v has a label, denoted $F(v)$, such that
$$F(v) \in \begin{cases} \{F_1, \cdots, F_d\} & \text{if } v \text{ is a nonterminal node,} \\ DS & \text{if } v \text{ is a terminal node.} \end{cases}$$
3) Each edge $e{:}u \rightarrow v$ is labeled with a nonempty set of integers, denoted $I(e)$, where $I(e)$ is a subset of the domain of u's label (*i.e.*, $I(e) \subseteq D(F(u))$).
4) A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label.
5) The set of all outgoing edges of a node v, denoted $E(v)$, satisfies the following two conditions:
   a) *Consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$.
   b) *Completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$. □

Figure 4 shows an example of a firewall decision diagram over the two fields F_1, F_2 where $D(F_1) = D(F_2) = [1, 100]$. Note that in labelling the terminal nodes, we use letter "a" as a shorthand for "$accept$" and letter "d" as a shorthand for "$discard$".

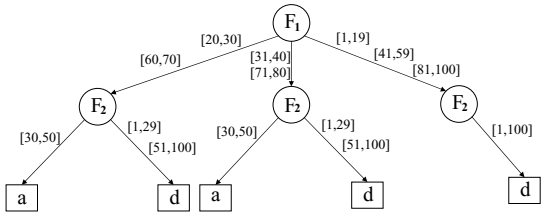

Fig. 4. A firewall decision diagram

B. Step 2: FDD Reduction

We next create a reduced FDD f_3. An FDD is *reduced* if and only if it satisfies the following two conditions: (1) no two nodes are isomorphic; (2) no two nodes have more than one edge between them. Two nodes v and v' in an FDD are *isomorphic* if and only if v and v' satisfy one of the following two conditions: (1) both v and v' are terminal nodes with identical labels; (2) both v and v' are nonterminal nodes and there is a one-to-one correspondence between the outgoing edges of v and the outgoing edges of v' such that every pair of corresponding edges have identical labels and they both point to the same node. Note that we relax the definition of reduced FDD in [5] by removing the requirement that no node has only one outgoing edge. This relation simplifies the implementation of our algorithm without losing any benefit of reducing FDDs.

A brute force deep comparison algorithm for FDD reduction was proposed in [5]. Here we present a more efficient FDD reduction algorithm that processes the nodes level by level from the terminal nodes to the root node using signatures to speed up comparisons.

1) At each level, first compute a signature for each node at that level. For a terminal node v, set v's signature to be its label. For a non-terminal node v, we assume we have the k children $v_1, v_2, \cdots, v_k$, in increasing order of signature ($Sig(v_i) < Sig(v_{i+1})$ for $1 \leq i \leq k-1$), and the edge between v and its child v_i is labeled with a sequence of non-overlapping intervals in increasing order E_i. Set signature of node v as follows:
$$Sig(v) = h(Sig(v_1), E_1, \cdots, Sig(v_k), E_k)$$
where h is a hash function.
2) After we have assigned signatures to all nodes at a given level, we check for redundancy as follows. For every pair of nodes v_i and v_j ($1 \leq i \neq j \leq k$) at this level, if $Sig(v_i) \neq Sig(v_j)$, then we can conclude that v_i and v_j are not isomorphic; otherwise, we explicitly determine if v_i and v_j are isomorphic. If v_i and v_j are isomorphic, we delete node v_j and its outgoing edges, and redirect all the edges that point to v_j to point to v_i. Further, we eliminate double edges between node v_i and its parents.

Figure 5 shows a reduced FDD, which is equivalent to the one in Figure 4.

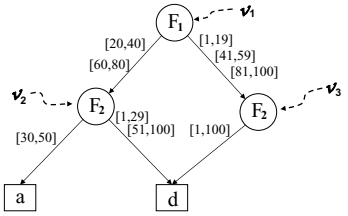

Fig. 5. A reduced firewall decision diagram

C. Step 3: Computing Firewall

Next, we present the core algorithm for compressing multi-dimensional firewalls. We start the discussion of our algorithm by examining the reduced FDD in Figure 5. We first look at the subgraph rooted at node v_2. This subgraph can be seen as representing a one-dimension firewall over field F_2. We can use the Firewall Scheduling Algorithm in Figure 3 to minimize the number of rules for this one-dimensional firewall. The algorithm is given the following 3 intervals as input:

$[30, 50]$ (with color *accept* and cost 1),
$[1, 29]$ (with color *discard* and cost 1),
$[51, 100]$ (with color *discard* and cost 1).

The algorithm will produce a minimum firewall of two rules: $F_2 \in [30, \ 50] \rightarrow accept$ and $F_2 \in [1, \ 100] \rightarrow discard$. Similarly, from the subgraph rooted at node v_3, we can get a minimum firewall of one rule $F_2 \in [1, \ 100] \rightarrow discard$.

Next, we look at the root v_1. Without changing the semantics of the FDD, we can split the outgoing edges of v_1 into 5 edges labeled with intervals $[20, 40]$, $[60, 80]$, $[1, 19]$, $[41, 59]$, and $[81, 100]$ respectively. As shown in Figure 6, we view the subgraph rooted at v_2 as a decision with a multiplication factor or cost of 2, and the subgraph rooted at v_3 as another decision with a cost of 1. Thus, the graph rooted at v_1 can be thought of as a "virtual" one-dimensional firewall over field F_1 with a generalized cost vector.

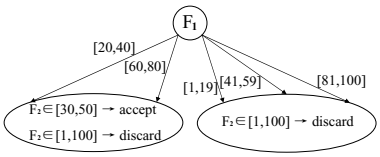

Fig. 6. "Virtual" one-dimensional firewall

Now we are ready to use the Firewall Scheduling Algorithm in Figure 3 to minimize the number of rules for this "virtual" one-dimensional firewall. The algorithm is given the following 5 intervals as input:

$[20, \ 40]$ (with color v_2 and cost 2),
$[60, \ 80]$ (with color v_2 and cost 2),
$[1, \ 19]$ (with color v_3 and cost 1),
$[41, \ 59]$ (with color v_3 and cost 1),
$[81, 100]$ (with color v_3 and cost 1),

Running the Firewall Scheduling Algorithm on the above input will produce the "virtual" one-dimensional firewall of three rules as shown in Figure 7:

$F_1 \in [41, \ 59] \rightarrow go \ to \ node \ v_3$
$F_1 \in [20, \ 80] \rightarrow go \ to \ node \ v_2$
$F_1 \in [1, \ 100] \rightarrow go \ to \ node \ v_3$

Fig. 7. A minimum firewall corresponding to v_1

Combining the "virtual" firewall in Figure 7 and the two firewalls that correspond to nodes v_2 and v_3, we get a firewall of 4 rules as shown in Figure 8.

$r_1: \ F_1 \in [41, \ 59] \ \wedge F_2 \in [1, \ 100] \ \rightarrow discard$
$r_2: \ F_1 \in [20, \ 80] \ \wedge F_2 \in [30, \ 50] \ \rightarrow accept$
$r_3: \ F_1 \in [20, \ 80] \ \wedge F_2 \in [1, \ 100] \ \rightarrow discard$
$r_4: \ F_1 \in [1, \ 100] \ \wedge F_2 \in [1, \ 100] \ \rightarrow discard$

Fig. 8. Firewall generated from the FDD in Figure 4

To summarize, in this step, we compute a firewall f_4 from a reduced FDD f_3 in the following bottom up fashion. For every terminal node of f_3, assign a cost of 1. For a non-terminal node v with z outgoing edges $\{e_1, \cdots, e_z\}$, formulate a firewall scheduling problem as follows. For every interval i in the label of edge e_j, $(1 \leq j \leq z)$, we set $c(i)$, the color of interval i, to be j, and the cost x_j is the cost of the node that edge e_j points to. Use the optimal firewall scheduling algorithm in Figure 3 to compute an optimal schedule of all the intervals that appear in the outgoing edges of node v, and set the cost of node v to be the cost of this schedule. After the root node has been processed, generate firewall f_4 using the interval schedules computed at each node. The cost of the root indicates the total number of rules in firewall f_4.

D. Step 4: Redundancy Removal

Next, we observe that rule r_3 in the firewall in Figure 8 is redundant. A rule in a firewall is redundant if and only if removing the rule from the firewall does not change the semantics of the firewall. Removing rule r_3, all the packets that used to be resolved by r_3, that is, all the packets that match r_3 but do not match r_1 and r_2, are now resolved by r_4, and r_4 has the same decision as r_3. Therefore, removing rule r_3 does not change the semantics of the firewall. Redundant rules in a firewall can be removed using the algorithms in [7]. Finally, after removing redundant rules, we get a firewall of 3 rules from the FDD in Figure 4. The geometric representation of this firewall (Figure 9) clearly shows that 3 is the minimum number of rules needed to represent this firewall.

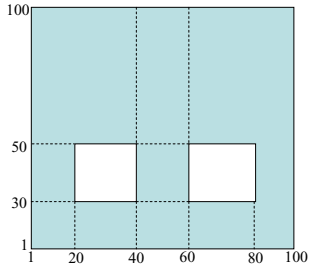

Fig. 9. Geometric representation

VI. Experimental Results

We now evaluate the effectiveness and efficiency of Firewall Compressor on both real-life and synthetic firewalls.

A. Methodology

Measurement Metrics. We first define the metrics that we used to measure the effectiveness of Firewall Compressor. In this paragraph, f denotes a firewall, S denotes a set of firewalls, and FC denotes Firewall Compressor. We then let $FC(f)$ denote the firewall produced by applying Firewall Compressor on f. We define the following two metrics for assessing the performance of FC on a set of firewalls S.

- The *average compression ratio* of FC over $S = \frac{\Sigma_{f \in S} \frac{|FC(f)|}{|f|}}{|S|}$.
- The *total compression ratio* of FC over $S = \frac{\Sigma_{f \in S} |FC(f)|}{\Sigma_{f \in S} |f|}$.

Real-life Firewalls. We next define a set RL of 17 real-life firewalls that we performed experiments on. We actually obtained 42 real-life firewalls including router ACLs from

distinct network service providers that range in size from dozens to hundreds of rules. Although this collection of firewalls is diverse, some firewalls from the same network service provider have similar structure and exhibited similar results under Firewall Compressor. To prevent this repetition from skewing the performance data, we divided the 42 firewalls into 17 structurally distinct groups, and we randomly chose one from each of the 17 groups to form the set RL.

Synthetic Firewalls. Firewall rules are considered confidential due to security concerns. Thus, it is difficult to get many real-life firewalls to experiment with. To address this issue and further evaluate the effectiveness of Firewall Compressor, we generated a set of synthetic firewalls, denoted SYN, in the following fashion. Every predicate of a rule in our synthetic firewalls has five fields: source IP address, destination IP address, source port number, destination port number, and protocol type. We first randomly generated a list of values for each field. For IP addresses, we generated a random class C address; for ports we generated a random interval; for protocols, we generated a random protocol number. Given these lists, we generated a list of predicates by taking the cross product of all these lists. We added a final default predicate to our list. Finally, we randomly assigned one of two decisions, accept or deny, to each predicate to make a complete rule.

B. Variable Ordering

The variable order that we used to convert a firewall into an equivalent FDD affects the effectiveness of Firewall Compressor. There are $5! = 120$ different permutations of the five packet fields (protocol type, source IP address, destination IP address, source port number, and destination port number). A question that naturally arises is: *which variable order achieves the best average compression ratio?* To answer this question, for each permutation, we computed the average and the total compression ratios that Firewall Compressor achieved over RL, which are shown shown in Figure 10 and 11 respectively. From these two figures, we can see that the effectiveness of Firewall Compressor does not significantly depend on variable order. For all variable orders, the average compression ratios achieved by Firewall Compressor fall in the range between 52.3% and 60.0%. Likewise, for all variable orders, the total compression ratios achieved by Firewall Compressor fall in the range between 61.4% and 72.6%.

Still, we are interested in the variable order that achieves the best average compression ratio. Six permutations achieve the best average compression ratio, which is 52.3%. Interestingly, these six permutations all achieve the same total compression ratio, which is 69.4%. The six permutations are all formulated as follows: *destination IP, source IP, any permutation of the other three fields*. To break the tie, we further evaluated the average compression ratios of the six permutations on synthetic rule sets. Finally, the permutation of (*destination IP, source IP, destination port, source port, protocol type*) was best. We use Firewall Compressor(12430) to denote the Firewall Compressor algorithm using this permutation.

The next natural question to ask is: *is permutation 12430 the best order for most firewalls?* The answer for RL is yes. For a given firewall f, we use Firewall Compressor(Best) to denote Firewall Compressor using the best of the 120 permutations for f. In Figure 12, for each firewall in RL, we show the compression ratios of Firewall Compressor(Best) and Firewall Compressor(12430). The results show that permutation 12430 achieves almost the best compression ratio for each rule set.

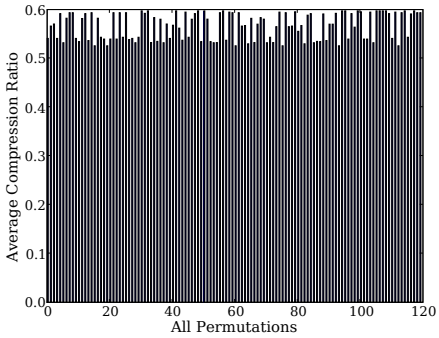

Fig. 10. The average compression ratio for each permutation

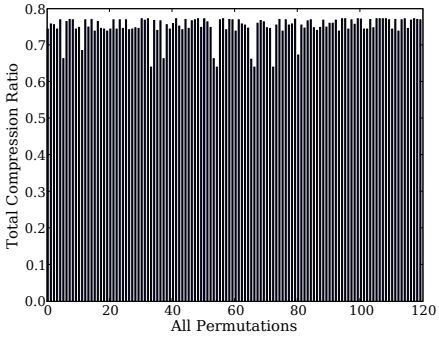

Fig. 11. The total compression ratio for each permutation

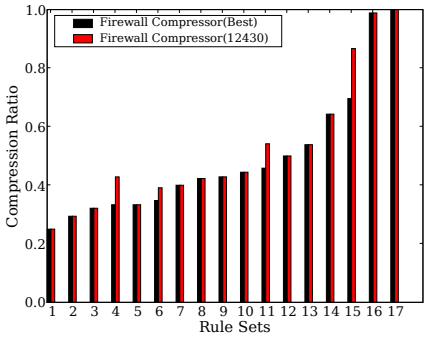

Fig. 12. Compression ratios of real-life firewall groups

C. Effectiveness

Compression Ratio of Real-life Rule Sets. The average and total compression ratios of Firewall Compressor(12430) over RL are 52.3% and 69.4%. Figure 13 shows the distribution of compression ratios achieved by Firewall Compressor(12430) over RL.

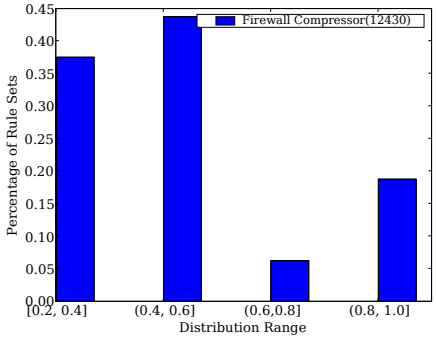

Fig. 13. Distribution of real-life firewalls by compression ratio

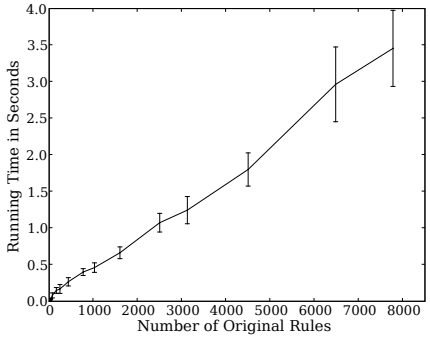

Fig. 15. Total running time of Firewall Compressor(12430) vs. number of original rules

Compression Ratio of Synthetic Rule Sets. The average and total compression ratios of Firewall Compressor(12430) over SYN are 32.0% and 7.4%respectively. Figure 14 shows the distribution of compression ratios achieved by Firewall Compressor(12430) over SYN.

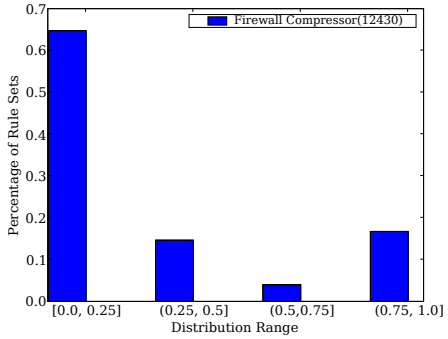

Fig. 14. Distribution of synthetic firewalls by compression ratio

D. Efficiency

We implemented our algorithm using C++. Our experiments were carried out on a desktop PC running Windows XP with 1G memory and Intel Pentium D Processor 820 of 2.8 GHz.

Efficiency on Real-life Rule Sets Table II shows the running time of Firewall Compressor(12430) for three representative rule sets.

Number of Original Rules	Running Time (seconds)
42	0.2
87	3.5
661	14.9

TABLE II
SAMPLE RUNNING TIME DATA FOR REAL-LIFE FIREWALLS

Efficiency on Synthetic Rule Sets Figure 15 shows the average running time of Firewall Compressor(12430).

VII. CONCLUSIONS

In this paper, we present Firewall Compressor, a framework for compressing firewall rules. Specifically, we make three major contributions. First, we give an optimal algorithm for compressing one-dimensional firewalls. Second, we present a systematic solution for compressing multi-dimensional firewalls. Third, we conducted extensive experiments on both real-life and synthetic rule sets. Our experimental results show that Firewall Compressor achieves an average compression ratio of 52.3%. Moreover, the algorithms proposed in this paper are not limited to firewalls. Rather, they can be applied to other rule-based systems such as packet filters on Internet routers.

Acknowledgement

The authors would like to thank Yun Zhou for his participation in the work and the anonymous reviewers for their constructive comments and valuable suggestions on improving the presentation of this paper. The work of Alex X. Liu is supported in part by the National Science Foundation under Grant No. CNS-0716407.

REFERENCES

[1] ipchains, http://www.tldp.org/howto/ipchains-howto.html.
[2] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.
[3] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proceedings of the ACM Sigmetrics*, pages 311–322, 2006.
[4] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proceedings of the IEEE INFOCOM*, pages 88–97, 1999.
[5] M. G. Gouda and A. X. Liu. Structured firewall design. *Computer Networks Journal (Elsevier)*, 51(4):1106–1120, March 2007.
[6] A. X. Liu and M. G. Gouda. Diverse firewall design. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-04)*, pages 595–604, June 2004.
[7] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. In *Proceedings of the 19th Annual IFIP Conference on Data and Applications Security, LNCS 3654*, pages 196–209, August 2005.
[8] C. R. Meiners, A. X. Liu, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proceedings of the 15th IEEE Conference on Network Protocols (ICNP)*, pages 266–275, October 2007.
[9] D. Rovniagin and A. Wool. The geometric efficient matching algorithm for firewalls. In *Proceedings of the 23rd IEEE Convention of Electrical & Electronics Engineers in Israel (IEEEI)*, pages 153–156, 2004.
[10] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.
[11] D. E. Taylor. Survey & taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.