A Difference Resolution Approach to Compressing Access Control Lists

James Daly Alex X. Liu Eric Torng
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824-1226
{dalyjame, alexliu, torng}@cse.msu.edu

***Abstract*—Access Control Lists (ACLs) are the core of many networking and security devices. As new threats and vulnerabilities emerge, ACLs on routers and firewalls are getting larger. Thereore, compressing ACLs is an important problem. In this paper, we propose a new approach, called Diplomat, to ACL compression. The key idea is to transform higher dimensional target patterns into lower dimensional patterns by dividing the original pattern into a series of hyperplanes and then resolving differences between two adjacent hyperplanes by adding rules that specify the differences. This approach is fundamentally different from prior ACL compression algorithms and is shown to be very effective. We implemented Diplomat and conducted side-by-side comparison with the prior Firewall Compressor algorithm on real life classifiers. The experimental results show that Diplomat outperforms Firewall Compressor most of the time, often by a considerable margin. In particular, on our largest ACLs, Diplomat has an average improvement ratio over Firewall Compressor of 30.6%.**

I. Introduction

A. Background and Motivation

Access Control Lists (ACLs) are the core of many networking and security devices, such as routers and firewalls, which perform services such as packet filtering, virtual private networks (VPNs), network address translation (NAT), quality of service (QoS), load balancing, traffic accounting and monitoring, differentiated services (Diffserv), etc. A packet can be viewed as a d-tuple over d fields with finite, discrete domains. For IP packets, d is typically 5 and the relevant fields are source IP address, destination IP address, source port number, destination port number, and protocol type, where the domains of these fields are $[0, 2^{32}-1]$, $[0, 2^{32}-1]$, $[0, 2^{16}-1]$, $[0, 2^{16}-1]$, and $[0, 2^{8}-1]$, respectively. An ACL is specified as a sequence (*i.e.*, ordered list) of predefined rules. Each rule is specified in the form $\langle predicate \rangle \rightarrow \langle decision \rangle$. The $\langle predicate \rangle$ over packet fields $F_1, F_2, \cdots, F_d$ is typically specified as $(F_1 \in S_1) \wedge (F_2 \in S_2) \wedge \cdots, \wedge (F_d \in S_d)$. If each S_i is specified as a range, we call the ACL a range-ACL. If each S_i is specified as a prefix, we call the ACL a prefix-ACL. The $\langle decision \rangle$ of a rule can be *accept*, or *discard*, or a combination of these decisions with other options such as a logging option. An ACL is essentially a function from the space of legal packets to a set of possible decisions. When a packet arrives at a router, the router extracts the relevant field values to form a search key and searches the ACL to find the first rule that the search key matches. The decision of this rule determines the appropriate action to take upon the packet. The rules in an ACL often conflict, which means that a packet may match multiple rules and these rules may have different decisions. The way that ACLs resolve conflicts is to follow the first match principle: for any packet, the decision for the packet is the decision of the first packet that the packet matches.

In this paper, we focus on the ACL Compression Problem for range-ACLs: *given a range-ACL L, find another equivalent range-ACL L' so that the number of rules in L', denoted $|L'|$, is as small as possible*. This problem has many motivations. First, as new threats and vulnerabilities emerge, ACLs on routers and firewalls are getting larger. For example, because the ACLs used for Quality of Services on routers are often automatically generated, they tend to be huge. Second, in the Open Flow architecture, which is gaining more popularity and real deployment, networking devices typically need to handle a large number of ACL-like rules; thus, compressing such ACL-like rules is very helpful for managing and optimizing such devices. A number of network system management tools, such as Yu *et al.*'s DIFANE work [15] and Sung *et al.*'s work [13], have used the prior ACL compression algorithm called Firewall Compressor to reduce system complexity. Third, some networking and security devices have strict limits on the number of rules that can be stored. For example, NetScreen-100 only allows ACLs with 733 rules. Fourth, fewer ACL rules often leads to higher system performance. As ACLs are used to examine every incoming and outgoing packet, such performance is critical for network throughput.

B. Summary and Limitations of Prior Art

Optimal polynomial time algorithms have been developed for 1-dimensional range-ACL compression (Top Coder Challenge in 2003, [2], [10]) and prefix-ACL compression [4], [14]. Applegate *et al.* proved that the 2-dimensional range-ACL compression problem is NP-hard [2]. The complexity of 2-dimensional prefix-ACL compression is still unknown. The only approximation algorithm with a non-trivial approximation bound is one given by Applegate *et al.* which has an approximation ratio of $O(\min(n^{1/3}, OPT^{1/2}))$ for range-ACL compression, where n is the number of rules in the input rule list and OPT is the number of rules in the optimal rule list [2].

Very few prior algorithms work for more than two dimensions. Liu *et al.*'s Firewall Compressor algorithm [10] handles this by converting the rule list into a canonical firewall decision diagram and applying an optimal weighted algorithm to each of the one-dimensional patterns formed between two layers of the diagram. In their algorithm, the "decision" used is actually the rule list that is produced by lower levels, which is replicated for each of the corresponding rules in the higher dimension. However, Firewall Compressor only considers whether or not the two lower rule lists are the same. If two lists are very similar, but not the same, both lists are replicated in their entirety.

C. Proposed Approach

We present a new approach, called Diplomat, to ACL compression. The key idea is to transform higher dimensional target patterns into lower dimensional patterns by dividing the original pattern into a series of hyperplanes. It then selects two adjacent planes and resolves their differences by adding rules to specify where the two planes differ. After resolution, the two planes are compatible and can be merged into a single plane. For example, in Figure 2, a single white rule in the middle row would allow it to be merged with the top row. Diplomat repeats this process until all of the planes have been merged together into a single plane. Diplomat then repeats the process on the reduced pattern. After recursive applications, the pattern is reduced to a 1-dimensional pattern for which optimal algorithms already exist. In terms of construction, the rules generated by each resolution step and the final compression step are joined together into one rule list with the rules from earlier steps appearing before the rules from later steps. Thus, any section specified by an earlier resolution cannot be undone by a later resolution. This is what allows Diplomat to consider the result of each resolution to be one homogeneous region. We illustrate this process for a two-dimensional range-ACL in Figure 1. We call this algorithm Diplomat because it compresses a rule list by repeatedly "resolving differences" between adjacent planes.

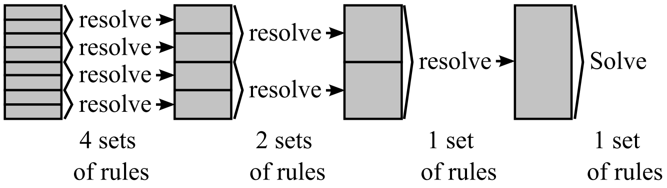
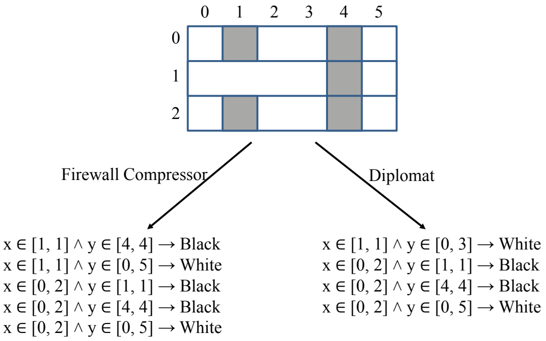


Fig. 1. Overview of Diplomat in two dimensions

Figure 2 shows that for an example input ACL, Firewall Compressor produces more rules than Diplomat.

Note that Diplomat is designed to be run offline so that network managers do not need to interact with the compressed ACL. Rather, the manager can interact with the rule list in a comfortable and understandable form. This can then be input into Diplomat, which will create the compressed rule list actually used by the network device.



x ∈ [1, 1] ∧ y ∈ [4, 4] → Black
x ∈ [1, 1] ∧ y ∈ [0, 5] → White
x ∈ [0, 2] ∧ y ∈ [1, 1] → Black
x ∈ [0, 2] ∧ y ∈ [4, 4] → Black
x ∈ [0, 2] ∧ y ∈ [0, 5] → White

x ∈ [1, 1] ∧ y ∈ [0, 3] → White
x ∈ [0, 2] ∧ y ∈ [1, 1] → Black
x ∈ [0, 2] ∧ y ∈ [4, 4] → Black
x ∈ [0, 2] ∧ y ∈ [0, 5] → White

Fig. 2. A two-dimensional pattern divided into three rows

D. Key Contributions

The key contribution of this paper is our new ACL compression approach that is totally different from all prior ACL compression algorithms. This novel approach is also very effective. We implemented Diplomat and conducted side-by-side comparison with the prior Firewall Compressor algorithm on real life classifiers. The experimental results show that Diplomat outperforms Firewall Compressor most of the time, often by a considerable margin. In particular, on our largest ACLs, Diplomat has an average improvement ratio over Firewall Compressor of 30.6%.

The rest of this paper is organized as follows. First, we review related work in Section II. Second, we present key definitions and some fundamental results in Section III. We then describe how Diplomat is applied to range-ACLs in Section IV. We cover the theoretical approximation bounds in Section V and the practical experimental results in Section VI. We conclude the paper by summarizing our results and discussing some open problems in Section VII.

II. Related Work

The 2-dimensional range-ACL compression problem was proven to be NP-hard in [2]. Whether the prefix-ACL problem is NP-hard is currently open. The 1-dimensional case of both problems is in P.

Applegate *et al.* [2] presented an algorithm for compressing 2-dimensional range-ACLs. For the special case of strip rules where each rectangle spans the entire canvas in one of the two dimensions and only 2 colors, their algorithm is optimal. They then describe a way to divide the ACL into regions which can each be solved with their strip rule solver. When applying their solution to the general 2-dimensional range-ACL problem, they gave a $O(\min(n^{1/3}, OPT^{1/2}))$-approximation algorithm. They also adapt their methods to prefix-ACL compression which adds a factor of w^2 to their approximation bounds where w is the number of bits in the prefix word size. This is the only approximation result we are aware of for either range or prefix-ACL compression for 2 or more dimensions. We adapt their methods in Section V to demonstrate that some versions of Diplomat can achieve the same bounds. Finally, they observe that the 1-dimensional range-ACL version was given as StripePainter in the 2003 Google TopCoder challenge and that a $O(Kn^3)$ running time can be achieved, where K is the number of colors [1].

For the 1-dimensional prefix-ACL compression problem, Draves *et al.* gave an optimal algorithm based on tries [4], and Suri *et al.* gave an optimal algorithm based on dynamic programming [14]. Suri *et al.* also gave an optimal dynamic programming solution for the special case where two prefix rectangles in the ACL can intersect only if one is fully contained within the other.

Liu *et al.* presented their own optimal 1-dimensional range-ACL compression solution based on dynamic programming methods [10]. They then applied this algorithm to each of the layers of a firewall decision diagram [6] to create a solution for any number of dimensions. In their later paper [9], they apply a similar idea using the optimal 1-dimensional prefix-ACL solver from [14] to expand their solution to operate on both range and prefix-ACLs. In [11], Liu *et al.* define ACL compressor which considers ACLs where some fields are ranges and some fields are prefixes. In this paper, we use Firewall Compressor to optimally compress the 1-dimensional range-ACLs which Diplomat generates as subproblems.

Finally, the Rectilinear Polygon Cover (RPC) Problem, which has been well explored in prior work [3], [5], [12], can be thought of as a special case of the range-ACL compression problem where the color of the last rule must be white and the color of all prior rules must be black. However, RPC is very different than range-ACL because the ability to use rectangles with different colors leads to much shorter rule lists. Thus, we cannot leverage any existing RPC algorithms to help with range-ACL compression.

III. Definitions and Notations

Let $\mathcal{K}$ be a finite set of decisions, such as {yes, no}, {accept, deny}, or {0, 1, 2, 3, 4}. A *color* $c \in \mathcal{K}$ is any one of those decisions.

We define a *canvas* $\mathbb{C}$ as the Cartesian product of d finite discrete dimensions, $[0, n_1 - 1] \times \cdots \times [0, n_d - 1]$. A *box* $B \subseteq \mathbb{C}$ is any region represented by a Cartesian product of intervals. A *pattern* or *coloring* $P : \mathbb{C} \to \mathcal{K}$ assigns each point in $\mathbb{C}$ a color in $\mathcal{K}$. An incomplete pattern is a pattern that does not define a color for some points in the canvas. Given two incomplete patterns P_1 and P_2 that do not both assign colors to any point in the canvas, $P_1 \cup P_2$ is the pattern formed by joining the two patterns P_1 and P_2.

A *rule* $r : B \to \mathcal{K}$ assigns exactly one color $c \in \mathcal{K}$ to each point in box B. A rule list $L = \{r_1, \ldots, r_n\}$ is an ordered list of rules and $|L| = n$ is the number of rules in L. A rule list L assigns colors to points as follows: for any point $p \in \mathbb{C}$, $L(p)$ is the value of $r_i(p)$ for the minimum i such that $r_i(p)$ is defined. A rule list is *complete* if $L(p)$ is defined for all points p in the canvas and is *incomplete* otherwise. We use ℓ to denote an incomplete rule list. We use the terms rule list and classifier interchangeably to refer to both a rule list L and the pattern defined by L. For a complete rule list L, we call the color of r_n the background color of L, and we can assume without loss of generality that $domain(r_n) = \mathbb{C}$ since r_n applies to all of $\mathbb{C}$ not covered by prior rules. For two rule lists L_1 and L_2, $L_1 \cdot L_2$ denotes the rule list formed by appending L_2 to the end of L_1. We will also write $\ell \cdot P$ where P is a pattern; in this case, the pattern implied by incomplete rule list ℓ overrides P for points where both are defined.

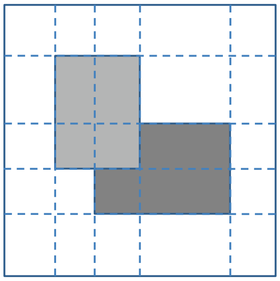


Fig. 3. An Effective Grid Denoted by the Dashed Lines

Two rule lists are equivalent, $L_1 \equiv L_2$, if $L_1(p) = L_2(p) \ \forall p \in \mathbb{C}$. We denote the set of all rule lists equivalent to P as $\langle P \rangle$. L is an optimal rule list for a pattern P if $L \in \langle P \rangle$ and $|L| \leq |L'| \forall L' \in \langle P \rangle$. Our objective is to find an optimal rule list for an input pattern P. This was shown in [2] to be NP-hard for patterns of two or more dimensions, so we try to find one as close to optimal as possible.

Given a pattern P, box B is P-monochromatic if $P(p_1) = P(p_2) \ \forall p_1, p_2 \in B$. We can break $\mathbb{C}$ into monochromatic boxes using Applegate *et al.*'s *effective grid* concept. Quoting [2], "For any rectilinear pattern P, call a horizontal or vertical line segment that separates and bounds two differently-colored regions a *boundary line*. Extend each boundary line in both directions so that it crosses the full canvas. We call the resulting grid the "effective grid" for the pattern, and note that all its grid cells will be monochromatic." Without loss of generality, we assume the input rule list L is converted to the effective grid space, so $\mathbb{C}$ is the effective grid and n_i ($1 \leq i \leq d$) is one more than the number of boundary planes in dimension i. We also order the dimensions so $n_1 \leq \cdots \leq n_d$. Applegate *et al.* also showed that $n_i \leq 2n$ [2]. An example of an effective grid can be seen in Figure 3.

IV. Methodology

We now formally describe the Diplomat classifier compression algorithm. We first describe how we reduce our d-dimensional pattern into a collection of $(d-1)$-dimensional patterns. Given a d-dimensional pattern P with a d-dimensional canvas $\mathbb{C} = [0, n_1-1] \times \cdots \times [0, n_d-1]$, define the $(d-1)$-dimensional canvas $\mathbb{C}' = [0, n_2-1] \times \cdots \times [0, n_d-1]$. We divide $\mathbb{C}$ into n_1 copies of canvas $\mathbb{C}'$ with $\mathbb{C}'_i$ being the $(d-1)$-dimensional plane with $f_1 = i$ for $0 \leq i \leq n_1 - 1$. We then create n_1 patterns P_i for $0 \leq i \leq n_1 - 1$ where P_i is the intersection of plane $\mathbb{C}'_i$ with the original pattern P. We define pattern P_i on the $(d-1)$-dimensional canvas $\mathbb{C}'$ as follows: for any point $p \in \mathbb{C}'$, $P_i(p) = P(i \times p)$.

We create the resulting rule list L for $\mathbb{C}$ from our sequence of n_1 patterns P_i ($0 \leq i \leq n_1 - 1$) by "merging" together adjacent patterns until we are left with a single pattern and then recursively solving that single pattern. More formally, let $P_{[i,j]}$ denote some merging of patterns P_i to P_j inclusive. For any point $p \in \mathbb{C}'$, $P_{[i,j]}(p) = P_k(p)$ for some $i \leq k \leq j$. For any $p \in \mathbb{C}'$, we define pattern $P^{\mathbb{C}}_{[i,j]}$ on $\mathbb{C}$ by setting $P^{\mathbb{C}}_{[i,j]}(k \times p) = P_{[i,j]}(p)$ if $i \leq k \leq j$; otherwise, $P^{\mathbb{C}}_{[i,j]}(k \times p)$ is undefined. Initially, $P_{[i,i]} = P_i \ \forall i \in [0, n_1 - 1]$. Initialize L to be the empty rule list. While we have at least two patterns remaining in our sequence of patterns, we select two adjacent patterns $P_{[i,j]}$ and $P_{[j+1,k]}$ to be merged. We merge these two patterns into a single pattern by providing extra information in a partial rule list ℓ defined on $\mathbb{C}$ which is appended to L. Intuitively, ℓ colors points where $P_{[i,j]}$ and $P_{[j+1,k]}$ differ; this allows the two patterns to then be merged. Let ℓ_q be the partial rule list used in the qth merger of patterns where $1 \leq q \leq n_1 - 1$. When we have a single pattern $P_{[0,n_1-1]}$ remaining, the rule list L is the concatenation of partial rule lists ℓ_q for $1 \leq q \leq n_1 - 1$. We then recursively solve the $(d-1)$-dimensional pattern $P_{[0,n_1-1]}$ which is then appended to L for our final solution. If $d = 2$ which means $d - 1 = 1$, we have reached our base case and use the optimal one-dimensional solver in [10] to generate a solution to $P_{[0,n_1-1]}$.

To merge two adjacent patterns $P_{[i,j]}$ and $P_{[j+1,k]}$, we need a *resolver* which we formally define as follows.

Definition 1. *A resolver of $P_{[i,j]}$ and $P_{[j+1,k]}$ returns a partial rule list ℓ defined on $\mathbb{C}$ and pattern $P_{[i,k]}$ defined on $\mathbb{C}'$ subject to the following constraints: $\forall p \in \mathbb{C}'$, $P_{[i,k]}(p)$ equals either $P_{[i,j]}(p)$ or $P_{[j+1,k]}(p)$ and $\ell \cdot P^{\mathbb{C}}_{[i,k]} \equiv P^{\mathbb{C}}_{[i,j]} \cup P^{\mathbb{C}}_{[j+1,k]}$.*

We observe that if $P_{[i,j]} \equiv P_{[j+1,k]}$ then $P_{[i,k]}$ will match both lists and ℓ is the empty list. While a resolver is formally the function that computes the partial rule list ℓ and the merged pattern $P_{[i,k]}$, we also often refer to partial rule list ℓ as a resolver and say that ℓ resolves $P_{[i,j]}$ and $P_{[j+1,k]}$.

Next, we review Firewall Compressor which is used as a sub-program by our resolvers. We then formally present several different resolvers to merge adjacent patterns and schedulers to determine the merging order for patterns.

A. Firewall Compressor

We review here the 1-dimensional version of Firewall Compressor from [10]. Given some input classifier L, compute the effective grid of L and label the cells from 0 to $n-1$. Now consider the rule, r_i that covers cell 0. Since this is the leftmost border of the the pattern, no rule can extend past its left border. Thus, r_i can always be considered to be the last rule since any rule that passes under r_i can be trimmed to the right terminous of r_i. However, many rules might be placed on top of r_i. They do not have to be contiguous; it may be better to have two sections be separated by a region where r_i is exposed. To this end, we try the various places, x where r_i can be exposed to create two smaller problems: one for the region $[1, x-1]$ that rests upon r_i and one for $[x, n-1]$. We solve both sets separately and then go back and extend r_i so that it covers cell 0 from under the first block. We observe that the only possible locations for x are cells within $[0, n-1]$ where $color(0) = color(x)$. We try each of those cells and select whichever one minimizes the total cost.

B. Resolvers

We first define two different types of resolvers that constrain the type of scheduler Diplomat can use.

Definition 2. *An in-plane resolver is a resolver where all of the rules in ℓ correspond to one of the two patterns being merged. Stated another way, all the rules in ℓ come from the canvas $\mathbb{C}'$ corresponding to only one of the two patterns being merged. A general resolver has no constraints placed upon the rule source.*

With an in-plane resolver, the resulting merged pattern will be identical to the pattern that is not the source of the rules in ℓ. With a general resolver, the merged pattern may be a new pattern that is different from either of the two input patterns. The general resolver has the advantage that ℓ can be smaller since we have more flexibility in resolving differences between the two patterns. The in-plane resolver has the advantage that because the resulting pattern will be one of the input patterns, the scheduler can use dynamic programming. We now describe our efforts to develop fast and effective resolvers. We begin by giving optimal general and in-plane resolvers for rows (one-dimensional planes) and then give heuristic higher dimensional resolvers.

1) Optimal one-dimensional resolvers: In this section, $\mathbb{C}'$ is a row. We use R_t and R_b to correspond to the two patterns $P_{[i,j]}$ and $P_{[j+1,k]}$ that are being merged. We use $R_t[i,j]$ and $R_b[i,j]$ to refer to the sub-pattern in row R_t and R_b, respectively, between columns i and j. Both resolvers we present use the optimal one-dimensional version of Firewall Compressor (FC) in [10] as a subroutine. For completeness, we review Firewall Compressor in the the appendix. Proofs of optimality for the resolvers can be found in Section V.

a) Optimal General Resolver: Here, we present an optimal way to merge R_t and R_b between columns 0 and k. Let ℓ_k denote the partial rule list returned by this optimal solution, let $C_k = |\ell_k|$, and let P_k denote the resulting pattern that is returned. If $R_t[k] = R_b[k]$, then $\ell_k = \ell_{k-1}$, $C_k = C_{k-1}$, and $P_k = P_{k-1} \cup R_t[k]$ as there are no differences in column k. Otherwise, there is a difference between R_t and R_b in position k. This means ℓ_k must include at least one rule that covers either $R_t[k]$ or $R_b[k]$. We prove in Section V-A that we can restrict our attention to resolvers that cover only R_t or R_b from column k back to some column s where $0 \leq s \leq k$. This is then combined with an optimal solution up to column $s-1$. We consider all possible choices of this split point s in the following description and set $C_{-1} = 0$, $\ell_{-1} = \emptyset$, and $P_{-1} = \emptyset$ for the case where $s = 0$. Then $C_k = \min_s(C_{s-1} + \min(|FC(R_t[s,k])|, |FC(R_b[s,k])|))$ and $S_k =$ the corresponding value of s. Now, using $\vec{S}$, we build the solution ℓ_k, P_k. If $R_t[k] = R_b[k]$ then $\ell_k = \ell_{k-1}$ and $P_k = P_{k-1} \cup R_t[k]$. Otherwise, we add the rules from the appropriate solution $FC(R_t[S_k,k])$ or $FC(R_b[S_k,k])$ to

ℓ_{S_k-1} to form ℓ_k, and we add the *other* pattern for columns S_k to k to P_{S_k-1} to form P_k.

b) Optimal In-Plane Resolver: This version assumes that all of the rules for ℓ have to come from R_t or R_b. Without loss of generality, assume the rules come from R_t. Then $P_{[t,b]} = R_b$. We use a dynamic programming technique similar to the general resolver to create our rule list ℓ. Since we have already determined that $P = R_b$ we do not need to worry about computing P. Furthermore, since we know no rules for ℓ can come from R_b, we only need to compute the partial solutions in R_t.

2) Higher Order Resolvers: If $d \geq 3$, then we need to merge two patterns P_1 and P_2 that are $(d-1)$-dimensional hyperplanes. This is more complicated than merging two rows. In particular, we have an optimal algorithm for compressing 1-dimensional patterns, but coloring 2-dimensional patterns is NP-hard as shown in [2]. This difference in complexity carries over to merging patterns.

We use two general methods for performing higher dimensional merges. First, we use the FDD comparison algorithm in [8] to find the differences between the two patterns. We can then create ℓ_{dif} by enumerating the differences in P_1 or P_2, whichever is smaller. Second, we find box, the minimum bounding box surrounding all of these differences. We then solve $P_1(box)$ and $P_2(box)$ using Diplomat. We call the smaller of these two solutions ℓ_{mbb}. We then let $\ell = \min(\ell_{dif}, \ell_{mbb})$. Both sets contain all of the differences and so ℓ is a valid solution.

In both cases, we focus on in-plane resolvers as this allows the use of a dynamic programming scheduler. However, we can easily define general resolvers for both methods.

C. Schedulers

To complete the Diplomat algorithm, we need a scheduler to determine an order for merging patterns. We present two such schedulers. The first uses a greedy scheme and works with any resolver while the second uses dynamic programming to get better overall results but requires in-plane resolvers. We can prove theoretical bounds for the greedy scheduler, but the dynamic programming scheduler outperforms the greedy scheduler in our experiments.

1) Greedy Scheduler: For the greedy scheduler, after every merge, if we have j remaining patterns, we relabel them from 0 to $j-1$ so the remaining patterns are P_0 through P_{j-1}. The first method is very simple. Pick $\min_{i=0}^{j-2} Resolve(P_i, P_{i+1})$ and do that merge. Repeat until only one pattern remains.

The basic version would require $O(n_1)$ calls to $Resolve$ for each of the $n_1 - 1$ merges. However, since only two computations change after each merge, we store the results and only perform the required new merges after each merge. Overall, we perform less than $2n_1$ calls to $Resolve$. For a d-dimensional classifier, we can consider all possible permutations of the d fields when performing the greedy scheduling.

Thinking more generally, there is no reason to perform merges in only one dimension at a time. Instead, we can greedily choose the best merge in any of the d dimensions at any point in time. In particular, for $d = 2$, we find the cheapest pair of rows or columns to merge. In section V, we show how to use this rotational greedy schedule to create an $O(\min(n^{1/3}, OPT^{1/2}))$-approximation for 2-dimensional patterns.

2) Dynamic Programming Scheduler: We also present a dynamic programming method that requires more upfront work and, as a result, outperforms the greedy scheduler in our experiments. For this scheduler, we retain the original numbering for each pattern even as merges are performed.

Using an in-plane resolver such as the one in Section IV-B1b, let $MergeCost[i,j] = |Resolve(P_i, P_j)| \forall i \neq j \in [0, n_1 - 1]$ where the returned pattern is P_i. This requires $O(n_1^2)$ calls to $Resolve$. However, because the merged pattern is always one of the original patterns, we do not need to do any more merges.

We then define the following subproblem. Given interval $[l, u]$ for $0 \leq l \leq u \leq n_1 - 1$, we find the minimum cost for merging patterns $P_l, \ldots P_u$ together while having P_r be the remaining pattern where $l \leq r \leq u$. We use standard dynamic programming techniques to compute the minimum cost solution to these $O(n_1^3)$ subproblems and the corresponding optimal schedule of merges.

V. Algorithmic Analysis

We now prove some results about our algorithms. We first prove that our one-dimensional resolvers are optimal. We then prove some approximation results for some two-dimensional implementations of Diplomat. Finally, we prove that dynamic schedule Diplomat should outperform the current state of the art algorithm, Firewall Compressor.

A. Optimal One-Dimensional Resolvers

In this section, we prove that our one-dimensional resolvers are optimal.

We first prove that there are optimal one-dimensional resolvers in which no rule spans both rows.

Theorem 1. *Let ℓ be a rule list that resolves R_t and R_b, Then there exists a rule list ℓ' that also resolves R_t and R_b such that $\ell' \leq \ell$ and that no rule $r \in \ell'$ spans both rows. In particular, there is an optimal resolver with this property.*

Proof: We prove this result by contradiction. Let ℓ be an optimal resolver with the minimum possible number of rules k that span both rows. If $k = 0$ then $\ell = \ell'$ and we are done. Otherwise, let r_i be the last rule in ℓ spanning both rows.

We first consider the case where there is no rule r_j with $j > i$ and $r_i \cap r_j \neq \emptyset$. In this case, we create a new rule list ℓ^* from ℓ by replacing r_i with $r_t = r_i \cap R_t$. Any point in R_t covered by r_i is also covered by $r_i \cap R_t$, so ℓ^* still resolves all of the columns. For any point $p \in R_b$, either some rule r_k with $k < i$ defines p for both ℓ and ℓ^* or no rule in ℓ^* does. Thus, there can be no color clash and ℓ^* is still a resolver. This means ℓ^* is an optimal resolver with only $k - 1$ rules that span both rows contradicting the definition of ℓ. Thus, this case is not possible.

The case where all r_j with $r_i \cap r_j \neq \emptyset$ for $j > i$ are in row R_t can be shown to be impossible using the same analysis. The reverse case where are all intersections occur in row R_b can be handled similarly; the only modification is that we replace r_i with $r_i \cap R_b$ to form ℓ^*.

We now consider the last case where some later rules that intersect r_i are in row R_t and others are in row R_b. We first trim all rules r_j with $j > i$ so that both its left and right end points are not covered by any prior rules. Since we merely removed the parts of r_j covered by other rules, the resulting rule list ℓ^* is equivalent to ℓ. This may reduce the problem to one of the above cases in which case we are done. If not, let S be the set of rules r_j in ℓ^* with $j > i$ that intersect r_i and let $S_t \subset S$ be the subset in row R_t and $S_b \subset S$ be the subset in row R_b. Assume without loss of generality that $r^* \in S_t$ is the first rule in S to appear scanning from left to right. Since r^* intersects r_i, its right end point must be to the right of r_i's end point given the trimming operation we performed earlier. We then apply a second trimming operation and update ℓ^* by replacing all rules $r_j \in S_b$ with the portion of r_j that is strictly to the right of r_i. This classifier still resolves R_t and R_b since r_t still covers any column that was formerly covered by r_b. We have thus reduced this case to the prior case where all rules after r_i that intersect r_i are from one row, and the proof is complete. ■

We next prove that there are optimal one-dimensional resolvers in which no column is covered by rules in both rows.

Theorem 2. *Let ℓ be a rule list that resolves R_t and R_b. Then there exists a rule list ℓ' that resolves R_t and R_b such that $|\ell'| \leq |\ell|$ and that for any $i \in [0, n_d - 1]$, either $\ell'(t, i)$ or $\ell'(b, i)$ is undefined.*

Proof: First, by Theorem 1, we can assume that ℓ has no rule that occupies both rows. Let ℓ be an optimal resolver with the minimum number of rules k in one row that overlaps the horizontal range of any rule in the other row. If $k = 0$, we are done. Otherwise, assume without loss of generality that $r^* \in R_t$ is the first such rule when scanned left to right. We create a modified rule list ℓ^* by trimming all rules in R_b that overlap the horizontal range of r^*, removing any that are entirely contained within the range of r^* (which cannot happen or else ℓ is not optimal). The rule list ℓ^* still resolves R_t and R_b since ℓ^* specifies values for all columns affected in row R_t. However, because the modified rule r^* does not overlap any rules in R_b, ℓ^* has a smaller k value contradicting the definition of ℓ, and the result follows. ■

We now prove the optimality of our general one-dimensional resolver.

Theorem 3. *The Optimal Resolver algorithm described in Section IV-B1a is an optimal one-dimensional resolver.*

Proof: By induction on k, the length of the rows. The base case $k = 0$ is trivial. Now assume this method produces an optimal resolver ℓ_j for $0 \leq j \leq k$. We now consider $k+1$. If $R_t[k] = R_b[k]$, then $\ell_{k+1} = \ell_k$ is clearly an optimal resolver as it resolves all differences between the two rows in the first $k + 1$ columns, and $|\ell_{k+1}| \leq |\ell_k|$. Thus, by our induction hypothesis, our one-dimensional resolver is an optimal one-dimensional resolver for this case.

If $R_t[k] \neq R_b[k]$, by Theorem 2 we restrict our attention to optimal resolvers ℓ_{k+1} in which no column is covered by rules in both rows. Column k must be covered by some rule since $R_t[k] \neq R_b[k]$. This means column k will be covered by a rule r_i in only R_b or R_t. Without loss of generality, let us assume r_i is only in R_t. By Theorem 2, it follows that for some $0 \leq q \leq k$ that ℓ_{k+1} must color $R_t[q, k]$ and ℓ_{k+1} does not color $R_b[q, k]$. Thus ℓ^* will be an optimal resolver ℓ_{q-1} combined with an optimal coloring of $R_t[q, k]$. By our induction hypothesis, the fact that Firewall Compressor will compute an optimal coloring of $R_t[q, k]$, and the fact we try all possible values of q, our general resolver algorithm is an optimal resolver that finds such a solution. ■

We also show that our one-dimensional in-plane resolver is an optimal one-dimensional in-plane resolver.

Theorem 4. *The Optimal In-Plane Resolver algorithm descrubed in Section IV-B1b is an optimal one-dimensional in-plane resolver.*

Proof: We again induct on k, the length of the rows. The base case $k = 0$ is trivial. Now assume this method produces an optimal in-plane resolver ℓ_j for $0 \leq j \leq k$. We now consider $k+1$. If $R_t[k] = R_b[k]$, then $\ell_{k+1} = \ell_k$ is clearly an optimal in-plane resolver as it resolves all differences between the two rows in the first $k + 1$ columns, and $|\ell_{k+1}| \leq |\ell_k|$. Thus, by our induction hypothesis, our in-plane resolver is an optimal in-plane resolver for this case.

If $R_t[k] \neq R_b[k]$, then ℓ_{k+1} must color $R_t[k]$. It follows that ℓ_{k+1} must color $R_t[q, k]$ for some $0 \leq q \leq k$. Thus ℓ_{k+1} will be an optimal in-plane resolver ℓ_{q-1} combined with an optimal coloring of $R_t[q, k]$. By our induction hypothesis, the fact that Firewall Compressor will compute an optimal coloring of $R_t[q, k]$, and the fact we try all possible values of q, our in-plane resolver algorithm is an optimal in-plane resolver that finds such a solution. ■

B. Two-Dimensional Approximation Results

In [2], the authors present their iterated strip-rule algorithm and prove that it is a $O(\min(n^{1/3}, OPT^{1/2}))$-approximation algorithm where n is the number of rules in the input list and OPT is the size of the optimal solution. Here, we simplify their algorithm slightly and generalize it to create a class of approximation algorithms with the same approximation ratio. We then show that some Diplomat variations belong to this class.

We first define strip-rule patterns identified by Applegate *et al.* [2].

Definition 3. *A strip-rule pattern is a two-dimensional pattern which can be created by a rule list where each rule spans an entire row or column.*

Applegate *et al.* prove several properties about strip-rule patterns, the most important of which is that any 2×2 or

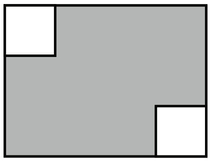


Fig. 4. A forbidden rectangle

3×3 sub-array must have a monochromatic row or column.

Definition 4. *A* forbidden rectangle *is a minimal subpattern containing either a 2×2 or 3×3 subarray with no monochromatic rows or columns.*

Any pattern that includes a forbidden rectangle is not a strip-rule pattern.

We now define the class of strip solver compression algorithms.

Definition 5. *A* strip solver *is a compressor that on an $m \times n$ strip-rule pattern always returns a rule list with at most $k(m+n)$ rules for some constant k.*

On other $m \times n$ patterns, a strip solver may either return such a rule list or it may fail. Any compressor that completes on all values and satisfies the bounds on strip rule patterns can be turned into a strip solver by checking to see if the limit was met and failing if it does not.

Theorem 5. *Diplomat with the rotational greedy scheduler is a strip solver compressor with $k = 1$.*

Proof: Let P be an arbitrary strip-rule pattern. We prove this by induction on $m+n$, the sum of the number of rows and columns in P. If $m+n \leq 1$, then we have the empty pattern which Diplomat solves optimally with 0 rules. If $m+n = 2$, then we have a pattern with 1 row and 1 column which means it is a single cell which Diplomat again solves optimally with 1 rule. If $m+n = 3$, then we have a pattern with either 1 row and 2 columns or 2 rows and 1 column which Diplomat solves optimally using 2 rules.

Now consider $m+n \geq 4$ and assume the inductive hypothesis holds for all patterns where the sum of the rows and columns is strictly smaller than $m+n$. Since P is a strip-rule pattern, there must exist a monochromatic row or column in P. Without loss of generality, assume there exists a monochromatic row R^*. This row can be merged with either of its neighbors at cost 1 by coloring R^* with a single rule. Thus, whichever pair of rows, R_t and R_b, that Diplomat selects, the cost to merge them must be exactly one.

We now argue that P', the resulting pattern of merging R_t and R_b, is still a strip-rule pattern. Assume otherwise for contradiction. This implies that the row created by the merge is part of a forbidden rectangle. However, since only a single rule is placed in one of the rows, the merged row must be identical to the other row. This implies that the forbidden rectangle already exists in the original pattern P, a contradiction. Thus, P' is a strip rule pattern where the sum of the rows and columns is at most $m+n-1$. Thus, by the inductive hypothesis, Diplomat with the rotational greedy scheduler solves P' with at most $m+n-1$ rules. Combining this with the one rule used to merge R_t and R_b, we see that Diplomat colors P with at most $m+n$ rules and the inductive case is complete. ■

We now define the class of *iterated strip solver* algorithms using ideas from Applegate *et al.*. Iterated strip solver algorithms can be applied to arbitrary 2-dimensional patterns P. The first step is to partition P as follows. Let P be a $2^h \times 2^w$ pattern for integers h and w and assume without loss of generality that $w \geq h$; we pad P with empty rows and columns as needed if they are not powers of 2. For each integer $q \in [1, w]$, do the following. Partition P into vertical sections of width 2^q. Partition each section into a sequence of disjoint blocks $B_1, B_2, \ldots$, where each block B_i is a maximal height subpattern using the full width of the section in which the strip solver returns a value. Apply a strip solver to each block. For each value of q, form a solution by taking the union of solutions for each block defined by q. Because the blocks are disjoint, the solutions for each block are completely independent. The final solution is the best of the w different solutions for each width 2^q.

Applegate *et al.* prove the following result.

Lemma 1. *Suppose there are r disjoint forbidden rectangles in a 2-dimensional pattern P. Then $OPT(P) \geq r/4$.*

Proof: Applegate *et al.* give a proof for this in [2]. The intuition is that the first rule to include any corner of a forbidden rectangle must itself have a corner within that forbidden rectangle. ■

We now prove an approximation bound for any iterated strip solver algorithm.

Theorem 6. *For any pattern P, any iterated strip solver is a $O(k \min(n^{1/3}, OPT^{1/2}))$-approximation algorithm.*

Proof: Let there be a total of $b + 2^q$ blocks, 2^q blocks for the vertical sections plus b additional blocks added because of sections that cannot be done by the strip solver. For any pair of vertically adjacent blocks, there must be at least one forbidden rectangle. Otherwise, the two blocks would together be a strip rule pattern and the strip solver would have returned a value. While some of the forbidden rectangles could overlap, if two pairs are disjoint then their forbidden rectangles must also be disjoint. Thus, there are at least $b/2$ forbidden rectangles.

$$\begin{aligned}
TotalCost &= k \textstyle\sum_{B_i} (Width(B_i) + Height(B_i)) \\
&= k \textstyle\sum_{B_i} Width(B_i) + k \sum_{B_i} Height(B_i) \\
&= k(b + 2^q)\tfrac{2^w}{2^q} + k(2^q 2^h) \\
&= k(b2^{w-q} + 2^w + 2^{q+h}) \\
&\leq k(OPT^2/2^q + OPT + 2^q OPT)
\end{aligned}$$

Since we try many values of q, we can pick the one that minimizes the above expression. If we let $2^q = \sqrt{OPT}$ we get $Cost \leq k(OPT + 2OPT^{3/2}) = O(kOPT^{3/2})$ and so the whole thing is a $O(k\sqrt{OPT})$-approximation.

We can choose to use L if it is better than our output classifier. Now assume $OPT \geq |L|^{2/3} = n^{2/3}$. In this case, $|L|/OPT \leq n^{1/3}$. Further, the two bounds are equal if $OPT = n^{2/3}$ and the result follows. ■

Corollary 1. *Diplomat with the rotational greedy scheduler is a $O(\min(n^{1/3}, OPT^{1/2}))$-approximation algorithm.*

Proof: The result follows immediately from Theorem 5 and Theorem 6. ■

C. Comparison with Firewall Compressor

We now show that on two-dimensional patterns, Diplomat with a dynamic programming scheduler produces a smaller rule list than Firewall Compressor.

Theorem 7. *Given some two-dimensional pattern P, let D be the rule list produced by Diplomat with a dynamic programming scheduler and F be the rule list produced by Firewall Compressor. Then for all P, we have $|D| \leq |F|$.*

Proof: We prove this by induction on the number of rows m in P. As a base case where $m = 1$, a $1 \times n$ pattern takes the same number of rules for both Diplomat and Firewall Compressor as both use an optimal one-dimensional solver. We now consider patterns P with $m \geq 2$. Both Diplomat and Firewall Compressor eliminate one row from P to create an $m-1$ row pattern. Let R^* be the row removed by Firewall Compressor with cost k and P' be the resulting $m-1$ row pattern. The key observation is that Diplomat considers removing all rows including R^*; more precisely, Diplomat considers merging R^* with each of its neighboring rows using rules that only color R^*. When Diplomat considers removing row R^*, its cost c for doing so is at most k because one option that Diplomat considers is coloring all of R^*. Furthermore, because Diplomat uses an in-plane resolver, the resulting $m-1$ row pattern is also P'. By our inductive hypothesis, Diplomat will color P' with smaller total cost than Firewall Compressor will color P'. Thus, one choice for Diplomat with dynamic programming has cost no more than Firewall Compressor's cost. Since Diplomat chooses the lowest cost solution, the result follows. ■

Theorem 7 does not necessarily hold when we include redundancy removal and other post-processing steps. That is, while Theorem 7 guarantees that Diplomat with a dynamic programming scheduler will produce a rule list with no more rules than Firewall Compressor before performing redundancy removal, no such guarantee can be made after performing redundancy removal. In our experiments, we observe that Firewall Compressor rarely but occasionally does outperform Diplomat with a dynamic programming scheduler due to this phenomenon.

VI. Experimental Results

In this section we evaluate the effectiveness of Diplomat on real-life classifiers. Specifically, we assess how much Diplomat improves over Firewall Compressor, the current state of the art compression algorithm [10]. We consider two variants of Diplomat, Diplomat with a greedy scheduler and a general resolver (GDip) and Diplomat with a dynamic programming scheduler and an in-plane resolver (DDip).

TABLE I
Size of Largest Dimension

	Min	Median	Max
Small	5	14	37
Medium	24	66	139
Large	55	506	1805

A. Methodology

Both Diplomat variants and Firewall Compressor (FC) are sensitive to the ordering of the five packet fields (source IP address, destination IP address, source port, destination port, and protocol). We run these algorithms using each of the $5! = 120$ different permutations across all of the fields and use the best of the 120 results for each classifier. We also use redundancy removal [7] as a post-processing step.

We now define the metrics for measuring the effectiveness of our two Diplomat variants. Let f and g denote compressors and L denote a classifier. Let $f(L)$ denote the resulting rule list obtained by applying compressor f to L, and $|L|$ is the number of rules in L. The *compression ratio* of f on L is $\frac{|f(L)|}{|L|}$, and the improvement ratio of f over g on L is $\frac{|g(L)|-|f(L)|}{|g(L)|}$. In our results, we focus on the mean compression ratio and mean improvement ratio given a set of classifiers S. It is desirable to have a low compression ratio and a high improvement ratio.

We assess the performance of our algorithms using a set of 40 real-life classifiers which we first preprocess by removing redundant rules. We divide this set into three smaller sets based on the number of rules after running redundancy removal. The small set contains the 17 smallest classifiers, the middle set contains the next 12 larger classifiers, and the large set contains the 11 largest classifiers. The classifiers in the large set all have at least 600 rules after redundancy removal, with the largest having more than 7600 rules.

In general, the larger classifiers also have larger effective grids with the largest classifier having $n_d = 1805$. Not all of the fields strictly increase as the list size goes up. In particular, the protocol field only uses a few distinct values across all classifiers, usually 6 (TCP) and 17 (UDP). As such, $n_1 \leq 9$ for all classifiers. The range of n_d for each set can be seen in Table I.

B. Compression Results

The mean compression ratio for FC, GDip, and DDip on each set of classifiers can be found in Table II. For FC, for each classifier, we use the better of FC and the classifier after redundancy removal. For GDip and DDip, for each classifier, we use the best of the given method, FC, and the classifier after redundancy removal. This allows us to assess how much improvement is achievable by adding Diplomat to the available suite of classifier compression algorithms. The mean improvement ratio of using GDip or DDip over FC can be found in Table III.

TABLE II
MEAN COMPRESSION RATIO

	FC	GDip	DDip
Small	67.4%	67.4%	67.2%
Medium	50.8%	50.6%	45.7%
Large	44.5%	44.5%	30.2%
All	56.1%	56.1%	50.6%

TABLE III
MEAN IMPROVEMENT RATIO

	GDip vs FC	DDip vs FC
Small	0.0%	0.6%
Medium	0.3%	12.5%
Large	0.0%	30.6%
All	0.1%	12.4%

As we can see from the tables, GDip offers little improvement over FC. In fact, GDip only has a better solution than FC on one classifier in the medium set. In many cases they provide equally good solutions but FC outperforms GDip on 21 of the 40 classifiers. As such, we conclude there is little reason to use Diplomat with a greedy scheduler instead of or in addition to Firewall Compressor.

On the other hand, our results show that DDip offers significant improvement over FC. Specifically, on the medium and large sets, DDip has mean improvement ratios of 12.5% and 30.6%, respectively. On the small set, DDip offers only modest improvement over FC and redundancy removal. Overall, DDip outperforms FC on 19 classifiers. Interestingly, FC outperforms DDip on two classifiers. As we mentioned in Theorem 7, redundancy removal may affect one method or the other more depending on various factors. From these results, we conclude that Diplomat with a dynamic programming scheduler significantly outperforms Firewall Compressor, particularly as classifiers increase in size and complexity.

C. *Efficiency*

We implemented our algorithms using a combination of C# and VB. We found that Diplomat runs in under a second on the smaller sets, between a second to a minute on the medium sets, and up to a few hours for some permuatations on the largest sets. Our implementation has not been designed with an emphasis on speed. In particular, multiple calls to Firewall Compressor are made for each row when compared to any other row. Remembering this result and other changes should result in significant speed ups.

VII. CONCLUSIONS

In this paper, we presented Diplomat, an algorithm for compressing range-ACLs. We first presented the general framework and then presented concrete algorithms for implementing this framework. In particular, we presented a greedy scheduler that has guaranteed approximation bounds and a dynamic programming scheduler that performs well in practice. We implemented Diplomat and conducted side-by-side comparison with the prior Firewall Compressor algorithm on real life classifiers. The experimental results show that Diplomat outperforms Firewall Compressor most of the time, often by a considerable margin. In particular, on our largest ACLs, Diplomat has an average improvement ratio over Firewall Compressor of 30.6%. Furthermore, we generalized the Iterated Strip Rule algorithm in [2] and used it to create a larger class of algorithms that achieve the same approximation bounds. We showed that the greedy scheduler Diplomat is a member of this class. While we showed that some Diplomat versions belong to this class, the dynamic programming scheduler that we normally use does not.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under Grant No. CNS-0916044 and the National Natural Science Foundation of China (Grant No. 61272546).

REFERENCES

[1] Topcoder statistics. http://community.topcoder.com/tc?module=Static&d1=match_editorials&d2=srm150. Accessed 27/7/2012.

[2] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.

[3] P. Berman and B. DasGupta. Complexities of efficient solutions of rectilinear polygon cover problems. *Algorithmica*, 17:331–356, 1997.

[4] Richard Draves, Christopher King, Srinivasan Venkatachary, and Brian Zill. Constructing optimal IP routing tables. In *Proceedings of the IEEE INFOCOM*, pages 88–97, 1999.

[5] V. S. Anil Kumar and H. Ramesh. Covering rectilinear polygons with axis-parallel rectangles. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 445–454, 1999.

[6] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-04)*, pages 595–604, June 2004.

[7] Alex X. Liu and Mohamed 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] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19:1237–1251, 2008.

[9] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2010.

[10] Alex X. Liu, Eric Torng, and Chad Meiners. Firewall compressor: An algorithm for minimizing firewall policies. In *Proceedings of the 27th Annual IEEE Conference on Computer Communications (Infocom)*, Phoenix, Arizona, April 2008.

[11] Alex X. Liu, Eric Torng, and Chad R. Meiners. Compressing network access control lists. *IEEE Transactions on Parallel and Distributed Systems*, 22:1969–1977, 2011.

[12] W. J. Masek. Some NP-complete set covering problems. 1978.

[13] Yu-Wei Eric Sung, Xin Sun, Sanjay G. Rao, Geoffrey G. Xie, and David A. Maltz. Towards systematic design of enterprise networks. *IEEE/ACM Trans. Netw.*, 19(3):695–708, June 2011.

[14] Subhash Suri, Tuomas Sandholm, and Priyank Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.

[15] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with difane. *SIGCOMM Comput. Commun. Rev.*, 40(4):351–362, August 2010.