Compressing Network Access Control Lists*

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*—An access control list (ACL) provides security for a private network by controlling the flow of incoming and outgoing packets. Specifically, a network policy is created in the form of a sequence of (possibly conflicting) rules. Each packet is compared against this ACL, and the first rule that the packet matches defines the decision for that packet. The size of ACLs has been increasing rapidly due to the explosive growth of Internet-based applications and malicious attacks. This increase in size degrades network performance and increases management complexity. In this paper, we propose ACL Compressor, a framework that can significantly reduce the number of rules in an access control list while maintaining the same semantics. We make three major contributions. First, we propose an optimal solution using dynamic programming techniques for compressing one-dimensional range based access control lists. Second, we present a systematic approach for compressing multi-dimensional access control lists. Last, we conducted extensive experiments to evaluate ACL Compressor. In terms of effectiveness, ACL Compressor achieves an average compression ratio of 50.22% on real-life rule sets. In terms of efficiency, ACL runs in seconds, even for large ACLs with thousands of rules.**

***Index Terms*—Access Control List, Packet Classification, Firewall, Algorithm.**

I. Introduction

A. Background and Motivation

Access control lists (ACLs) 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 an ACL is to examine every packet's field values and decide how to enforce the network policy. This policy is specified as a sequence of (possibly conflicting) rules. Each rule in an ACL 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 ACLs are typically 4-dimensional (over 4 packet fields: source IP address, destination IP address, destination port number, and protocol type) or 5-dimensional (over 5 packet fields: source IP address, destination IP address, source port number, destination port number, and protocol type).

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

Rule	SIP	DIP	SPort	DPort	Proto	Act
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 ACL

In this paper, we study a general ACL compression problem: *given an ACL f, generate another ACL f' that is semantically equivalent to f but has the minimum possible number of rules.* We call this process "*ACL compression*". We focus on five versions of ACL compression that differ only in the format of field constraints of the output ACL: (1) *range ACL compression* where field constraints are specified by a range of integers (e.g., source port $\in [5000, 6000]$), (2) *prefix ACL compression* where field constraints are specified by a prefix string (e.g., source IP $=$ $192.168.*.*$), (3) *ternary ACL compression*, where field constraints are specified by a ternary (including prefix) string (e.g., source IP $= 192.*.0.*$), (4) *range-prefix ACL compression* where some field constraints are specified by ranges and the remaining field constraints are specified by prefix strings, and (5) *range-ternary ACL compression* where some field constraints are specified by ranges and the remaining field constraints are specified by ternary strings. In most ACLs, the source port number and destination port number fields use a range field constraint whereas the source IP address, destination IP address, and protocol type fields use a prefix or ternary field constraint.

We give an example that illustrates the possibilities of ACL compression. The input ACL with five rules is depicted in Figure 1(A). For simplicity, we assume this ACL only examines one packet field F, the domain of F is $[1, 100]$, and F uses a range field constraint. The geometric representation of this five rule ACL is given in Figure 1(a) where the predicate of each rule is a line segment, the decision of each rule is the color of its line segment, a packet corresponds to a point on the line, and the decision for a packet is the color of the first line segment that contains the point. To generate another sequence of rules that is equivalent to the ACL 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

*The preliminary version of this paper titled "Firewall Compressor: An Algorithm for Minimizing Firewall Policies. " was published in the proceedings of the 27th Annual IEEE Conference on Computer Communications (INFOCOM), pages 176-180, April, ©IEEE, 2008.

(A)	(B)	(C)
$F \in [41, 60] \rightarrow d_1$	$F \in [41, 60] \rightarrow d_1$	$F \in [41, 60] \rightarrow d_1$
$F \in [21, 55] \rightarrow d_2$	$F \in [81, 100] \rightarrow d_3$	$F \in [21, 80] \rightarrow d_2$
$F \in [45, 80] \rightarrow d_2$	$F \in [21, 40] \rightarrow d_2$	$F \in [1, 100] \rightarrow d_3$
$F \in [1, 65] \rightarrow d_3$	$F \in [61, 80] \rightarrow d_2$	
$F \in [75, 100] \rightarrow d_3$	$F \in [1, 20] \rightarrow d_3$	

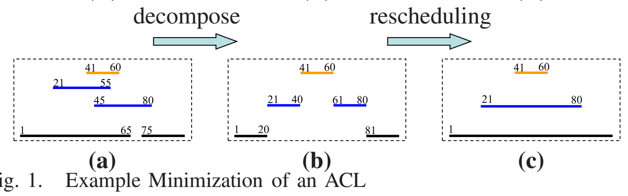


Fig. 1. Example Minimization of an ACL

rules is in Figure 1(b). We now reschedule the intervals to generate a shorter semantically equivalent ACL as follows. We first schedule the interval $[41, 60]$. This allows us to schedule the two intervals $[21, 40]$ and $[61, 80]$ together using one interval $[21, 80]$ based on first-match semantics. Finally, we can schedule intervals $[1, 20]$ and $[81, 100]$ together using one interval $[1, 100]$ again based on first-match semantics. The three ACLs in Figures 1(A), 1(B) and 1(C) are equivalent, but the rightmost ACL has fewer rules.

Our work on ACL compression has two important motivations. First, ACL compression is useful for network system management and optimization because minimizing large ACL rule sets greatly reduces the complexity of managing and optimizing network configurations. As a result, ACL compression tools in general and our ACL compression tool in particular have been used or proposed for use in several prominent network management and optimization projects, such as Yu et al.'s DIFANE work [18] and Sung et al.'s work on systematic design of enterprise networks [16], [17]. Second, some network products have hard constraints on the number of rules that they support. For example, NetScreen-100 only allows ACLs with at most 733 rules. ACL compression may allow users with larger ACLs to still use such devices. This may become an increasingly important issue for many users as ACL size has grown dramatically due to an increase in Internet applications and services as well as an increase in known vulnerabilities, threats, and attacks [2]. For example, our older ACLs have at most 660 rules whereas the ACLs we have more recently acquired have as many as 7652 rules.

B. Summary and Limitations of Prior Art

The main limitation of prior work is, to the best of our knowledge, the lack of work on two key ACL compression problems. First, no prior work has considered range ACL compression for more than two dimensions, and we are aware of only one paper that has considered range ACL compression for two dimensions [1]. Second, no prior work has considered ACL compression where different fields use different field constraints. There is prior work that considers prefix ACL compression [2], [1], [12] and ternary ACL compression [11], [13], but none of these algorithms can be directly used to compress ACLs where different fields have different field constraints.

C. Our Approach

We use a divide-and-conquer approach where we first decompose a multi-dimensional ACL into a hierarchy of one-dimensional ACLs using decision diagrams. We minimize each one-dimensional ACL using appropriate ACL compression algorithms. For one-dimensional range and prefix ACLs, we achieve optimal compression. Finally we combine the many one-dimensional ACL compression solutions into one multi-dimensional solution to the original multi-dimensional ACL minimization problem. Our approach has two key features. First, the hierarchical representation of ACLs is canonical. That is, two semantically equivalent ACLs will have the same hierarchical representation no matter how they are specified. Thus, our approach eliminates variance due to human factors in the design of given ACLs. Second, our approach allows range, prefix, and ternary fields to be optimized independently using customized algorithms because it deals with one field at a time. We name our approach "*ACL Compressor*".

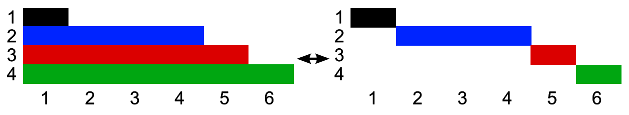


Fig. 2. Converting a schedule to a canonical schedule

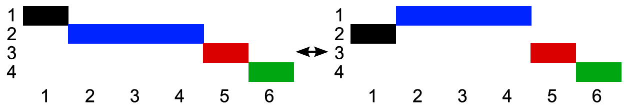


Fig. 3. Swapping two adjacent intervals

D. Key Contributions

In this paper, we make three key contributions: (1) We propose an optimal algorithm for the one-dimensional range ACL compression problem. This algorithm uses dynamic programming techniques. (2) We present a systematic and efficient framework for generating good solutions to the NP-hard multi-dimensional range, range-prefix, and range-ternary ACL compression problems. Our framework combines the locally optimized one-dimensional solutions into a good but not necessarily optimal multi-dimensional solution. (3) We conducted extensive experiments on both real-life and synthetic ACLs. The results show that ACL Compressor achieves an average compression ratio of 50.22% on real-life range-prefix ACLs.

ACL Compressor is designed to run off-line so that network managers do not need to read or manage the compressed ACL. Instead, network managers can continue to design and maintain an intuitive and understandable ACL f while using ACL Compressor to generate and deploy a minimal yet semantically equivalent ACL f' on their network device.

The rest of the paper proceeds as follows. In Section II, we describe optimal solutions using dynamic programming techniques to two weighted one-dimensional ACL compression problems. In Section III, we give a solution to the multi-dimensional ACL compression problem. We show experimental results in Section IV. We give concluding remarks in Section V. The digital supplemental material of this paper include all the proofs for the lemmas and theorems in this paper, the pseudocode for some algorithms in this paper, and detailed review of related work.

II. One-dimensional ACL Compression

We focus primarily on the *weighted one-dimensional range ACL compression problem*. We briefly discuss the *weighted* *one-dimensional prefix and ternary ACL compression problems*. For range and prefix ACL compression problems, we present optimal algorithms that use dynamic programming. For ternary ACL compression, we present a bit merging heuristic. We use these algorithms for one-dimensional ACLs as building blocks in our multi-dimensional ACL Compressor framework, which we describe in our next section. Table II lists the notations used throughout this paper.

Symbol	Description
F_i	field i
$D(F_i)$	domain of F_i
d	# of dimensions
b	# of bits
Σ	set of all packets
f	an ACL
p	packet
$f(p)$	decision of f on p
$\{f\}$	set of all ACLs equivalent to f
$\mathcal{P}$	prefix
$f_{\mathcal{P}}$	ACL equivalent to f on $\mathcal{P}$
δ	serialization mapping function
U	universe of tasks
C	set of colors

Symbol	Description
X	cost vector of colors
$c(i)$	color of task i
$\|c(i)\|$	number of tasks with color i
$S(I)$	ACL schedule for input $I = (U, C, X)$
d_i	decision i
w_{d_i}	cost of decision i
$C(f_{\mathcal{P}})$	minimum cost of an ACL equivalent to $f_{\mathcal{P}}$
$C(f_{\mathcal{P}}^{d_i})$	minimum cost of an ACL equivalent to $f_{\mathcal{P}}$ whose last rule's decision is d_i

TABLE II
NOTATIONS USED IN THIS PAPER

A. One-dimensional Range ACL Compression

The weighted one-dimensional range ACL compression problem is the fundamental problem for compressing range-based domains such as port number ranges. We solve this problem by mapping an ACL to a scheduling problem via two processes: *decomposition* and *serialization*. Once the ACL is rewritten as a scheduling problem, we use dynamic programming to find the optimal schedule from which we compute a minimum ACL.

1) ACL Decomposition and Serialization: In a non-overlapping one-dimensional range based ACL, for any two rules, say $F \in [a, b] \to d_x$ and $F \in [c, d] \to d_y$, 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., $F \in [a, d] \to d_x$ if $b + 1 = c$, and $F \in [c, b] \to d_x$ if $d + 1 = a$). A non-overlapping one-dimensional range ACL is called *canonical* if and only if no two rules in the ACL can be merged into one rule. For example, Figure 1(B) shows a canonical ACL that is equivalent to the ACL in Figure 1(A).

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

We then *serialize* the canonical ACL 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 ACL f'' is called a *serialized ACL*. 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 ACL 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 ACL 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 weighted one-dimensional range compression problem as the following ACL scheduling problem.

2) The ACL Scheduling Problem: In the ACL 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 $1 \leq j \leq z$, let $|j|$ denote the number of tasks with color j.
- Let $X = \{x_1, \ldots x_z\}$ be the cost vector where it costs x_j to execute any task that has color j for $1 \leq j \leq z$.

Then an input instance to the ACL 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 ACL where each task in U represents a rule in the ACL and the color of the task represents the decision of the rule. In the one-dimensional ACL compression problem, the cost of every task is 1; that is, we assign the value 1 to every x_j ($1 \leq j \leq z$). We consider the weighted one-dimensional range compression problem because its solution can be used as a routine in solving the multi-dimensional ACL range compression problem.

For any ACL scheduling input instance $I = (U, C, X)$, an ACL 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 an ACL 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 $C(S(I))$.

3) 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 observation is to define the notion of a canonical schedule.

Definition 2.1 (Canonical Schedule): For any input instance $I = (U, C, X)$, a legal schedule $S(I) = \{r_1, \dots, r_m\}$ is a canonical schedule if for each interval $r_i = [p_i, q_i]$, $1 \le i \le m$, it holds that $p_i \in f_i$ and $q_i \in f_i$.

We then observe that for any schedule including an optimal schedule, there exists an equivalent canonical schedule that has the same cost. For example, Figure 2 depicts two equivalent schedules with identical costs where the one on the right is canonical. This allows us to consider only canonical schedules for the remainder of this section.

Lemma 2.2: 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 for any canonical schedule S, swapping two adjacent intervals that do not overlap results in a canonical schedule with the same cost. Figure 3 illustrates this observation for an example canonical schedule.

Lemma 2.3: 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 for any canonical schedule including an optimal canonical schedule, there exists an equivalent 1-canonical schedule that has the same cost. This implies that for any input instance I, there exists an optimal 1-canonical schedule.

Lemma 2.4: 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 $C(S'(I)) = C(S(I))$.

Let k be the number of tasks with the same color as task 1 including task 1 itself. Given Lemma 2.4 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 $S(I)$. For example, 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 2.5: 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 $1 \le j \le i-1$, 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 $1 \le i \le j \le n$:

- $I(i, j)$ denotes an input instance with a universe of tasks $\{i, \dots, j\}$ 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 $Opt(I(i, j))$.

Given that there exists an optimal 1-canonical schedule for any input instance, we derive the following Lemma.

Lemma 2.6: 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) Suppose task 1 is not the only task fired in the last interval of $Opt(I(1, n))$. Let 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 ACL scheduling problem. For $1 \le j \le z$, 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 ($1 \le i \le n$). Therefore, for $1 \le i \le n$, $G_{c(i)}$ denotes the set of all the tasks that have the same color as task i.

Theorem 2.7: $C(i, j)$ can be computed by the following recurrence relation.
For $1 \le i \le n$, $C(i, i) = x_{c(i)}$.
For $1 \le i < j \le n$, $C(i, j) = \min(x_{c(i)} + C(i+1, j), \min_{l \in G_{c(i)} \wedge i+2 \le l \le j}(C(i+1, l-1) + C(l, j)))$.

B. One-dimensional Prefix ACL Compression

In [12], we proposed a polynomial time optimal algorithm for the weighted one-dimensional prefix ACL compression problem using dynamic programming. This algorithm is based on three observations. First, the last rule of f can always have its predicate changed to a default predicate. This change is possible because f is complete and therefore extending the range of the last rule interval cannot change the semantics of f. Second, we can append an additional default rule to f without changing the semantics of the resulting ACL. Third, the structure imposed by the prefix rules provides an efficient mechanism to divide the problem space into isolated subproblems. For example given a prefix domain of **** we only have to consider two cases: ****, or 0***, and 1***. The dynamic programming solution subdivides f along prefix boundaries until each prefix contains only a single decision. These adjacent prefixes are combined onto a minimal prefix rule list that covers both prefixes. This process is repeated until we are left with a single prefix and classifier. These observations lead to a the completely different dynamic programming formulation [9]:

Theorem 2.8: Given a one-dimensional packet classifier f on $\{*\}^b$, a prefix $\mathcal{P}$ where $\mathcal{P} \subseteq \{*\}^b$, the set of all possible decisions $\{d_1, d_2, \cdots, d_z\}$ where each decision d_i has a cost w_{d_i} ($1 \le i \le z$), we have that

$$C(f_{\mathcal{P}}) = \min_{i=1}^{z} C(f_{\mathcal{P}}^{d_i})$$

where each $C(f_{\mathcal{P}}^{d_i})$ is calculated as follows:
(1) If f has a single decision on $\mathcal{P}$, then

$$C(f_{\mathcal{P}}^{d_i}) = \begin{cases} w_{f(x)} & \text{if } f(x) = d_i \ \forall x \in \mathcal{P} \\ w_{f(x)} + w_{d_i} & \text{if } f(x) \neq d_i \ \forall x \in \mathcal{P} \end{cases}$$

(2) If f does not have single decison on $\mathcal{P}$, then

$$C(f_{\mathcal{P}}^{d_i}) = \min \begin{cases} C(f_{\underline{\mathcal{P}}}^{d_1}) + C(f_{\overline{\mathcal{P}}}^{d_1}) - w_{d_1} + w_{d_i}, \\ \dots, \\ C(f_{\underline{\mathcal{P}}}^{d_{i-1}}) + C(f_{\overline{\mathcal{P}}}^{d_{i-1}}) - w_{d_{i-1}} + w_{d_i}, \\ C(f_{\underline{\mathcal{P}}}^{d_i}) + C(f_{\overline{\mathcal{P}}}^{d_i}) - w_{d_i}, \\ C(f_{\underline{\mathcal{P}}}^{d_{i+1}}) + C(f_{\overline{\mathcal{P}}}^{d_{i+1}}) - w_{d_{i+1}} + w_{d_i}, \\ \dots, \\ C(f_{\underline{\mathcal{P}}}^{d_z}) + C(f_{\overline{\mathcal{P}}}^{d_z}) - w_{d_a} + w_{d_i} \end{cases}$$

where $f_{\mathcal{P}}^{d_i}$ is a classifier f on prefix $\mathcal{P}$ with a background decision d_i. □

C. *One-dimensional Ternary ACL Compression*

We address the NP-hard weighted one-dimensional ternary ACL compression problem by first producing an optimal weighted prefix ACL and then applying bit merging [13] to further compress the prefix ACL. We use bit merging rather than McGeer and Yalagandula's heuristics [11] since we need to handle more than two decisions. This algorithm is not guaranteed to produce an optimal weighted one-dimensional ternary ACL.

III. Multi-dimensional ACL Compression

In this section, we present ACL Compressor, our framework for compressing multi-dimensional ACLs. Similar to [12], we take a divide-and-conquer approach to this multi-dimensional problem. First, we decompose a multi-dimensional ACL into a hierarchy of one-dimensional ACLs using decision diagrams. Second, for one-dimensional range ACLs, we use our optimal weighted one-dimensional range ACL optimization algorithm; for one-dimensional prefix ACLs, we use the optimal weighted one-dimensional prefix ACL optimization algorithm in [12]; for one-dimensional ternary ACLs, we use the same prefix ACL optimization algorithm followed by bit merging [12]. Third, we combine the multiple one-dimensional solutions into one multi-dimensional solution to the original multi-dimensional ACL minimization problem. Note that the multi-dimensional solution is not guaranteed to produce a minimal classifier. In this section, we assume we are dealing with a range-prefix ACL compression problem. We handle range-ternary ACL compression by simply running bit merging after optimal one-dimensional prefix ACL compression.

A. *ACL Decomposition*

To leverage our one-dimensional ACL optimization algorithms, we first decompose the given multi-dimensional ACL into a hierarchy of one-dimensional ACLs by converting the given ACL to a canonical representation called *Firewall Decision Diagram (FDD)*, which was introduced by Liu and Gouda in [5], [4]. At a fundamental level, a d-dimensional FDD is an annotated acyclic directed graph with d levels of nodes. Each node in an FDD can be viewed as a smaller FDD. For example, a d-dimensional FDD is composed of labeled edges from a root node that connect to several $(d-1)$-dimensional FDDs. This hierarchical view of ACLs facilitates sophisticated optimization techniques such as identification and reuse of critical low dimensional ACLs. Using a canonical FDD representation, our approach is insensitive to the input ACL syntax because any two semantically equivalent ACLs will result in the same FDD after reduction. This key feature of our algorithm eliminates variance due to human factors in specifying ACLs.

We now formally describe FDD's using a description from [7]. "An (FDD) with a decision set DS and over fields $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following 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 $F(v) \in \{F_1, \cdots, F_d\}$ if v is a nonterminal node and $F(v) \in DS$ if v is a terminal node. (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: (i) *Consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$. (ii) *Completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$."

Given an ACL such as the one shown in Figure 4, we convert it to an equivalent FDD using the FDD construction algorithm in [7]. Figure 5 shows an example FDD over the two fields F_1 and F_2 where $D(F_1) = [0, 10]$ and $D(F_2) = [0, 15]$. We use letter "a" as a shorthand for "*accept*" and letter "d" as a shorthand for "*discard*" when labeling the terminal nodes.

We next perform FDD reduction where we identify and eliminate redundant or isomorphic low dimensional ACLs that may be reused multiple times within a high dimensional ACL. 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. A reduced FDD is an FDD with no redundant nodes.

The core operation in FDD reduction is to identify isomorphic nodes, which can be sped up using signatures as follows. 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 the signature of node v as follows: $Sig(v) = h(Sig(v_1), E_1, \cdots, Sig(v_k), E_k)$ where h is a one-way and collision resistant hash function such as MD5 [14] and SHA-1 [3]. 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

#	F_1	F_2	Decision
1	[0,2]	11**	discard
2	[0,2]	****	accept
3	[5,6]	0***	accept
4	[5,6]	10**	accept
5	[5,6]	****	discard
6	[0,10]	****	discard

Fig. 4. Input ACL

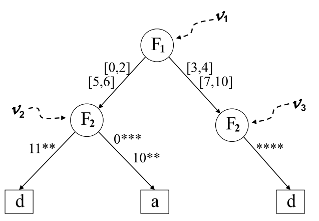


Fig. 5. An FDD

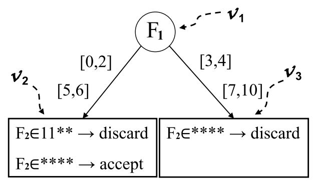


Fig. 6. "Virtual" one-dimensional ACL

#	F_1	F_2	Decision
1	[3,4]	****	discard
2	[7,10]	****	discard
3	[0,10]	11**	discard
4	[0,10]	****	accept

Fig. 7. Compressed output ACL

point to v_j to point to v_i. Further, we eliminate double edges between node v_i and its parents. Note that we process nodes in the FDD level by level from the terminal node level to the root node level.

B. Computing the Compressed ACL

Next, we present the core algorithm for compressing multi-dimensional ACLs using the FDD in Figure 5 as a running example. We observe that v_1, v_2, and v_3 can be seen as one-dimensional ACLs over their respective fields. We compute a compressed multi-dimensional ACL by first applying the appropriate one-dimensional ACL compression algorithm to each node's ACL and then composing their compressed ACLs into a multi-dimensional ACL. For example, for the FDD in Figure 5, for F_1 nodes, we use our weighted one-dimensional range ACL compression algorithm, and for F_2 nodes, we use the weighted one-dimensional prefix ACL compression algorithm.

Given an FDD, we start generating one-dimensional ACLs from the bottom nodes. For each node, we first create an ACL of non-overlapping rules from the node's outgoing edges. For example, in Figure 5, we start with v_2 and v_3. The ACLs for v_2 and v_3 before compression are listed in Table III. Given these two prefix ACLs, we apply the weighted one-dimensional prefix ACL compression algorithm, which produces the two minimal prefix ACLs in Figure 6.

v_2			
#	F_2	Decision	Cost
1	0***	accept	1
2	10**	accept	1
3	11**	deny	1

v_3			
#	F_2	Decision	Cost
1	****	deny	1

TABLE III
ACLS FOR v_2 AND v_3 BEFORE COMPRESSION

Now consider the root node in Figure 5. Treating the two minimal ACLs computed for v_1 and v_2 as terminal nodes, Figure 6 shows a one-dimensional FDD rooted at v_1. The corresponding input instance is given in Table IV. We now explain the costs that we assigned to decisions v_2 and v_3. Given a one-dimensional FDD for v_1, we can use the ACLs generated for v_1's children to form a two-dimensional ACL. For example, using the ACL for v_1 in Table IV, we expand rule 1 into two rules by prepending rule 1's F_1 field to both rules in v_2's ACL. Likewise, rule 2 is converted into a single rule that is derived from v_3's ACL. To account for this expansion, we assign the cost for decision v_i to be the total number of rules in v_i's ACL after compression. For this example, v_2 has cost 2 and v_3 has cost 1. We use these costs to produce a minimal table upon ACL composition. For this example, after applying our weighted one-dimensional range ACL compression algorithm on the ACL in Table IV, we get the minimal one-dimensional ACL in Table V. Finally, composing this ACL with the minimal ACLs for v_2 and v_3 in Figure 5, we get the final multi-dimensional ACL in Figure 7.

v_1			
#	F_1	Decision	Cost
1	[0,2]	v_2	2
2	[3,4]	v_3	1
3	[5,6]	v_2	2
4	[7,10]	v_3	1

TABLE IV
ONE-DIMENSIONAL ACL FOR v_1 BEFORE COMPRESSION

v_1			
#	F_1	Decision	Cost
1	[3,4]	v_3	1
4	[7,10]	v_3	1
3	[0,10]	v_2	2

TABLE V
ONE-DIMENSIONAL ACL FOR v_1 AFTER COMPRESSION

To summarize, in this step, we compute a compressed multi-dimensional ACL from a reduced FDD in the following bottom up fashion. For each terminal node of the FDD, we generate a (non-overlapping) ACL from the labels of v's outgoing edges, assign a cost of 1 to each decision, and finally apply the appropriate weighted one-dimensional ACL compression algorithm to compress the ACL. For each non-terminal node v, we generate a (non-overlapping) ACL from the labels of v's outgoing edges where each decision is a node pointed to by an outgoing edge of v, assign a cost to each decision where the cost is the number of rules in the compressed ACL for the corresponding node, apply the appropriate weighted one-dimensional ACL compression algorithm to compress the ACL, and finally compose the resulting one-dimensional compressed ACL with the compressed ACLs of v's children to form a multi-dimensional ACL.

C. Redundancy Removal

We observe that ACL composition can produce ACLs with redundant rules. We are fortunate that the ACL in Figure 7 is optimal and therefore contains no redundant rules; however, as a postprocessing step, we run a redundancy removal algorithm [10], [8] on the resultant ACL. Note that it is also possible to run redundancy removal on each non-terminal node's ACL. In some cases, this results in a more accurate cost value for each node and can lead to smaller ACLs.

D. Rule Logging

Devices that use ACLs commonly provide facilities to log matches against specific rules. However, ACL Compressor generates a new set of rules which conflicts with logging. We propose preserving logging information by assigning a unique decision to each rule that has logging enabled. The unique decision ensures that the logged rule cannot be merged with any other rule and thus ensures the correct logging behavior. For example, suppose the rules $F \in [21, 55] \rightarrow d_2$ and $F \in [48, 80] \rightarrow d_2$ in Figure 1 have logging enabled. ACL compressor then redefines these rules to be $F \in [21, 55] \rightarrow d_2'$ and $F \in [48, 80] \rightarrow d_2''$. After optimization we will have four rules $F \in [41, 60] \rightarrow d_1$, $F \in [21, 40] \rightarrow d_2'$, $F \in [61, 80] \rightarrow d_2''$, and $F \in [1, 100] \rightarrow d_3$.

As the percentage of logged rules increases, the effectiveness of ACL Compressor decreases as fewer rules have common decisions and thus fewer rules can be combined together. In extreme cases where most rules are logged, few rules will have common decisions, and we do not recommend running ACL Compressor because the large number of rules with unique decisions makes the rule list uncompressable.

We can still reduce the number of rules by removing upward redundant rules via the process described in [6]. An upward redundant rule is a rule that is redundant and will never match any packets because all packets that match the rule also match earlier rules in the ACL.

IV. Experimental Results

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

A. Methodology

a) Measurement Metrics: We first define the metrics that we used to measure the effectiveness of ACL Compressor. In this paragraph, f denotes an ACL, S denotes a set of ACLs, and AC denotes ACL Compressor. The variable order that we use to convert an ACL into an equivalent FDD affects the performance of ACL Compressor. We number the five packet fields as follows: protocol type = 0, source IP address = 1, destination IP address = 2, source port number = 3, and destination port number = 4. We represent the $5! = 120$ different permutations of the five packet fields with these numbers. For example, permutation 01342 corresponds to (protocol type, source IP address, source port number, destination port number, destination IP address). For any permutation p, we use AC$_p$ to denote ACL Compressor using permutation p and $AC_p(f)$ denotes the ACL produced by applying ACL Compressor with permutation p on f. For a given classifier f, we use AC$_{Best}$ to denote ACL Compressor using the best of the 120 variable orders for f. For a set of classifiers S, we again use AC$_{Best}$ to denote ACL compressor using the best of the 120 variable orders for each classifier $f \in S$ where different classifiers may use different variable orders. We define the compression ratio of AC_p on f as $\frac{|AC_p(f)|}{|f|}$. We define the following two metrics for assessing the performance of AC on a set of ACLs S.

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

Within our experiments, we use two sets of ACLs which we describe below. We always treat the source and destination port fields as range fields. We create three separate treatments where we view the source IP, destination IP, and protocol fields as range fields, prefix fields, and ternary fields, respectively. We thus report results for range ACL compression, range-prefix ACL compression, and range-ternary ACL compression.

b) Real-life ACLs: We first define a set RL of 40 real-life ACLs from a set of 65 real-life ACLs that we performed experiments on. RL is chosen from a larger set of real-life ACLs obtained from various network service providers where the ACLs range in size from dozens to thousands of rules. We eliminated structurally similar ACLs from RL because similar ACLs exhibited similar results for each method. Structurally similar ACLs have identical rule structure and differ only in the range or prefix values in the given predicates. ACL compressor will produce the same number of rules for these structurally similar ACLs so we eliminate structurally similar ACLs to prevent biasing both the average and total compression ratios. We created RL by randomly choosing a single ACL from each set of structurally similar ACLs.

c) Synthetic ACLs: Because ACLs are considered confidential due to security concerns, it is difficult to acquire a large sample of real-life ACLs. To address this issue and further evaluate the performance of ACL Compressor, we generated SYN, a set of synthetic ACLs of 7 sizes, where each size has 25 independently generated ACLs. Every predicate of a rule in our synthetic ACLs has five fields: source IP, destination IP, source port, destination port, and protocol. We based our generation method upon Rovniagin and Wool's [15] model of synthetic rules.

B. Effectiveness

We now assess the effectiveness of ACL Compressor. Because ACL Compressor is generally run offline, we assume that network administrators will typically try all 120 different permutations to generate the best possible compression. For the range, range-prefix, and range-ternary compression problems on RL, AC$_{Best}$ achieves average compression ratios of 44.87%, 50.22%, and 42.26% with standard deviations of 22.82%, 22.40%, and 20.65%, respectively, and total compression ratios of 41.77%, 53.12%, and 38.99%. In Figure 8, we show the cumulative percentage graph for the range-prefix compression ratio of AC$_{Best}$ for each classifier in RL.

We now assess how much impact variable order has on the effectiveness of ACL Compressor and whether or not one variable order performs well for most classifiers. For each permutation p, we computed the average and total range-prefix compression ratios that AC$_p$ achieves on RL and display the cumulative percentage graphs of these values in Figures 9 and 10, respectively. For all variable orders, the average range-prefix compression ratios achieved by ACL Compressor fall in the range between 56.88% and 71.40%, and the total compression ratios achieved by ACL Compressor fall in the range between 59.32% and 83.41%. From these figures, we see that variable order does significantly influence the effectiveness of ACL Compressor but also that many of the variable orders are very effective.

As we noted earlier, since ACL Compressor runs offline and is efficient, we assume network managers will try all 120 permutations. If time does not permit, we suggest using permutation 01342 as it achieved the lowest range-prefix average compression ratio of 56.93% with a standard deviation of 24.15% on RL, and it achieved a total compression ratio of 59.32%. In Figure 8, we show the cumulative percentage graph for the range-prefix compression ratio of AC$_{01342}$ for each classifier in RL.

ACL Compressor works very well on Rovniagin and Wool's model of synthetic rules with average and total range-prefix

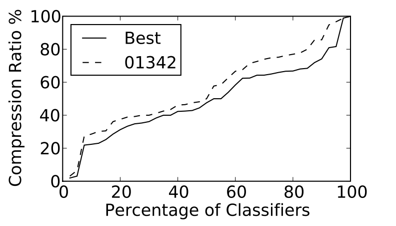


Fig. 8. CPG of range-prefix compression ratios for AC$_{Best}$ and AC$_{01342}$ for RL

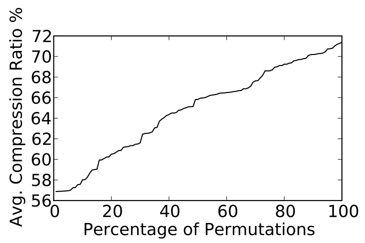


Fig. 9. CPG of range-prefix average compression ratios for AC$_p$ for RL

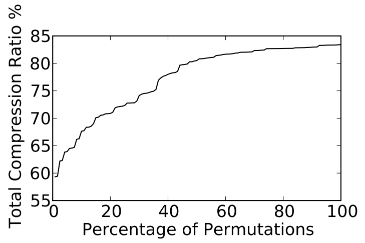


Fig. 10. CPG of range-prefix total compression ratios for AC$_p$ for RL

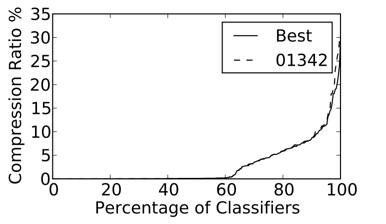


Fig. 11. CPG of range-prefix compression ratios for AC$_{Best}$ for SYN

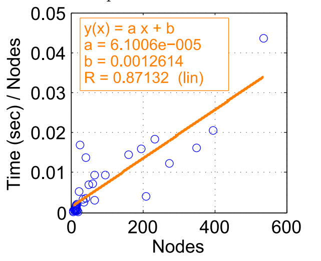


Fig. 12. Execution time per FDD node according to FDD size for RL and $p = 01342$.

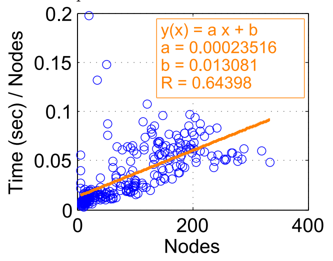


Fig. 13. Execution time per FDD node according to FDD size for SYN and $p = 01342$.

compression ratios on SYN of 2.99% and 14.04%, respectively. Figure 11 shows the cumulative percentage graphs of range-prefix compression ratios achieved by AC(01342) over SYN. From this figure we can see that 90% of the classifiers compress to at most a tenth of their original size.

C. Efficiency

We implemented all algorithms on Microsoft .Net framework 2.0. Our experiments were carried out on a desktop PC running Windows XP with 1G memory and a single 2.2 GHz AMD Opteron 148 processor. ACL Compressor is quite efficient taking at most a few minutes to compress any of our real life classifiers from RL. We observe that although ACL compressor does take more time as classifiers become more complex, it is still relatively efficient with essentially a quadratic running time in classifier complexity where we estimate classifier complexity by the total number of FDD nodes required to represent the classifier. We observe that the average amount of time that ACL compressor spends per FDD node increases in roughly a linear fashion with the classifier complexity as measured by the total number of FDD nodes. Specifically, Figure 12 shows a scatter plot of the the average amount of time ACL Compressor spends per FDD node versus the total number of FDD nodes required to represent the same classifier where each point in Figure 12 is a classifier in RL. We observe similar trends for synthetic classifiers as can be seen from a similar scatter plot in Figure 13.

V. Conclusions

In this paper, we present ACL Compressor, a framework for compressing ACLs and make three major contributions. First, we give an optimal algorithm for the one-dimensional range ACL compression problem. Second, we present a systematic solution for compressing multi-dimensional ACLs with mixed field constraints. Third, we conducted extensive experiments on both real-life and synthetic ACLs. Our experimental results show that ACL Compressor achieves an average compression ratio of 56.93% for range-prefix ACLs.

References

[1] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.

[2] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proc. ACM Sigmetrics*, pages 311–322, 2006.

[3] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1). *RFC 3174*, 2001.

[4] M. G. Gouda and A. X. Liu. Firewall design: consistency, completeness and compactness. In *Proc. 24th IEEE Int. Conf. on Distributed Computing Systems (ICDCS-04)*, pages 320–327, March 2004.

[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. Complete redundancy detection in firewalls. In *Proc. 19th Annual IFIP Conf. on Data and Applications Security, LNCS 3654*, pages 196–209, August 2005.

[7] A. X. Liu and M. G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(8), 2008.

[8] A. X. Liu and M. G. Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear.

[9] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE Transactions on Networking*, 18:490–500, 2010.

[10] A. X. Liu, Y. Zhou, and C. R. Meiners. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proc. 27th Infocom*, April 2008.

[11] R. McGeer and P. Yalagandula. Minimizing rulesets for TCAM implementation. In */Proc IEEE Infocom*, 2009.

[12] C. R. Meiners, A. X. Liu, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proc. 15th ICNP*, pages 266–275, October 2007.

[13] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proc. IEEE ICNP*, 2009.

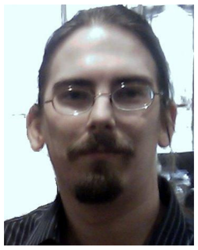
[14] R. Rivest. The md5 message-digest algorithm. *RFC 1321*, 1992.

[15] D. Rovniagin and A. Wool. The geometric efficient matching algorithm for firewalls. Technical report, July 2003. `http://www.eng.tau.ac.il/ yash/ees2003-6.ps`.

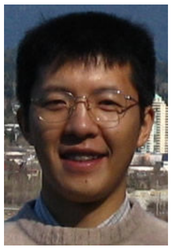
[16] Y.-W. E. Sung, X. Sun, S. G.Rao, G. G. Xie, and D. A. Maltz. Towards systematic design of enterprise networks. In *Proc. ACM CoNEXT*, 2008.

[17] Y.-W. E. Sung, X. Sun, S. G.Rao, G. G. Xie, and D. A. Maltz. Towards systematic design of enterprise networks. *IEEE Transactions on Networking*, to appear, 2010.

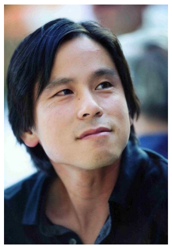
[18] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *Proc. SIGCOMM*, 2010.



Chad R. Meiners Chad Meiners received his Ph.D. in Computer Science at Michigan State University in 2009. He is currently a technical staff at MIT Lincoln Laboratory. His research interests include networking, algorithms, and security.



Alex X. Liu Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the IEEE & IFIP William C. Carter Award in 2004 and an NSF CAREER Award in 2009. His research interests focus on networking, security, and dependable systems.



Eric Torng Eric Torng received his Ph.D. degree in computer science from Stanford University in 1994. He is currently an associate professor and graduate director in the Department of Computer Science and Engineering at Michigan State University. He received an NSF CAREER award in 1997. His research interests include algorithms, scheduling, and networking.

Compressing Network Access Control Lists - Supplement

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

I. Formal Definitions

We now formally define the concepts of fields, packets, ACLs, and the ACL Compression Problem. A *field* F_i is a variable whose domain, denoted $D(F_i)$, is a finite interval of nonnegative integers. Fields can support range, prefix, or ternary field constraints. In most ACL networking devices that implement sequential search of an ACL when processing packets, all the fields support range constraints. However, some existing products such as Linux's ipchains [1] support a mixture of field constraints. For example, ipchains supports prefix field constraints for the source and destination IP address fields. 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), but only some integer intervals correspond to prefix strings. Prefix and ternary field constraints require special treatment, so we use a separate optimization algorithm for fields with prefix and ternary field constraints.

A *packet* over 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.

An ACL 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 S_i is a subset of $D(F_i)$ that conforms to the F_i field constraint format. That is, if F_i supports range field constraints, S_i is a nonempty interval that is a subset of $D(F_i)$. If F_i supports prefix or ternary field constraints, S_i must correspond to a legal prefix or ternary string, respectively. 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.

An *ACL* f over d fields $F_1, \cdots, F_d$ is a sequence of ACL 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 an ACL. To ensure that an ACL is complete, the last rule is typically a *default rule* which is matched by every packet. Table I shows an example ACL over five fields.

Rule	SIP	DIP	SPort	DPort	Proto	Act
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 ACL

Two rules in a ACL overlap if there exists a packet that matches both rules. Two rules in an ACL conflict if they overlap and have different decisions. ACLs typically use a first-match rule conflict 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 ACL f makes for packet p is denoted $f(p)$.

We can think of an ACL f as defining a many-to-one mapping function from Σ to α. Two ACLs 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 ACL f, we use $\{f\}$ to denote the set of ACLs that are semantically equivalent to f.

We define the *ACL Compression Problem* as follows:

Definition 1.1 (ACL Compression Problem): Given an ACL f_1, find an ACL $f_2 \in \{f_1\}$ such that $\forall f \in \{f_1\}$ the condition $|f_2| \leq |f|$ holds.

If all fields support range field constraints, prefix field constraints, or ternary field constraints, we have the range ACL compression problem, the prefix ACL compression problem, and the ternary ACL compression problem, respectively. A key special case is the one-dimensional ACL compression problem when f has a single field.

We illustrate many of our definitions with the example four rule ACL in Table I. The prefix entry $192.168.*.*$ in the Source IP column represents the range of IP addresses from 192.168.0.0 to 192.168.255.255 while the entry $*$ in the Source Port column represents the entire range of port values from 0 to $65,535$. Because every rule has $*$ in the Source Port column, source port number is irrelevant in the example ACL. A packet matches rule 1 of the example ACL if its source IP address begins with 192.168, its destination IP address begins with 1.2.3, its destination port number is in the interval $[4000, 5000]$, and its protocol type is TCP. Any packet matching rule 1 is discarded. Rules 3 and 4 conflict. The first-match rule conflict resolution strategy dictates that the decision of rule 3, acceptance, is the action applied to any packet that matches both rule 3 and rule 4.

II. Related Work

As we noted earlier, to the best of our knowledge, no prior work has considered ACL compression problems with a mixture of field constraints. We now describe prior work on ACL compression problems where all fields have the same field constraint.

Applegate et al. studied the two-dimensional range and prefix ACL compression problems [4]. They proved that the two-dimensional range ACL compression problem with two decisions is NP-hard. They then gave optimal, polynomial time algorithms for restricted two-dimensional range and prefix variants where there are only two decisions and all rules are strip rules, which means that only one field can be a proper subset of its domain. They used these algorithms to create $O(\min(n^{1/3}), OPT^{1/2})$-approximation algorithms for the general two-dimensional range compression problem where n is the number of rules in the input firewall and OPT is the optimal firewall size. For prefix ACL compression, their approximation ratios are multiplied by b^2 where b is the number of bits required to represent a field. 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 range ACL compression problem and state this problem can be solved via dynamic programming with running time $\Theta(Kn^3)$ where K is the number of distinct decisions. The StripePainter problem is a special case of our weighted one-dimensional range ACL compression problem. 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.

Draves et al. proposed an optimal solution for one-dimensional prefix ACL compression in the context of minimizing routing tables in [6]. Subsequently, in the same context of minimizing routing tables, Suri et al. proposed an optimal dynamic programming solution for one-dimensional prefix ACL compression. 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 [13]. Suri et al. noted that their dynamic program would not be optimal for rules with more than 2 dimensions. Applegate et al. developed approximate solutions for two dimensional prefix ACL compression [4]. In [5], Dong et al. proposed four techniques of expanding rules, trimming rules, adding rules, and merging rules that applies to d-dimensional prefix ACL compression. Meiners et al. proposed a systematic approach call TCAM Razor for d-dimensional prefix ACL compression [9].

In [11], McGeer and Yalagandula formulate the ternary ACL compression problem as a two-level logic minimization problem, prove that ternary ACL compression is NP-hard, and give exact and heuristic solutions for ternary ACL compression. However, their heuristic solutions only work for ACLs with two decisions. In [12], Meiners et al. proposed a polynomial-time Bit Weaving heuristic for the d-dimensional ternary ACL compression problem that works for any number of decisions.

Redundancy removal techniques [10], [8] can be used to reduce the number of rules in a given ACL. However, they are only useful when the ACL has redundant rules, and they cannot combine and rewrite ACL rules to produce a smaller ACL. In our ACL Compressor, redundancy removal is an important component of our ACL compression algorithm.

The only other work on ACL optimization that we are aware of has a completely different focus: creating an ACL that minimizes average packet processing time given an input ACL and a traffic model that specifies a probability distribution for packet arrivals [3], [2], [7], [14]. The goal in such work is to construct an ACL that minimizes the expected number of rules examined for any packet assuming that the rules in the ACL are searched in sequential order and that the traffic model accurately specifies the arrival probabilities for all possible packets. In such work, it may be beneficial to have a longer ACL if this decreases the expected number of rules examined. We do not explicitly compare our results to any algorithms that perform this type of ACL optimization as the goals are different and thus any comparison would be an apples to oranges comparison.

III. Proofs

Lemma 3.1: 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 $C(S'(I)) = C(S(I))$.

Proof: If $S(I)$ is canonical, we are done. Without loss of generality, we assume that $f_i \neq \emptyset$ for $1 \leq i \leq m$ because any interval whose core is empty can be removed with no effect on the legality or cost of the schedule. For $1 \leq i \leq m$, let p'_i be the smallest numbered task in f_i, and let q'_i be the largest numbered task in f_i. Then for $1 \leq i \leq m$, interval r'_i of schedule $S'(I)$ is $[p', q']$. It is obvious that $S'(I)$ is canonical, $S(I')$ has at most m intervals, and that $C(S'(I)) = C(S(I))$. ■

Lemma 3.2: 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, $C(S'(I)) = C(S(I))$. □

Lemma 3.3: 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 $C(S'(I)) = C(S(I))$.

Proof: If task 1 is fired in interval r_m of canonical schedule $S(I)$, we are done. Thus, we assume that task 1 is fired in interval r_j of schedule $S(I)$ where $j < m$. Let q_j be the right endpoint of interval r_j in $S(I)$; that is interval $r_j = [1, q_j]$. We create 1-canonical schedule $S'(I)$ as follows. For $1 \leq i \leq j-1$, interval r'_i of schedule $S'(I)$ is identical to interval $r_i = [p_i, q_i]$ of schedule $S(I)$. For $j \leq i \leq m-1$, interval r'_i of schedule $S'(I)$ is interval

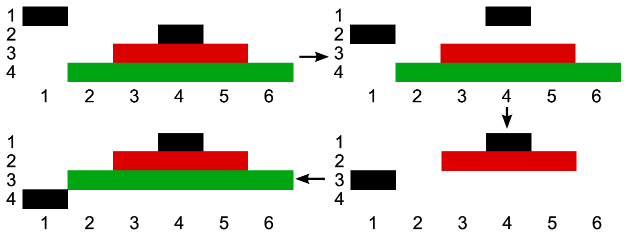


Fig. 1. Repeated application of Lemma 3.2

$r_{i+1} = [p_{i+1}, q_{i+1}]$ of schedule $S(I)$. Finally, interval r'_m of schedule $S'(I)$ is interval $r_j = [1, q_j]$. Because $S(I)$ is a canonical schedule, $p_i \in f_i$ for $1 \le i \le m$. Thus, p_i does not appear in any interval prior to interval r_i. This means that $p_i > q_j$ for $j+1 \le i \le m$; otherwise, p_i would appear in interval r_j which precedes interval r_i. Thus, applying Lemma 3.2 repeatedly, we conclude that $S'(I)$ is a 1-canonical schedule, and $C(S'(I)) = C(S(I))$. This repeated application of Lemma 3.2 is illustrated in Figure 1. ■

Lemma 3.4: 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 $1 \le j \le i-1$, let $r_j = [p_j, q_j]$. It must be the case that either $t < p_j$ or $q_j < t$.

Proof: Suppose the statement is not true. Then for some $j \le i-1$, it is the case that $p_j \le t \le q_j$. This means that $t \in r_j \subseteq \bigcup_{h=1}^{i-1} r_j$. This implies $t \notin f_i$, which is a contradiction since we assumed that $t \in f_i$, and the result follows. ■

Lemma 3.5: 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) Suppose task 1 is not the only task fired in the last interval of $Opt(I(1,n))$. Let 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 $C(1,n) = C(2,t'-1) + C(t',n)$.

Proof: Suppose the last rule of optimal 1-canonical schedule $Opt(I(1,n))$ is $[1,1]$. By Lemma 3.4, all the previous intervals r_i have $p_i > 1$. Thus, these intervals form a schedule for problem $I(2,n)$. These intervals can be replaced by $Opt(I(2,n))$ with no increase in cost and the first observation follows.

We now consider the case where the last rule of optimal 1-canonical schedule $Opt(I(1,n))$ includes task t'. By Lemma 3.4 and the definition of t', any interval $[p_i, q_i]$ prior to the last interval with $p_i < t'$ also has $q_i < t'$. Repeatedly applying Lemma 3.2, we can move all the intervals of $Opt(I(1,n))$ that are in the range $[2, t'-1]$ to the beginning of the schedule without increasing the cost of the resulting schedule. These intervals form a schedule for the problem $I(2,t'-1)$. In addition, these intervals can be replaced by $Opt(I(2,t'-1))$ with no increase in cost.

If we modify the last interval $[p_m, q_m]$ so that $p_m = t'$ instead of 1, the remainder of $Opt(I(1,n))$ forms a schedule for problem $I(t',n)$. These intervals can be replaced by $Opt(I(t',n))$ with no increase in cost. Finally, we modify the last interval r' of $Opt(I(t',n))$ so that its left endpoint is 1 instead of t'. This modification does not increase the cost of the resulting schedule, and the result follows. ■

Theorem 3.6: $C(i,j)$ can be computed by the following recurrence relation.
For $1 \le i \le n$, $C(i,i) = x_{c(i)}$.
For $1 \le i < j \le n$, $C(i,j) = \min(x_{c(i)} + C(i+1,j), \min_{l \in G_{c(i)} \wedge i+2 \le l \le j}(C(i+1,l-1) + C(l,j)))$.

Proof: The base case is correct as there is only one canonical schedule for this instance, $r_1 = [i,i]$.

The correctness of the recursive case is a bit more involved. The first term of the minimization in the recursive case, $x_{c(i)} + C(i+1,j)$, corresponds to the case that i is the only task fired the last interval of $Opt(I(i,j))$. The correctness follows from case 1 of Lemma 3.5.

The second term of the minimization in the recursive case, $C(i+1,l-1) + C(l,j)$, refers to the case where task l is the smallest numbered task that also fires in the last interval of $Opt(I(i,j))$. There are at most $|c(i)| - 1$ possible choices for l because task l has the same color as task i. Also, task l cannot be task $i+1$ because adjacent tasks cannot have the same color in a canonical or serialized ACL. By case 2 of Lemma 3.5, we have $C(i,j) = C(i+1,l-1) + C(l,j)$. ■

IV. Pseudocode for ACL Scheduling Algorithm

Algorithms 1, 2 and 3 show the pseudocode of the ACL scheduling algorithm based on Theorem 3.6. 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 $M[i,j] \ne i$, it means that the smallest numbered task (other than i) that is also fired in the last interval of $Opt(I(i,j))$ is $M[i,j]$. Figure 2 shows the resultant tables and schedule for a sample input.

The function ACLSA-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 ACLSA-Cost is used by the function Print-ACLSA. The function Print-ACLSA(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 $k = \max_{i \in C} |i|$ is the maximum number of tasks in U that exhibit the same color. Note that $\lceil n/z \rceil \le k \le \lceil n/2 \rceil$. 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 $i \ge 1$, we need to compute $C(i,n)$. In addition, for any task $i+1$ where $i \ge 1$, 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 $C(i,j)$.

Algorithm 1: ACL Scheduling Algorithm

```
Data: (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 color $j$; 3) array $group[1..z]$ where $group$ is
    the set of all tasks with color $h$;
Result: (1) an optimal schedule of the $n$ tasks; (2) the cost of
    the optimal schedule;
ACLSA-Cost$(1, n)$ /*compute optimal cost, store
trace info in $M$*/ ;
Print-ACLSA$(1, 1, n)$ /*print an optimal schedule
using array $M$*/ ;
print the optimal cost $C[1, n]$ ;
```

Algorithm 2: ACLSA-Cost(i, j)

```
if $C[i, j] = 0$ then
    $min \leftarrow cost[color[i]]$ + ACLSA-Cost$(i + 1, j)$ ;
    $M[i, j] \leftarrow i$ ;
    for every element $l$ in $group[color[i]]$ do
        if $i + 2 \leq l \leq j$ then
            $tmp \leftarrow$
            ACLSA-Cost$(i + 1, l - 1)$ + ACLSA-Cost$(l, j)$ ;
            if $tmp$ ¡ $min$ then
                $min \leftarrow tmp$ ;
                $M[i, j] \leftarrow l$ ;
    $C[i, j] \leftarrow min$ ;
return $C[i, j]$ ;
```

Algorithm 3: Print-ACLSA(t, i, j)

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

color = [0,1,2,0,2,1]
cost = [1,1,1]
group = [[0,3],[1,5],[2,4]]

C[i,j]	1	2	3	4	5	6
1	0	0	0	0	0	4
2	0	0	2	0	0	3
3	0	0	1	0	2	3
4	0	0	0	1	2	3
5	0	0	0	0	1	2
6	0	0	0	0	0	1

M[i,j]	1	2	3	4	5	6
1	0	0	0	0	0	1
2	0	0	2	0	0	6
3	0	0	3	0	5	5
4	0	0	0	4	4	4
5	0	0	0	0	5	5
6	0	0	0	0	0	6

Schedule:
[4,4]
[3,5]
[2,6]
[1,1]

Fig. 2. Resultant tables and schedule for a sample input

Acknowledgement

The authors would like to thank Yun Zhou for his participation in the early stage of this work. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-0716407, CNS-0916044, and CNS-0845513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] ipchains, http://www.tldp.org/howto/ipchains-howto.html.

[2] S. Acharya, B. N. Mills, M. Abliz, T. Znati, J. Wang, Z. Ge, and A. G. Greenberg. OPTWALL: A hierarchical traffic-aware firewall. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[3] S. Acharya, J. Wang, Z. Ge, T. Znati, and A. Greenberg. Simulation study of firewalls to aid improved performance. In *Proc. IEEE Annual Simulation Symposium*, 2006.

[4] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.

[5] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proc. ACM Sigmetrics*, pages 311–322, 2006.

[6] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proc. IEEE INFOCOM*, pages 88–97, 1999.

[7] E.-S. M. El-Alfy and S. Z. Selim. On optimal firewall rule ordering. In *AICCSA*, pages 819–824, 2007.

[8] A. X. Liu and M. G. Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear.

[9] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE Transactions on Networking*, 18:490–500, 2010.

[10] A. X. Liu, Y. Zhou, and C. R. Meiners. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proc. 27th Infocom*, April 2008.

[11] R. McGeer and P. Yalagandula. Minimizing rulesets for TCAM implementation. In */Proc IEEE Infocom*, 2009.

[12] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proc. IEEE ICNP*, 2009.

[13] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.

[14] W. Wang, H. Chen, J. Chen, and B. Liu. Firewall rule ordering based on statistical model. In *International Conference on Computer Engineering and Technology, 2009.*, pages 185 –188, 2009.

Alex X. Liu Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the IEEE & IFIP William C. Carter Award in 2004 and an NSF CAREER Award in 2009. His research interests focus on networking, security, and dependable systems.

Eric Torng Eric Torng received his Ph.D. degree in computer science from Stanford University in 1994. He is currently an associate professor and graduate director in the Department of Computer Science and Engineering at Michigan State University. He received an NSF CAREER award in 1997. His research interests include algorithms, scheduling, and networking.

Chad R. Meiners Chad Meiners received his B.S. in Computer Science at Truman State University and his M.S. in Computer Science at Michigan State University. He is currently pursuing a Ph.D. in Computer Science with a focus on applying algorithmic techniques to networking and security problems. His research interests include networking, algorithms, and security.