Alpaca: Compact Network Policies with Attribute-Carrying Addresses

Nanxi Kang[1] Ori Rottenstreich[1] Sanjay Rao[2] Jennifer Rexford[1]

[1]Princeton University, {nkang, orir, jrex}@cs.princeton.edu
[2]Purdue University, sanjay@ecn.purdue.edu

ABSTRACT

In enterprise networks, policies (e.g., QoS or security) are often defined based on the categorization of hosts along dimensions such as the organizational role of the host (faculty vs. student), and department (engineering vs. sales). While current best practices (VLANs) help when hosts are categorized along a single dimension, policy may often need to be expressed along multiple orthogonal dimensions. In this paper, we make three contributions. First, we argue for *Attribute-Carrying IPs (ACIPs)*, where the IP address allocation process in enterprises considers attributes of a host along all policy dimensions. ACIPs enable flexible policy specification in a manner that may not otherwise be feasible owing to the limited size of switch rule-tables. Second, we present Alpaca, algorithms for realizing ACIPs under practical constraints of limited-length IP addresses. Our algorithms can be applied to different switch architectures, and we provide bounds on their performance. Third, we demonstrate the importance and viability of ACIPs on data collected from real campus networks.

1. INTRODUCTION

Managing large enterprise networks is challenging. A typical enterprise has many users who belong to different departments (*e.g.,* sales and engineering, or computer science and history), and play different roles (*e.g.,* faculty, staff, administrators, and students). In addition, the network supports diverse end-hosts running different operating systems and offering different services. In response, network administrators want to enforce policies—such as access control and quality of service—that group hosts along multiple different *dimensions*. For instance, one policy may restrict access to a database to all employees in the sales department, while another may offer a higher bandwidth limit to senior managers across all departments, and yet another may restrict access for old hosts running a less secure operating system.

CoNEXT '15 December 01-04, 2015, Heidelberg, Germany
 ISBN 978-1-4503-3412-9/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2716281.2836092

1.1 Enforcing Policies in Today's Enterprises

To enforce policies in today's enterprises, network administrators typically rely on virtual local area networks (VLANs) [1]. A host joining the network is assigned to a VLAN based on its MAC address or the physical port of the access switch. Hosts in the same VLAN are assigned an IP address in the same IP prefix, even if they are not located near each other. Traffic flows freely between hosts in the same VLAN, while traffic between different VLANs traverses an IP router that can enforce policy. Since a host can only belong to a single VLAN, administrators typically assign hosts to VLANs based on a single dimension (*e.g.,* department or role), which has several major limitations:

- The routers interconnecting VLANs need long lists of data-plane "rules" to classify traffic along all relevant "dimensions" of the source and destination hosts.
- No security or QoS policies can be imposed on intra-VLAN traffic, forcing administrators to use a VLAN only to group hosts that should "trust" each other.
- Since VLAN tags are removed from traffic destined to the Internet, the border router must classify all return traffic from the Internet to assign VLAN tags.

For instance, suppose a security policy depends on both user role and department. If VLANs were created based on the user's department, then expressing a policy based on role in a concise fashion is challenging. And, if two users in the different department are close to each other, the traffic follows inefficient paths through intermediate routers to go between VLANs.

The rise of more flexible network switches, with open interfaces to separate control software, enables an attractive alternative. In recent years, switches built with commodity chipsets expose a pipeline of rule-tables that perform match-action processing on packet headers. While the rule tables are relatively small (with small thousands of rules per stage, to limit power and cost), the switches support programmatic

interfaces (*e.g.,* OpenFlow, OF-Config, and OVSDB [2]) that enable new ways of controlling the network by installing the rules in support of higher-level policies. One natural approach, adopted in the early Ethane system [3], directs the first packet of each flow to a central controller, which consults a database—containing all the relevant host attributes and high-level policies—and reactively install rules for forwarding the remaining packets of the flow. However, reactive, fine-grained solutions like Ethane have high overhead and do not scale to large enterprises. FlowTags [4] tags packets and uses the tags to enforce network policies. Yet, this approach, when applied to enterprise networks, requires installing many extra tagging rules at the edge switches to classify hosts along all dimensions. Instead, we need a proactive design that can aggregate hosts along many dimensions, while keeping switch rule tables small.

1.2 Attribute-Carrying IP Addresses

In the current enterprise networks, although IP addresses are assigned based on attributes (*e.g.,* a separate IP prefix per VLAN), it is just on a *single* dimension. We argue that a complete IP address management scheme should consider *all* dimensions and enable compact representation of policies. Our key idea is to assign each host an IP address based on *all* dimensions of the policy—that is, an *attribute-carrying IP address* (ACIP). Our solution, Alpaca (algorithms for policies on attribute-carrying addresses), computes *an efficient address assignment*, based on the policy dimensions and the attributes of individual hosts, and *a compact list of switch rules that* realize a specific network policy.

Alpaca proactively generates a small number of coarse-grained rules in the switches, without using VLANs. Alpaca greatly simplifies enterprise management by (i) enabling administrators to specify policies on many orthogonal dimensions while achieving an order of magnitude reduction in rules; (ii) allowing policy to be correctly maintained even when a host connects to the network at a new location; and (iii) simplifying federated management, where different teams manage different parts of the enterprise network.

While Alpaca has many potential advantages, several practical issues must be considered:

Limited IP address space: Since most enterprises have limited IP address space, a naive ACIP assignment can easily exhaust all the available bits.

Heterogenous group sizes: Some combinations of policy attributes are much more common than others. As such, an efficient address assignment cannot simply devote a portion of the bits to representing group size.

Minimizing churn: It is important to support changes in the policy attributes associated with a host (*e.g.,* due to a user moving to a different role or department), or changes in the set of attributes themselves (*e.g.,* due to the creation of a new department).

Multi-stage switch pipelines: Rather than assuming switches have a single rule table (as in OpenFlow 1.0 [5]), a practical solution should capitalize on the multi-stage pipelines in modern switch chipsets [6–8].

We propose a family of algorithms to generate ACIPs and the associated rule tables, starting with a simple strawman that devotes a separate set of address bits to each policy dimension. This solution minimizes the number of rules but consumes too much IP address space, making it infeasible in most practical settings. We then propose two other algorithms that can keep the number of rules small, while respecting constraints on the number of bits in the IP address space. Our algorithms optimize based on the characteristics of modern switch chipsets. Conventionally, switch chip-sets have a single TCAM rule-table that supports a few thousands of wildcard rules matching on multiple header fields [5, 9, 10]. More recently, we are seeing the emergence of switch chip-sets with a pipeline of multiple tables, where each table could be a TCAM or a larger SRAM that supports prefix matching on source IP or destination IP [6–8]. As such, our first algorithm generates ACIPs that enable policies to be expressed by solely IP *prefixes*, useful for rule tables that support IP prefix matching. Our second algorithm targets both single-table switches and multi-table switches, generating rules that perform arbitrary *wildcard* matching on IP addresses, in exchange for a reduction in the number of rules. Together, these algorithms can capitalize on the unique capabilities of a variety of commodity switch architectures.

In the next section, we present a case study of multiple campus networks, to underscore the need for policies along multiple dimensions. Section 3 introduces ACIPs and formulates the optimization problem Alpaca must solve, followed by Section 4 that presents our two algorithms. Using access control data from two large campuses, the experiments in Section 5 show that Alpaca can reduce the number of ACL rules on existing networks by $60\% - 68\%$ for switches with multiple tables and by $40\% - 96\%$ for switches with a single table, while requiring only 1 more bit of the IP address space than needed to represent the number of hosts in the network. Further, Alpaca can support futuristic scenarios with policies based on multiple dimensions, while requiring an order of magnitude fewer rules than VLAN-based configurations optimized for a single dimension. Section 6 presents related work, and Section 7 concludes the paper.

2. CASE STUDY: DIVERSE ENTERPRISE POLICIES

In this section, we present a case study of 25 enterprise networks, to identify the challenges in representing sophisticated policies, and the implications for Alpaca. Specifically:

• We present a qualitative analysis of the security and quality-of-service policies employed by 22 universities, plus one individual department that runs its network separately from the campus IT group. The analysis indicates that networks must often apply policy along several logical *dimensions*, with multiple *attributes* as possible categories in each dimension. However, the analysis also points to policies that are desirable but difficult to realize in practice.

• We analyze router configuration data from two other large campuses. The analysis provides further confirmation that there is significant commonality in policy across hosts, but

also points to how an inefficient assignment of IP addresses can lead to an unnecessary "blow-up" in rule-table size.

• We study host-registration data for one department-level network, to understand the dimensions of network policies and the number and size of host "groups" with these attributes. The analysis has important implications for the design of Alpaca.

2.1 Policies on Multiple Dimensions

Many universities make descriptions of their high-level network policies available online (see http://tinyurl.com/pwvlygx for a summary). Most schools classify hosts by the owner's *role* (*e.g.,* faculty, students, staff, visitors), *department*, *residence* (*e.g.,* a particular dormitory), and *usage* (*e.g.,* research vs. education). In addition, many schools associate each host with a *security level* (with around ten different integer values) and whether the host is currently viewed as *compromised* (with a "yes" or "no" value). Some schools also classify hosts by *bandwidth quota* and *past usage*, to inform rate-limiting policies, and by whether they offer *core services* (*e.g.,* email and web servers). Based on these documents, and our discussions with the administrators of the computer-science department's network of one university (University A), we learned about the following example policies.

Security: Schools use the security level to limit which external users can access a given host (and in what way). For example, hosts at the lowest security level might be blocked from receiving unsolicited traffic from external hosts; that is, these hosts cannot run public services. Other security levels correspond to different restrictions on which transport port numbers are allowed (*e.g.,* port 80 for HTTP, but not port 22 for SSH or 109, 110, and 195 for POP3). Some schools allow individual departments to state their own access-control lists, applicable only to hosts with IP addresses in that department's address block. When administrators identify an internal host as compromised, they change the *compromised* attribute and significantly restrict the host's access to network services. In addition, users in the *visitor* category typically have access to a limited set of services on the campus (*e.g.,* no access to the printers or campus email servers and compute clusters). One school restricts access to compute clusters in dormitories to the students residing in that particular dorm.

Quality of Service: Some universities impose a different default bandwidth quota based on the host's *role*, but allow students and postdocs to purchase a higher quota. Some universities employ rate-limiting policies that depend on the user's bandwidth usage on previous days (*e.g.,* users whose bandwidth usage exceeded a certain level were rate-limited to a lower level). Hosts offering core services are excluded from bandwidth usage calculations for both the users responsible for the service machines and the owners of the access machines, to avoid that traffic counting against their usage caps. Also, some schools offer higher quality-of-service for hosts assigned for educational use (*e.g.,* for streaming high-quality media in a classroom). The administrators of University A also expressed a desire to perform server load balancing for internal Web services based on user role, to prevent heavy load from one group of users from compromising the performance of other users.

Administrator "wish-lists": Our discussions with the administrators of University A also indicated that there were many additional policies that they would ideally like to implement in the network, but did not do so since they were hard to realize in practice. University A assigns hosts to VLANs based on role (*e.g.,* faculty, staff, and students), for traffic isolation, to prevent packet sniffing and excessive broadcast traffic. The administrators would like to apply access-control policies based on device usage, device ownership, and OS, but do not do so today, since this would require exhaustive enumeration of IP addresses in the switch configuration. Likewise, the administrators expressed a desire to apply flexible QoS policies based on (i) the way a device is used (*e.g.,* research vs. infrastructure machines) and (ii) whether the host is owned by the department (as opposed to a personal "Bring Your Own Device" host).

Our discussions also revealed additional challenges with federated network management. The campus network assigns IP addresses in blocks based on location (*e.g.,* building). This raises challenges in applying security policies that restrict access to users affiliated with computer science department. Currently, the policy works correctly for hosts that are physically in the CS building, since these hosts are assigned a prefix from the CS subnet. However, when a CS user works in another building (common for faculty with dual appointments in other departments), the host receives a different IP address outside the CS subnet and the user is no longer able to access the CS resources. While the administrators could conceivably update network configurations dynamically to reflect the IP addresses that should have access, the management complexity is a deterrent. More generally, federated management would be much easier if network administrators had concise ways to represent security and QoS policies based on host attributes.

2.2 Potential for Concise Rules with ACIPs

Existing techniques for assigning IP addresses to hosts can lead to a large numbers of rules in the switches. To quantify this problem, we analyzed the access-control policies in router configuration files for two university networks (University B and University C). Prior work shows that hosts in a network may be partitioned into a small number of policy units [11]—*i.e.,* a set of hosts that have identical reachability policies in terms of their communication with the rest of the network. Though the number of policy units is small, the number of ACL rules to express policy could still be as large as the square of the number of policy units. Thus, we go beyond [11], and not only identify policy units, but also calculate the number of rules required if ACLs were written in terms of policy units rather than the existing IP assignment.

Specifically, we consider two hosts as belonging to the same source policy unit (SPU) if and only if packets sent by these hosts to all destinations are treated identically in every ACL across all routers. Likewise, we consider two hosts as

Dimensions	#Attributes	Example Attributes
Role	8	Faculty, Students
Security Level	16	1, 2, ..., 16
Status	6	In service, In testing
Location	7	–
Usage	3	Research, Infrastructure, ...
CS_owned	2	Yes, No
OS	5	MacOS, Windows, Linux, ...

Table 1: Host data for CS department (University A)

belonging to the same destination policy unit (DPU) if and only if packets from all sources to these hosts are treated identically in every ACL across all routers. We then compute the total number of rules needed to represent each ACL if it were more compactly expressed in terms of its source and destination policy units. Our results show that the number of ACL rules required is much smaller than the product of the source and destination policy units, and indicates that a smart ACIP allocation, which classifies hosts into their SPUs and DPUs efficiently, can potentially offer significant reduction ranging from 48% to 98% for ACLs of the two universities (Section 5).

2.3 Diverse Attributes and Group Sizes

To better understand the attributes of hosts, we collected data about the 1491 registered hosts of the CS department of one university (University A). Each host is associated with seven dimensions of information, as summarized in Table 1. In this network, (i) hosts are assigned to separate VLANs based on role and (ii) role, security level, and status are considered in access-control policy.

Given the number of attributes in each dimension, hosts could theoretically have $161,280$ (*i.e.,* $8 \times 16 \times 6 \times 7 \times 3 \times 2 \times 5$) combinations of attributes. In practice, only 287 unique combinations exist; for example, no visitor has a CS_owned host. In addition, some combinations are much more popular than others. One group of hosts—belonging to one Linux-based compute cluster—has 109 members (more than 7% of all hosts). The large number of attributes and the diversity of group sizes have important implications for address assignment in Alpaca, which encodes host attributes in the IP address to enable more compact representations of policies.

Consider a naive address allocation scheme that performs *BitSegmentation*, by (1) concatenating a binary encoding of the host attributes along each dimension, where dimension i with a set of attributes D_i requires $\lceil \log \|D_i\| \rceil$ bits, and (2) using the remaining bits to distinguish hosts with the same attributes along all dimensions, requiring $\lceil \log X \rceil$ bits, where X is the size of the largest group.

The resulting encoding would enable very compact rules in the switches, using wildcard patterns to match on any attribute. However, this solution is impractical, even for this small network. Representing the seven dimensions would require 19 bits, and representing the largest group (with 109 members) would requires 7 bits, for a total of 26 bits—a highly inefficient allocation of IP address space.

3. ALPACA OVERVIEW

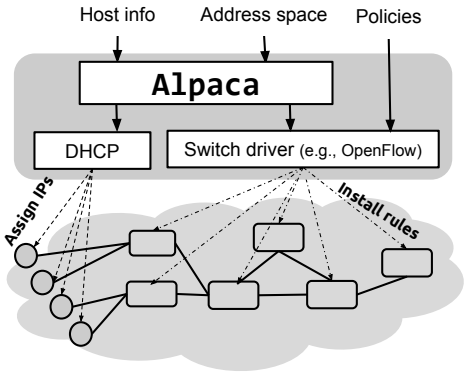


Figure 1: Use Alpaca in a network.

3.1 ACIP allocation with Alpaca

We present Alpaca, a system that embeds host attributes in IP addresses to enable compact policies. Figure 1 shows an overview of the system. Alpaca takes as inputs from the network operator the set of policy dimensions, and a database that lists the attributes associated with each host. It instructs DHCP servers how to assign IP addresses to hosts based on the results of Alpaca algorithm. If a host needs to be assigned a new address (*e.g.,* it moves to a new location), the original attributes along with the new location are used in determining the new IP address. Meanwhile, the switch driver instantiates the network policies, which are defined on attributes, by installing match-action processing rules on switches. Alpaca coordinates with the switch driver such that rules can correctly classify the IP addresses of hosts to the corresponding attributes along different dimensions.

While ACIP allocation enables operators to express policies defined on multiple dimensions with switch rules, Alpaca must adapt to the following constraints:

Switch rule-table sizes: Switches impose hard constraints on the maximum number of rules to be installed. Therefore, Alpaca should optimize the classification of hosts for different switch architectures so that the attribute-based policies can be compactly expressed with switch rules that stay within the rule-table sizes.

Adddress space: IPv4 is widely deployed and likely to remain for the foreseeable future. Many enterprises have 16 or fewer bits for public IPv4 address space, or up to 24 bits for private IPv4 address space (*i.e.,* 10.0.0.0/8). The limited address space calls for an efficient ACIP allocation that encodes attributes without wasting IPs. Though the address space constraint may be relaxed if IPv6 is fully deployed in a network, the allocation still needs to take the size of address space into account to correctly represent attributes.

Dynamics: Attributes of a host along any dimension may change. Alpaca must be able to handle changes in the attributes of a host while ensuring that only the IP address of that host changes, and that IP addresses of other hosts are not impacted. Alpaca should also handle (1) addition or removal of new attributes in existing dimensions without impacting the existing IP address allocation, and (2) addition or removal of dimensions, which is however relatively rare and may require significant changes.

3.2 Problem Formulation

Given an IP address space of W bits, a set U of N hosts ($N \leq 2^W$), a set of M dimensions and the attributes for the hosts along each dimension, an Alpaca algorithm computes an assignment of IPs to individual hosts and M sets of classification rules. Each rule-set corresponds to a dimension. A classification rule consists of an address pattern p and an attribute a; rule (p,a) means that any host with IP matching p has attribute a. The classification rules in the same rule-set have disjoint address patterns. An example is shown in Figure 2(a)(b).

The classification rule-sets must be optimized. Consider a multi-table switch architecture. We install the rule-sets to classify source or destination (or both) to the corresponding attributes for the use by following tables. Figure 2(c) shows an example, where the first two tables decide the attributes of "department"and "role" of the source by appending values to metadata, the last table decides to permit or deny the source based on attributes.

Our primary optimization goal is to *minimize the total sizes of M rule-sets,* i.e., *the number of classification rules that decide the attributes of all hosts along all dimensions in a multi-table switch architecture*. We also extend our algorithms (Section 4.2) to optimize the rules for a single-table switch architecture, where the installed rule-set is the *product* of all rule-sets in a multi-table architecture (*e.g.,* the three tables in the example) and the total number of installed rules heavily depends on how frequent attributes (or the combination of attributes) are used in the network policies.

3.3 Overview of Alpaca algorithms

Alpaca consists of a series of algorithms targeted at different scenarios.

The Prefix algorithm computes prefix classification rules with a proven approximation ratio to the optimal case. It uses address space efficiently, requiring exactly $\lceil log_2 N \rceil$ bits. It is specially designed for multi-table switches with SRAMs that support a large number of prefix rules.

The Wildcard algorithm computes wildcard classification rules. It uses a small address space and can be applied to single-table and multi-table switch architectures.

Both prefix and wildcard algorithms by themselves do not handle dynamics and host attribute changes.

The Slack algorithm refines the prefix and wildcard algorithms by taking advantage of one more bit in the address space for an allocation that works well under dynamics in host attributes.

4. ALPACA ALGORITHMS

In this section we describe Alpaca algorithms for assigning IPs to individual hosts. The first algorithm is designed for switch chipsets that allow prefix rules while the second solution applies for more general chipsets with tables allowing wildcard rules.

4.1 Prefix Solution

This section presents an address allocation algorithm that

Input			Output
Hosts	**Dept**	**Role**	**Addresses**
$h_1 - h_5$	CS	Faculty	0000, 0001, 0010, 0011 0111
$h_6 - h_7$	CS	Students	1010, 1011
$h_8 - h_{10}$	EE	Faculty	0100, 0101 0110
$h_{11} - h_{16}$	EE	Students	1000, 1001 1100, 1101 1110, 1111

(a) Address assignment.

Output	
Dept	
p	**a**
0111	CS
101*	CS
00**	CS
0110	EE
010*	EE
100*	EE
11**	EE
Role	
p	**a**
0***	Faculty
1***	Students

(b) Prefix rules.

Dept	
Match src	**Action** append
0111	1
101*	1
00**	1
0110	2
010*	2
100*	2
11**	2

→

Role	
Match src	**Action** append
0***	1
1***	2

→

Match metadata	**Action**
1,2	permit
2,1	permit
*	deny

(c) Rules on multiple tables to permit CS Students and EE Faculty

Figure 2: Example allocation: $W = 4, N = 16, M = 2$**.**

optimizes the number of *prefix rules* to represent attributes along multiple dimensions. It targets at multi-table switch architectures with IP prefix matching tables. We first introduce the notation, then discuss the optimal solution for a single dimension and the generalization to multiple dimensions.

We use the following notations when illustrating the algorithms. Let α be a dimension and $A = \{a_1, a_2, ...\}$ be the set of associated attributes. We view α as a function that maps every host to an attribute, *i.e.,* $\alpha(x) \in A$ is the attribute of host x. Let T be an ACIP allocation. We use $C_\alpha(T)$ to denote the minimum number of rules to represent dimension α. Likewise, for a set of dimensions $\mathscr{D} = \{\alpha, \beta, \dots\}$, $C_\mathscr{D}(T)$ represents the total number of rules to present all the dimensions in $\mathscr{D}$ using the allocation T, *i.e.,* $C_\mathscr{D}(T) = \sum_{\phi \in \mathscr{D}} C_\phi(T)$. We define $opt_\alpha = \min_T C_\alpha(T)$ and $opt_\mathscr{D} = \min_T C_\mathscr{D}(T)$ to be the minimum number of rules to represent dimension α and the set of dimensions, respectively.

A single dimension: We start with the simplest case: assigning addresses to represent exactly *one dimension*. Consider the dimension *Role*: each attribute of Role, such as Faculty, Students or Visitors, should have its own set of rules for the hosts. As a prefix pattern matches a power-of-two number of hosts (*e.g.,* 0*** stands for 8 hosts and 111* stands for 2 hosts), one attribute might need several rules. The rules of different attributes do not overlap, i.e., matches are disjoint. Below, we describe a simple algorithm that finds an optimal address allocation to represent one dimension.

Given the dimension function $\alpha : U \to A$, the algorithm returns the address allocation function $T : U \to \{0,1\}^W$. The core idea is to treat the number of hosts for each attribute as the sum of power-of-twos and use a prefix rule for each power-of-two. Specifically, we first partition hosts into $|A|$ sets based on their attributes. Let n_i be the number of hosts with attribute a_i ($i = 1, , |A|$) and $bin(n_i)$ be the binary repre-

sentation of n_i. We represent n_i as the sum of *distinct* power-of-twos based on $bin(n_i)$. For example, $bin(14) = b1110$ and 14 is represented as $8+4+2$. Next, for each attribute a_i, we further partition the set of hosts into subsets according to sum representation of n_i. For example, if $n_i = 14$, we will partition the set of hosts into 3 subsets with size $2, 4$ and 8 respectively. The last step is to sort all the subsets for different attributes in non-increasing sizes. Hosts are ordered based on the subsets they belong to. The resulting address allocation gives the k-th host address $bin(k)$. The pseudo-code of the algorithm is shown in Algorithm 1.

Algorithm 1 Optimal algorithm for a single dimension α

```
for all attribute a_i ∈ A do
  U_i = {x ∈ U, where α(x) = a_i}
  n_i = |U_i|
  for j = 0 to W do
    if 2^j & n_i > 0 then
      Create a subset of 2^j hosts selected from U_i
      Remove these 2^j hosts from U_i
    end if
  end for
end for
Sort all subsets by their sizes in a non-increasing order
Sort hosts according to the order of subsets
Assign k-th host an address of bin(k), k = 0, 1, ..., N
return the address allocation of hosts
```

Let $\|bin(n_i)\|$ be the number of 1s in the binary representation of n_i, *e.g.,* $\|bin(14)\| = 3$. The above algorithm constructs $\sum_{i=1}^{|A|} \|bin(n_i)\|$ subsets. We show that the resulting address allocation needs exactly $\sum_{i=1}^{|A|} \|bin(n_i)\|$ rules for α. In other words, each subset takes a single rule to represent. To prove it, we consider a subset of size 2^i. Since subsets are sorted in non-increasing sizes, any previous subset must have a size of 2^j for some $j \geq i$. Hence, the sum of the sizes of all previous subsets are multiples of 2^i. This guarantees that all 2^i hosts in the current set can be all represented by a single prefix rule with exactly i wildcards. To prove the optimality of the algorithm, we further show that an attribute shared by n_i hosts requires at least $\|bin(n_i)\|$ rules.

PROPERTY 1. *For a dimension α, let $n_i = |\{x \in U | \alpha(x) = a_i\}|$ be the number of hosts that have to be mapped to an attribute a_i, $i = 1, 2, \cdots, |A|$. The minimal number of rules that can represent α in any ACIP allocation satisfies $opt_\alpha = \min_T C_\alpha(T) = \sum_{i=1}^{|A|} \|bin(n_i)\|$.*

Two dimensions: Let $\alpha : U \to A, \beta : U \to B$ be the dimensions under consideration, where $B = \{b_1, \cdots, b_{|B|}\}$ is the set of attributes in the second dimension. We observe a clear tradeoff between shortening the representation of these two dimensions. While we could choose the address allocation to be the optimal for α and use the minimal number of rules for α, we may have to use many more rules to represent β. Since the address allocation is shared by both dimensions, in most cases we cannot find an allocation that favors both dimensions. Below, we show the property on the relationship between the optimal allocation for two dimensions and the optimal allocation for each dimension. We recall that the optimal allocation minimizes the sum of the number of rules to represent each dimension.

PROPERTY 2. *The optimal allocation for the dimensions α, β satisfies $opt_{\alpha,\beta} \geq opt_\alpha + opt_\beta$. An equality $opt_{\alpha,\beta} = opt_\alpha + opt_\beta$ is achieved if there exists an allocation that is optimal for the dimension α as well as for the dimension β.*

To obtain an upper bound on the optimal number of rules, we construct a special allocation below. Let γ be a new dimension, which is the product of α and β. The corresponding set of attributes $C = \{c_1, \ldots, c_{|A|\cdot|B|}\}$. For a host $x \in U$, if $\alpha(x) = a_i, \beta(x) = b_j$ then $\gamma(x) = c_{(i-1)\cdot|B|+j}$. The dimension γ, denoted by $\gamma = \alpha \times \beta$, has the property that $\gamma(x)$ determines the attributes $\alpha(x), \beta(x)$ for the same host x. Consider the representation of γ under some allocation T with $C_\gamma(T)$ rules. We can obtain a representation of the dimension α (or of β) with the same number of rules by only modifying the attribute of each rule, *i.e.,* replacing attribute in C with the corresponding attribute in A (respectively in B). Therefore, $C_\alpha(T) \leq C_\gamma(T)$, $C_\beta(T) \leq C_\gamma(T)$. Finally,

$$C_\alpha(T) + C_\beta(T) = C_{\alpha,\beta}(T) \leq 2C_\gamma(T) \tag{1}$$

We remark here that these are not necessarily the minimum representations of α, β with the address allocation T: we can further compress rules for each dimension.

Next we show that an optimal allocation for γ is *a 2-approximation* of the optimal allocation for α, β, *i.e.,* the number of rules it generates is at most twice of the minimum. Consider some allocation T, it must satisfy

$$C_\gamma(T) \leq C_\alpha(T) + C_\beta(T) = C_{\alpha,\beta}(T) \tag{2}$$

This is because a group of hosts that cannot be represented using a single rule in γ must have different attributes in at least one of α and β, thus requiring a minimum of two rules to represent in that dimension. Let T_γ be the optimal solution of γ and $T_{\alpha,\beta}$ be the optimal solution for α, β together. Substituting T with $T_{\alpha,\beta}$ in Equation 1 and 2, we obtain

$$C_\gamma(T_{\alpha,\beta}) \leq C_\alpha(T_{\alpha,\beta}) + C_\beta(T_{\alpha,\beta}) = C_{\alpha,\beta}(T_{\alpha,\beta}) \leq 2C_\gamma(T_{\alpha,\beta})$$

Meanwhile, since $C_{\alpha,\beta}(T_{\alpha,\beta}) = opt_{\alpha,\beta}$ and $C_\gamma(T_{\alpha,\beta}) \leq C_\gamma(T_\gamma) = opt_\gamma$, we conclude the following property:

PROPERTY 3. *Let T_γ be an optimal allocation for $\gamma = \alpha \times \beta$. Then, $C_{\alpha,\beta}(T_\gamma) \leq 2 \cdot opt_{\alpha,\beta}$.*

To summarize, for two given dimensions α, β, we calculate $\gamma = \alpha \times \beta$ and find its optimal allocation T_γ by the algorithm for a single dimension. We then use this allocation to represent each of the dimensions α, β.

General number of dimensions: We can generalize the above solution for two dimensions to handle a set $\mathscr{D}$ of an arbitrary number of dimensions $M = |\mathscr{D}|$. Similar to computing γ for the two dimensions, we introduce a dimension $\Pi_\mathscr{D}$, whose attributes for a host $x \in U$ is a vector of length $|\mathscr{D}|$ with the attributes of all dimensions in $\mathscr{D}$ for that

	CS	EE
Faculty	5	3
Student	2	6

(a) Group sizes

⇒

Id	(Dept, Role, Value)
1	CS, F, 1
2	EE, F ,1
3	EE, S, 2
4	CS, S, 2
5	EE, F, 2
6	EE, S, 4
7	CS, F, 4

(b) Initial nodes

Figure 3: Create nodes from input of Figure 2(a).

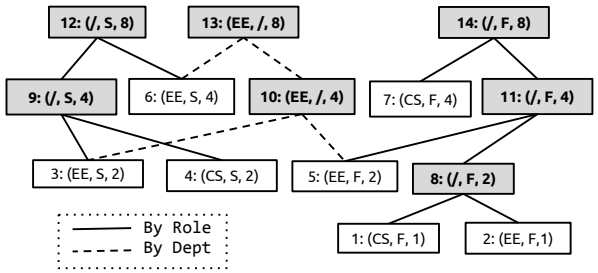


Figure 4: The compression graph: a node has an id, attributes and a value. Colored nodes are super-nodes.

host. We show that the optimal allocation for $\Pi_{\mathscr{D}}$ is an M-approximation to the optimal allocation for $\mathscr{D}$. We omit the proof for brevity.

4.2 Wildcard Solution

In this section, we present an algorithm that generates wildcard rules by optimizing the output of the prefix solution. To illustrate the algorithm, we use the same example in Figure 2. For clarity, we use *host group* to refer the set of hosts with the same attributes along all dimensions; we use *rules* and *patterns* interchangeably to refer the compact ACIP representation of host groups and attributes.

We revisit our example. The prefix solution uses 1 pattern for the CS Students group and 2 patterns for each of the rest groups, because it views the size of a host group as the sum of powers-of-twos (*e.g.,* $5 = 1+4$), each of which corresponds to a prefix pattern. Hence, it uses $2+2=4$ prefix rules to represent Faculty attribute. But if we assign {0111, 010*} to EE Faculty and {0110, 11**} to CS Faculty, then we can compress these patterns to a single pattern *1** to represent Faculty attribute. The key observation is that *if two host groups share common attribute(s), it is beneficial to assign them similar patterns that can be compressed to reduce the number of rules for the common attribute*.

Potential compression. Our first task is to find out all the potential compression of patterns among host groups. Starting with the output of prefix solution, which uses the sum of power-of-two terms to denote the size of a host group, we map every power-of-two term to a *node*. The node saves the value of the term and copies the attributes of the host group. For example, we can create two nodes for the CS Faculty group: (CS, Faculty, 1) and (CS, Faculty, 4), as the group size $5 = 1+4$. Figure 3(b) shows the full list of nodes.

Two nodes can be compressed if their values are equal and they share some common attribute(s). The result of compression is a new node that (1) has a value equal the sum of the values of the two nodes, (2) "inherits" the shared attributes and (3) has $\emptyset$ attributes for other dimensions. For example, (CS, Faculty, 1) and (EE, Faculty, 1) can be compressed into a new node ($\emptyset$, Faculty, 2). We call the new node a *super-node* and the two original nodes *sub-nodes*. The compression suggests that we could use the super-node instead of listing two sub-nodes individually to represent their common attributes.

A super-node can be compressed with other nodes, as long as they share the same attributes (except $\emptyset$). But a node cannot be compressed twice on the same dimension, *i.e.,* once

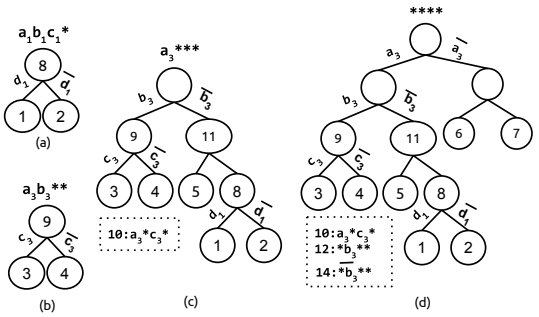


Figure 5: Flip bits to compress nodes

(CS, Faculty, 1) and (EE, Faculty, 1) are compressed, neither of them can be compressed with other nodes that have Faculty attribute. We repeat the compression until no new super-nodes can be produced. We plot graphs to denote the compression relationship by creating edges from sub-nodes to their super-nodes (Figure 4).

In the graph, a node with value 2^k can be assigned a wildcard pattern with exactly k wildcards, which represent 2^k hosts. As we work on the output of the prefix solution, the initial nodes (Figure 3(b)) should be assigned prefix patterns.

Compressible patterns. Two patterns are compressible if they negate at exactly one bit, *e.g.,* *00* and *10* are compressed into **0*. When two sub-nodes (of a super-node) are assigned compressible patterns (*e.g.,* *00* and *10*), the resulting pattern (*e.g.,* **0*) can be assigned to the super-node to achieve a reduction of one rule in representing the common attribute, where we can use the compressed pattern instead of listing two patterns independently. In the example, we can use the pattern for ($\emptyset$, Faculty, 2) to represent CS attribute rather than two patterns for (CS, Faculty, 1) and (EE, Faculty, 1). Therefore, our goal is to assign patterns to nodes to *maximize the number of pairs of sub-nodes with compressible patterns*, *i.e.,* the total reduction in the number of rules to represent attributes.

Key idea: flip one bit. Let $a_ib_ic_id_i$ be the pattern assigned to the i-th node, where $a_i, b_i, c_i, d_i \in \{0, 1, *\}$. Consider Node 1 (CS, Faculty, 1) and Node 2 (EE, Faculty, 1). $a_1b_1c_1d_1$ and $a_2b_2c_2d_2$ are compressible if they negate at one bit, *i.e.,* $a_1b_1c_1d_1 = \overline{a_2}b_2c_2d_2$, or $a_2\overline{b_2}c_2d_2$, or $a_2b_2\overline{c_2}d_2$, or $a_2b_2c_2\overline{d_2}$. Our key idea to enable compression is *to choose a bit* (e.g., *a, b, c or d) to flip*. If we flip d, the compressed pattern $a_1b_1c_1*$ (or $a_2b_2c_2*$) can be assigned to the super-node ($\emptyset$, Faculty, 2), *i.e.,* Node 8. As a result, $a_8b_8c_8d_8 = a_1b_1c_1*$. We plot the equality in Figure 5(a).

Dept	
p	**a**
0110	CS
001*	CS
11**	CS
0111	EE
10**	EE
0*0*	EE

Role	
p	**a**
*1**	Faculty
*0**	Students

Figure 6: Wildcard rule-sets.

Similarly, we can compress patterns of Node 3 and Node 4 by flipping c ($d_3 = d_4 = *$) as shown in Figure 5(b). We can further compress (1) Node 5 and Node 3 by flipping b (as c_3 is flipped before) and (2) Node 5 and Node 8 by flipping c (shown in Figure 5(c)). However, we are unable to compress Node 6 ($a_6b_6c_6d_6 = \overline{a_3}b_3**$) and Node 10 ($a_{10}b_{10}c_{10}d_{10} = a_3 * c_3*$), because their patterns do not match. We finish the procedure by compressing Node 6 and Node 9, Node 7 and Node 10 (Figure 5(e)). To translate the results to patterns, we set all variables, *i.e.,* a_3, b_3, c_3, d_1, to 0.

To summarize, with the idea of bit flipping, we construct *equality and inequality* between the bits of patterns, *i.e.,* a, b, c, d, assigned to nodes. For the patterns of each pair of sub-nodes, if the equality (or inequality) is not determined before, we choose the last possible bit to flip. When there is no such a bit (*i.e.,* all the patterns negating at one bit are already used), then we choose to flip more than one bit until the resulting patterns do not overlap with any used ones. After checking all the pairs of sub-nodes, we obtain the full equality and inequality. The final step is to set all free bit variables to 0.

We can represent each attribute with rules given the pattern assignment (Figure 6). For example, to represent Student, we can use *0** for super-node ($\emptyset$, Student, 8), *i.e.,* Node 12. Similarly, we use 10**, 0*0* and 0111 to represent EE. In total, we need $3+3+1+1 = 8$ rules to represent all the attributes, whereas prefix solution needs 9 rules (Figure 2(b)).

In what follows, we discuss the order to process pairs of sub-nodes (or super-nodes) to achieve the optimization goal, extend the solution to generate prefix rules and how to handle weighted attributes to support the single-table switch architecture (Section 3).

Processing order of sub-nodes. The order we use to process sub-nodes matters, as the compression of one pair of sub-nodes may restrict the compression of another (due to the equality and inequality between bits). The algorithm calculates the *order values* for super-node n as the total number of super-nodes in the tree rooted at n in the graph. For example, the tree rooted at ($\emptyset$, S, 4) only contains one super-node (*i.e.,* itself); the tree rooted at ($\emptyset$, F, 8) contains three. Super-nodes are sorted according to their order values and examined one by one. When examining one super-node, we process all the pairs of sub-nodes in its tree. If the compression failed for one pair (*i.e.,* we cannot find a bit to flip), we roll back all the previous compressions of sub-node pairs in the tree and continue to examine the next super-node in the sorted list; if the compressions of all pairs of sub-nodes succeed, we remove these sub-nodes from the trees of other super-nodes, re-calculate order values of the affected super-nodes and sort again.

Extension: minimize prefix rules. Although the above algorithm is designed to generate wildcard rules, with a simple trick we could use it to minimize prefix rules as well. The key observation is that wildcard patterns are produced when we choose to flip *non-trailing bits* to compress patterns. For example, when compressing nodes $a_1b_1c_1*$ and $a_2b_2c_2*$, if we choose c_1 then the result a_1b_1** is a prefix pattern, otherwise the pattern (*e.g.,* $a_1 * c_1*$ or $*b_1c_1*$) is a wildcard pattern. Hence, to generate prefix rules, we only need to constrain the algorithm to flip the last non-wildcard bit (*e.g.,* c_1 in the pattern $a_1b_1c_1*$).

Extension: weighted attributes. The basic algorithm minimizes the total number of rules to represent all the attributes. But attributes may not be equally important in the single-table switch architecture. For example, Students may be used more often than Faculty. It is preferred to use fewer rules to represent Students despite the increased number of rules to represent Faculty. We can extend the wildcard algorithm to minimize the total number of rules when attributes are weighted. The intuition is to change how the order values of super-nodes are calculated. We introduce the weight of a super-node as the sum of weights of its non-$\emptyset$ attributes. To calculate the order value of a super-node, instead of counting the number of super-nodes in its tree, we sum up the weights of the super-nodes in the tree. The sorting and compression procedure remains the same. We can also handle weighted combinations of attributes (*e.g.,* CS Faculty) with a similar modification to the calculation of weights and order values of super-nodes.

4.3 Handle Changes in Host Attributes

Our algorithms proposed so far support address allocation given the attributes of each host. In practice, attributes of a host may change over time (*e.g.,* the department of the corresponding user might change), or new attributes may be added (*e.g.,* a new department may be created). In handling changes, a key consideration is ensuring that only the IP addresses of impacted hosts are modified to the extent possible.

We employ two techniques to handle changes in attributes. First, to handle growth in the number of hosts that have a certain attribute, we introduce *slack,* and budget for more hosts than actually exist. A straight-forward solution is to provision for a growth in the number of hosts corresponding to a given attribute by a fixed percentage (*e.g.,* 10%), though information about projected trends could be used when available. For example, a university can estimate the number of hosts in the coming semester based on the number of newly admitted students.

Second, to handle growth in the number of attributes along each dimension, we introduce a *"ghost" attribute* for each dimension (an additional attributes with which no host is currently associated) and decide the group sizes for combinations of ghost and real attributes (*e.g.,* the number of hosts with ghost department and Students, or the number of hosts with ghost department and ghost role).

Given the input with slacked group sizes and ghost at-

	CS (9)	EE (8)	Ghost_dept (6)
Faculty (6)	3	1	2
Students (11)	4	5	2
Ghost_role (6)	2	2	2

↓

	CS **(16)**	EE (8)	Ghost_dept (6)
Faculty (**8**)	**5**	1	2
Students (**16**)	**9**	5	2
Ghost_role (6)	2	2	2

Table 2: An example of slack

tributes, Alpaca algorithms compute ACIP allocation. When the updates only occur for the existing attributes, we change the addresses of the affected hosts to unused ACIP from the patterns computed for their new attribute. In the case that the provisioned slack of a group is exhausted, we partition the address space of the associated ghost groups, whose attributes are either ghost attributes or attributes of the exhausted group, and allocate part of the space to the exhausted one. For example, if the ACIPs of (Student, CS) are used up, we could partition the address space of (Student, GhostDept), (GhostRole, CS) or (GhostRole, GhostDept) and assign new space to (Student, CS). When the updates involve a new attribute in one dimension, *e.g.,* Department, we run Alpaca algorithms on the address space for the ghost attribute to split the space into two parts: one for the new attribute and the other for the ghost attribute. Afterwards, the addresses of affected hosts are changed accordingly.

Benefits of slack and ghost attributes. The above two techniques offer another important advantage: *further compacting network policies beyond the optimal solution*. Consider an example where there are 7 CS hosts and 7 EE hosts. Alpaca needs at least 3 rules for each attribute, as $7 = 4+2+1$ (Section 4.1). With slack, we can round 7 to 8, thus allowing Alpaca to use only 1 rule per attribute. In fact, if we round the group size for every attribute to the nearest power-of-two upper bound, we at most double the number of addresses to use. Namely, *we use at most one extra bit to encode attributes given the slacked group sizes.*

We create extra hosts with "mix-matched" attributes such that the number of hosts for every attribute is power-of-two (Algorithm 2). Let p_a be the target power-of-two and g_a be the number of hosts for the attribute a. We choose attribute v_i from i-th dimension such that $p_{v_i} > g_{v_i}, \forall i \in [1,M]$, and create $h = \min_i\{p_{v_i} - g_{v_i}\}$ hosts with attribute $v_1,..,v_M$. When all attributes in a dimension reach their target power-of-two (*i.e.,* $p_a = g_a$), we use the ghost attribute as default, assuming its target power-of-two is infinite. We repeat the procedure until all attributes reach their target power-of-two (except ghost attributes). Consider the example in Table 2. The numbers of hosts for CS, Faculty and Students should be rounded to $16, 8$ and 16. We create 2 CS Faculty hosts in the first iteration and create 5 CS Students afterwards.

For more complex updates that involve additions of new dimensions, it may be desirable to recompute IP allocations from scratch. However, we make several points. First, such scenarios are relatively infrequent. We envision that Alpaca algorithms are run with a conservative set of dimensions, even if some of these dimensions are not currently used as part of network policy. Addition of new dimensions is likely to happen over long time-scales — operators typically collect host attribute information using device registration information filled by owners, and introducing new dimensions would require new data collection for registered devices. Second, when such scenarios do occur, it is feasible to temporarily deal with it by splitting the unused address space of other dimensions and introducing less compact classification rules to identify a given set of hosts. Finally, changes in address allocation can be incrementally handled using DHCP.

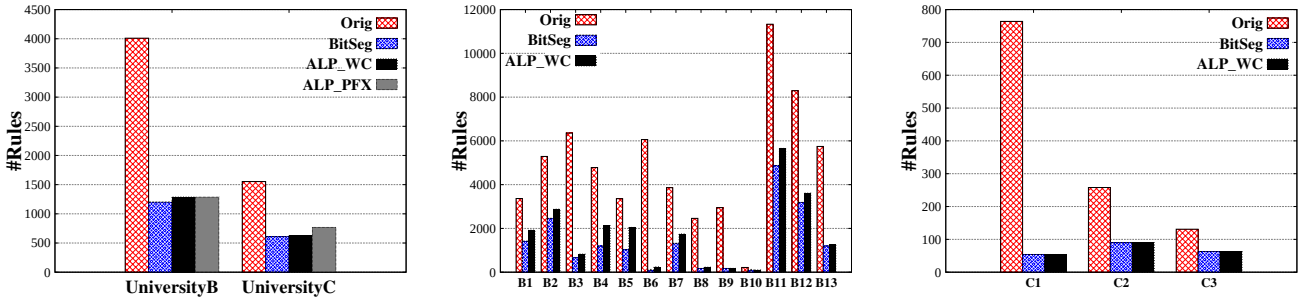
Algorithm 2 Slack algorithm

```
while true do
  for all Dimension d_i do
    v_i = ghost attribute
    for all Attribute a ∈ d_i do
      if p_a > g_a then
        v_i = a
      end if
    end for
  end for
  if ∀i, v_i is ghost attribute then
    break
  end if
  h = min p_{v_i} − g_{v_i}
  g_{v_i} = g_{v_i} + h, ∀i
  Create h hosts with attribute v_1, ..., v_M
end while
```

4.4 Practical Issues

Layer-3 routing. In Alpaca, we consider L3 routing as a policy that forwards packets based on the "location" of their destinations (*e.g.,* the edge switches of the L3 network). Hence, Location is regarded as one dimension in the ACIP allocation. We can run Alpaca to generate classification rules for location dimension, *i.e.,* the rule-set for routing. In some cases, operators may want to pre-assign subnets to the edge switches, *i.e.,* the classification rules for Location are predetermined. Alpaca can work with the requirement as well. The prefix solution naturally decides the prefix patterns for one dimension after another, it can compute the rules given the pre-assigned prefixes for Location; the wildcard solution can construct the equality and inequality of bits in the patterns for nodes based on the Location prefixes first, and make the later ACIP allocation to comply the prefixes.

Mobility. There are two common solutions to ensure connection affinity when hosts move. One approach is to keep the IPs of end-hosts unchanged and update routing rules instead [12]; the other proposes protocols for end-hosts to maintain connections when both IPs can change [13, 14]. While seamless migration is orthogonal to our work, Alpaca can work well with either approach. In the former case, we do not update Alpaca's classification results, as the attributes of the host is unchanged except location, and the change of location (used by the routing policy) is handled by the proposed solution; in the latter case, we can freely assign a new



(a) Multi-table switches (b) Single-table switches (University B) (c) Single-table switches (University C)

Figure 7: Optimize network policies.

	#ACLs	Total #Rules	#SPU	#DPU
University B	13	17868	577	624
University C	5027	32401	523	87

Table 3: Network policies of two universities.

ACIP to the host based on its updated attributes (including location).

5. EVALUATION

In this section, we evaluate Alpaca's effectiveness in producing concise rules under two scenarios: (i) actual policies in existing networks and (ii) futuristic scenarios where operators may express policy along many orthogonal dimensions. For existing settings, we evaluate Alpaca using the network configuration files of University B and University C (Section 5.1). For futuristic settings, we use the host attribute data obtained from University A (Section 5.2). Details of both data-sets were presented in Section 2.

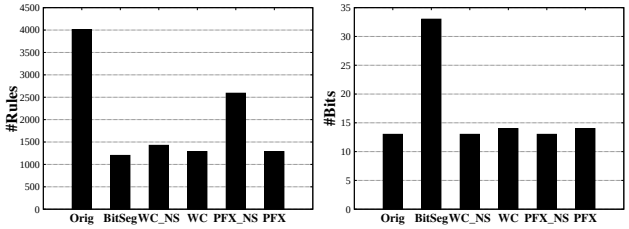
Overall, our results show that Alpaca can reduce the number of rules by $60\% - 68\%$ and $40\% - 96\%$ as compared to the current IP address allocation for multi-table switches and single-table switches, respectively. Meanwhile, it has the potential to reduce the total number of rules by over an order of magnitude as compared to the traditional single dimensional approaches (*e.g.,*VLAN) in futuristic scenario where the policy is expressed on many dimensions. Our experiments further demonstrate that Alpaca can handle changes in hosts gracefully, with only a small extra number of rules.

Our evaluations explore the performance of both Alpaca variants: Prefix (ALP_PFX) and Wildcard (ALP_WC), and for comparison purposes we also consider the BitSegmentation scheme (BitSeg). Unless otherwise mentioned, both our prefix and wildcard algorithms use the algorithm (with prefix extension) in Section 4.2 and the extension with slack in Section 4.3 by default. We evaluate the schemes for multi-table and single-table switches. However, our evaluations with single-table switches is limited to the wildcard algorithm, since the prefix algorithm can only be applied to the multi-table architectures with prefix matching tables.

5.1 Benefits with Existing Policies

5.1.1 Alpaca for multi-table switches

We extract the source and destination policy units (SPUs and DPUs) from the low-level configuration files for both



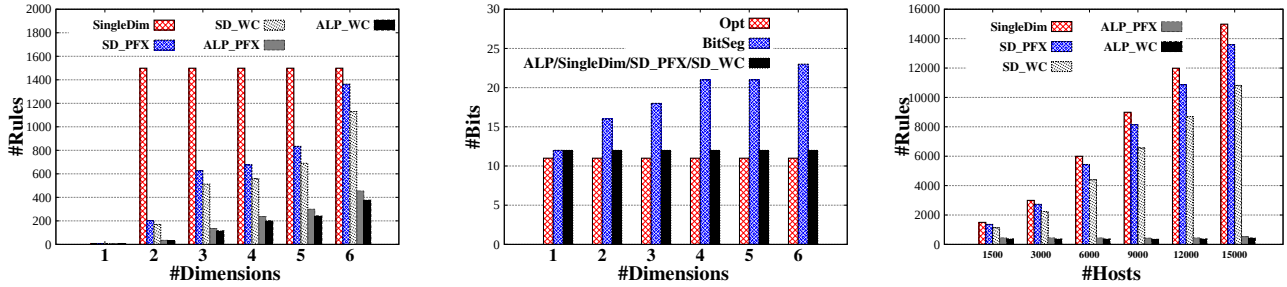
(a) Rules (b) Bits

Figure 8: Benefits of slack. BS, WC and PFX denote Wildcard, Prefix and BitSegmentation schemes. NS indicates variant without slack.

University B and University C, as discussed in Section 2. Table 3 shows the total number of ACL rule-sets, ACL rules across rule-sets, and the number of SPUs and DPUs for both universities. Given a pipeline of tables, we install the classification rules that associate a given IP with its appropriate SPU and DPU in the first two tables, and the actual policy action (*e.g.,* permit or deny) based on the SPUs and DPUs in the last table. We focus on the number of classification rules in the first two tables for a given policy, since the last table is the same in all approaches.

Figure 7(a) compares the number of rules used by (1) the original IP allocation (Orig), (2) BitSeg, (3) ALP_PFX and (4) ALP_WC for University B and University C, respectively. The original IP allocation needs the most rules. BitSeg takes the least, as it uses one rule for each policy unit. Specifically, the number of rules used by BitSeg equals the number of SPUs and DPUs. Both ALP_PFX and ALP_WC perform closely to BitSeg, achieving 68% reduction in rule consumption as compared to the original. It confirms that Alpaca can efficiently encode policy units.

Benefits of slack. We compare the case with and without slack operations to show the benefits of trading an extra bit for significant reduction in number of rules. Figure 8(a) presents the reduction in the number of rules for University B. We use NS to indicate running Alpaca without slack. While WC_NS (3rd bar) is competitive with other approaches, PFX_NS (5th bar) performs slightly worse, giving a reduction of 35.4%. The reasons are two-fold: for one thing, prefix patterns fundamentally restrict the potential of using fewer rules (as compared to wildcard patterns); for the other, PFX_NS solutions represent the exact group size of every combination (*i.e.,* each SPU and DPU pair) without any slack. If the group size is not power-of-two, PFX_NS so-



(a) Increased #dims (Rules) (b) Increased #dims (Bits) (c) Increased #hosts

Figure 9: Encode attributes in addresses.

lutions need many more rules. We bridge the gap by adding slack and rounding group sizes. As a result, PFX offers similar performance to the optimal (*i.e.,* BitSeg). In the remainder of the evaluation, we run slack algorithm before IP allocation by default.

Moving to the number of bits for encoding (Figure 8(b)), we find BitSeg performs the worst, using as many as 34 bits (more than IPv4!). We calculate the least number of bits that can sufficiently number all the hosts (13 for University B). While the original allocation uses the least bits, PFX_NS takes exactly one bit more as the slack algorithm makes use of an extra bit to round group sizes (Section 4.3). Alpaca strikes a balance in both the number of rules and bits, using almost as few rules as BitSeg and one extra bit than the least number of bits.

5.1.2 Alpaca for single-table switches

Generally, a switch with a single table takes more rule space than the one with multiple tables to implement the same policy, because the rules installed in the former case are the cross-product of rules in the multiple tables in the latter case. Alpaca uses the frequency of attributes (*i.e.,* SPU or DPU) used in ACLs to minimize the resulting rules. We compare three approaches: ALP_WC, BitSeg and the original IP allocation. Figure 7(b)(c) demonstrate the effectiveness of Alpaca in compacting large ACL rule-set. Alpaca wildcard compacts the original policies by $40\% - 96\%$, competitive with BitSeg. We would like to point out that the original ACLs are written with respect to the resource constraints of the deployed switches in the networks. As a result, all the original ACLs could fit into the switches. But even so, the reduction by Alpaca is significant. It suggests that with Alpaca, the network operators can use cheaper switches with smaller rule-tables to support today's policies, or plan for larger policies in the future with the current switches.

5.2 Benefits with Futuristic Policies

We demonstrate Alpaca's capability to support flexible attribute-based policies with a series of experiments on the host information at the CS department of University A (Table 1). In the current CS network, operators deploy VLANs to group hosts with the same Role, which is used in most network policies. But they would like to use Security Level, Status and Operating System for access control and have flexible QoS policies defined on Usage, CS_owned as well. Hence, we examine the cost in terms of rules and address space to support the futuristic scenarios, where policies are defined on attributes along multiple dimensions.

We compare Alpaca with three approaches:

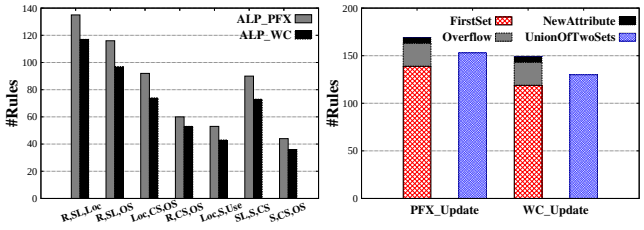
(1) *SingleDim* (*e.g.,* VLAN), which assigns addresses based on a single dimension. SingleDim uses a few rules to represent attributes for one dimension: VLAN uses one rule (the subnet) for each Role attribute; a host is assigned a random address in the subnet corresponding to its Role attribute. However, given a second dimension or more, SingleDim has to enumerate every single host and list their attributes.

(2) *SD_PFX*, which applies an optimal algorithm [15] to minimize the number of prefix rules for attributes, given the SingleDim address assignment.

(3) *SD_WC*, which uses an efficient heuristic [16] to compute the wildcard rules to represent attributes based on the SingleDim address assignment, as minimizing wildcard rules is NP-hard [17].

We remark that SD_PFX and SD_WC minimize the number of rules by assuming rule priority. Both methods generate overlapping rules for different attributes. In contrast, Alpaca generates non-overlapping rules for different attributes, i.e., does not apply rule priority. Below, we show that even without using rule priority, Alpaca significantly outperforms the two compression methods.

Scale with more dimensions. We evaluate Alpaca's encoding efficiency and scalability with increasing number of dimensions. Six dimensions are chosen (in order): Role, Security Level, Location, Status, CS_owned and Usage. The initial set of dimensions only contains Role. Then, in each iteration, we add one more dimension to the current set and run Alpaca algorithms to generate classification rules. Figure 9(a) plots the number of rules generated by SingleDim, SD_PFX and SD_WC and Alpaca over the six iterations. Given one dimension, all approaches generate a small number of rules. Moving to two dimensions (*i.e.,* Role and Security Level), SingleDim has to potentially enumerate hosts and their attributes, taking as many rules as the number of hosts in the data. The number of rules used by SingleDim is unchanged over the iterations then. SD_PFX and SD_WC generates less rules than SingleDim, as they apply compression algorithms for a smaller rule set to represent attributes. Yet, when there are six dimensions, the number of rules (1130 wildcard rules and 1363 prefix rules) is very close to the number of hosts. In contrast, both Alpaca prefix and



(a) Different sets of dims (b) Update assignment

Figure 10: Property of Alpaca algorithm

wildcard scales well with increasing number of dimensions. Alpaca uses 376 wildcard rules or 456 prefix rules for six dimensions, which are significantly smaller. We show the number of bits in Figure 9(b). Both Alpaca and SingleDim approaches use 12 bits, while the least number of bits (denoted as Opt) is 11 ($1491 < 2^{11}$). BitSeg is infeasible in practice after two dimensions, as it takes more than 16 bits to encode attributes.

Scale with more hosts. Our data only covers a single department, but an entire university (with dozens of departments) has more hosts and enterprises can be even larger in sizes. We examine Alpaca's scalability with more hosts, by synthesizing the host information. We copy each host 2 to 10 times and obtain the scaled-up host data. Figure 9(c) compares several approaches to classify hosts in 6 dimensions. Alpaca scales well, using 436 wildcard rules or 528 prefix rules for around 15000 hosts. Its performance is very stable, due to the use of aggregated patterns (*e.g.,* wildcard or prefix matches) to classify groups of hosts. The number of hosts does not impact its performance. In contrast, SingleDim potentially needs 15000 rules to enumerate every host; it does not scale to larger networks. The compression algorithms do not help much: SD_PFX and SD_WC need 13589 and 10821 rules, respectively, because the single dimension based allocation does not help the aggregation on other dimensions.

Encode different sets of dimensions. While Alpaca performs well for three dimensions: Role, Security Level and Location (as shown above), we are curious about its performance on a different set of three dimensions, such as Role, CS_owned and Operating System. Hence, we fix the number of dimensions to encode and run the algorithm on various sets of dimensions. Figure 10(a) shows the performance of Alpaca to encode seven sets of three dimensions. We do observe the fluctuation: the number of rules generated by Alpaca ranges from 36 to 117 for wildcard case and 44 to 135 for prefix case. Upon closer examination, we find out that the performance is highly correlated with the possible combinations of attributes. Specifically, for the set of dimension {Role, Security Level, Location}, there are 80 combinations of attributes which at least one host is associated with; for the set {Status, CS_owned, OS}, there are only 22 combinations. Given increasing numbers of combinations of attributes, Alpaca is more likely to generate many rules.

Update the assignment for new hosts. We divide hosts into two equal-sized sets based on their created time and run Alpaca to encode four dimensions: Role, Security Level, Status and Usage. We use the first set for the initial hosts and the second set for the newly added hosts. For the first set, Alpaca provisions slack and creates ghost attributes for all dimensions. The second set not only inserts more hosts with the existing attributes but also introduces 4 more new attributes in Security Level. To assign addresses to new hosts with existing attributes, Alpaca uses the slack in the corresponding group. But if the group size is insufficient, Alpaca has to "stealâĂŹâĂŹ flow space from the related ghost groups (Section 4.3). To handle the new attributes, Alpaca splits the address space of ghost attributes in the same dimension as well. In our evaluation (Figure 10(b)), the first set (left red bar) uses 139 prefix rules or 119 wildcard rules to represent the four dimensions. Fixing the assignment for the first set, we calculate the extra rules needed to handle the second set. The extra rules come from two parts: (1) hosts with new attributes and (2) overflowed group sizes. As a result, we need an extra 6 rules for the new attributes and 24 rules for the overflowed groups. The overhead is very small compared to 153 prefix rules or 130 wildcard rules for the union of the two sets, where the assignment is computed from scratch without any incremental updates.

6. RELATED WORK

Rule optimization: Minimizing prefix rules matching one header field is easy [18], but minimizing prefix rules or wildcard rules in general cases is NP-hard [15, 17]. Optimal solutions are developed to minimize prefix rules in special cases [18–22]. Heuristics [15, 23, 24] are presented to compress rules in general cases. In particular, [25] suggests to decompose a single rule list into a pipeline of rule lists to minimize the total number of rules. All of these works take the rule-set as an input and explore the potential for minimization, which, in fact, is limited by the original (unoptimized) address allocation. In comparison, Alpaca *generates* the rule list as an output through a smart address allocation process to minimize the number of rules.

Address permutation: Wei et al. propose to swap addresses between two blocks of users to reduce the number of rules [26], but the algorithm can only handle up to two dimensions. Meiners et al. use permutation of the bits in addresses to create prefix patterns so compression algorithms can apply [16]. But it only discovers the optimization potential within the original address allocation.

Information encoding: Huffman coding encodes attributes of a single dimension using prefixes, but its goal is to minimize the weighted sum of the prefix lengths of all attributes. Hence the prefixes do not match the group sizes. SoftCell [12] embeds two dimensional information, *i.e.,* location and middlebox service chain, in the NAT-ed IP addresses. Its encoding mechanism is a special case of BitSegmentation. Algorithms to encode forwarding rules with minimum bits are proposed in [27, 28].

Attribute-based policy enforcement: Ethane [3] proposes to implement access control at the network edge by directing the first packet of every flow to a controller, which consults the attributes of hosts and install microflow rules on the switch. FlowTags [4] tag packets based on host at-

tributes and match tags to enforce network policies. Another approach, NetAssay [29], supports network traffic monitoring policies by pushing specific switch-rules for each host given their current IPs. All these work do not optimize IP allocation and install many host-specific rules.

7. CONCLUSION

In this paper, we have made three contributions. First, we show the importance and feasibility of considering attributes in IP address allocation. Second, we present the Alpaca system, and two algorithms which cope well with constraints on the IP address space, enterprise churn, and heterogeneity in group sizes. When evaluated with configuration data from two universities, ALP_WC and ALP_PFX reduce the number of rules by $50\% - 68\%$ and $60\% - 68\%$ respectively for multi-table switches. Further, the algorithms have the potential to reduce the total number of rules by over an order of magnitude compared to single dimension based IP address allocation schemes. This can in turn lower the barriers for network administrators to express richer policies involving multiple dimensions. While promising, our results are only a first step. In the future, we hope to build and deploy an actual prototype, as well as evaluate the system with more networks and richer data-sets.

8. ACKNOWLEDGMENT

We would like to thank the CoNEXT reviewers and our shepherd Richard Ma. This work was supported by the NSF grant TC-1111520 and the NSF Career Award No.0953622.

9. REFERENCES

[1] M. Yu, J. Rexford, X. Sun, S. G. Rao, and N. Feamster, "A survey of virtual LAN usage in campus networks," *IEEE Communications Magazine*, vol. 49, no. 7, pp. 98–103, 2011.

[2] "Production quality, multilayer open virtual switch." http://openvswitch.org/.

[3] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, 2009.

[4] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *NSDI*, 2014.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM*, 2013.

[8] R. Ozdag, "Intel®Ethernet Switch FM6000 Series-Software Defined Networking," *Intel Coroporation*, 2012.

[9] M. Appelman and M. D. Boer, "Performance analysis of OpenFlow hardware," tech. rep., University of Amsterdam, Feb 2012. http://www.delaat.net/rp/2011-2012/p18/report.pdf.

[10] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *ACM HotSDN*, 2013.

[11] T. Benson, A. Akella, and D. A. Maltz, "Mining policies from enterprise network configuration," in *ACM IMC*, 2009.

[12] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "SoftCell: Scalable and flexible cellular core network architecture," in *ACM CoNEXT*, 2013.

[13] P. Zave and J. Rexford, "The design space of network mobility," in *Recent Advances in Networking. ACM SIGCOMM*, 2013.

[14] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman, "Serval: An end-host stack for service-centric networking," in *USENIX NSDI*, 2012.

[15] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 490–500, Apr 2010.

[16] C. R. Meiners, A. X. Liu, and E. Torng, "Bitweaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 20, pp. 488–500, Apr 2012.

[17] R. McGeer and P. Yalagandula, "Minimizing rulesets for TCAM implementation," in *IEEE INFOCOM*, 2009.

[18] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal IP routing tables," in *IEEE INFOCOM*, 1999.

[19] S. Suri, T. Sandholm, and P. R. Warkhede, "Compressing two-dimensional routing tables," *Algorithmica*, vol. 35, no. 4, pp. 287–300, 2003.

[20] D. L. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *ACM-SIAM SODA*, pp. 1066–1075, 2007.

[21] O. Rottenstreich and I. Keslassy, "On the code length of TCAM coding schemes," in *IEEE ISIT*, 2010.

[22] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal In/Out TCAM encodings of ranges," *IEEE/ACM Trans. Netw.*, 2015.

[23] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Trans. Netw.*, 2015.

[24] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *ACM/IEEE ANCS*, 2015.

[25] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: Optimizing space, power, and throughput for TCAM-based classification," in *ACM/IEEE ANCS*, 2011.

[26] R. Wei, Y. Xu, and H. J. Chao, "Block permutations in boolean space to minimize TCAM for packet classification," in *IEEE INFOCOM*, 2012.

[27] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables for datacenter scalability," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 32, no. 1, pp. 138 – 151, 2014.

[28] O. Rottenstreich, A. Berman, Y. Cassuto, and I. Keslassy, "Compression for fixed-width memories," in *IEEE ISIT*, 2013.

[29] S. Donovan and N. Feamster, "NetAssay: Providing new monitoring primitives for network operators," in *ACM HotNets*, 2014.