Concise Encoding of Flow Attributes in SDN Switches

Robert MacDavid*, Rudiger Birkner†, Ori Rottenstreich*
Arpit Gupta*, Nick Feamster*, Jennifer Rexford*
*Princeton University †ETH Zürich

Abstract

Network devices such as routers and switches forward traffic based on entries in their local forwarding tables. Although these forwarding tables conventionally make decisions based on a packet header field such as a destination address, tagging flows with *sets* or *sequences* of attributes and making forwarding decisions based on these attributes can enable richer network policies. For example, devices at the edge of a network could add a tag to each packet that encodes a set of egress locations, a set of host permissions, or a sequence of middleboxes to traverse; simpler devices in the core of the network could then forward packets based on this tag.

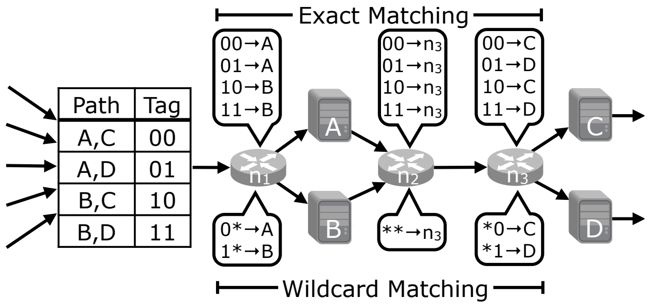
Unfortunately, naive construction of these tags can create forwarding tables that grow quadratically with the number of elements in the set or sequence—prohibitive for commodity network devices. In this paper, we present PathSets, a compression algorithm that makes such encodings practical. The algorithm encodes sets or sequences (*e.g.*, middlebox service chains, lists of next-hop network devices) in a compact tag that fits in a small packet-header field. Our evaluation shows that PathSets can encode attribute sets and sequences for large networks using tag widths competitive with existing approaches and that the number of forwarding rules grows linearly with the number of attributes encoded.

CCS Concepts:
Networks → Network management; Programmable networks;

Keywords:
Software-Defined Networks; Network Management; OpenFlow; Commodity Switch; TCAM

1 Introduction

Forwarding network traffic increasingly involves more sophisticated forwarding operations than simply traversing a shortest path to a single destination. The values in a packet header may cause the packet to traverse a sequence of middleboxes, be subject to various access-control policies, or be forwarded to one of several distinct destinations. These types of flexible forwarding decisions can require network switches to have many forwarding table rules, each of which may match on multiple packet header fields, thus causing switches to need large tables to handle all possible cases.



Path	Tag
A,C	00
A,D	01
B,C	10
B,D	11

Figure 1: *An illustration of the scaling issues of flat tagging. There are three switches n_1, n_2, n_3 and four middleboxes A, B, C, D. Packets enter the network on the left and are classified as needing to traverse one of four service chains, and are tagged as such. If flat matching is used, no aggregate rules are possible on the four tags.*

Fortunately, although the number of possible combinations of forwarding decisions could in theory result in an inordinate number of flow table entries, in *practice* many distinct traffic flows may be subject to the same forwarding actions, or forwarding equivalence class (FEC). Along an end-to-end path, a switch could mark all packets that belong to the same FEC with a tag that subsequent switches could use to make forwarding decisions. Switches further downstream on the path could then make forwarding decisions based on this tag, as opposed to a complex combination of header fields, potentially reducing switch forwarding table size if the number of FECs is smaller than the number of unique header field combinations.

The simplest way to mark a FEC is with a flat tag. MPLS [22] and VLANs [1] use flat tags, for example, and newer SDN-based architectures such as FlowTags [6] and our original SDX architecture [9] also make forwarding decisions based on flat tags. Flat tags work well in many scenarios because hardware switches can perform exact matches on these fields (*e.g.*, using a content-addressable memory); yet, forwarding tables that match on flat tags do not scale, because the number of forwarding rules that the switch must store

SOSR'17, April 03–04, 2017, Santa Clara, CA, USA
© 2017 ACM. ISBN 978-1-4503-4947-5/17/04 ...$15.00
DOI: http://dx.doi.org/10.1145/3050220.3050227

grows linearly with the number of FECs which traverse that switch, and small changes to forwarding decisions can trigger recomputation of a large number of FECs (and tags).

In practice, a FEC constitutes a *collection* of forwarding decisions (*e.g.*, a sequence of middleboxes), but two FECs that differ only slightly might have arbitrarily different flat tags, making it difficult to aggregate forwarding table rules. For example, Figure 1 shows an example where a FEC constitutes a sequence of middleboxes. Two FECs may differ in only a single middlebox yet matching on flat tags requires the switch to maintain an entry for each unique sequence. In this example, switch n_1 uses separate rules to forward tags 00 and 01 to A, even though the two sequences are forwarded equally by n_1. In contrast, wildcard matching in the switch, coupled with a careful assignment of tag bits, can enable the smaller rule tables shown in the bottom of the figure.

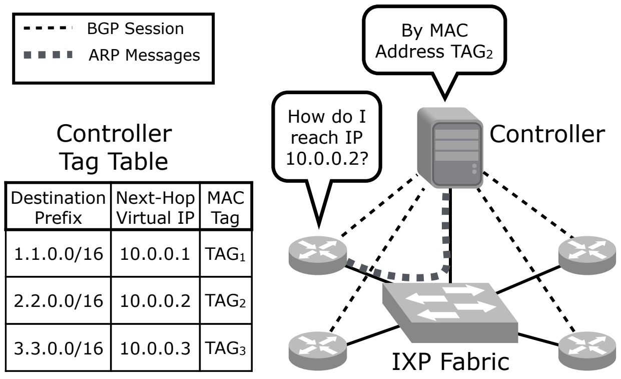
A forwarding mechanism that assigns tags that reflect the commonality between sequences or sets between different forwarding equivalence classes could make it possible to aggregate rule-table entries in switches. Fortunately, commodity switches that support OpenFlow 1.3 [17] can now perform wildcard matching on arbitrary packet-header fields; emerging protocol-independent switches supported by languages such as P4 [5] also support more flexible matching based on wildcards, as well as the ability to create new header fields that can serve as tags. These new matching capabilities make it possible to redesign FEC tagging architectures to achieve more efficient encodings.

In this paper, we present PathSets, an encoding suite that takes advantage of these emerging capabilities. PathSets can encode similar FECs with tags that share common bits, as opposed to simply assigning each FEC a distinct flat tag. These *attribute-carrying tags*, where attributes can be forwarding decisions or flow properties, are more compact and reduce forwarding table size compared to flat tags. We present efficient algorithms for encoding sets and sequences of attributes in tags and demonstrate how these encodings can reduce switch rule-table size for a variety of applications, including an SDN-based Internet Exchange Point and a service chaining application. We have publicly released the source code for the PathSets library that implements this encoding algorithm [18].

The rest of this paper proceeds as follows. Section 2 presents three motivating applications. Section 3 presents the basic ideas of PathSets. In Section 4, we extend PathSets to support sequences of attributes; Section 5 improves the space efficiency of the encoding using variable-length prefix codes. In Section 6, we evaluate the encodings over both real and synthetic data sets for two uses cases. We discuss related work in Section 7 and conclude in Section 8.

2 Motivating Applications

There are many applications where encoding and reading attributes is of interest, and in particular we will look at three of them, which are outlined in Table 1.



Destination Prefix	Next-Hop Virtual IP	MAC Tag
1.1.0.0/16	10.0.0.1	TAG_1
2.2.0.0/16	10.0.0.2	TAG_2
3.3.0.0/16	10.0.0.3	TAG_3

Figure 2: *A Software-Defined IXP (SDX) [8].*

2.1 Software-Defined IXPs

At an Internet Exchange Point (IXP), multiple autonomous systems (ASes) connect at a single logical interconnect to exchange traffic and interdomain routing information. At an IXP with support for SDN policies, the connected ASes may wish to enact fine-grained policies based on attributes beyond a packet's destination address. Suppose that AS_1 wants to send as much of its HTTP traffic to AS_2 as possible. AS_2 may not have routes to every HTTP destination, so it is incorrect for this AS to receive all HTTP traffic, regardless of destination. If the routing policies do not account for the BGP routes advertised by each AS, traffic may be forwarded to networks that cannot handle it.

The initial SDX system used flat tags to encode these distinct forwarding decisions [9], as shown in Figure 2. Border routers connect to the IXP fabric and initiate BGP sessions with the controller. Each router announces BGP routes for IP prefixes to the controller, and the controller aggregates these, computes FECs for each IP prefix, and announces a Virtual Next-Hop (VNH) IP address for IXP participants to reach that prefix. When a participant attempts to send traffic to a VNH IP, it first sends an ARP request for the MAC of that VNH to the IXP fabric. The controller intercepts the ARP request, and responds with a tag as the destination MAC. The participant then forwards packets to the IXP fabric with the appropriate tag in the destination MAC address field. The tagging technique works without requiring any modifications to legacy BGP-speaking border routers. This approach works with unmodified BGP-speaking border routers, simply by changing the destination MAC address associated with the border routers' own forwarding rules.

Unfortunately, the flat tagging approach introduces scalability challenges at the IXP by creating large rule tables and frequent changes in the installed rules when BGP routes change. We remedied these issues in the follow-up work on iSDX [8]. iSDX used a precursor to PathSets for sets of attributes, taking advantage of OpenFlow 1.3's support for wildcard matching on destination MAC addresses.

Application	Existing Solution	Attributes	Tag Field	Tags Conveyed By
SDN-Enabled IXP (SDX)	iSDX [8]	Advertising peers	Destination MAC	ARP
Service chaining	FlowTags [6]	Middleboxes	IP Fragment Field	First Middlebox
Policy enforcement	Alpaca [14]	Host permissions	IP Source Address	DHCP

Table 1: *Example applications and systems which have solved them by some form of tagging.*

2.2 Service Chaining

Network operators often want traffic to traverse a sequence of middleboxes, such as load balancers or firewalls. Different flows may be subject to different chains of middleboxes, and it can be a challenge to design the network in such a way that every flow traverses only the needed set of middleboxes, and only in the correct order. Additionally, middleboxes may modify packet headers, obscuring the original source of the flow and making it unclear which middlebox chain should be followed.

FlowTags [6] uses tags to encode how each middlebox should process each packet. To compress policy information into small, repurposed header fields, FlowTags uses flat tags, where each tag maps to a middlebox sequence and the packet's origin host. Whenever a middlebox sees a new flow, it communicates with a central controller to establish a new tag, which introduces delay and overhead. The FlowTags paper does not evaluate the number of rule-table entries that such a flat tagging scheme might require. Because many flows may traverse similar sequences of middleboxes, we expect that service chaining may benefit from PathSets, the attribute-based tagging approach we present in this paper.

2.3 Host Attributes for Network Policies

In some situations, network policies depend on the sending or receiving host. For example, users in certain departments within an organization may be subject to different quality-of-service or access-control policies. If department information is not attached to packets directly, it must be inferred from some combination of packet-header fields, which can result in unnecessarily complex rule tables.

Alpaca [14] encodes policy information in the low-order bits of host IP addresses and assigns these addresses to the hosts via DHCP. Although not explicitly a tag, these IP addresses can be thought of as a tag appended to the network's IP prefix. Alpaca takes advantage of prefix and wildcard matching when constructing tags to overcome the memory scalability challenges of flat tagging, yet each host must receive a different tag, because each IP must be unique. FEC-based tagging could provide significant opportunities for aggregation in this setting.

3 Encoding Attribute Sets

In this section, we describe the intuition behind attribute-carrying tags and the metrics by which they can be evaluated. We then present a concise encoding structure that achieves a fair tradeoff between the metrics, and we also present an algorithm for computing such an encoding.

The simplest way to encode a set of attributes is a bitmask with one bit for each attribute, at the expense of large tags. In this section, we present a concise encoding that uses multiple smaller bitmasks on different subsets of attributes, at the expense of slightly larger rule tables. We also present an algorithm for computing the concise encoding.

3.1 Strawman: Bitmask Tags

To illustrate the use of attribute-carrying tags, we begin with a strawman encoding scheme that takes full advantage of forwarding tables capable of wildcard matching. The method proceeds as follows: If there are N possible attributes that may be encoded in any tag, construct tags of length N: the i^{th} bit corresponds to the i^{th} attribute. When a packet is classified and the tag is attached, the i^{th} bit is set to 1 if the i^{th} attribute is present for that flow, and 0 if it is not. As a result, testing for the presence of attribute x, requires only checking that x's bit is 1 in the tag, rather than exact matching on every tag that contains x.

This approach is a strawman because the tag is of width N, and N can be quite large. This can be viewed as an extreme opposite of flat tagging. Flat tagging consumes too much switch memory as the number of FECs grows, but only needed a tag width logarithmic in the number of FECs. In summary, we have traded an extreme in memory usage for an extreme in tag width. An ideal design would strike the right balance between these two extremes. In general, any tagging scheme must simultaneously consider three different metrics:

1. **Tag Width:** Tags should not be too wide, to avoid wasting packet-header space. Tags should be able to either be inserted into small repurposed header fields or contribute little size to custom packet headers.
2. **Switch Memory:** The amount of memory required to decode attributes from any tag should be able to easily fit in modern commodity switches rule tables.
3. **Churn:** Normal network events should not cause too many rule additions and removals in switch tables.

3.2 Multiple Smaller Subsets of Attributes

In the strawman solution of a simple bitmask, the tag has one bit for each boolean attribute. For example, Figure 3(a) shows multiple forwarding equivalence classes (S_1-S_5) that each correspond to a different subset of five attributes (A-E). The network policy determines which traffic has which attributes,

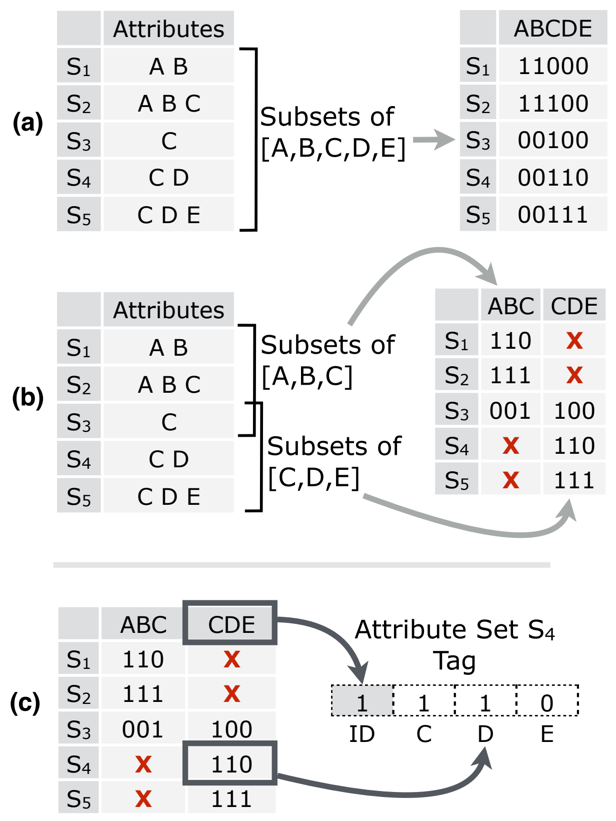


Figure 3: *Two different ways to recover attribute sets. In (a), the sets are recovered by masking over* $[A,B,C,D,E]$*. In (b), the sets are recovered by masking over either superset* $[A,B,C]$ *or set* $[C,D,E]$*. An X denotes that the set cannot be fully recovered by masking over the given set. (c) shows how the matrix in (b) can be mapped to tags.*

and which combinations of attributes can hold together. For example, all packets with attributes A and B true, and C, D, and E false, belong to forwarding equivalence class S_1, and can be encoded with the bitmask `"11000"`. Certain combinations of attributes may not occur for any traffic (*e.g.*, attributes B and D are never true together, though both can be false as in S_3). A single rule can test for any combination of attributes (*e.g.*, comparing a tag to `"1*1**''"` identifies whether attributes A and C hold, without caring whether B, D, or E hold). However, when the number of attributes is large, a bitmask tag becomes quite large.

The PathSets encoding identifies groups of attributes that commonly appear together, and creates one shorter bitmask for each such group. In Figure 3, attributes A, B, and C commonly appear together, as do C, D, and E, leading to two groups. The right side of Figure 3(b) shows how each forwarding equivalence class can be encoded as a bitmask on $[A,B,C]$, $[C,D,E]$, or both. To distinguish the two groups, the tag can include a one-bit group identifier (*e.g.*, a 0 for $[A,B,C]$ and a 1 for $[C,D,E]$), as shown in Figure 3(c). The result is a four-bit identifier where, for example, S_4 with attributes C and D is encoded as `"1110"`.

The forwarding rules can match on attributes by considering both the group identifier and the associated bitmask. For example, a switch could test for attribute D by matching on (i) group 1 and (ii) the D bit in group 1's bitmask. Thus, the rule would have a match of `"1*1*"`. Naturally, however, some attributes appear in multiple groups, such as attribute C in the example. The switch can use two rules to test for C: (i) `"0**1"'` for group 0 and (ii) `"11**"'` for group 1. Thus, PathSets slightly increases the forwarding table size over the simple bitmask approach (by only one rule in this example).

3.3 Computing Concise Encodings of Sets

The tag and rule-table sizes depend on the nature of the encoding. A natural starting point is to have one group for each attribute equivalence class (*e.g.*, $[A,B]$, $[A,B,C]$, $[C]$, $[C,D]$, and $[C,D,E]$), at the cost of a large group identifier. A simple first step is to remove any group with attributes that are a *subset* of another group (*e.g.*, removing $[A,B]$ and $[C]$ that are subsets of $[A,B,C]$, and removing $[C,D]$ that is a subset of $[C,D,E]$). In the simple example, this step results in the two groups ($[A,B,C]$ and $[C,D,E]$). In examples with more FECs, this step could still create too many groups. Instead, the algorithm uses the set of groups ($\mathbb{S}$) produced by the subset removal as an input to an algorithm for optimizing the selection of groups.

Algorithm 1: Greedy Memory Minimization

```
Input: Groups $\mathbb{S}$, Attribute Test Counts $\{q_k\}$, Tag Width
       Limit, $W_{max}$, Tag Width Calculator $W()$.
Output: Groups $\mathbb{S}$ with minimal rule-table size.
begin
    while $|\mathbb{S}| > 1$ do
        $bestPair \leftarrow (None, None)$
        $bestGain \leftarrow 0$
        for $(s_i, s_j) \in \mathbb{S} \times \mathbb{S}$ do
            $\mathbb{S}_{temp} \leftarrow (\mathbb{S} - \{s_i, s_j\}) \cup \{s_i \cup s_j\}$
            if $W(\mathbb{S}_{temp}) \leq W_{max}$ then
                $gain \leftarrow \sum_{k \in s_i \cap s_j} q_k$
                if $gain > bestGain$ then
                    $bestGain \leftarrow gain$
                    $bestPair \leftarrow (s_i, s_j)$
        if $bestPair = (None, None)$ then
            break
        $(s_a, s_b) \leftarrow bestPair$
        $\mathbb{S} \leftarrow (\mathbb{S} - \{s_a, s_b\}) \cup \{s_a \cup s_b\}$
    return $\mathbb{S}$
```

The encoding algorithm iteratively merges pairs of groups to minimize the rule-table size while staying within some limit W_{max} on the tag size. Suppose a switch has q_a clauses that test whether attribute a is true, and the attribute appears in k_a groups. Then, the switch would require $q_a \cdot k_a$ rules to test for that attribute. Merging two groups can reduce the number of rules but may increase the tag size depending on the number of attributes the two groups have in common. Suppose $\mathbb{S} = \{s_1, s_2, \ldots, s_N\}$, where group s_i is a subset of attributes. Replacing any pair of groups $\{s_i, s_j\}$ with their union $s_i \cup s_j$ would decrease the number of rules by $\sum_{a \in s_i \cap s_j} q_a$ because every attribute a in the intersection of the two groups would appear in one fewer group after the merge.

We can extend this observation to a greedy algorithm that repeatedly merges the pair of groups that maximally decreases the number of rules without exceeding the W_{max} tag size. Algorithm 1 shows the pseudocode. The algorithm takes as input a set of attribute groups $\mathbb{S} = \{s_1, s_2, \ldots, s_N\}$, the attribute test counts $\{q_k\}$, a maximum tag width W_{max}, and a tag-width calculator $W()$. The tag-width calculator determines the number of bits in the tag, given the current set of groups $\mathbb{S}$. We present closed-form equations for the tag-width calculator later in Section 5.

The algorithm runs for at most $N = |\mathbb{S}|$ iterations, and for each iteration considers N^2 pairs of groups, causing the inner loop to execute $O(N^3)$ times.

The inner loop performs two significant operations: a call to the tag width calculator $W()$, and a merging of two attribute groups. If the former takes time T_W and the latter takes time T_M, the final running time is $O(T_W T_M N^3)$. In our final implementation, T_W is constant and T_M is linear in the average group size.

It is straightforward to extend this algorithm to handle set updates. If, after groups have been computed, any attribute set decreases in size, the decreased set is clearly still a subset of an existing group. As a result, only the tag for that set must change; no new forwarding table rules are generated. If a set grows and is no longer a subset of any group, we create a new group equal to the new set. We then run one more iteration of the greedy algorithm to attempt to merge the new set into an existing group, taking $O(T_M N)$ time to consider the N possible merges. If the new set cannot be added without exceeding the maximum tag width, a full recomputation is needed. This can be made unlikely by stopping the initial greedy algorithm at width $W'_{max} = W_{max} - x$ for some $x > 0$, to allow space for the groups to grow.

4 Encoding Attribute Sequences

In some applications, the order of attributes is important. For example, in service chaining, traffic must traverse middleboxes in a specified order. In this section, we extend PathSets for encoding sets of attributes to support sequences of attributes. We first propose a simple representation of the attribute sequences in a sequence graph. Next, we discuss how to encode sequences when the attributes follow a partial order. Then, we show how to encode sequences in general, even when the attributes do not form a partial order.

4.1 Sequence Graph of Attribute Orderings

When the tag must encode a sequence of attributes, we need an effective way to identify what ordering of attributes can occur. Figure 4(a) shows four equivalence classes (S_1-S_4) with different sequences of attributes (A-F); for example, S_2 has the attribute sequence E-A-B, whereas S_4 has A-B-C-D. We can generate a *sequence graph*, where each node is an attribute, and a directed edge from node u to node v exists if u appears before v in any of the sequences. For example, the sequence graph in Figure 4(b) has an edge from E to A, from E to B, and from A to B because of S_2, and from D to F because of S_3. Each sequence in the input data corresponds to a path through the sequence graph. If the sequence graph is acyclic, the attributes form a partial order, making it easier to encode all of the sequences concisely.

We expect that many use cases impose a natural order on the attributes. For example, in traffic steering, typically a compression middlebox would occur before an encryption middlebox, and encryption would occur before decryption. Similarly, a firewall would usually appear before a network address translator, so the firewall can act on the end-host IP addresses rather than the address of the NAT. Still, sometimes exceptions can easily arise. In extending PathSets to encode sequences, we optimize for the case where attributes mostly follow a natural order, and handle the (presumably) few exceptions as they arise.

4.2 Sequences Forming a Partial Order

If all of the attributes form a (partial) order, we can easily construct a single ordered list of all attributes where each input sequence is a subsequence. We refer to such a sequence as a *supersequence*. A supersequence can be computed by processing the nodes of the (acyclic) sequence graph in order. For example, suppose the input sequences are X-Y, Y-Z, and X-Z; then, the resulting supersequence would be X-Y-Z. To generate the tags, Algorithm 1 can be used “as is” to generate a concise encoding. The resulting rules in the switches would simply match on the attributes in the appropriate “order”, *e.g.*, a rule that checks for attribute Y would include a 0 bit for X, to ensure that middlebox X was already visited (if necessary) and the associated bit in the tag cleared. We discuss how to compute the rules in more detail in Section 4.4.

4.3 Sequences Forming a Cycle

When the sequence graph contains a cycle, no supersequence of attributes exists that is consistent with all of the input sequences. For example, the sequence graph in Figure 4(b) contains a cycle with attributes B and C because S_1 has sequence A-C-B-D (with C before B) while S_4 has sequence A-B-C-D (with B before C). We considered two approaches to overcome this obstacle.

Group sequences with consistent attribute orderings (Approach I): The approach merges groups only if they have consistent ordering of the attributes. For example, S_2 and S_4 could merge because E-A-B-C-D is consistent with both

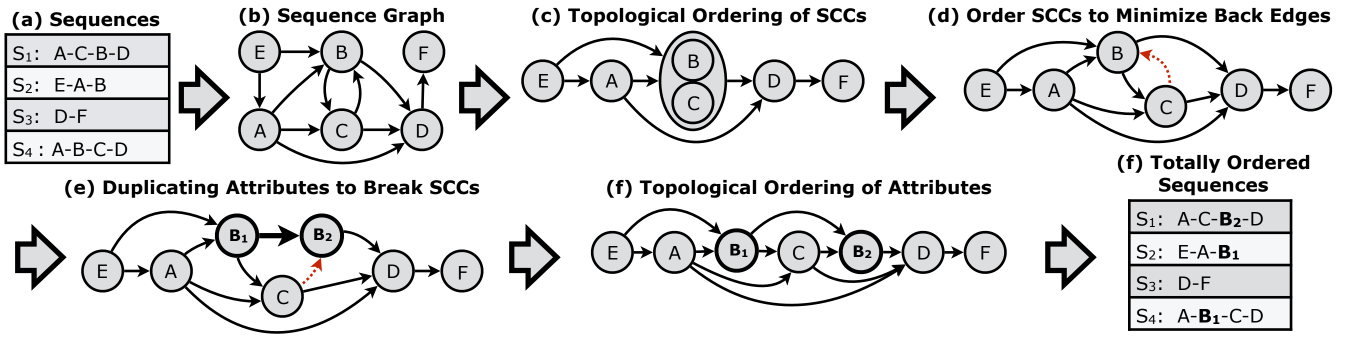

(a) Sequences
S1: A-C-B-D
S2: E-A-B
S3: D-F
S4 : A-B-C-D
(b) Sequence Graph
(c) Topological Ordering of SCCs
(d) Order SCCs to Minimize Back Edges
(e) Duplicating Attributes to Break SCCs
(f) Topological Ordering of Attributes
(f) Totally Ordered Sequences
S1: A-C-B2-D
S2: E-A-B1
S3: D-F
S4: A-B1-C-D

Figure 4: *Sequences forming a cycle of attributes. (a) shows the four input sequences. (b) illustrates the corresponding sequence graph G (c) shows the disjoint SCCs (Strongly Connected Components) of G. Each SCC corresponds to a set of incomparable elements. In (d), we order the elements of the SCC to minimize backward edges. (e) and (f) show how nodes of the SCC can be split to make all backward edges become forward edges, allowing a complete ordering to be found. Splitting nodes results in duplicated elements, which slightly changes the original sequences, as shown in (f).*

sequences. Pairs of groups could merge based on similar criteria as in Section 3.3, subject to the constraint on consistent attribute order. At the completion of the algorithm, each group would be assigned a group id and a single ordering of their combined attributes, represented as a bitmask.

Break cycles by duplicating nodes in the sequence graph (Approach II): The first approach can sometimes result in too many groups that cannot merge. The second approach is strictly more flexible because it permits merging two sequences even if they impose a different order on some attributes. To resolve the inconsistency, we break cycles by duplicating some nodes in the sequence graph, and representing each copy of the attribute with a different bit in the bitmask. For example, we can break the cycle in Figure 4(b) by splitting node B into B_1 and B_2. This makes it possible to construct a supersequence E-A-B_1-C-B_2-D-F (Figure 4(f)) that is consistent with each sequence (Figure 4(g)).

To break cycles in a systematic fashion, we run an algorithm that finds the Strongly Connected Components (SCCs) on the sequence graph. An SCC is a set of nodes such that for every pair of nodes u and v in the set, u has a directed path to v and vice versa. In this context, every SCC corresponds to a set of attributes that cannot be placed in order. Figure 4(c) shows the result of finding SCCs on the sequence graph, identifying the pair of attributes B and C. We then identify a minimal set of "back edges" to remove, and split nodes to remove each of these back edges. For example, creating nodes B_1 (with an edge to C) and B_2 (with an edge from C) breaks the cycle, resulting in the acyclic sequence graph in Figure 4(e). It is redrawn in Figure 4(f) to highlight a suitable ordering of the attributes, leading to the modified input sequences. Now, we can simply run Algorithm 1 "as is" to generate the groups and their corresponding bitmasks.

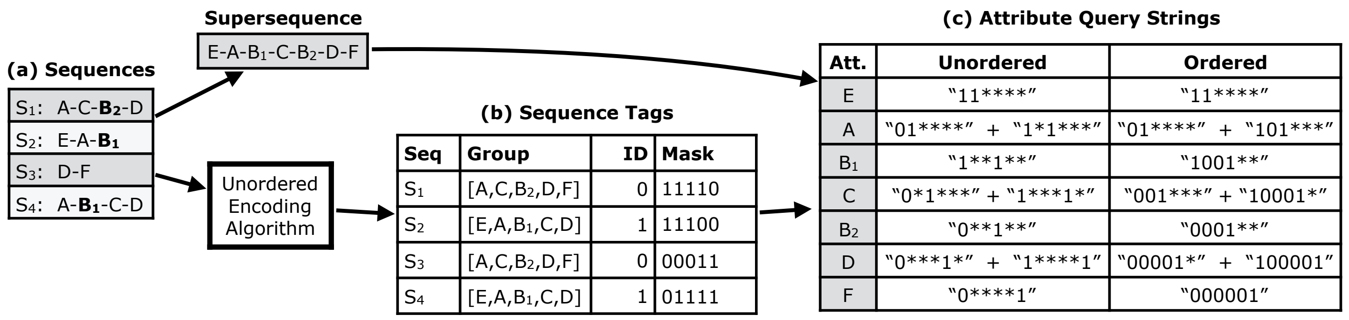
We must note that finding a minimal set of "back edges" in step (d) is NP-Hard and is commonly known as the Feedback Arc Set problem. To complete this step, we use a simple 2-approximation for the dual problem of Maximum Acyclic Subgraph, which is described in [12]. Additionally, if sequences are allowed to have repeating attributes, then finding a total ordering becomes the more general Shortest Common Supersequence problem. This is known to be NP-Hard even for the case of only two attributes [19]. We are not aware of any works which prove or disprove that our variant is NP-Hard, and our algorithm seems to perform well when there are not too many conflicts.

4.4 Constructing Rules That Respect Order

We previously showed how to construct tags and match strings for attribute sets. Figure 5 shows the process for constructing rules for sequences of attributes. The process assumes that the input sequences are drawn from a supersequence (as in Figure 4(g), shown again in Figure 5(a)), so the input to this process is the output of the conflict-resolution algorithm described in Section 4.3.

The input sequences are treated as attribute sets and run through the PathSets encoding scheme (Algorithm 1), to produce a tag for each sequence (Figure 5(b)) and a set of unordered match strings for each attribute (middle column of Figure 5(c)). To produce rules that respect ordering, we consider the concrete example of attribute A. The unordered tests for A include a rule for each group that contains A. The first rule (`"01****"`) matches on group id 0 and the first bit of the bitmask ($[A,C,B_2,D,F]$), and the second rule (`"1*1***"`) matches on group id 1 and the second bit of the bitmask ($[E,A,B_1,C,D]$).

The second rule is not appropriate when considering attributes in a sequence because attribute E appears ahead of A in the sequence E-A-B_1-C-D. If attribute E holds for a packet (*i.e.*, the bit for attribute E is set to 1), the packet should match a rule concerning attribute E (*e.g.*, `"11****"`) rather than A. So, policies that impose an ordering on the attributes must check that all earlier attributes in the bitmask are set to 0. So, in the right column of Figure 5(c), there is a match string for each group that contains A, with any bits before A in the bitmask set to 0 rather than a wildcard. That is, the second



(a) Sequences

S_1:	A-C-**B_2**-D
S_2:	E-A-**B_1**
S_3:	D-F
S_4:	A-**B_1**-C-D

(b) Sequence Tags

Seq	Group	ID	Mask
S_1	[A,C,B_2,D,F]	0	11110
S_2	[E,A,B_1,C,D]	1	11100
S_3	[A,C,B_2,D,F]	0	00011
S_4	[E,A,B_1,C,D]	1	01111

(c) Attribute Query Strings

Att.	Unordered	Ordered
E	"11****"	"11****"
A	"01****" + "1*1***"	"01****" + "101***"
B_1	"1**1**"	"1001**"
C	"0*1***" + "1***1*"	"001***" + "10001*"
B_2	"0**1**"	"0001**"
D	"0***1*" + "1****1"	"00001*" + "100001"
F	"0****1"	"000001"

Figure 5: *Match strings for attribute sets versus sequences. (a) shows the set of input sequences which, when treated as unordered sets and run through Algorithm 1, produce a set of tags in (b). The unordered column of (c) shows the query strings produced for checking each attribute in an unordered fashion. The ordered column is produced by taking the unordered strings and replacing wildcard characters with 0 for every attribute that appears before the current attribute in the ordering.*

rule is "101⋆⋆⋆" for an ordered test on A, instead of the "1⋆1⋆⋆⋆" match for the unordered test.

5 Variable-length Group Identifiers

In the previous sections, the PathSets encoding imposes a fixed division of the tag bits into the *group identifier* and the *bitmask* on the attributes in the group. However, some groups include more attributes than others, making a static split inefficient. In this section, we describe an enhanced encoding scheme that reduces the total size of the tag by allowing variable-length group identifiers, so some groups can devote more tag bits to the bitmasks.

5.1 Prefix Codes for Group Identifiers

Table 2 illustrates how a fixed division between the group identifier and the bitmask can waste space in the tag. Table 2(a) shows an example output of Algorithm 1 with four groups with anywhere from two to four attributes. With a fixed-length group identifier, the tags require a two-bit identifier (to represent the four groups) and a four-bit bitmask (to represent the largest bitmask of $[W,X,Y,Z]$), for a total of six bits, as shown in Table 2(b). The other three groups do not make effective use of the four-bit bitmask. In the general case, given N groups where group i has ℓ_i attributes, the width of the tag is determined by the largest group and equals $W_f = \lceil \log_2(N) \rceil + \max_{i \in [1,N]} \ell_i$.

To reduce the size of the tag, we introduce variable-length group identifiers. In particular, groups that need larger bitmasks should have shorter group ids, and vice versa. Table 2(c) uses group identifiers 0, 10, 110, and 111, enabling a smaller tag with just five bits. Notice that the group identifiers are selected such that none is a prefix (start) of another, *i.e.*, the identifiers are codewords in a *prefix code*. Selecting the group identifiers in this way allows the switch rules to distinguish between the groups for any values for their bitmasks in the second part of the tag.

(a) *Groups*

Group attributes
A B C
C D
E F
W X Y Z

(b) *Fixed-length group ids*

Id	Attributes
00	A B C
01	C D
10	E F
11	W X Y Z

(c) *Variable-length group ids*

Id	Attributes
10	A B C
110	C D
111	E F
0	W X Y Z

Table 2: *Illustration of variable-length group identifiers. While with fixed-length ids the maximal width in (b) is 6 bits, with variable-length identifiers it is reduced in (c) to only 5 bits.*

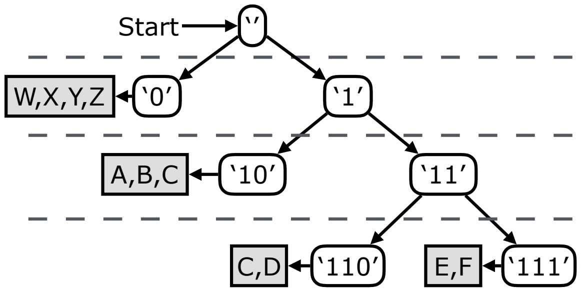


Figure 6: *An example prefix code tree, based upon the prefix code identifiers from Table 2c which are used to identify the attribute sets* $[A,B,C]$, $[C,D]$, $[E,F]$ *and* $[W,X,Y,Z]$.

5.2 Computing Efficient Prefix Codes

Kraft's inequality [2] formally determines whether prefix codes of given lengths exist. It says that a code with N codewords of lengths $L_1, L_2, \ldots, L_N$ exists if and only if the following inequality holds:

$$\sum_{i=1}^{N} 2^{-L_i} \leq 1.$$

For instance, the lengths of the identifiers in Table 2(c) satisfy $2^{-1}+2^{-2}+2^{-3}+2^{-3}=1$. This also enables us to exactly calculate the minimal width W_v that can be achieved for a given N groups of size $\ell_1,\ldots,\ell_N$ using variable-length identifiers as described in the following property:

Property 1: *For given groups of size $\ell_1,\ldots,\ell_N$ the optimal (minimal) tag width that can be derived using variable-length identifiers is given by inequality holds:*

$$W_v = \left\lceil \log_2 \sum_{i=1}^{N} 2^{\ell_i} \right\rceil.$$

Proof. A width of W_v allows assigning an identifier of length $W_v - \ell_i$ to the i-th group. By Kraft's inequality the width W_v must satisfy:

$$1 \geq \sum_{i=1}^{N} 2^{-(W_v-\ell_i)} = 2^{-W_v} \cdot \sum_{i=1}^{N} 2^{\ell_i}.$$

Accordingly we have $2^{W_v} \geq \sum_{i=1}^{N} 2^{\ell_i}$. Since W_v is defined as the minimal possible width with that property, the result follows. □

The fixed-length group identifier is a special case of the variable-length identifiers, so $W_v \leq W_f$.

A selection of variable-length group identifiers satisfying Kraft's inequality can be described as a subset of leaves in a binary tree. A path from the root node to a node corresponds to a binary string where visits of the left child or the right child of a node stand for bits of 0 and 1, respectively. The path length to a leaf corresponds to the identifier length in bits. Figure 6 illustrates the corresponding tree for the four identifiers from Table 2(c). Shorter identifiers appear higher in the tree.

For a given set of groups, after calculating W_v as the minimal possible width enabled by variable-length identifiers, we can easily find identifiers that realize it. If a group has ℓ_i elements, then the maximum size the group identifier may have is $W_v - \ell_i$ bits. To assign every group an identifier, we begin with a complete version of the binary tree seen in Figure 6, but with a depth of W_v. Recall the i^{th} level of the tree contains all binary strings of length i, and each node's string is a prefix of all its descendants' strings.

The tree is traversed in level-order, beginning at the root. At each level i, if a group s without an identifier exists such that $i = W_v - \ell_s$, a binary string from a tree node in that level is assigned as the group's identifier, and all descendants of that tree node are deleted. By Kraft's inequality, we are guaranteed to not run out of tree nodes until all groups are assigned identifiers. A naive implementation of this description can result in an exponential running time, but a linear running time can be achieved by avoiding explicitly constructing the full tree, and instead only constructing each tree node as it is traversed.

5.3 Optimizing the Selection of Groups

Recall that Algorithm 1 takes a list of groups $\mathbb{S}$ and greedily merges them to minimize memory usage. The algorithm assumes that $\mathbb{S}$ does not require tags that are close to the maximum tag width in size, so that it is able to perform some meaningful number of merges to minimize memory. To aid this assumption, we can perform some preprocessing on $\mathbb{S}$ using our new knowledge of variable-length prefix identifiers. By Property 1 the number of groups and their identifiers should be selected such that their sizes $\ell_1,\ldots,\ell_M$ minimize the term $T = \sum_{i=1}^{M} 2^{\ell_i}$ f.

We begin with the input list of attribute groups $\mathbb{S} = s_1,\ldots,s_N$. Next, we consider each pair of groups, and calculate the benefit to the term T if that pair of groups were to be replaced by their union, decreasing the number of groups by 1. For two groups s_i, s_j of size ℓ_i, ℓ_j and intersection size ℓ_{ij}, the impact to T of replacing them with their union $s_i \bigcup s_j$ is equal to $-2^{\ell_i} - 2^{\ell_j} + 2^{\ell_{ij}}$. However, it may be the case that no pair of groups has an impact less than 0. In such a case, we choose the pair which has minimum impact to T, and replace them with their union anyway. We do this because it may be the case that such a step will enable beneficial union steps later on. After each union, we record the current value of T and the set of groups $\mathbb{S}$ that achieved this value. This process is repeated until there is only one group remaining. After one group remains, we go back and identify the set of groups $\mathbb{S}$ that achieved the minimum value of T, and return this as our answer. This process closely resembles Algorithm 1, with the exception that we are minimizing the sum T, rather than switch memory usage.

5.4 Handling Updates

We briefly touched on handling changes to sets at the end of the section 3, but we must take into account the addition of sequence encoding and variable-length identifiers. An update occurs when the attributes associated with a FEC change, and a new tag must be generated which encodes these attributes. Forwarding table entries may also need to change if any encoding group changes. For the case of unordered attributes, PathSets makes no changes to forwarding tables if the new set is a subset of an existing group. Only the tag associated with the updated FEC need change.

If the set is not a subset, we either attempt to merge the new set into an existing group, or create an entirely new group just for this set. New forwarding table entries need to be generated in both cases. In the latter case, the new group must be assigned a prefix-code identifier. If we do not have an unused prefix-code identifier, we can take the identifier I from an existing group and "split" it into two identifiers $I +' 0'$ and $I +' 1'$, if there is space. If there is not enough space, we must generate the encoding from scratch, and replace all forwarding table entries. As we said in section 3, this can be made unlikely by forcing each tag to have some amount of padding during the initial compilation.

For the case of ordered attributes, if the new sequence is a subsequence of the topological ordering of attributes discussed in section 4, then the sequence can be converted to a totally ordered set and treated as an update to an unordered encoding. We leave for the future work the case where a new sequence does not adhere to the topological ordering; currently this results in a full recomputation. In practice this can be avoided if all sequences are known in advance.

6 Evaluation

We now evaluate how the PathSets encoding performs on tag width and memory usage for the applications of service chaining and Software-Defined IXPs (SDX). Our implementation was written in about 1000 lines of Java. All experiments were run on a single laptop with a 2.5 GHz 4-core processor and 16GB of RAM. Full computation of an encoding scheme was under a few seconds in all cases.

6.1 Software-Defined IXPs

To evaluate the Software-Defined IXP setting, we must identify the characteristics of an IXP that ultimately determine tagging efficiency. For each packet in this setting, the attributes of a packet are the set of next-hop IXP participants which have announced routes to that packet's destination. Two packets share an FEC if they can take the same list of next-hops to exit the IXP switch. An attribute (next-hop) is queried if there is some policy which wants to forward traffic to that next-hop. Thus, to evaluate the proposed encoding algorithm over Software-Defined IXPs, we need (1) route announcements and (2) forwarding policies.

For route announcements, we used routing tables and messages from the AMS-IX exchange point [3], retrieved from the Routing Information Service raw data webpage [21]. This dataset includes 63 participating ASes advertising over 600,000 prefixes. To simulate smaller instances when determining growth functions, we randomly sample subsets of these IXP participants. We repeat this sampling procedure multiple times to average together multiple results. To simulate interdomain forwarding policies, we generate a fixed number of forwarding actions that each forward to a random IXP participant. Our evaluation only depends upon the destination of each forwarding policy, and not the decision logic, so a more realistic policy simulation is not needed. To evaluate update performance, we parsed 24 hours of BGP update messages. Our tests show that new routing table entries are virtually *never* needed, as the vast majority of changes do not result in expanding sets. Each update results in only a new tag for the associated FEC.

For the SDX [9] and iSDX [8] systems, tags are attached to the destination MAC field of packets, which has a limit of 48 bits. iSDX attaches additional information to the MAC field, leaving only 37 bits available for tagging. For PathSets to be used in iSDX, the tags must fit within 37 bits. We operate

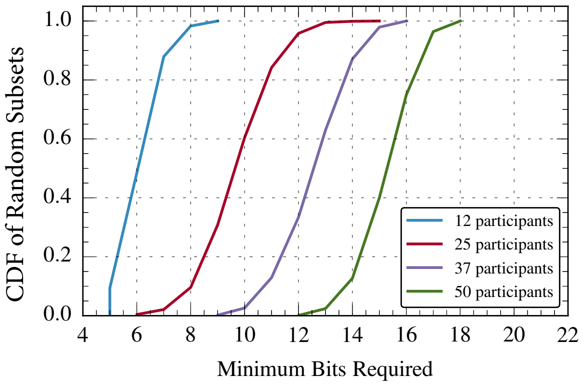


Figure 7: *Minimum number of bits required by a feasible solution for a random policy which forwards to a random subset of participants.*

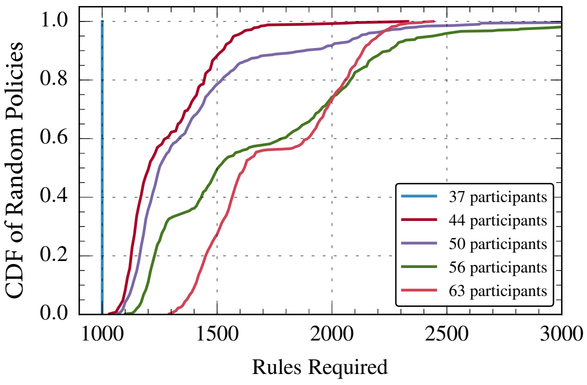


Figure 8: *Number of rules required after encoding for a random policy of 1000 rules to random subsets of participants.*

under this constraint when evaluating the number of rules required for the evaluation we present in this paper.

Figure 7 shows the number of bits required by PathSets with uniformly randomly chosen IXP participant subsets, repeated 500 times for each subset size. In the worst case, 18 bits were required when considering all 63 participants. The graphs appear to show that the number of bits required scales linearly with the number of participants present in the active set. If this is the case, extrapolating the data yields that PathSets can encode over 100 attributes in the worst case, allowing for very complex forwarding policies.

Figure 8 shows the number of rules required by PathSets after running the greedy algorithm up to the limit of 37 bits. In this experiment, we began with a baseline forwarding policy of 1000 rules, with each rule forwarding to a next-hop chosen uniformly at random from the set of all next-hops. The policy was then augmented with our tagging scheme's attribute testing strings.

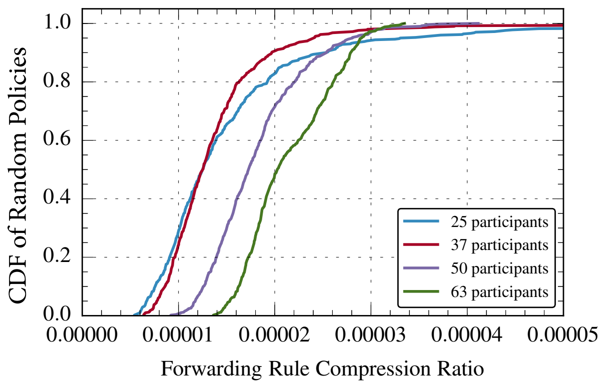


Figure 9: *Ratio of flow rules required by PathSets versus the uncompressed case on random policies involving random subsets of participants.*

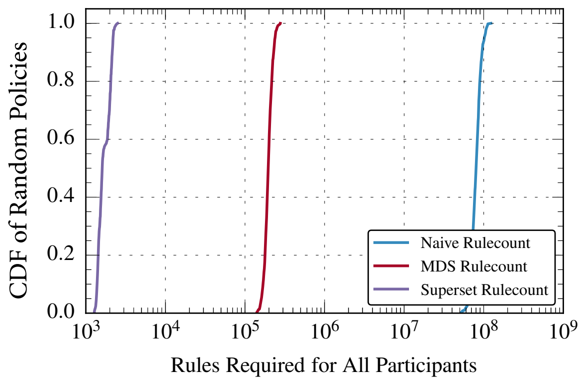


Figure 10: *Comparison of the number of flow rules required by PathSets to the previous state-of-the-art MDS encoding algorithm and the uncompressed case for a random policy involving all participants.*

Figure 9 shows how the number of flow rules generated by PathSets compares to the naive case of zero compression. The compression ratio of our approach versus the naive approach is 20,000 to 1 in the worst case for all active set sizes, and 50,000 to 1 in the median case.

Figure 10 compares PathSets to the uncompressed case, as well as to the previous state-of-the-art, the MDS algorithm used in the original SDX system [9]. The comparisons were all made using the same approach of generating 1000 random rules, with the exception of the MDS simulation. The MDS algorithm requires each prefix's default next-hop as part of the input, so in each trial we chose next-hops uniformly at random from the list of available next-hops. The graph shows that our approach consistently compresses the number of flow rules by two orders of magnitude greater than the MDS algorithm, which itself compressed the number of flow rules required by the naive case by three orders of magnitude.

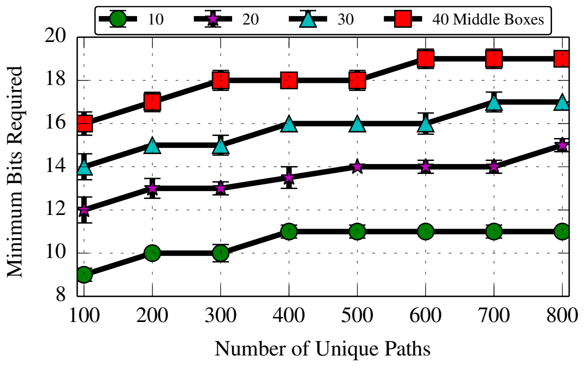


Figure 11: *Minimum number of bits required by PathSets to encode every service chain as the number of service chains increases. All paths generated had a 0.05 probability of reordering.*

6.2 Service Chaining

Recall that for the service chaining setting, attributes of an FEC are middleboxes that packets in the FEC must visit, and that these attributes are ordered. To evaluate service chains, we simulate a network policy by generating a set of random middlebox paths. The number of attributes in the service chaining application is equal to the number of distinct types of middleboxes. Sherry *et al.* [29] provide a lower bound of eleven middlebox categories, so we evaluate up to $M = 40$ middleboxes to emulate a reasonably sized network.

To generate random paths over these M middleboxes, we assume that all service chains follow a fixed underlying ordering. A random starting point in the ordering is chosen, and a path is constructed by making random jumps to middleboxes further down the ordering, stopping once the end of the ordering is reached. To simulate conflicting orderings, we post-process each random path to add ordering conflicts. For each pair of adjacent middleboxes in a path, we swap their positions with some probability p_{err}. We chose a value of 5% for p_{err}, as we believe reordering of middleboxes to be rare. We did not simulate changes to service chains, because we did not have a realistic model of how ordinary network events can affect middlebox paths.

Figure 11 evaluates the minimum widths required by PathSets across varying numbers of random paths and middleboxes. Recall that if there are P distinct paths, flat tagging requires $log_2(P)$ bits to encode every path. Since $log_2(800) = 9$, PathSets requires roughly twice the tag width of flat tagging for this experiment.

Figure 12 compares the number of rules required by PathSets to the number of rules required by flat tags. Since PathSets uses less memory when the tag is permitted to be wider, we evaluate the memory usage for tag widths of 20, 30, and 40. We observed that 20 bits was the minimum necessary for our tagging scheme on this dataset, as seen in Figure 11, so we begin at that width. Flat tagging does not have any tunable parameters for making a width-memory tradeoff, so it is only evaluated once.

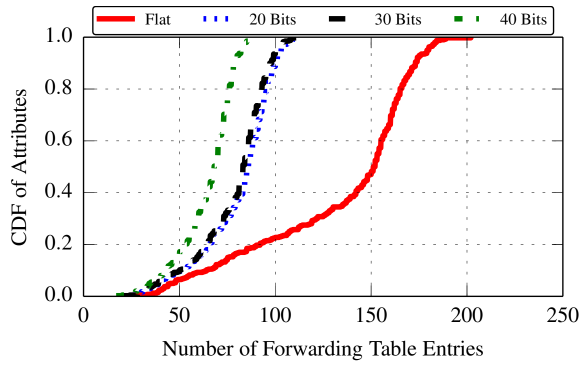


Figure 12: *Number of forwarding table entries required for tags of different widths, over a distribution of paths with different sets of attributes, compared to the number of entries required by a flat tagging scheme.*

The number of rules needed by PathSets is less than half that used by flat tagging in this application. While this is an improvement, we suspect the benefits will be much greater for real datasets, as there should be more redundancy in the paths for PathSets to exploit.

7 Related Work

Software-defined Internet exchange points. The first SDX [9] prototype in 2014 used a flat tag to identify the set of valid next hops (*i.e.*, BGP neighbors) that can direct traffic to a destination prefix to avoid installing forwarding table entries for each destination IP prefix in the IXP switch; this prototype SDX controller proactively installed rules to decode each tag. The follow-up industrial scale iSDX [8] project reduced forwarding-table size and rule churn to make operation at large IXPs practical. The iSDX uses the technique for encoding sets of attributes described in Section 3, although our iSDX paper did not describe the encoding scheme in much detail. In addition, the iSDX design supports neither the encoding of sequences of attributes (Section 4) nor the optimizations possible with variable-length group identifiers (Section 5).

Service chaining with flat tags. Recent works on traffic steering through middleboxes uses a flat tag to identify the service chain each packet should traverse. FlowTags [6] is a prominent example, where the first middlebox in a path tags the packet with the rest of the service chain. Rules to decode each tag are installed reactively when switches see a new tag for the first time. In contrast, PathSets results in fewer rules, and allows for proactive rule installation. Incorporating PathSets would make FlowTags scale to larger sequences of middleboxes when switches have limited table size.

Encoding end-host attributes in IP addresses. Alpaca [14] embeds the attributes of end hosts into the packet header with the goal of easing network policy enforcement. Alpaca focuses on a special case where attributes are embedded in the least significant bits of the IP address. This mechanism requires a unique attribute encoding for each host. In addition, Alpaca does not support the encoding of sequences of attributes. As such, PathSets can solve the problem introduced in Alpaca, but the reverse is not true.

Compressing Forwarding Entries in Routing Tables The topic of memory efficient representation of network policies such as QoS and forwarding information is well studied. The policies include matching rules examining packet fields and are associated with an action that has to be applied on the matching traffic. Various compression schemes rely on properties of the special memory like TCAM [16, 24, 27] or SRAM [28]. These schemes also take advantage of the type of represented information (e.g., rules that allow a range of consecutive values, disjointness of rules) [15]. It was shown that schemes can often achieve compression rates close to theoretical lower bounds from Information theory [20]. Our work achieves the same of goal of decreasing switch memory usage, but by compressing information in packets, rather than in switches.

Forwarding with flat tags. Throughout the paper we have compared PathSets to flat tagging. MPLS [22] is a well-established source-routing protocol for forwarding packets by writing and matching on flat labels in packet headers. By default, MPLS tags only instruct the packet how to reach the next node in a path. To use MPLS across multiple hops in succession, MPLS performs label swaps, where the tag is swapped out for a new tag. MPLS can extended to support segment routing [7], by having each label represent a path segment, rather than a single hop. While powerful, an issue with MPLS is that labels are often stacked for path aggregation, causing packets to have variable length as they are forwarded. The Path Switching work [10] presents an alternative to MPLS for source routing which has the advantage of encoding forwarding information in a fixed amount of existing space in the packet headers. PathSets can be used for this purpose, if there is redundancy in the paths chosen.

Encoding sets concisely in Bloom filters. A Bloom filter [4, 25, 26] is a common data structure that represents a set of items with a fixed amount of memory. A Bloom filter supports membership queries but suffers from false positives, where some elements can be wrongly reported as members of the set. In contrast, PathSets has no false positives. Although minimizing the tag width is an key property of our scheme, the memory for a Bloom filter is several times larger than the number of elements—10-20 times for a false-positive probability of 0.01–1%. Bloom filters also require each value to be hashed into a large bit vector, which might not be possible in some switch architectures.

Efficient coding using prefix codes. Variable-length prefix codes have been used for various applications. The seminal work of Huffman [13] describes an algorithm for an optimal selection of prefix codes for lossless compression of source symbols. The selection minimizes the encoding length by using fewer bits for common symbols, achieving results close to lower bounds from information theory. More recently, prefix codes were suggested as a way to encode paths, while reducing the maximum length of any encoded path [11]. A similar approach was suggested for the encoding for fixed-width memories [23]. In all of these schemes, the encoding concatenates the codes of the attributes and thus is often long when the number of attributes is large. In contrast, PathSets combines input sets or sequences with common attributes (or common orderings of attributes) and uses prefix codes for the group identifiers to further reduce tag size.

8 Conclusion

Many network architectures and mechanisms, from SDN-based exchange points to middlebox service chains, encode sets or sequences in forwarding equivalence classes. Previous work has generally encoded each FEC with a flat tag, which is amenable to exact matching but scales poorly as the size of a set or the number of unique orderings increases. In this paper, we propose PathSets, a mechanism that takes advantage of the ability of commodity switches to perform wildcard matching on arbitrary packet header fields; this capability, which has been enabled by protocols such as OpenFlow 1.3, facilitates more efficient encodings that allow aggregation of similar FECs, through wildcard-based encoding.

Our evaluation for two deployment scenarios—for service chaining and an SDN-based IXP—demonstrates that a wildcard-based encoding can reduce the number of forwarding table entries in each switch by several orders of magnitude, given typical levels of redundancy in the sets or sequences that are typical in forwarding policies for these scenarios. PathSets can capture both unordered sets and ordered sequences; it can also efficiently capture rare perturbations to sequences that appear more frequently.

The PathSets encoding algorithm reduces forwarding table size, at the cost of larger tags. A possible avenue for future work involves exploring this tradeoff more thoroughly, as well as exploring alternative ways to handle attribute sequences that have conflicting attribute orderings. Every application that we presented relies on a different method to attach tags to packets. To make it easier to tag packets for arbitrary network deployment scenarios, we plan to investigate more general techniques for tagging packets that could apply to a wider range of use cases.

Acknowledgments The authors would like to thank the anonymous SOSR reviewers for their valuable comments and helpful suggestions. We would also like to thank Rob Harrison for his feedback, as well as Naman Agarwal, Brian Bullins, and Karan Singh for their insight on the algorithmic challenges. This work was supported in part by the NSF under grants CNS-1409056 and CNS-1539920.

References

[1] 802.1 Q/D10, IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks, 1997. (Cited on page 1.)

[2] N. Abramson. *Information Theory and Coding*. McGraw-Hill, 1963. (Cited on page 7.)

[3] AMS Internet Exchange. `https://www.ams-ix.net/ams-ix-route-servers/`, 2013. (Cited on page 9.)

[4] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 1970. (Cited on page 11.)

[5] 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 Computer Communication Review*, 44(3):87–95, 2014. (Cited on page 2.)

[6] 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 *USENIX NSDI*, 2014. (Cited on pages 1, 3 and 11.)

[7] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois. The segment routing architecture. In *IEEE GLOBECOM*, 2015. (Cited on page 11.)

[8] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever. An industrial-scale software defined internet exchange point. In *USENIX NSDI*, 2016. (Cited on pages 2, 3, 9 and 11.)

[9] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A software defined internet exchange. In *ACM SIGCOMM*, 2014. (Cited on pages 1, 2, 9, 10 and 11.)

[10] A. Hari, T. Lakshman, and G. Wilfong. Path switching: reduced-state flow handling in sdn using path information. In *ACM CoNEXT*, 2015. (Cited on page 11.)

[11] A. Hari, U. Niesen, and G. T. Wilfong. Optimal path encoding for software-defined networks. In *IEEE ISIT*, 2015. (Cited on page 12.)

[12] R. Hassin and S. Rubinstein. Approximations for the maximum acyclic subgraph problem. *Inf. Process. Lett.*, 51(3):133–140, 1994. (Cited on page 6.)

[13] D. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, 1952. (Cited on page 11.)

[14] N. Kang, O. Rottenstreich, S. Rao, and J. Rexford. Alpaca: Compact network policies with attribute-carrying addresses. In *ACM CoNEXT*, 2015. (Cited on pages 3 and 11.)

[15] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. T. Eugster. Exploiting order independence for scalable and expressive packet classification. *IEEE/ACM Trans. Netw.*, 24(2):1251–1264, 2016. (Cited on page 11.)

[16] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: a systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Trans. Netw.*, 18(2):490–500, 2010. (Cited on page 11.)

[17] OpenFlow 1.3 specifications. `http://bit.ly/1eyrkxY`. (Cited on page 2.)

[18] PathSets GitHub Repo. `https://github.com/PrincetonUniversity/PathSets`. (Cited on page 2.)

[19] K. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is np-complete. *Theor. Comput. Sci.*, 16:187–198, 1981. (Cited on page 6.)

[20] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM*, 2013. (Cited on page 11.)

[21] RIPE Routing Information Service (RIS). `http://www.ripe.net/ris`. (Cited on page 9.)

[22] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture, 2001. RFC 3031. (Cited on pages 1 and 11.)

[23] O. Rottenstreich, A. Berman, Y. Cassuto, and I. Keslassy. Compression for fixed-width memories. In *IEEE ISIT*, 2013. (Cited on page 12.)

[24] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. Exact worst case TCAM rule expansion. *IEEE Trans. Computers*, 62(6):1127–1140, 2013. (Cited on page 11.)

[25] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting bloom filter. *IEEE/ACM Trans. Netw.*, 22(4):1092–1105, 2014. (Cited on page 11.)

[26] O. Rottenstreich and I. Keslassy. The bloom paradox: When not to use a bloom filter. *IEEE/ACM Trans. Netw.*, 23(3):703–716, 2015. (Cited on page 11.)

[27] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. Optimal in/out TCAM encodings of ranges. *IEEE/ACM Trans. Netw.*, 24(1):555–568, 2016. (Cited on page 11.)

[28] 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*, 32(1):138–151, 2014. (Cited on page 11.)

[29] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012. (Cited on page 10.)