Memory-Efficient Membership Encoding in Switches

Mengying Pan
mengying@cs.princeton.edu
Princeton University

ABSTRACT

Network applications often define policies to manage network traffic based on its attributes. For example, service chaining forwards traffic to reach the middleboxes it wants to visit, and access control restricts traffic by checking the permission flags it carries. These policies match against packets' attributes in switches before being applied. However, the prior works of attribute encoding all incur a high memory cost to identify the attributes in the data plane. This paper presents MEME, an encoding scheme that clusters the attributes which tend to appear together in the traffic to reduce the memory usage. Naive clustering would still fail since it is ineffective when a cluster contains an excessive number of attributes. To tackle this, MEME breaks the clusters into smaller ones by encoding a minimal number of attributes separately and by taking advantage of the special structures within the attributes. MEME also leverages match-action tables and reconfigurable parsers on modern hardware switches to achieve a final 87.7% lower memory usage, and applies an approximate graph algorithm to achieve 1-2 orders of magnitude faster compilation time than the prior state of the art [13]. These performance gains pave the way for deployment of a real traffic management system desired by the world's largest Internet Exchange Points.

1 INTRODUCTION

With the rise of SDN switches come new opportunities in managing traffic based on sophisticated policies rather than conventional routing protocols. Some examples include:

- *Service Chaining* [7]: Service chaining involves having network traffic traverse a sequence of middleboxes. To traverse flexibly, each packet carries in its header the set of middleboxes to visit.
- *Software-Defined Internet Exchange Points (SDX)* [9, 10]: At an Internet exchange point (IXP), hundreds of autonomous systems (AS's) exchange routing information and network traffic. A traditional IXP uses BGP to select a single next-hop AS from the set of AS's that announced routes to an IP destination. In contrast, an SDX allows AS's to define finer-grained policies using various packet-header fields to choose next-hops from those available.

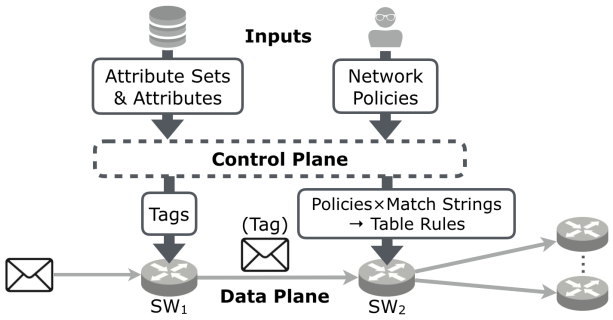


Figure 1: Membership encoding in applications.

- *Traffic Management with Host Attributes* [11]: Network operators often want to apply access-control or quality-of-service policies based on user groups. To achieve this, each packet is tagged with a set of user attributes, which the policies match on.

In these applications, a packet travels with a set of attributes (e.g., middleboxes, next-hop AS's, or host attributes). Policies match on these attributes to make routing, quality of service, or access control decisions. Identifying which attributes the packet carries, also known as *membership encoding*, is a fundamental building block of these functions.

A membership encoding scheme encodes each attribute set as a *tag* and each attribute as one or more *match strings* in the control plane. It guarantees that a set contains an attribute if and only if the tag matches any of the attribute's match strings. Thus, a membership encoding scheme has no false positives, unlike a Bloom filter [2, 16] which can falsely report some attributes as set members.

In order to identify packet attributes in the data plane, each packet is assigned a tag that represents its attribute set. Network policies are then combined with match strings and compiled into a switch match-action table as rules that query the existence of attributes before applying policies. In Figure 1, a packet is first tagged by the switch SW_1 before traversing the switch SW_2 that contains the compiled policies. SW_2 parses out the tag and compares it with match strings before applying a policy to the packet. This design can be adopted by any switches that have customizable match-action processing, such as OpenFlow switches [14].

However, due to the limited parsing capability of SW_2, tags must be short enough to parse at line rate. Moreover, due to the limited memory of SW_2, match strings should require

as little memory as possible to fit in the match-action table. For instance, SW_2 in an SDX is the IXP fabric, which installs interdomain forwarding policies defined by hundreds of AS's. If the SDX simply uses the IP destination to tag the set of next-hop AS's and the IP prefixes, of which there are over 500,000, as match strings, it takes at least half a million rules in total for a single AS to define one forwarding policy for each peer AS, overwhelming even high-end switches [10]. SW_1, in contrast, may use existing tables to assign tags, such as an edge router's ARP table, so SW_1 is not the bottleneck in resource constraints.

Prior works [7, 9, 10, 13] in membership encoding have succeeded in lowering tag width, but they all incur a high memory cost in the switches because they generate a large number of match strings. Our evaluation shows that with the prior state-of-the-art scheme, PathSets [13], only a limited number of policies fit in commodity switches. There have been works on reducing forwarding entries in TCAM tables to reduce memory [5, 12, 17, 20], but these techniques do not shorten tags at the same time. They generate semantically equivalent TCAM tables with fewer rules, while our work constructs an encoding scheme directly from attribute membership.

To improve the scalability of membership encoding, we present *MEME*, a **M**emory-**E**fficient **M**embership **E**ncoding scheme. It reduces tag width and optimizes the number of match strings at the same time, reaching the minimal memory cost compared to all the existing works on real-world datasets in SDN switches. Moreover, capitalizing on the emerging PISA (Protocol Independent Switch Architecture) switches capable of parsing an arbitrary portion of headers [3], MEME further cuts the memory cost by splitting a big match-action table into multiple smaller ones. Finally, MEME computes the encoding fast in the control plane, finishing within seconds on our largest dataset. We evaluated MEME on a 691-attribute dataset from the world's largest IXP, showing that MEME cuts the memory cost of match strings by 87.7% and the computation time by one order of magnitude compared to PathSets.

2 MEMBERSHIP ENCODING PROBLEM

In this section, we formalize the membership encoding problem using an SDX example. At an SDX, AS's can define policies to forward packets to specific next-hop AS's. For instance, an AS may want to forward TCP traffic to different next-hops based upon the service type (i.e., TCP ports). One policy can be to forward all HTTP traffic to AS A (Figure 2b). However, A may not have a BGP route for every destination IP prefix. Thus, the policy also needs to check whether A exists in S, the set of next-hop AS's that announce the packet's destination IP prefix.

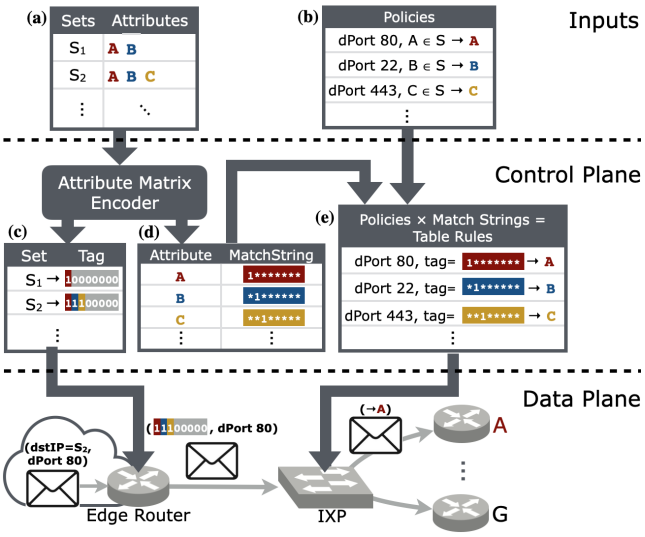


Figure 2: The overall information flow of the bitmap encoding scheme at an SDX.

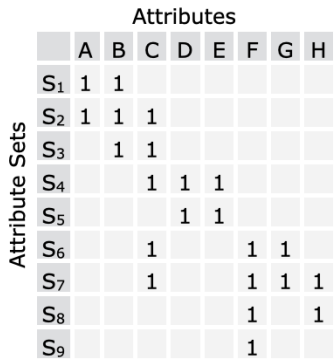


Figure 3: A 9×8 attribute matrix.

2.1 Input: Attribute Matrix

The input to a membership encoding problem can be formalized as an attribute matrix, where each column represents an attribute, and each row represents a possible set of attributes. For instance, Figure 3 is an attribute matrix of 9 attribute sets $S_1 = \{A, B\}$, $S_2 = \{A, B, C\}$, $S_3 = \{B, C\}$, $S_4 = \{C, D, E\}$, $S_5 = \{D, E\}$, $S_6 = \{C, F, G\}$, $S_7 = \{C, F, G, H\}$, $S_8 = \{F, H\}$ and $S_9 = \{F\}$. We define the matrix width (the number of attributes) as N, height (the number of attribute sets) as M, and density as D.

Figure 2a shows a truncated attribute matrix in the context of the SDX. Its columns are all the AS's connecting to the IXP; its rows are the sets of AS's that reach the same IP prefix. Such a matrix from a real IXP can contain $O(10^3)$ columns and $O(10^6)$ rows, but its density can be less than 1%—setting the stage for compression in membership encoding.

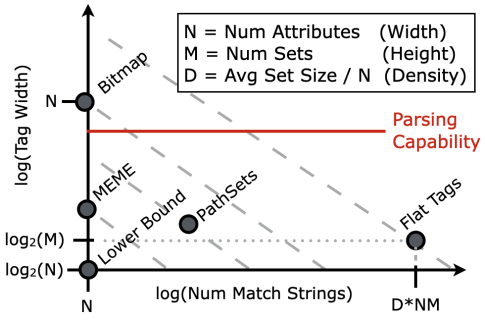


Figure 4: Traditional solution space w/o PISA switches on a log-log scale. Solutions on the same dashed line require the same memory , and solutions closer to the origin require less memory.

2.2 Output: Packet Tags and Match Strings

Given an attribute matrix, each row is encoded as a *tag* and each column as one or more *match strings* (Figure 2c,d). A row's tag matches one of a column's match strings if and only if the corresponding cell has a value of 1.

The control plane *augments* policies with the match strings to generate rules in the match-action table. A policy querying for an attribute is converted to match-action rules that match the tag with the attribute's match strings. For example, the HTTP policy inspecting the set S of next-hop AS's for A in Figure 2b is turned into a rule that matches the dPort and the tag with A's match string in Figure 2e. If an attribute requires multiple match strings, the table rules must be duplicated with every match string individually. This duplication increases the memory requirements, so it is preferable that each attribute has only one match string.

A packet is first given a tag at an edge router, either as a new field or an existing one (e.g. dstMAC in the SDX [9, 10]). In the network, the packet eventually enters a switch that implements the match-action table augmented from policies. There, the tag is parsed and, together with other header fields, matched with the table rules. Figure 2 shows that an HTTP packet gets the tag of $\{A, B, C\}$, "11100000", from the edge router. After entering the IXP fabric, the packet's tag matches the first rule and is forwarded to A.

Traditionally, match strings are assumed to be of the same width as tags, so the memory cost of match strings is the product of tag width and the number of match strings. This generates the solution space in Figure 4, where the lower bound of tag width is the entropy $\log_2(N)$ bits, the lower bound of the number of match strings is N, and the lower bound of memory cost is the origin $N \log_2(N)$ bits. Minimizing tag width and minimizing the number of match strings give the two strawman approaches.

Bitmap encoding translates each matrix row to a tag with a one bit for the contained attributes and zero bit for the others. Each column of the matrix requires one ternary match string with a one bit for itself and wildcards for the others. This scheme is optimal in the number of match strings. However, it suffers from prohibitively long tags, of the same width N as the matrix. As seen in Figure 4, N^2 bits are needed in total to store match strings, which correspond to 64 bits even for the very small attribute matrix in Figure 3.

Flat tags are adopted in SDX [10] and FlowTags [7]. The flat tag scheme encodes each unique row with an ID. In Figure 3, since no rows are duplicated, their tags are simply the row numbers starting from 0. This scheme achieves a small tag width upper bounded by $\lceil \log_2 M \rceil$ bits. However, for each attribute, its match strings include the IDs of all the rows that contain the attribute, leading to an enormous number of match strings. As a result, when D is the matrix density, $DNM\lceil \log_2 M \rceil$ bits are needed in the worst case. For instance, Figure 3 requires 22 match strings and 88 bits.

The two strawman approaches are points in the solution space, optimal in some aspect but poor in total memory. A key way to remedy this issue is clustering matrix rows, to trade off short tags for a smaller number of match strings.

A clustering-based encoding scheme: (i) clusters multiple rows in the attribute matrix, (ii) assigns each cluster a unique ID, and (iii) generates tags and match strings in the form of concatenation of a cluster ID and a bitmap of the same width as the cluster. (The *cluster width*, in analogy to matrix width, is the number of attributes in all rows of the cluster.) The key feature of the clustering schemes is that they generate one match string for every attribute in a cluster. An attribute's match string is a concatenation of the cluster ID and the bitmap whose bit is one for the attribute and wildcard for the others. A row's tag is a concatenation of the cluster ID and the bitmap whose bit is one for the attributes in the row and wildcard for the others. For practical reasons, all tags are padded with zeros and match strings with wildcards to the same length (represented as "-" to avoid confusion), so the tag width takes the maximum of (cluster width + cluster ID length) among all the clusters. For example, in Figure 5 with three clusters, $\mathbb{C}_1 = \{S_1, S_2, S_3\}$, $\mathbb{C}_2 = \{S_4, S_5\}$ and $\mathbb{C}_3 = \{S_6, S_7, S_8, S_9\}$, if we use the IDs "00", "01", "10" respectively, the tag of S_3 is "00|011−", the tag of S_4 is "01|111−", the tag of S_6 is "10|1110" and the match strings of C are "00|$**1-$" in $\mathbb{C}_1$, "01|1$**-$" in $\mathbb{C}_2$ and "10|1$***" in $\mathbb{C}_3$.

2.3 Related Work: Clustering Matrix Rows

PathSets [13], adopted by iSDX [9], relies on clustering. PathSets starts by treating every row as a cluster. It uses a greedy algorithm to iteratively merge *intersecting clusters*, i.e., clusters that share at least one attribute, until no merging results in a memory reduction. In general, clustering reduces the number of match strings by generating only one match string for each attribute in a cluster, while at the potential

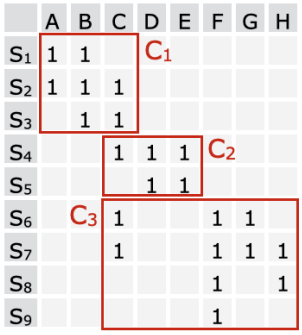
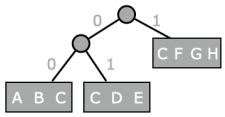


Figure 5: PathSets clustering.

cost of increasing the cluster width and hence the tag width. PathSets follows a path from flat tags (every row as a cluster eliminates bitmaps) to bitmaps (clustering all rows eliminates IDs), finding the solution closest to the origin in the middle. For the example matrix, this yields the clustering rendered in Figure 5.

PathSets also proposes the variable-length cluster ID algorithm to shorten tag width. Instead of using IDs of the same length, it assigns larger clusters, which need longer bitmaps, shorter IDs, and vice versa, in order to reduce the total tag width. Given a list of m clusters of size $l_1, l_2, \ldots, l_m$, the minimum tag width is $w = \lceil \log_2 \sum_{i=1}^{m} 2^{l_i} \rceil$ bits based on Kraft's Inequality [1, 13]. Since encoding a cluster i requires a bitmap of l_i bits, its cluster ID must be no longer than $(w - l_i)$ bits. The algorithm builds a binary tree from top down and places each cluster i as a leaf node on or above the level $(w - l_i)$ along the way. By assigning the left branches with bit 0 and the right branches with bit 1, a path from the root node to a leaf node yields the ID for its cluster (Figure 6a). With variable-length IDs, PathSets generates ten 5-bit match strings, taking 50 bits in total (Figure 6b).

Nevertheless, as the matrix grows, the memory requirement of PathSets still overwhelms commodity switches. Section 5 shows that PathSets only supports 108 out of 691 AS's to define one policy for each AS in the IXP. Also, the greedy clustering algorithm runs M iterations, and for each iteration, considers every pair of intersecting clusters for merging. Thus, PathSets has time complexity $O(NM^3)$, becoming extremely slow as the matrix size grows.



(a) Binary ID tree

	Attributes	Cluster width	Fix-len ID	Var-len ID
C_1	A, B, C	3	00	00
C_2	C, D, E	3	01	01
C_3	C, F, G, H	4	10	1

(b) Cluster IDs

Figure 6: Tags width is 6 bits with fixed-length cluster IDs, and 5 bits with variable length IDs.

3 MEME CLUSTERING ALGORITHM

No current approaches scale well for all three metrics: tag width, memory cost, and computation time. MEME achieves significant scalability gains by capitalizing on common properties of attribute matrices. MEME lowers the tag width while always generating the optimal number of match strings.

3.1 Extracting Bridging Attributes

The minimum possible number of match strings is N since there must be at least one string for each column in the matrix. To achieve this, MEME first conducts *complete clustering*, i.e., MEME iteratively clusters any intersecting attribute sets until all clusters are mutually exclusive in columns. This guarantees that each attribute appears in only one cluster. Since every attribute in a cluster has only one match string, complete clustering leads to N strings.

However, complete clustering can force some clusters to be very wide. As a result, the tag, consisting of a cluster ID and a bitmap of the same width as the cluster, can become unacceptably long. In Figure 3, complete clustering outputs the entire matrix as a cluster, falling back to the bitmap scheme with 8-bit tags.

We found that the "culprit" of the wide clusters is a few attributes that "bridge" over an excessive number of distinct sets. The *bridging attributes* make some clusters extremely wide. For instance, C is the bridging attribute in Figure 5, forcing the merging of $\mathbb{C}_1$, $\mathbb{C}_2$ and $\mathbb{C}_3$. This observation also holds in real-world matrices. In our IXP matrix, complete merging produces only three clusters, the largest one of which has 563 attributes resulted from 147 bridging attributes.

To handle large clusters after complete clustering, MEME identifies and extracts the fewest bridging attributes from the matrix with a minimum vertex cut algorithm to break up the clusters.

MEME first converts the attribute matrix into a graph, where each attribute is a vertex, and an edge exists between the vertices of every pair of attributes in a row. The left of Figure 7 depicts the graph of the matrix found in Figure 3. It is easy to see that a row in the matrix is a clique in the graph. We claim that *each cluster from complete clustering of the matrix corresponds to a connected component in the graph.* To explain that, we note that any row in the cluster shares at least one attribute with some other row in the cluster by the definition of complete clustering. This means that the cliques

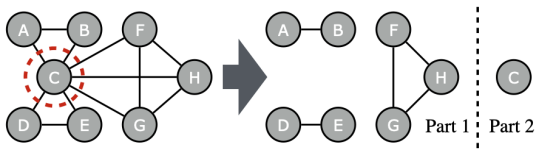
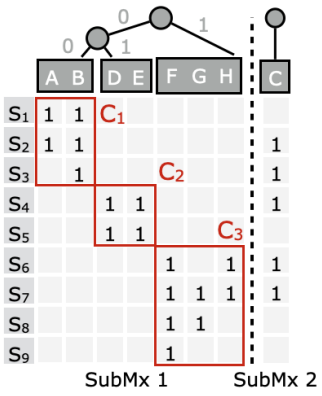


Figure 7: Extraction of minimum vertex cut.



(a)

Attr Sets→Tags

		Tag			
		Subtag1		Subtag2	
		ID	bMap	ID	bMap
S1	A B	00	11	0	
S2	A B C	00	11	1	
S3	B C	00	01	1	
S4	C D E	01	11	1	
S5	D E	01	11	0	
S6	C F G	1	110	1	
S7	C F G	1	111	1	
S8	F H	1	101	0	
S9	F	1	100	0	

Attr→Match Strings

	String			
	Substr1		Substr2	
	ID	bMap	ID	bMap
A	00	1*	*	
B	00	*1	*	
C		****	1	
D	01	1*	*	
E	01	*1	*	
F	1	1**	*	
G	1	*1*	*	
H	1	**1	*	

(b)

Figure 8: Extracting the bridging attribute yields the 2 submatrices in 8a, which are encoded in 8b.

from all rows in the cluster are connected. On the other hand, for any attribute X from a cluster and any attribute Y from another cluster, X and Y never appear together in a row, again by the definition of complete clustering. This means that the subgraphs of any two clusters are disconnected.

Therefore, the original problem of breaking up a cluster by extracting the fewest bridging attributes is converted to disconnecting its connected component by extracting the minimum number of vertices, i.e., the minimum vertex cut. As seen in Figure 7, the minimum vertex cut of the left graph is indeed the bridging attribute C. If there are multiple minimum vertex cuts, MEME picks the one that, if extracted, produces the most components. MEME extracts the minimum vertex cut iteratively from the graph till all the components have fewer than λ vertices. Equivalently, all the clusters have fewer than λ attributes at the end of this process.

This greedy algorithm extracts the minimum number of bridging attributes to make all clusters bounded in size. The bridging columns are then combined into a new submatrix, which is the input to MEME again. This process repeats until the input submatrix width is below λ. In the example, by choosing $\lambda = 5$, extracting C yields two submatrices, one with three clusters $\mathbb{C}_1 = \{A, B\}$, $\mathbb{C}_2 = \{D, E\}$ and $\mathbb{C}_3 = \{F, G, H\}$, and the other with one cluster $\{C\}$ (Figure 8a).

Each submatrix is then encoded separately to generate *subtags* for each row and *match substrings* for each column. Figure 8b shows the subtags and match substrings from Figure 8a. MEME adopts the variable-length cluster ID algorithm from PathSets, assigning a 1-bit ID to $\mathbb{C}_3$ and 2-bit IDs to $\mathbb{C}_1$ and $\mathbb{C}_2$. Since the cluster in the second submatrix has a single attribute, no bitmap is needed. Also, the match substrings for attributes not in the submatrix are simply wildcards. In the end, all subtags of a row are concatenated to construct its full tag; all match substrings of a column are concatenated to construct its full match string.

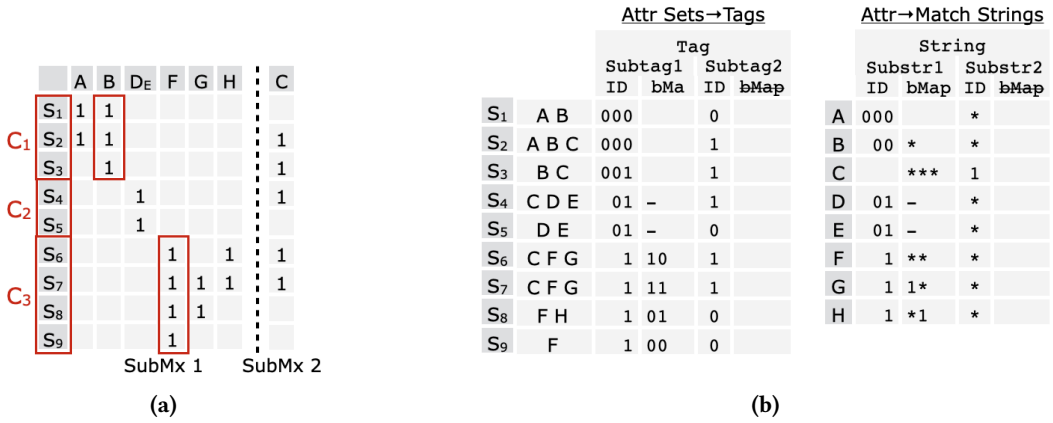
Extracting bridging attributes, while keeping the clusters mutually exclusive and the number of match strings optimal, avoids huge clusters inflating the tag width. Even though the new submatrices require multiple subtags to fully represent the entire rows, the total width becomes much smaller than after naïve complete clustering. The tags in Figure 8b are 5 bits, 3 bits fewer than before, and the improvement is much more profound when the matrix is larger. The current tag width is the same as produced by PathSets in Section 2.3, and since only one string is required for each attribute, the total memory cost is already smaller than PathSets'.

3.2 Sibling and Ancestor Attributes

In addition to extracting bridging columns, MEME exploits two special relationships which can be found among some columns in the attribute matrix to further shorten the tags: (i) *sibling* columns that are identical and (ii) *ancestor* attributes that exist in every row of a cluster. Though occurrences of these relationships are matrix dependent, they are not uncommon. In Figure 8a, column D and E are siblings, column B is the ancestor of the cluster $\mathbb{C}_1$ and column F is the ancestor of $\mathbb{C}_3$.

MEME takes advantage of these two relationships to further reduce the tag width. Given an attribute matrix, MEME detects all siblings and keeps only one in each sibling group. Since the siblings always appear together in a row, they can be treated as one attribute, identified with one match string.

In addition, MEME makes use of cluster IDs to encode ancestors implicitly. The algorithm from Section 3.1 is modified so that upon encountering a large cluster that has an ancestor, MEME removes the ancestor. Then MEME assigns hierarchical cluster IDs so that all the rows in the cluster share a common cluster ID prefix, which does not overlap with any other cluster's ID prefix. Since the ancestor exists with and only with all the rows in its cluster, MEME encodes the ancestor implicitly with that common ID prefix.



(a) (b)

Figure 9: Merging the siblings yields 9a, which is encoded in 9b with hierarchical cluster IDs for the ancestors.

Figure 9a depicts these two techniques, and Figure 9b lists the resulting tags and match strings. The siblings D and E are merged, sharing the same match string "$01-*$". Also, after removing the ancestor F from $\mathbb{C}_3$, all the rows in $\mathbb{C}_3$ share a cluster ID prefix, "1", so MEME encodes F as "$1***$". Indeed, this matches the tags of any attribute set that contains F, namely $\{S_6, S_7, S_8, S_9\}$. Similarly, the ancestor B is encoded as "$00**$", matching the tags of $\{S_1, S_2, S_3\}$.

Before describing how to assign such hierarchical cluster IDs, we take a look at the overall algorithm of MEME. Combining the designs on bridging, sibling and ancestor attributes, Algorithm 1 shows the full pseudocode. After removing the siblings from the input matrix (Line 3), MEME splits every submatrix into clusters by removing ancestors and extracting bridging attributes until all the resulting clusters are below the threshold λ and do not contain any ancestors (Line 16-33). Compared to bridging attributes which are encoded explicitly in new submatrices, siblings and ancestors are removed and encoded implicitly. MEME reduces the memory cost of the example matrix to 32 bits with eight 4-bit match strings.

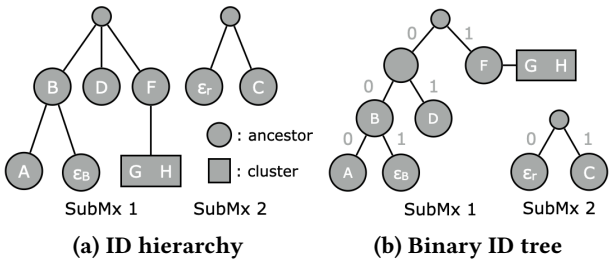
3.3 Hierarchical Cluster IDs

Variable-length IDs have effectively shortened tags, but the algorithm from PathSets [13] does not take into account the ID hierarchy required by the ancestors. To remedy that, MEME constructs the ID hierarchy and binarizes it to assign IDs of vairable length.

For each submatrix, MEME constructs a tree of ancestors and clusters so that every one of them is the child node of the previously removed ancestor in Algorithm 1 (Line 19, 26). For example, MEME removes the ancestors B and F from the matrix in Figure 9a and generates a tree as shown in Figure 10a. This tree also corresponds to the hierarchy of IDs in that the ID of an node is the prefix of another if and only if the latter is the descendant of the former in the tree. For instance, the ID of $\{G, H\}$ must match F's IDs but not B's or D's. A single-element cluster, such as $\{A\}$ and $\{D\}$, is regarded as an ancestor with width equal to 0.

One case that requires special handling is a matrix row consisting solely of ancestors. After removing the ancestors, such a row becomes empty, like S_3 and S_9 in Figure 8a. There are two situations. If the final ancestor removed has in the hierarchy any direct child that represents a cluster, the empty row can be encoded by its ID with an all-zero bitmap. For example, S_9 is encoded with the ID of $\{G, H\}$, "1", leading to the tag "1000". However, if the final ancestor removed only has ancestor children, that empty row cannot be encoded with any of their IDs since that would falsely imply the existence of the child. For example, since B only has an ancestor child, A, encoding S_3 with its IDs would match the match string of A. To handle this, an ϵ node is added to the final ancestor to assign a distinct ID to the empty row (Line 21-22, 27), such as ϵ_B in Figure 10a. Similarly, an ϵ node is added to the root as the place holder for the empty set if needed (Line 8- 9), such as ϵ_r in Figure 10a.

After building the ID hierarchy, MEME uses the variable-length ID algorithm as a subroutine to binarize it from bottom up. When binarizing the children of an ancestor, this portion of the hierarchy has a known minimum tag width and is equivalent to a cluster of the same size as its tag width for



(a) ID hierarchy (b) Binary ID tree

Figure 10: Tags width is 6 bits with fixed-length cluster IDs, while with variable length IDs it is 5 bits.

further binarization at the upper node. In Figure 10a, the portions under B and F, after binarization, are regarded as two "clusters" of size 1 and 2 respectively, and D, as an ancestor, has width 0. Then binarizing those three children at the root gives the minimum tag width of 3 bits. With B placed at or above the level 2, D at or above the level 3, and F at or above the level 1, a binary tree is generated to assign IDs to every ancestor and cluster (Figure 10b).

3.4 Computation Optimization

Finding bridging attributes depends on the minimum vertex cut algorithm [6]. On a graph $G = \langle V, E \rangle$, this algorithm calls a subroutine that finds the minimum vertex cut between a pair of vertices. This subroutine, of time complexity $O(|E||V|^{2/3})$, is run on $O(|V|)$ pairs of vertices to search for the actual minimum vertex cut for G. Thus, its total time complexity is $O(|E||V|^{5/3}) = O(HN^{8/3})$, where $H = DN$ is the average set size, which can be treated as a constant due to the sparse nature of the matrix. Thus, in the worst case, the time complexity of MEME is $O(N^{11/3})$. Even though this is a tighter bound than PathSets since $M >> N$ for attribute matrices in practice, it is still slow when N increases.

To speed this up, we observe that due to the sparsity of matrices, the minimum vertex cut size is small for large graphs. Therefore, when the graph size is above some threshold (e.g., 150), we approximate the minimum vertex cut algorithm by ceasing to call the subroutine once a small enough cut is found between any vertex pair (e.g., a cut of fewer than three vertices). We set multiple thresholds, and the stopping points are self-adjusted during the computation if they do not end searching early. This gives MEME considerable speedup with negligible impact on memory efficiency of encoding results. For smaller graphs, we still run the original algorithm to find the minimum vertex cut with the maximum number of resulting components, but the time cost is affordable.

3.5 Dynamic Updates

Attribute matrices are rarely static, with rows or columns being added or removed when either network conditions or policies change. For example, a column is added if a new AS joins the IXP, and a new row is added if a prefix's set of announcers changes. The former happens on the order of days, while the latter happens several times per second. When a matrix changes, either its tags, match strings, or both need to change. In an SDX, tags are updated via gratuitous ARPs, hence considered low-cost [10]. However, updating match strings require changing rules in the IXP fabric and must be done sparingly. For simplicity, we mainly consider bridging attributes (Section 3.1) and only briefly siblings and ancestors (Section 3.2) in the following discussion.

Algorithm 1: MEME

Input: Matrix M, Cluster size threshold λ
Output: Match strings $S = \{s_i | 1 \le i \le N\}$,
Tags $T = \{t_j | 1 \le j \le M\}$

```
Function MainAlgorithm(M, λ):
    M_curr ← M
    M_curr.delCol(getSibling(M_curr))
    tree_list ← []
    while width(M_curr)> 0 do
        M_brdg ← new Matrix(); T ← new Tree()
        root ← new Node(); T.addNode(root)
        if {} ∈ M_curr then
            T.addChild(root, ε_root)
        MatrixSplit(M_curr, M_brdg, λ, T, root)
        M_curr ← M_brdg; tree_list.append(T)
    B ← binarize(tree_list)
    S ← generateStrings(B, M)
    T ← generateTags(S, M)
    return S, T
Function MatrixSplit(M_curr, M_brdg, λ, T, anct):
    new_anct ← getAncestor(M_curr)
    if new_anct ≠ Null then
        T.addChild(anct, new_anct)
        anct ← new_anct
        if width(M_curr) > 1 & {anct} ∈ M_curr then
            T.addChild(anct, ε_anct)
        M_curr.delCol(anct)
    else
        if width(M_curr) < λ then
            T.addChild(anct, M_curr)
            T.delChild(ε_anct)
            return
        M_brdg.addCol(getBridging(M_curr))
        M_curr.delCol(getBridging(M_curr))
    for cluster in M_curr do
        MatrixSplit(cluster, M_brdg, λ, T, anct)
    return
```

Recall that MEME partitions the matrix's attributes into submatrices, clusters each submatrix's attributes, and assigns cluster IDs. When a new row appears, we first attempt to assign it a tag. The new row is partitioned into submatrices. If each partition is either empty or contained within a single cluster, then a tag can be generated without modifying the encoding. This is cheap and fast, only requiring finding

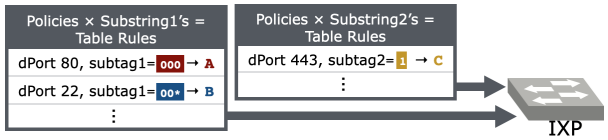


Figure 11: MEME on PISA switches.

matches in a few clusters and concatenating the corresponding IDs and bitmaps. However, if one of the row's partitions is not contained within a cluster, then the current encoding is insufficient. Since this may happen several times per second, re-encoding the entire matrix in response is infeasible.

To address this issue, we describe an efficient update procedure that minimizes the number of tag and match string modifications while keeping the number of strings optimal. If, in some submatrix, the new row's partition cannot be contained by an existing cluster, then it must span multiple clusters. Connecting these disjoint clusters may result in a cluster whose size is above the threshold λ. We re-encode those clusters the usual way, by extracting bridging attributes to split clusters and then reassigning IDs if needed. The extracted bridging attributes are moved to the next submatrix, and any attribute that co-occurs with them in a row of the next sub-matrix is also re-encoded. The subtags and match substrings of the affected columns need updates.

This procedure results in only a few updates of match substrings. Due to the sparse nature of the matrix, rows are quite small and updates often involve adding one attribute to an existing row, so it re-encodes only several attributes. The extraction of new bridging attributes causes the attributes to migrate from the starting submatrices to the ending ones, so the ending subtags may grow in size during updates. To address this, MEME can be configured to reserve bits in the initial encoding or to insert bridging attributes in the sparsest submatrix.

Taking siblings and ancestors into account, if an update disrupts such relationships, all affected clusters need re-encoding. Since the existence of these relationships implies an internal hierarchy of attributes, they are expected to rarely change. If that is untrue, sibling and ancestor encoding can be disabled to allow faster updates.

4 PISA MATCHING OPTIMIZATION

MEME partitions the columns of an attribute matrix into multiple submatrices, each encoded independently. The tag for a row in the original matrix is the concatenation of every submatrix's subtag; similarly, the match string for a column is the concatentation of every submatrix's match substring.

In prior membership encoding schemes, tags and match strings are compared in their entirety in switches. However, this matching design wastes memory. After MEME splits the matrix into submatrices, each attribute exists in only one submatrix. Only one subtag determines whether any given attribute is present or not, and only one match substring is something other than wildcards. Since it is known in advance which submatrix an attribute belongs to, we can save memory by only comparing the subtag with the substring of the submatrix that the attribute belongs to.

Traditional switches are restricted to operating on complete fields. In order to implement our design, we take advantages of the reconfigurable parsers in PISA switches that support flexible definition of header fields to parse each subtag separately. Then, instead of one match-action table for all policies, one table is created for each submatrix to match on its subtag. If a policy queries an attribute a from a submatrix M, the policy is augmented with the match substring of a from M to generate a rule in M's table. In Figure 11, adopting the encoding scheme from Figure 9b, the original match-action table (Figure 2e) is split into two, one matching on the $subtag1$ for the attributes $\{A, B, D, E, F, G, H\}$, and the other matching on the $subtag2$ for the attribute $\{C\}$.

Initially, we calculate the memory cost as the product of tag width and the number of match strings (Figure 4), but this is no longer true. The memory required to store all attributes' match strings is now the sum of the width of every match substring. This drastically cuts the memory cost, making MEME require even less memory than in Figure 4. The encoding scheme in Figure 9b requires only 22 bits for the seven 3-bit and one 1-bit match substrings, lower than the 32 bits derived at the end of Section 3.2.

Even though other membership encoding schemes can partition the attribute matrix to adopt the same design in PISA switches, we argue that MEME is a natural fit for this design. MEME finds the best partition of the attribute matrix that lowers the width of every substring greedily. In addition, it is possible to use MEME to find a submatrix partition which satisfies specific hardware constraints (e.g., the available number of tables and table width) by adjusting the cluster size threshold λ for each iteration of the algorithm.

5 EVALUATION

We evaluate MEME on memory cost, computation time, and tag width with two attribute matrices of routing information basis (RIB) from IXPs. All experiments were run on an Intel Xeon 4114 2.2Ghz processor with 96GB RAM. Our prototype has ~1500 lines in Python and is available on GitHub [15].

The first attribute matrix is converted from the RIB table dump of a 691-participant IXP with ~300K distinct IP prefixes on November 11, 2019. The second attribute matrix is constructed from BGP announcements of the PEERING testbed [18], containing routes from 4 IXPs. Since this process basically merges the four matrices, its density is much higher than the IXP matrix (Table 1).

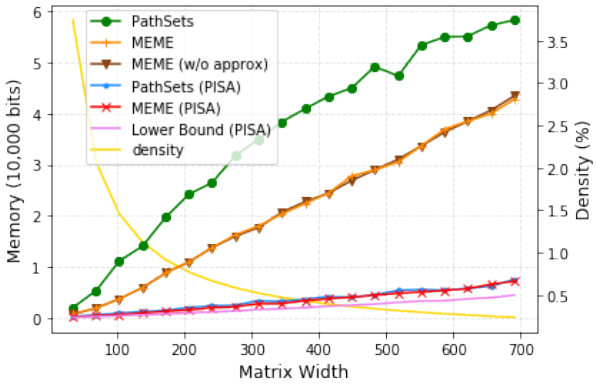


Figure 12: Memory cost of IXP

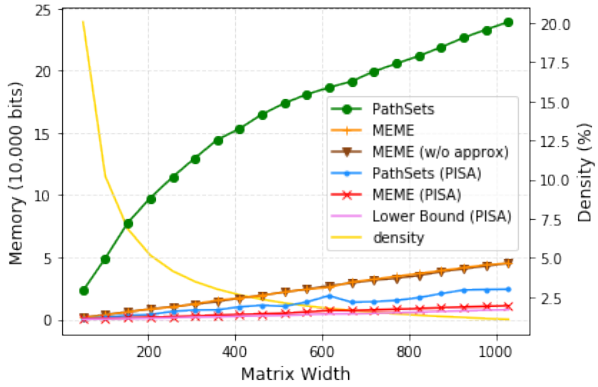


Figure 13: Memory cost of PEERING

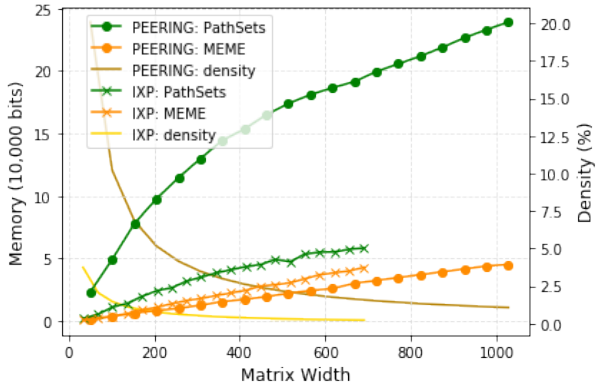


Figure 14: SDN-switch memory

To measure performance with growth of matrix size, we select subsets of the densest columns from the original matrices and combine them in smaller matrices. Instead of random sampling, selecting the columns with the most 1's not only yields the densest matrices, which are the hardest to compress, but also corresponds to real-world applications, where AS's tend to define policies for large ISP AS's rather than stub AS's that advertise only one prefix.

Memory Cost in SDN switches. Memory costs of the two experiments are shown in Figure 12 and 13, showing that MEME (orange lines) always outperforms PathSets (green lines). Comparing the two datasets in Figure 14, the memory usage of PathSets is highly dependent on the density. Specifically, for two matrices of the same width, the denser one (PEERING) requires on average 5.3× the memory to encode. In contrast, the memory cost of MEME is almost unaffected by density. Actually, the PEERING matrix requires slightly less memory than the IXP matrix because as a union of 4 matrices, it can be broken up by extracting a small number of bridging attributes. Indeed, only 55 bridging attributes are extracted from the PEERING matrix compared to 147 from the IXP matrix (Table 1). Consequently, MEME cuts the memory cost in SDN switches by 26.6% for the full IXP matrix and 81.1% for the full PEERING matrix.

Memory Cost in PISA switches. The Optimization for PISA switches brings substantial memory reduction. Our PISA-based design further cuts MEME's memory usage by 80.0% on average (red lines in Figure 12, 13). In both datasets, this leads to a memory cost of only 1.6× the lower bound (violet lines), $N \log_2 \frac{N}{m}$, where m is the number of submatrices produced by MEME. To quantify the benefits of this optimization in isolation, we also apply the same design to PathSets by partitioning the matrix randomly into m submatrices and encoding each with PathSets (blue lines). This also improves PathSets, reaching 2.0× the lower bound for the IXP dataset and 3.6× the lower bound for the PEERING dataset.

Dataset	N	M	D	Brdg.	Sibl.	Anct.
IXP	691	293,801	0.23%	147	0	11
PEERING	1028	805,865	1.06%	55	2	4

Table 1: Attribute matrix properties

Computation Time. Throughout our evaluation of MEME, we use initial stopping points of 2, 3, 4, and 5 for graph sizes of 200, 300, 400 and 500 (Section 3.4). It affects the memory cost negligibly (orange and brown lines in Figure 12, 13) while making the computation time almost flat, reaching 16.6% of PathSets' for the full IXP matrix and 2.3% for the full PEERING matrix (Figure 15). It takes MEME longer to compress the IXP matrix than the PEERING matrix of the same size since more calls of the minimum vertex cut algorithm are invoked to extract the larger number of bridging attributes.

Tag Width. Handling the bridging, sibling and ancestor attributes, MEME generates tags of 62 bits for the IXP matrix and 44 bits for the PEERING matrix (Figure 15). This tag width is far below the hundreds of bytes that modern switches can parse. Therefore, even though MEME's tag width is ~2.1× PathSets', it still permits line-rate processing.

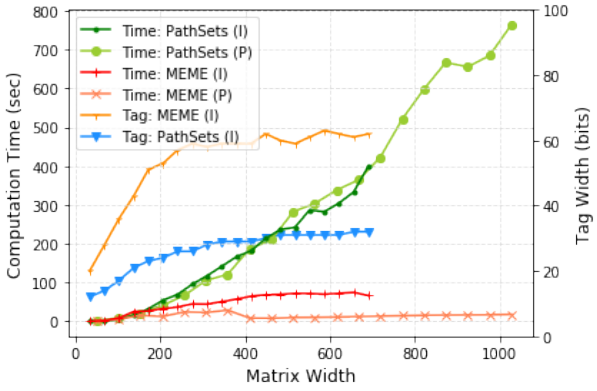


Figure 15: Time & Tag Width

Practical Implications in SDX. For the full IXP matrix, MEME with PISA optimization reduces the memory usage by 87.7%. To visualize this, it requires 10.4 bits to encode one attribute's match string(s) on average while PathSets requires 84.5 bits. Assuming that the available memory for match strings is 6Mb, the typical TCAM size in modern switches, PathSets allows only ~74K policies to be defined, which is 107 participants if everyone defines one policy for each peer. In comparison, MEME with PISA optimization allows ~607K policies and supports all the participants. Another practical benefit of MEME is minimizing policy update churn due to its optimal number of match strings.

We evaluated the update procedure (Section 3.5) by adding a random AS to any existing IXP matrix row, and it takes on average ~13 ms if such an addition leads to changes in match strings. On the other hand, simulations on 15 minutes' updates (~500,000 BGP messages) of the AMS-IX IXP on Jan. 7, 2020, retrieved from RIPE RIS [19], shows that although tags change on average 15 times per second, only *a single update* in the 15 minutes incurs match string changes, echoing observations from [9] that the vast majority of BGP updates preserve the matrix's clustering structure.

6 CONCLUSION

Many network applications rely on membership encoding. We propose a novel membership encoding algorithm and a PISA-based matching design. MEME drastically reduces both memory and computation time on large and real datasets.

Lastly, although this paper focuses on IXP networks as the application, MEME is potentially useful in other types of networks as well, For example, MEME could be useful in virtual networks [4, 8]. In this use case, operator-controlled virtual switches attach tags that impact forwarding in network core, and MEME is used to reduce the size of routers' forwarding tables.

ACKNOWLEDGMENTS

I would like to thank Robert MacDavid for his mentorship throughout the project. I would also like to thank my advisor Jennifer Rexford for her guidance throughout my research.

REFERENCES

[1] Norman Abramson. 1963. *Information Theory and Coding*. McGraw-Hill.

[2] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*. 99–110.

[4] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[5] Qunfeng Dong, Suman Banerjee, Jia Wang, Dheeraj Agrawal, and Ashutosh Shukla. 2006. Packet classifiers in ternary CAMs can be smaller. In *SIGMETRICS/Performance*. 311–322.

[6] Abdol-Hossein Esfahanian. 2006. On computing the connectivities of graphs and digraphs. *Networks* 14 (10 2006), 355 – 366. https://doi.org/10.1002/net.3230140211

[7] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. 2014. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Seattle, WA, 543–546.

[8] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[9] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. 2016. An Industrial-scale Software Defined Internet Exchange Point. In *USENIX NSDI*.

[10] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. 2014. SDX: A Software Defined Internet Exchange. In *ACM SIGCOMM*.

[11] Nanxi Kang, Ori Rottenstreich, Sanjay Rao, and Jennifer Rexford. 2017. Alpaca: Compact Network Policies with Attribute-Encoded Addresses. *IEEE/ACM Transactions on Networking* (June 2017).

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

[13] Robert MacDavid, Rudiger Birkner, Ori Rottenstreich, Arpit Gupta, Nick Feamster, and Jennifer Rexford. 2017. Concise Encoding of Flow Attributes in SDN Switches. In *ACM SIGCOMM Symposium on SDN Research*. 48–60.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM*

SIGCOMM Computer Communications Review 38, 2 (March 2008), 69–74.

[15] GitHub Repo. 2020. https://github.com/PrincetonUniversity/Meme.

[16] Ori Rottenstreich and Isaac Keslassy. 2015. The Bloom Paradox: When Not to Use a Bloom Filter. *IEEE/ACM Transactions on Networking* 23, 3 (June 2015), 703–716. https://doi.org/10.1109/TNET.2014.2306060

[17] Ori Rottenstreich, Isaac Keslassy, Avinatan Hassidim, Haim Kaplan, and Ely Porat. 2016. Optimal In/Out TCAM Encodings of Ranges. *IEEE/ACM Transactions on Networking* 24, 1 (2016), 555–568.

[18] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. 2019. PEERING: Virtualizing BGP at the Edge for Research. In *ACM CoNEXT*. Orlando, FL.

[19] RIPE Routing Information Service. 2020. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[20] Brent Stephens, Alan L. Cox, and Scott Rixner. 2016. Scalable Multi-Failure Fast Failover via Forwarding Table Compression. In *ACM SIGCOMM Symposium on SDN Research (SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2890955.2890957