ST. PETERSBURG DEPARTMENT OF V.A.STEKLOV INSTITUTE OF MATHEMATICS OF THE RUSSIAN ACADEMY OF SCIENCES

As a manuscript

Vitalii Yurievich Demianiuk

NEW METHODS, ALGORITHMS, AND THEORETICAL GUARANTEES FOR ALGORITHMS IN NETWORK ELEMENT DESIGN

PhD Dissertation

for the purpose of obtaining academic degree
Doctor of Philosophy in Computer Science HSE

Academic supervisors:
Researcher, PhD, Sergey I. Nikolenko
Senior Lecturer, PhD, Kirill Kogan

St. Petersburg — 2019

Contents

1 Introduction **3**
1.1 Dissertation topic and its relevance 3
1.2 Research Objectives 5

2 Key results and conclusions **7**

3 Publications and approbation of the research **9**

4 Contents **11**
4.1 Packet classifier representations 11
4.1.1 Representations of range based packet classifiers 12
4.1.2 Combined representations of multiple policies 14
4.1.3 Approximate classifiers with controlled accuracy 19
4.2 Robust Distributed Monitoring of Traffic Flows 22

5 Conclusion **25**

References **27**

Appendices **31**

A Paper “New Alternatives to Optimize Policy Classifiers” **31**

B Paper “Approximate Classifiers with Controlled Accuracy” **46**

C Paper “Robust Distributed Monitoring of Traffic Flows” **56**

D Paper “How to deal with range-based packet classifiers” **68**

1 Introduction

This section explores the tradeoffs appearing in the design of network elements, and then introduces the main research objectives of this thesis.

1.1 Dissertation topic and its relevance

A broad spectrum of services running on top of an exponentially growing number of interconnected devices make network operations more complex than ever. New complex network-wide behaviors, the variability of desired objectives incorporating different intents into final decisions together with increasing scalability levels require network infrastructure to be more intelligent, expressive, and robust. Usually, these requirements lead to significant operational complexity and an increased cost of network infrastructure. Reducing a manageable network state by better exploiting an expensive network infrastructure (efficiency) without compromising flexibility (expressiveness) can overcome new levels of operational complexity and scalability constraints; finding the right balance among them and finding ways to represent the manageable network state efficiently is a serious problem that requires fundamental understanding based on analytic observations. Recent developments in software-defined networking (SDN) partially improve expressiveness by adding new programmability levels but, unfortunately, fail to address the fundamental tradeoff between expressiveness and operational complexity. This calls for new design approaches that require efficient implementations of manageable state in network elements.

In this thesis, we deal with an efficient representation of a manageable state in a single network element and consider two fundamental questions:

(1) how to represent packet processing programs, addressing the fundamental tradeoff between efficiency and expressiveness;

(2) how to exploit existing network resources to address local constraints due to an exponentially growing number of interconnected devices and increased granularity of operations.

Packet processing programs. The core building block of packet processing programs is a packet classifier. Packet classifiers implement various commodity services such as QoS (Quality of Service), access control lists, packet forwarding, etc. With the adoption of OpenFlow [1] and P4 [2], packet classification has become even more prominent. In particular, a novel programming language P4 [2] is designed for the development of packet processing pipelines; a packet classifier is a basic element of P4 pipelines. The main purpose of packet classification is to determine the processing logic (*action*) that should be applied on an incoming packet. A packet classifier is an ordered set of rules, where a rule consists of the filter (matching packet headers) and the associated action to be applied on packets with matched headers. The first rule whose filter matches an incoming packet header defines which action to apply. A filter is a concatenation of field representations participating during classification.

In the simplest form, fields in a filter are represented by exact values (see Fig. 1a). In this case, packet classifiers can be implemented with constant lookup time but the number of rules can be so large that it is infeasible to fit the classifier in memory. In the other extreme case, to address the scalability

Rule	Filter		Action
#1	0	2	A_1
#2	0	3	A_1
#3	1	2	A_1
#4	1	3	A_1
#5	4	5	A_2
#6	4	6	A_2
#7	5	5	A_2
#8	5	6	A_2

(a)

Rule	Filter		Action
#1	[0,1]	[2,3]	A_1
#2	[4,5]	[5,6]	A_2

(b)

Rule	Filter		Action
#1	00*	01*	A_1
#2	10*	101	A_2
#3	10*	110	A_2

(c)

Figure 1: A packet classifier $\mathcal{K}$ matching headers by two 3-bit fields; fields in $\mathcal{K}$ filters are represented by: (a) exact values; (b) ranges of values; (c) ternary bit strings.

constraint, fields can be represented by ranges of values (see Fig. 1b). In this case, the number of rules is significantly reduced but the lookup complexity increases. As a result, some intermediate forms of field representations were introduced as prefixes or more general ternary bit strings, where every bit has three values: zero, one, or *don't care* denoted as $*$ (see Fig. 1c). For classifiers with fields represented by ternary bit strings, a specialized ternary content addressable (TCAM) memory was introduced; TCAM allows to classify headers in a constant time [3]. Note that TCAM memory is a power hungry and expensive resource. Thus, we should minimize the size of ternary classifier representations.

In this thesis we present three main results dealing with packet classifier representations:

(1) study how the encoding of ranges by ternary bit strings affects the size of a ternary representation of a range-based classifier,

(2) propose efficient combined representations of multiple classifiers that share identical classification patterns, and

(3) introduce the notion of approximate classifiers, allowing to trade classification accuracy for additional reduction in the number of classification rules (and hence memory).

Exploiting network resources. Traffic monitoring requires a significant amount of in-network resources and is crucial for efficient, reliable, and secure network operations. Understanding traffic properties allows network operators to achieve better capacity planning, QoS assurance, service differentiation, attack mitigation, and more. Scalable monitoring of traffic flows is challenging due to unrelenting traffic growth, device heterogeneity, and load unevenness. First, while traffic keeps growing in both volume and number of flows, the processing and storage needed for traffic monitoring in network elements grow as well. Second, networks comprise elements of increasing heterogeneity ranging from basic IoT (Internet of Things) access devices with greatly limited capabilities to high-end core routers that forward millions of concurrent flows. Third, the traffic-monitoring load on different network elements is uneven, and one element might get overloaded even when other elements have spare resources.

Even the relatively simple problem of computing all flow sizes becomes very challenging when the number of flows becomes significant. As a result, a large body of related work explores approximate solutions for traffic accounting that give up computation accuracy in order to reduce memory required in a single element. Previous approximate solutions for flow size computation include estimators [4, 5, 6, 7] and sketches: CM [8], CU [9], Pyramid Sketch [10], UnivMon [11], and Elastic Sketch [12]. A trace-driven evaluation of CEDAR [7], SAC [6], and DISCO [5] shows their average relative errors in excess

of 12% for 8-bit per-flow estimators [7]. The average relative error of Elastic Sketch, one of the most advanced current proposals, can be equal to 4 even when using 0.2 MB to represent 110K flows, i.e., around 15 bits per flow (see Fig. 9a in [12]). Such accuracy might be too low for many practical purposes.

An alternative approach to address the scalability constraint in a single network element is to monitor traffic flows by utilizing resources in multiple network elements as a shared pool. If a traffic-monitoring task overloads an element, then task execution can be shifted from the overloaded element to another element over the flow path that has sufficient resources. This empowers the network to leverage its global processing and storage resources to effectively cope with local traffic monitoring overloads. Due to the network noise consisting of packet loss and reordering, the maintenance of distributed packet counters is a challenging problem in itself. In this thesis, we propose a solution that efficiently maintains distributed packet counters under network noise without using control packets or any other additional communication between the involved network elements.

1.2 Research Objectives

In this section, we define specific objectives that we set for the research outlined in the thesis. We introduce three projects developing efficient packet classifier representations and one project for the maintenance of distributed packet counters.

Impact of range encoding. To maintain a range-based classifier $\mathcal{K}$ in TCAM memory, $\mathcal{K}$ filters should be encoded by ternary-bit strings. Since the TCAM is an expensive and limited resource, the total size of the ternary representation of $\mathcal{K}$ in bits and in ternary bit strings should be minimized by this encoding.

One way to encode a range-based classifiers into a ternary bit classifier is to encode each range in a classifier separately by one of the following methods [13, 14, 15, 16, 17]. These methods encode every field range by multiple prefixes or ternary bit strings whose number is at most linear to the field size (in bits). As a result, the number of ternary bit strings in the encoding of a range-based filter can exponentially depend on the number of filter fields. Other range encoding methods exploiting structural properties can achieve more compact representations [18, 19] but usually, they perform well only when the number of different encoded ranges is relatively small. Note that both these lines of research consider transformations to equivalent classifiers.

Representations [20, 21, 22, 23] construct ternary lookup tables on subsets of fields preserving structural properties of classifiers like rule disjointness. These representations become equivalent to originally given classifiers only when the constructed lookup tables are complemented by a false-positive check on a single matched rule. This allows balancing which fields are required to implement desired structural properties in the lookup table and which participate in false-positive checks where range encoding is unnecessary. The number of encoded ranges in [20, 21, 22, 23] significantly decreases that lead to significant reduction of bits and ternary bit strings in ternary representations of $\mathcal{K}$.

In this dissertation, we are going beyond [20] and propose RR (range reduction) representations implementing structural properties of classifiers with per-bit resolution (rather than per-field as in [20]). Representations narrowing down covered field ranges can significantly improve memory requirements for classifier representations.

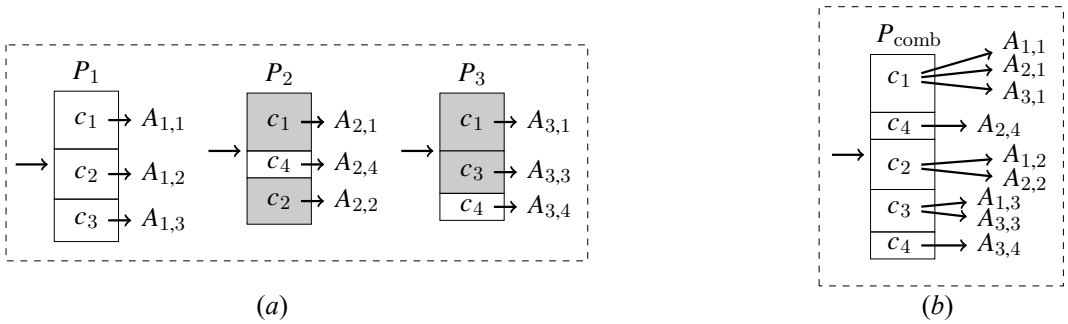


Figure 2: (a) separate policies P_1, P_2, P_3; (b) a combined representation P_{comb} that emulates the policies; class instances that have been cut in P_{comb} are shown in gray.

Policy-based representations of packet classifiers. In modern network elements, many services run in parallel. These services are implemented by multiple packet classifiers that often share common classification patterns (even packet classifiers corresponding to different services). To avoid the specification of every filter for every classifier, the abstraction "class" was presented. Classes are sets of classifier filters representing common classification patterns. In this case, a packet classifier (policy) can be defined as an ordered set of classes with associated actions. In a nutshell, class-based representations allow to produce object-oriented representations of packet classifiers that ease at least their declaration. Various vendors already support the notion of classes in policy declarations [24, 25], allowing them to abstract and manage classification patterns more efficiently. For instance, Cisco IOS supports up to 256 different QoS policies and up to 4096 classes per box [26]. In real deployments, the number of classes per policy ranges from tens to hundreds depending on the application model [26]. The size of a class depends on the complexity of the represented pattern. Our ideas hinge on the fact that classes are reused in different policies. Traditionally, a separate class instance is allocated for each policy containing it (see policies P_1, P_2, and P_3 in Figure 2). Each allocated class instance has an attached action specified by the corresponding policy (in Figure 2, an action $A_{i,j}$ is attached to the instance of c_j in the policy P_i). Since classes are reused in different policies, it allows us to look at combined policy representations, where ideally each class appears only once, providing substantial savings in representations of underlying classifiers in TCAM memory. Informally, combined policy representations "emulate" behaviors of represented policies. In this thesis, we consider combined policy representations that do not increase the number of classification lookups.

Approximate packet classifiers. Exact computations may require excessive resources. Approximate computing deals with potentially inaccurate computations, helping to alleviate resource constraints [27]. In this thesis we generalize the classical packet classification problem to the approximate case. There is a long line of research exploring various optimization methods to find semantically equivalent packet classifiers, where each header matches the same action in an originally given and optimized classifiers [28, 29, 30]. We consider the case when semantically equivalent classifiers fail to achieve desired optimization results and introduce approximate representations of packet classifiers allowing to "multiplex" multiple actions. This additional level of flexibility allows to improve resource requirements while still keeping the desired level of accuracy. The majority of proprietary heuristics minimizing the number of rules in regular packet classifiers can be reduced to well-known operators minimizing the size of Boolean expressions [31, 32]. Unfortunately, in the approximate case optimization capabilities of these

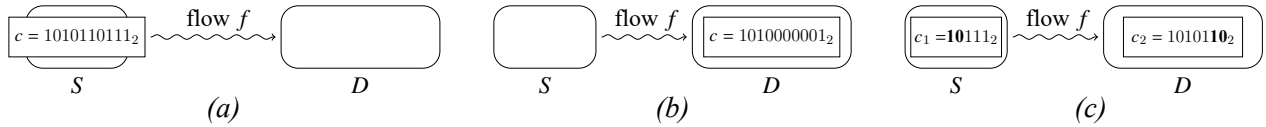


Figure 3: Computing the number of packets in a flow f: (a) at source S, (b) at destination D, and (c) distributively at both elements S and D. In (c) counter states overlap to keep the state representation consistent despite network noise.

operators are limited and using these operators may lead to an exponential blowup in the corresponding exact classifiers enumerations. To avoid this complexity and improve optimization results, we generalize basic operators to the approximate case and study the properties of sequences consisting of applied operations.

Distributed packet counters. To maintain flow packet counters distributively we could move the counter from a flow source to other network element on a flow paths (see Figure 3b). Unfortunately, moving the entire execution of a flow packet counter from the flow source to another network element would result in inaccurate results due to packet loss that occurs on the path between these two elements. Note that such inaccuracies arise regardless of whether the source element uses an unreliable or reliable transport protocol for sending traffic. Thus, at least some state of flow packet counter should be maintained in the source element (see Figure 3c). We introduce a robust distributed approach that maintains a packet counter in multiple network elements correctly despite network noise in the form of packet reordering and loss on the network paths between the elements. Instead of sacrificing the accuracy as in approximate solutions, our robust distributed approach supports scalable exact reconstruction of the number of packets in a flow. The explored distributed approach does not require bidirectional transport, does not introduce any control packets and communicates flow state by piggybacking a few, on the order of 2 or 4, control bits.

2 Key results and conclusions

This section summarizes the key contributions of this thesis, addressing specific objectives introduced above. Here, we provide a brief list of key results, and then will expand upon each research direction in more detail in Section 4.

Representations of range-based packet classifiers. In this thesis we define equivalent representations RR preserving rule-disjointness of range-based packet classifiers with per-bit resolution [33]. The basic element of the proposed RR representations is a *subrange* on a predefined set B of range bit-indices allowing to implement structural properties of a classifier on a per-bit level without intermediate range expansion. We show that each subrange on $|B|$-bit indices can be represented either as a $|B|$-bit range or as a union of two special ranges on $|B|$ bits; the proposed Algorithm 1 constructs such subrange representations. The number of ternary bit strings in prefix expansion [13] and in SRGE encoding [15] of any $|B|$-bit subrange is at most $2 \cdot |B| - 2$ (similarly to the case of a regular $|B|$-bit range). We formulate Problem 1 minimizing the encoding size of RR representations and propose Algorithm 3 solving Problem 1.

Combined representations of multiple policies. For a given set of policies $\mathcal{P}$ we propose a combined representation that consists of a *combined policy* P_{comb} emulating $\mathcal{P}$ [34, 35]. This thesis explores two basic cases: (1) ideal representations containing each class only once, and (2) non-ideal representations with duplicated class instances. In the first case, we show conditions for the existence of ideal representations and propose methods constructing P_{comb} when these conditions are satisfied. In the second case, we formulate the PSP problem finding the sequence of classes $\mathbb{S}$ in P_{comb} minimizing the number of filters $W^+(\mathbb{S})$ in duplicated class instances in $\mathbb{S}$. The proposed algorithms `AllOrOne` and `CliqueShare` constructing $\mathbb{S}$ have analytic guarantees on $W^+(\mathbb{S})$. We prove that the approximation factor of `CliqueShare` on $W^+(\mathbb{S})$ is at most $\alpha(G^{\mathsf{pair}}) \cdot \frac{|\mathcal{P}|^2}{4}$, where $|\mathcal{P}|$ is a number of policies in $\mathcal{P}$, G^{pair} is an auxiliary graph used by `CliqueShare`, and $\alpha(G^{\mathsf{pair}})$ is an approximation of the algorithm finding minimal Feedback Vertex Set [36, 37, 38] in G^{pair}. The lower bound on the approximation factor of `CliqueShare` is $\frac{|\mathcal{P}|^2}{4}$. In addition, we show that there is no polynomial algorithm for the PSP problem with a constant approximation factor (unless **P** = **NP**). The constructed $\mathbb{S}$ can be further optimized by the proposed efficient heuristics `GreedyGluing` and `LocalDescend`. To maintain the P_{comb} in a valid state after modifications of policies in $\mathcal{P}$, we propose the method modifying P_{comb} and $\mathbb{S}$ properly after each insert/remove operation in time $O(|\mathbb{S}| + \sum_{P \in \mathcal{P}} |P| + D(P))$, where $D(P)$ is the number of intersecting class pairs from P. Also we discussed combined representations consisting of multiple combined policies. In particular, we show that usage of multiple P_{comb}s can not relax conditions guaranteeing the existences of an ideal P_{comb}, and propose methods constructing ideal combined representations with multiple combined policies minimizing the maximum number of policies corresponding to a single P_{comb}.

Approximate classifiers with controlled accuracy. We introduce a notion of approximate packet classifiers with controlled errors [39]. Each rule R_i in the approximate classifier contains an *action set* $\mathcal{A}_i$ such that any action in $\mathcal{A}_i$ is a valid classification result of a header matched by R_i. The notion of approximate classification allows to additionally reduce the number of classification rules sacrificing the classification accuracy. In particular, we demonstrate that approximate classifiers can improve scalability of QoS and packet forwarding tables. To minimize approximate classifiers we generalize conventional optimization operators: forward subsumption, backward subsumption, and resolution [30]. The generalized operators optimize approximate classifiers strictly better than original ones.

Then we study properties of *operation sequences* consisting of applied operations. Let **O** be a set of operations containing only those that are applicable on the original classifier and not containing two operations using the same rule. Such sequences define lower bounds on the length of a longest operation sequence and illustrate the combinatorial complexity of the approximate classifier minimization. We prove that in the case of regular classifiers, there is an operation sequence containing all operations from **O**, and in the case of approximates classifiers, there is an operation sequence containing all forward subsumption and resolutions from **O**. To demonstrate the complexity of the approximate classifier minimization we show the example of **O** containing backward subsumptions and resolutions that can not be in the same operation sequence. Also, we prove that in the case of approximate classifiers, the construction of a longest operation sequence consisting of operations from **O** is an intractable problem.

We show that it is always possible to transform any operation sequence (including longest) into another one of the same length containing forward subsumptions and resolutions only before backward

subsumptions. This observation can be used for the development of heuristics finding longest operation sequences. Also, we show that after the proposed transformation of an operation sequence, action sets of rules in the optimized classifier either remains the same or even becomes bigger. Hence, this transformation can be used as a post optimization allowing to extend already constructed operation sequences.

Distributed packet counters. To reduce memory requirements in a single switch we split a flow packet counter into two chunks maintained by flow source and by flow destination (see Figure 3c). We propose methods for maintenance of counter chunks in consistent states despite network noise. There are two major contributions of this thesis: (1) two network noise models allowing to describe the tolerance of the proposed distributed approaches to packet loss and packet reordering; (2) Algorithm 1 and Algorithm 2 supporting flow counter chunks in consistent states; tolerance of these algorithms to the network noise is analytically proved in the corresponding network noise models. The tolerance of Algorithm 2 to the network noise is significantly bigger than the tolerance of Algorithm 1. Also we bound the error of Algorithm 2 when its correctness conditions are not satisfied.

3 Publications and approbation of the research

Each of the key results presented in the previous section has been published in one of the peer-reviewed research papers listed below. In all these papers the author of the dissertation is the main author.

First-tier publications (CORE ranking A/A^*):

- Demianiuk V., Nikolenko S., Chuprikov P., Kogan K. New Alternatives to Optimize Policy Classifiers // IEEE Transactions on Networking, vol. 28, no. 3, 2020, pp. 1088–1101. Conference version previously appeared in Proceedings of IEEE ICNP 2018. The content is in Appendix A.

  Contribution of the dissertation's author: an ideal representation for the case when all classes are pairwise disjoint, described in Section 3.A; the method prepending filters in P_{comb} containing intersecting classes, described in Section 3.B (except the lower bound shown by Theorem 2); Theorem 3 defining conditions for the existence of ideal representations; Theorem 4 and Theorem 6 analyzing the properties of ideal combined representations containing multiple combined policies; definition of the PSP problem (Problem 1); the idea of `AllOrOne` algorithm solving PSP; Theorem 11 presenting a lower bound on the approximation factor of `AllOrOne`; Theorem 12 showing an inapproxiamability of PSP with a constant factor; the `CliqueShare` algorithm solving PSP; Theorem 13 proving the correctness of `CliqueShare`; Theorems 14 and 15 presenting the upper and lower bounds on the approximation factor of `CliqueShare`; a local descent and a greedy gluing optimization, described in Section 5.E; Algorithm 3 for efficient dynamic updates proposed in Section 6; Theorem 16 proving the correctness of Algorithm 3.

- Demianiuk V., Kogan K., Nikolenko S. Approximate Classifiers with Controlled Accuracy // Proceedings of IEEE INFOCOM 2019. The content is in Appendix B.

Contribution of the dissertation's author: a notion and a motivation of approximate classifiers with controlled accuracy, described in Sections 2,3; generalization of conventional optimization operators, described in Section 4; examples in Section 4.C demonstrating that generalized operations are strictly more efficient than original ones; Theorem 1 showing that in the case of regular classifiers, it is always possible to apply all operations from **O**; Corollary 1 finding the biggest **O**; Theorem 2 showing that in the case of approximate classifiers, all forward subsumptions and all resolutions in **O** are applicable together; Theorem 3 showing that in the approximate case, the construction of the longest operation sequence consisting of operations from **O** is an intractable problem; Lemma 1 proposing a transformation of any operational sequence.

- Demianiuk V., Gorinsky S., Nikolenko S., Kogan K. Robust Distributed Monitoring of Traffic Flows // Proceedings of IEEE ICNP 2019. The content is in Appendix C.

  Contributions of the dissertation's author: algorithms `PL` and `PR` maintaining distributed packet counters under a partial network noise that either loses packets loss or reorders packets; the first network noise model, described in Section 4; Algorithm 1 for distributed packet counting under a general network noise that loses and reorders packets; Theorem 2 showing the tolerance of Algorithm 1 to the network noise in the first model; the second network noise model, described in Section 5; Algorithm 2 that is an extension of Algorithm 1 for the second network noise model; Theorem 4 showing the tolerance of Algorithm 2 to the network noise; Theorem 6 showing that correctness conditions of Algorithm 2 are wider than correctness conditions of Algorithm 1; Theorem 7 bounding the error of Algorithm 2 when its correctness conditions are not satisfied.

Other publications:

- Demianiuk V., Kogan K. How to deal with range-based packet classifiers // Proceedings of ACM SOSR 2019. * The content is in Appendix D.

  Contributions of the dissertation's author: Observation 1 showing that any range can be represented as a union of the left border range and the right border range; a notion of a subrange, described in Section 2; Lemma 3.2 showing that a subrange of a left (right) border range is a left (right) border range; Theorem 3.3 showing that any subrange can be represented as a union of two border ranges; Algorithm 1 constructing such subrange representation; Lemma 3.4 and Theorem 3.5 proving the correctness of Algorithm 1; Theorem 3.6 presenting the upper bound on the number of ternary bit strings in the encoding of a $|B|$-bit subrange; the formalization of equivalent representations with per-bit resolution, described in Section 4; the definition of Problem 1 minimizing the size of such equivalent representations; Algorithm 3 solving Problem 1.

The obtained results are supported in two ways. First, for each optimization algorithm we present a rigorous theoretical study analyzing its properties (proofs of correctness, worst case performance guarantees, structural analysis of the constructed and optimal solutions, etc.). Second, for practical purposes, we evaluate proposed methods on synthetic traces showing their behavior on the average case. Algorithms

*SOSR is the premiere venue for research publications on Software Defined Networking, established in 2015.

minimizing approximate classifiers (Section 4.1.3) have been evaluated on the classifiers generated by *Classbench* [40] framework and on five IPv4 FIB classifiers used in real systems. The action sets of rules in these classifiers were generated synthetically according to packet forwarding and QoS use cases. Methods constructing combined policies (Section 4.1.2) have been evaluated on synthetic sets of policies $\mathcal{P}$. During the input generation we vary the following major characteristics of $\mathcal{P}$ affecting optimization results: the number of policies, the number of class intersections and the average number of common classes in two policies. Algorithms constructing equivalent representations of range based classifiers (Section 4.1.1) have been evaluated on randomly generated classifiers where the number of fields in a filter and field bit-width varies. Methods maintaining distributed counter chunks (Section 4.2) have been evaluated in YAPS simulator [41] with implemented unreliable transport; traffic traces were obtained from the "data-mining" distribution of flow sizes [42, 43, 44].

4 Contents

This section provides an overview of the research projects that led to the results presented in Section 2 and describes how these results achieve the objectives stated in Section 1.2.

4.1 Packet classifier representations

As we mentioned before, a packet classifier is a core building block of packet processing programs. The main purpose of packet classification is to determine an action that should be applied on an incoming packet. Here, we propose different methods improving the efficiency of packet classifier representations.

An incoming packet is classified by a packet *header* H consisting of k fields $\{h_1, \ldots, h_k\}$, where each h_i is an integer on w_i bits. A packet *classifier* $\mathcal{K} = \{R_1, \ldots, R_N\}$ is a sequence of rules, where each *rule* $R_i = (F_i, A_i)$ consists of a *filter* F_i matching packet headers and the associated *action* A_i. We consider two types of field representations in a classifier $\mathcal{K}$: (1) a range of values (see Figure 1b); and (2) a ternary bit string, where each bit can have one of the three values: zero, one, or *don't care* denoted as * (see Figure 1c). In the first case, a filter F is a sequence of k ranges $\{I_1, \ldots, I_k\}$; a header H is matched by F if h_i belongs to I_i for each $1 \leq i \leq k$. In the second case, a filter F is a ternary bit string of length $l = \sum_{i=1}^{l} w_i$ obtained as the concatenation of the ternary bit strings representing corresponding fields; a header H is a binary string of length l obtained as the concatenation of the binary notations of H fields; F matches H if for each bit index $1 \leq i \leq l$ either $H[i] = F[i]$ or $F[i] = *$. A rule in $\mathcal{K}$ *classifying* H is a first rule in $\mathcal{K}$ matching H. The classification process of H finds the action of a rule in $\mathcal{K}$ classifying H. If there is no rule matching H in $\mathcal{K}$, the classification result is a default action. In the following we say that two filters F_1 and F_2 are *disjoint* if no single header matches both of them. Otherwise, F_1 and F_2 *intersect*. Two rules intersect (are disjoint) if their filters intersect (are disjoint).

Section 4.1.1 gives an overview of the approach published in the paper titled "How to deal with range-based packet classifiers" (see Appendix D and [33]) corresponding to the first result from Section 2. Section 4.1.2 describes the paper "New Alternatives to Optimize Policy Classifiers" (see Appendix A and [34, 35]), where the second result from Section 2 is presented. Section 4.1.3 presents approximate representations of packet classifiers proposed in the paper "Approximate Classifiers with

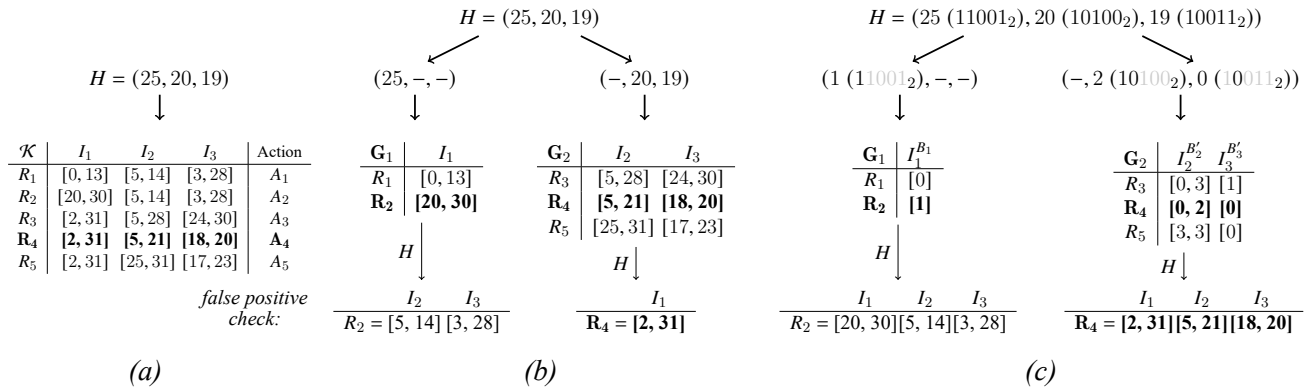


Figure 4: RR vs. SAX-PAC [20]: (a) a given classifier $\mathcal{K}$; (b) SAX-PAC representation of $\mathcal{K}$ consisting of two groups defined by $\mathbf{F}_1 = \{I_1\}$ and $\mathbf{F}_2 = \{I_2, I_3\}$, respectively; (c) RR representation of $\mathcal{K}$ consisting of two groups defined by $\mathcal{B}_1 = \{\{1\}, \emptyset, \emptyset\}$ and $\mathcal{B}_2 = \{\emptyset, \{1, 2\}, \{2\}\}$.

Controlled Accuracy" (see Appendix B and [39]) corresponding to the third result from Section 2.

4.1.1 Representations of range based packet classifiers

To store a range-based classifier $\mathcal{K}$ in TCAM, $\mathcal{K}$ filters should be encoded by ternary bit strings. Recall that the number of ternary bit strings (and the number of bits in these strings) in the $\mathcal{K}$ encoding exponentially depends on the number of $\mathcal{K}$ fields. To reduce the number of encoded fields, SAX-PAC [20] propose equivalent classifier representations where only fields preserving rule-disjointness property participate in a classification lookup; the remaining fields of matched rules are verified after the lookup. In this project, we are going beyond [20] and propose range reduction (RR) methods allowing to implement rule-disjointness with a per-bit resolution that provides additional savings in the encoding size. Firstly, we describe SAX-PAC classifier representations, then we introduce a notion of a subrange that is a basic element of RR representations, and then we define RR representations.

SAX-PAC representations. Consider β disjoint groups of $\mathcal{K}$ rules $\mathbf{G}_1, \mathbf{G}_2, \ldots \mathbf{G}_\beta$ such that rules in each group $\mathbf{G}_i$ are pairwise disjoint on a subset of fields $\mathbf{F}_i$. A SAX-PAC representation contains a separate lookup table for each such group $\mathbf{G}_i$. Due to rule disjointness only a single rule in $\mathbf{G}_i$ can match an incoming header H even if only fields in $\mathbf{F}_i$ participate in a lookup. Therefore, the lookup table representing $\mathbf{G}_i$ contains only fields in $\mathbf{F}_i$ (i.e, only ranges corresponding to fields in $\mathbf{F}_i$ are encoded) and is followed by a false positive check validating remaining fields of a matched rule. Since not all rules can be covered by these β groups, there is a portion of remaining rules C from $\mathcal{K}$ not belonging to any group. We represent C as a regular classifier (i.e., all ranges of rules in C are encoded). The value of $\beta + 1$ corresponds to a number of "pseudo-parallel" lookups that can be issued at line-rate. The classification process in SAX-PAC is the following: (1) find a classifying rule in every group (lookup table) and perform a false-positive check for every matched rule (at most β overall); (2) independently with (1) find a classifying rule in C; (3) from at most β matched rules passing a false-positive check and a classifying rule in C return the action of a rule that appears first in $\mathcal{K}$ or the default action if there is no rule matching a given header (see Figure 4b). In Figure 4, the number of ternary string in the encoding of the SAX-PAC representation is 8 times smaller than the number of ternary strings in the $\mathcal{K}$ encoding if ranges are encoded by the prefix expansion [13].

Subranges. Here, we introduce a notion of a *subrange* allowing to implement a rule-disjointness property of a given classifier with per-bit resolution avoiding an intermediate range expansion. Intuitively, taking values on a subset of bit indices of a given range leads to the consideration of a smaller set of values and potentially more efficient representations in ternary bit strings than the originally given range. But this is not enough, for correctness, we need that a value matched by the original range continues to be matched by the corresponding subrange on the representing subset of bit indices. The proposed below subranges are efficient and satisfy correctness property.

Consider a non-empty subset of bit indices $B \subseteq \{1, 2 \ldots, w\}$ of a w-bit range I. For a value x, denote by x^B a value obtained from x by taking the values of bits at the positions in B. A *subrange* I^B of a range I is a set of values on B bit indices obtained from the values in I. For instance, if I is a 5-bit range $[23, 25]$ and $B = \{1, 3, 5\}$, then $I^B = \{4, 5, 7\}$; the following table consists of values in I and in I^B.

x	$23\ (10111_2)$	$24\ (11000_2)$	$25\ (11001_2)$
x^B	$7\ (111_2)$	$4\ (100_2)$	$5\ (101_2)$

This example shows that a subrange is not necessary a range. Note that, if I consists of one value x then I^B consist of one value x^B. In the following we consider ranges containing at least two different values.

For a range $I = [l, r]$, let $sim(I)$ be a first index of a bit whose value differs in the binary representations of l and r; for $I = [23, 25]$, $sim(I) = 2$. We say that a w-bit range $I = [l, r]$ is a *left border range* if $l \bmod 2^{w-sim(I)} = 0$; and I is a *right border range* if $(r + 1) \bmod 2^{w-sim(I)} = 0$. For instance, $I = [24(11000_2), 29(11101_2)]$ is a left border range; and $I = [18(10010_2), 23(10111_2)]$ is a right border range. A range $I = [l, r]$ can be represented as a union of a right border range and a left border range. For instance, $[9, 14]$ is a union of a right border range $I_1 = [9, 11]$ and a left border range $I_2 = [12, 14]$. The following lemma and theorem show that each subrange also can be represented as a union of a right border range and a left border range.

Lemma. *For a subset of bit-indices $B \subseteq \{1, \ldots, w\}$ and a w-bit range $I = [l, r]$, if I is a left (right) border range, the corresponding subrange I^B is a $|B|$-bit left (right) border range.*

Theorem. *For a subset of bit-indices $B \subseteq \{1, \ldots, w\}$ and a w-bit range I, the set I^B can be represented as a union of the $|B|$-bit left border range and $|B|$-bit right border range.*

We propose an algorithm that for a range I and a set of bit indices B computes two border ranges forming I^B in time $O(w)$ (see Algorithm 1 in Appendix D). The correctness of this algorithm is not obvious and is proved in Appendix D by Lemma 3.4 and Theorem 3.5. Note that we can operate on subranges as on regular ranges since the intersection of two subranges can be verified in a constant time.

The encoding of a w-bit regular range consists of $2 \cdot w - 2$ ternary bit strings for the prefix expansion method [13] and $2 \cdot w - 4$ for the SRGE method [15] in the worst case. The following theorem shows that, in the worst case, the number of ternary bit strings in the encoding of a $|B|$ bit subrange is almost the same as in the encoding of a regular $|B|$ bit range for both prefix expansion and SRGE.

Theorem. *A subrange I^B can be encoded by at most $2 \cdot |B| - 2$ ternary bit strings using prefix expansion or SRGE encoding.*

RR representations. In RR we follow the same classification process as in SAX-PAC but now each group $\mathbf{G}_i$ implements rule disjointness on a subset of subranges (at most one per field) defined by $\mathcal{B}_i =$

$\{B_1, B_2, \ldots, B_k\}$, where B_j is a subset of bit indices of jth field (see Figure 4c). Note that B_j can be empty meaning that the corresponding field does not participate in the lookup table. The false-positive checks in RR representations is done on all filter fields. In Figure 4, the number of ternary entries in the encoding of the RR representation is 60 times smaller than the number of ternary strings in the $\mathcal{K}$ encoding if ranges are encoded by the prefix expansion.

Problem (RR construct, RR-C)**.** *For a given range encoding method and a range-based classifier $\mathcal{K}$ find the RR representation $\mathcal{R}$ of $\mathcal{K}$ minimizing the encoding size of $\mathcal{R}$ in bits or in ternary bit strings.*

Even in the case of a $\beta = 1$, the RR-C problem is intractable that can be shown by the reduction from SetCover [45]. Since SAX-PAC representations are included in the set of RR-C solutions, the resulting encoding of any SAX-PAC representation is no smaller than the encoding of optimal RR representation. To solve the RR-C problem we propose the greedy heuristic that starts with the same assignment of $\mathcal{K}$ rules into multiple groups as in SAX-PAC, but now for every range-based field a subrange preserving rule-disjointness property is selected to minimize the total encoding size in bits or in ternary entries.

Evaluations. In the evaluation study we compare the total encoding size in bits and in ternary bit strings of constructed SAX-PAC representations and RR representations for synthetically generated classifiers. In our experiments we vary a number of fields in a classification rule and a range width in bits. The evaluations show that the difference between the encoding size of RR representation and the encoding size of SAX-PAC representation rapidly increases with growth of the number of fields in a rule and the range width. In particular, for a classifier on six 32-bit ranges, the total number of ternary strings in the encoding of the RR representation can be up to 137 times small than in the encoding of SAX-PAC representation. See Section 5 in Appendix D for more details.

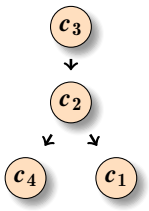
4.1.2 Combined representations of multiple policies

As we mentioned before, classes allow object-oriented representations of packet classifiers (*policies*) that ease at least their declaration. Classes are reused by different policies that allow us to construct combined representations maintaining same class instances for different policies. Sharing class instances can substantially reduce the total size of policy representations. In this work we propose methods constructing combined representations of a given set of policies $\mathcal{P}$. These representations do not increase the number of classification lookups.

Definition of policy-based representations. Classes represent an intermediate level of abstraction: a *class* c is a set of filters represented by ternary bit strings. A header H *matches* a class c if H matches at least one filter in c. To define a *policy* P one needs to specify a sequence $\mathbb{S}(P)$ of classes in P and associate an action with each class in $\mathbb{S}(P)$. For an incoming header H, the action of a first matched class in $\mathbb{S}(P)$ is returned (or the policy default action if there are no such classes). The policy P in $\mathcal{P}$ where the classification of H should be performed is retrieved from internal switch data structures; in this case, we say that H is coming in the *context* of the policy P. Our goal is to construct a *combined policy* P_{comb} with minimum number of filters classifying each H incoming in the context $P \in \mathcal{P}$ into the same action as P. In this project, we consider combined representations consisting of one combined policy and show

Class	Filters	Action
c_3	01** *1*0	A_1
c_2	**00	A_2
c_1	10**	A_3
c_4	00**	A_4

(a)



(b)

Figure 5: (a) a policy P with $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) partial order $\prec_P$: $c_3 \prec_P c_1$, $c_3 \prec_P c_2$, $c_3 \prec_P c_4$, $c_2 \prec_P c_1$, $c_2 \prec_P c_4$.

that representations with multiple combined polices do not provide additional savings in the number of filters.

Class disjointness. Two classes c and c' are *disjoint* ($c \perp c'$), if there are no headers matching both c and c', otherwise, they *intersect*. Due to class disjointness, a permutation of classes in $\mathbb{S}(P)$ can lead to an *equivalent* policy yielding the same action as P on every header. For instance, in Figure 5a swap of c_4 and c_1 in $\mathbb{S}(P)$ lead to an equivalent policy. To define permutations of $\mathbb{S}(P)$ leading to policies that are equivalent to P, we introduce the following partial order $\prec_P$ of classes in $\mathbb{S}(P)$ (see Figure 5b). We say that $c_i \prec_P c_j$ if at least one of the following conditions is satisfied: (1) c_i intersects with c_j and c_i appears before c_j in $\mathbb{S}(P)$; (2) there is a class $c_k \in P$ such that $c_i \prec_P c_k$ and $c_k \prec_P c_j$ (transitivity). Consider a policy P' obtained from P by permutation of classes in $\mathbb{S}(P)$. If $\prec_{P'}$ coincide with $\prec_P$ then P and P' are equivalent. The proposed below methods construct P_{comb} by partial orders of policies in $\mathcal{P}$.

Ideal representations. Let C be a set of all classes appearing in policies in $\mathcal{P}$. In the optimal case, the combined policy P_{comb} contains only one instance of every class from C, we call such P_{comb} as *ideal*. For a given $\mathcal{P}$, we identify conditions guaranteeing the existence of ideal representations and explain how to construct an ideal P_{comb}.

If any two different classes in C do not match the same headers, we can construct an ideal policy P_{comb} that contains all classes from C in any order. A header H can be looked up in P_{comb} instead of a configured policy P_i, and if the first matched class c belongs to P_i, the action of c in P_i is returned. Otherwise, the classification result is the default action in P_i. Note that, the pairwise disjointness of all classes in C is not necessary for the existence of ideal P_{comb}. To deal with more general structural properties, we must guarantee that for a header H incoming in the context of P_i, a first matched class $c \in P_{\text{comb}}$ belongs to P_i. To implement this requirement, we prepend each filter of a class c in P_{comb} with the ternary *policy prefix* $\text{pp}(c)$, and each header H incoming in the context of P_i with the binary *header prefix* pp_i satisfying the following property: $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$. One of the possible variants of $\text{pp}(c)$ and pp_i satisfying the defined above property can be the following: $\text{pp}(c)$ is a ternary bit string of length $|\mathcal{P}|$ such that $\text{pp}(c)_i = *$ if $c \in P_i$ and $\text{pp}(c)_i = 0$ otherwise; pp_i is a bit string $0 \ldots 010 \ldots 0$ of length $|\mathcal{P}|$ that has a 1 only at position i (see Figure 6a).

Now we define condition that still guarantee the existence of ideal representations and show how to build them. For this purpose, we introduce a notion of the *joint graph* G^{jnt} for a set of policies $\mathcal{P}$ over classes C; this is a directed graph $G^{\text{jnt}}(\mathcal{P}) = (C, E^{\text{jnt}})$, where E^{jnt} contains an edge from c_i to c_j for $c_i, c_j \in C$ if and only if $c_i \prec_P c_j$ for at least one policy $P \in \mathcal{P}$ (see Figure 6b).

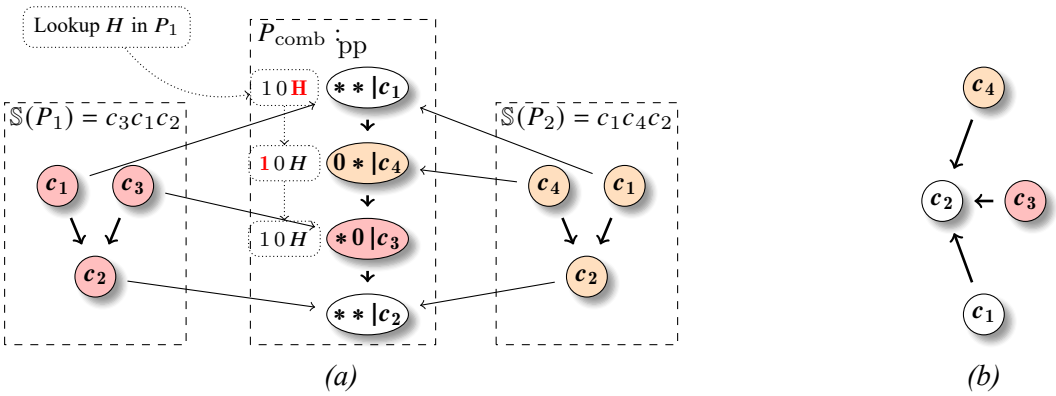


Figure 6: (a) $\mathcal{P} = \{P_1, P_2\}$, ideal P_{comb} and policy prefixes; (b) $G^{\text{jnt}}(\mathcal{P})$.

Theorem. *For a given set of policies $\mathcal{P}$, there exists an ideal P_{comb} if the corresponding G^{jnt} is acyclic.*

This theorem implies an algorithm constructing an ideal P_{comb}: we take a topological order of classes in G^{jnt} and prepend each class by a policy prefix as described above.

Non-ideal representations. For the cases when G^{jnt} is not acyclic we construct combined policy containing duplicated class instances. In the following we say that a sequence $\mathbb{S}$ defining the order of class instances in P_{comb} is *compatible* with a policy $P_i \in \mathcal{P}$ if there exists a subsequence $\mathbb{S}'_i$ of $\mathbb{S}$ that consists of a single instance of every class in P_i and for any two classes $c_j, c_t \in P_i$, if $c_j \prec_{P_i} c_t$ then c_j appears before c_t in $\mathbb{S}'_i$. A sequence $\mathbb{S}$ of the constructed P_{comb} should be compatible with all policies in $\mathcal{P}$. Only instances of classes from $\mathbb{S}'_i$ participate in the classification by policy P_i, i.e., in the corresponding P_{comb} only for them the ith bit of the policy prefix is set to $*$, while for all other instances the ith bit of the policy prefix is set to zero. Clearly, the number of filters in classes should be taken into account during class duplications. We denote by $\mathrm{W}^+(\mathbb{S})$ the total number of filters in duplicated class instances in $\mathbb{S}$.

Problem (Policy Sequence Packing, PSP)**.** *Given a set of policies $\mathcal{P}$, find a sequence of classes $\mathbb{S}$ compatible with all policies in $\mathcal{P}$ that minimizes $\mathrm{W}^+(\mathbb{S})$.*

Theorem. *Unless $\mathrm{P} = \mathrm{NP}$, there is no polynomial algorithm for the PSP problem with a constant approximation factor on $\mathrm{W}^+(\mathbb{S})$.*

We propose algorithms `AllOrOne` and `CliqueShare` solving PSP. These algorithms exploits solution of the *Weighted Feedback Vertex Set* (WFVS) problem [36], which is NP-complete. The *feedback vertex set* is a set of vertices in a vertex-weighted directed graph $G = (V, E)$ such that the removal of these vertices makes G an acyclic; the WFVS problem asks to find a feedback vertex set of minimal total weight. For instance, the work [38] proposes an algorithm for WFVS with approximation factor $O(\log |V| \log \log |V|)$, but there are other alternatives [37]. We denote by $\alpha(G)$ the approximation factor of an algorithm for the WFVS problem on a graph G. Note that WFVS problem is not harder than PSP.

AllOrOne. Recall that, the main reason for class duplications are cycles in the joint graph. Firstly, `AllOrOne` constructs G^{jnt}; then finds a feedback vertex set V^{wfvs} in G^{jnt} with minimal total weight, where vertex weight equals the number of filters in the corresponding class; and then constructs $\mathbb{S}$ by V^{wfvs}. An induced subgraph on vertices that are not in V^{wfvs} is acyclic, therefore, the corresponding classes appear only once in $\mathbb{S}$. For a class $c \in V^{\text{wfvs}}$, $\mathbb{S}$ contains a separate c instance for each policy containing c.

`AllOrOne` correctly solves the PSP problem and its approximation fact is at most $\alpha(G^{\text{jnt}}) \cdot (|\mathcal{P}| - 1)$ (see Theorem 9 and Theorem 10 in Appendix A, journal version).

Theorem. *The approximation factor of the `AllOrOne` algorithm is at least* $|\mathcal{P}| - 1$.

CliqueShare. The efficiency of an algorithm based on WFVS heavily depends on the information about $\mathbb{S}$ provided by FVS. In the `AllOrOne` algorithm this information is very limited: FVS only provides the set of classes appearing in $\mathbb{S}$ more than once. To overcome this limitation, we propose another algorithm `CliqueShare`: for each class c, FVS in `CliqueShare` provides the pairs of policies containing c that do not share the instance of c in $\mathbb{S}$. In the `CliqueShare` algorithm we construct another graph G^{pair} allowing to operate with a finer resolution. For each class c and each pair A of policies containing c, G^{pair} contains a vertex c^A. For each P_i and any two classes $c_1 \prec_{P_i} c_2$, G^{pair} has an edge $(c_1^A, c_2^{A'})$ for all pairs A, A' containing P_i. At the beginning, `CliqueShare` finds a feedback vertex set V^{wfvs} in G^{pair} with minimal total weight. If c^A is in V^{wfvs} then the resulting $\mathbb{S}$ contains different instances of c for the policies $P_i, P_j \in A$. A set of policies can share the same instance of a class c if for any two policies from this set, the corresponding vertex for a class c in G^{pair} is not in V^{wfvs}, we call such sets as *admissible* subsets. For each class c, `CliqueShare` finds a partition of the set of policies containing c into minimal number of admissible subsets.

Theorem. *`CliqueShare` correctly solves the PSP problem.*

Theorem. *`CliqueShare` has an approximation factor of at most* $\alpha(G^{pair}) \lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

Theorem. *The approximation factor of `CliqueShare` is at least* $\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

The approximation factor of `CliqueShare` is quadratic on $|\mathcal{P}|$ and worse than for `AllOrOne` for all $|\mathcal{P}| > 3$. But on practice, `CliqueShare` significantly outperforms `AllOrOne` since it operates with a better resolution.

Local optimizations. Both `AllOrOne` and `CliqueShare` algorithms can be further improved by two additional optimizations. These algorithms construct an auxiliary acyclic graph G^*, whose topological order of vertices forms the resulting $\mathbb{S}$. The first optimization `GreedyGluing` iteratively shrinks pairs of vertices in G^* corresponding to the same class while G^* remains acyclic. At every iteration, `GreedyGluing` peeks vertices with the maximum weight. The time complexity of `GreedyGluing` is $O(n^3)$, where n is the number of vertices in G^*.

The second optimization comes from the fact that the proposed algorithms do not usually guarantee that $\mathbb{S}$ will be a *local minimum* solution, i.e., it might happen that one can remove some class instances from the resulting $\mathbb{S}$ and still get a valid sequence for P_{comb}. We propose a `LocalDescent` post optimization that is defined in the following way: given $\mathbb{S}$, try to remove class instances in $\mathbb{S}$ one by one, while $\mathbb{S}$ is a valid solution.

Dynamic updates. Although economic models rarely change, support of dynamic updates in represented policies can be important in some deployment scenarios. We support two basic operations on policies in $\mathcal{P}$: (1) $\text{delete}(P, c)$, remove a class c from a policy P; (2) $\text{insert}(P, c, c_{succ})$, add a class c

to a policy P such that c appears in $\mathbb{S}(P)$ just before c_{succ}. After each operation P_{comb} should be updated properly. To support fast and efficient dynamic updates we propose method modifying P_{comb} after each operation in time $O(|\mathbb{S}| + \sum_{P \in \mathcal{P}} (|P| + D(P)))$, where $|\mathbb{S}|$ is a number of class instances in $\mathbb{S}$, $|P|$ is a number of classes in P and $D(P)$ is the number of intersecting class pairs in P (see Section 6 in Appendix A, journal version).

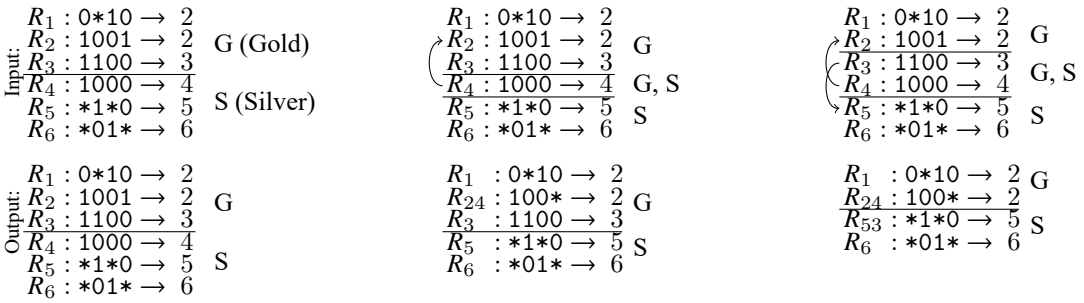
Combined representations consisting of multiple P_{comb}s. Here, we consider representations consisting of multiple P_{comb}s. In this case, due the to single lookup constraint, every input policy correspond to only one combined policy in the resulting representation. Hence, the representations consisting of multiple P_{comb}s do not allow to reduce the number of maintained filters compared to representations consisting of a single P_{comb}. In particular, usage of multiple combined policies does not allow to construct an ideal representation maintaining only one instance of every class if there is no ideal P_{comb} representing the given set of policies.

Theorem. *For a given set of policies, if there exists an ideal representation consisting of multiple combined policies, then there is an ideal representation consisting of a single P_{comb}.*

However, the usage of multiple combined policies allows to reduce lengths of extra prefixes in constructed combined policies. In the general case, the number of policies corresponding to the same combined policy in the constructed representation addresses a tradeoff between the maximum length of an extra prefix and the maximum possible reduction of filters in duplicated class instances. Theorem 6 in [35] defines a method allowing to construct an ideal representation with multiple P_{comb}s minimizing the number of policies in $\mathcal{P}$ represented by the same P_{comb}.

Evaluation. In the evaluation study, we compare total number of filters in all policies in $\mathcal{P}$ with the number of filters in the corresponding combined representations constructed by `CliqueShare`, `AllOrOne`, the algorithm `WSCS` that constructs the shortest weighted common supersequence [46] of all class sequences defining policies in $\mathcal{P}$, and the algorithm `MajorityMerge` proposed in [46] with (3, 1)-lookahead extensions [47, 48]. For each considered algorithm we also evaluate its version extended by additional optimizations. In particular, we extend all algorithms by `LocalDescent`, and additionally extend graph-based algorithms `AllOrOne` and `CliqueShare` by `GreedyGluing`.

We generate inputs with different values of the following main characteristics: (1) number of intersecting classes, (2) number of policies that contain a class, (3) total number of policies. The evaluation study confirms the usefulness of class-based abstractions in the optimization of policy classifiers both based on sequences and partial orders of classes, but the latter performs much better. The algorithm `CliqueShare` with `LocalDescend` and `GreedyGluing` optimizations significantly outperforms all other evaluated algorithms. Moreover, `CliqueShare` without local optimizations outperforms `WSCS` and `MajorityMerge` even if they followed by `LocalDescend` optimization, and constructs class sequences with almost the same value of $W^+(\mathbb{S})$ as `AllOrOne` with `LocalDescend` and `GreedyGluing`. See Section 7 in Appendix A (journal version) for more details.



(*a*) Irreducable classifier (*b*) R_4 can be in Gold or Silver (*c*) R_3, R_4 can be in G or S

Figure 7: Tradeoff between accuracy and required memory in approximate classifiers.

4.1.3 Approximate classifiers with controlled accuracy

In this project we consider approximate classification allowing to further minimize the number of rules in a given classifier by sacrificing the classification accuracy. The classification errors are controlled in the following way: each classification rule R_i has an action set $\mathcal{A}_i$ such that each action in $\mathcal{A}_i$ can be a classification result of a header classified by R_i. In this work, we consider classifiers whose filters are ternary bit strings. In the following we say that $\mathcal{K}$ is an *exact* classifier if for each rule $R_i \in \mathcal{K}$, $\mathcal{A}_i$ consists of a single element (i.e., $\mathcal{K}$ is a regular packet classifier); otherwise $\mathcal{K}$ is an *approximate* classifier.

We say that an approximate classifier $\mathcal{K}'$ obtained after the optimization of a given approximate classifier $\mathcal{K}$ is *approximately equivalent* to $\mathcal{K}$ if for every header H, the action set of the rule in $\mathcal{K}'$ classifying H is a subset of the action set of the rule in $\mathcal{K}$ classifying H. In the following we propose methods that construct approximately equivalent classifiers minimizing the number of classification rules.

Motivating examples. We begin with two motivating examples that introduce the idea of approximated representations. First, consider a common situation when the rules of a classifier map headers to a quantitative characteristic, e.g., desired latency, which is further aggregated into service classes which the packets are classified into. E.g., on Figure 7a values 1-3 are mapped to the *Gold* class; 4-6, to *Silver*. The classifier in Figure 7a is irreducible, i.e., no smaller classifier maps exactly the same headers to the same classes. However, let us consider "borderline" rules between service classes. Suppose that we can allow packets falling under R_4, which currently maps to *Silver* but has value 4, very close to the lower bound of the *G*old class, to be associated with either *Gold* or *Silver* classes. In this case we can perform further optimizations, associating R_4 with *Gold* or *Silver* to better reduce the classifier; Figure 7b shows how R_4 can be merged with R_2 into R_{24} that maps to *Gold*. We can take the approximation one step forward. Since R_3 has value 3, which is very close to the upper bound of the *Silver* class, we can allow R_3 to associate with *Silver* (Figure 7c), and classifier optimization can now exploit this additional flexibility, merging R_3 with R_5 to get R_{53}. Note that even in this specific example, not only borderline rules between different classes may be extended with additional actions. Although R_6 maps to 6, very far from the lower bound of the *Gold* class, it might still be allowed to extend R_6 with the *Gold* option due to some additional considerations (for instance, if we know *a priori* that required bandwidth for the traffic classified by R_6 is relatively small compared to the total volume of *Gold* traffic).

The second motivating example is of a different nature, showing that approximate classifier representations are a much more general tool. The scalability problem of forwarding tables (FIBs) is a largely

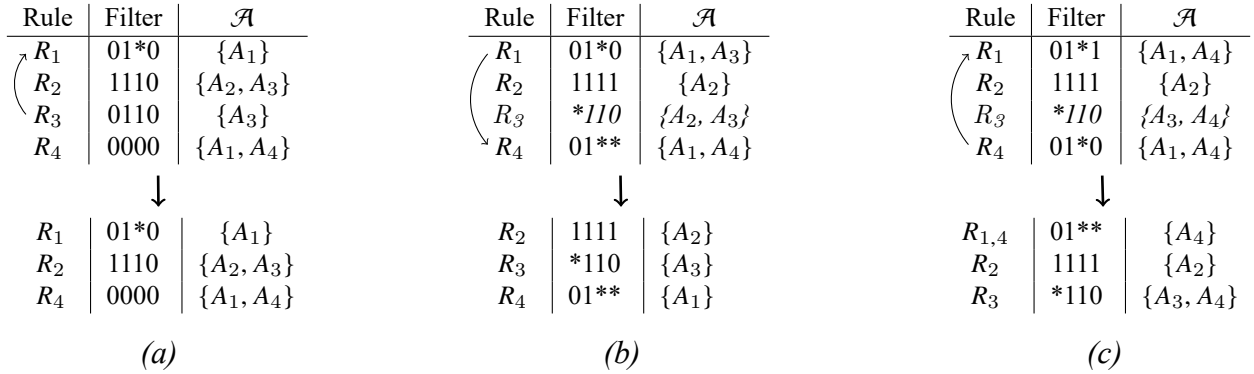


Figure 8: Illustration of the generalized optimization operators: (a) forward subsumption; (b) backward subsumption; (c) resolution.

unsolved problem to date [49]. One can run a third-party process that estimates the "quality" of different paths for the header space covered by a given traffic matrix. It turns out that by exploiting approximate representations and extending the actions of already computed FIBs with acceptable alternatives (based on the "quality" of estimated paths), we can often reduce the classifier size (see Figure 2 in Appendix B).

Generalized optimization operators. Optimization methods minimizing the size of classifiers [28, 29, 30] usually can be reduced to methods minimizing the size of Boolean expressions. Existing heuristics include a set of basic operators applied to a classifier while possible. They consist of two major blocks: *basic operators* and an *optimization process* that constructs a sequence of applied operations reducing the classifier. Usually, three basic operators are considered in the scope of packet classifiers [50] since they have the right balance between operational complexity and applicability. These basic operators decide when a rule can be removed or replaced by another rule. We generalize them for the approximate case. In the following we say that a rule R_i *covers* R_j if all headers matching R_j also match R_i.

Forward Subsumption $\mathcal{F}(R_i)$. This operator removes all unreachable rules and does not depend on the structure of action sets. Formally, a rule $R_i \in \mathcal{K}$ can be removed if there exists a rule that covers R_i and appears in $\mathcal{K}$ before R_i. Since forward subsumptions do not depend on actions, the resulting classifier is approximately equivalent to the original one. In Figure 8a, R_3 is removed since R_1 covers it.

Backward Subsumption $\mathcal{B}(R_i)$. Sometimes a rule can still be removed if it is covered by a rule appearing after R_i in $\mathcal{K}$. Formally, $R_i \in \mathcal{K}$ can be removed if the following conditions hold: (1) there is a rule R_j such that R_j covers R_i, R_j appears in $\mathcal{K}$ after R_i, and $\mathcal{A}_j \cap \mathcal{A}_i \neq \emptyset$; (2) $\mathcal{A}_t \cap \mathcal{A}_i \neq \emptyset$ for every rule $R_t \in \mathcal{K}$ that intersects with R_i and is located in $\mathcal{K}$ between R_i and R_j; after applying backward subsumption, we set $\mathcal{A}_t := \mathcal{A}_t \cap \mathcal{A}_i$. If several R_j satisfies the first condition, we choose R_j appearing in $\mathcal{K}$ first. In Figure 8b, R_1 can be removed by backward subsumption since R_1 is covered by R_4, even though R_1 intersects with R_3 since R_1 and R_3 have intersecting action sets; after applying this backward subsumption action sets $\mathcal{A}_3$ and $\mathcal{A}_4$ are properly modified.

Resolution $\mathcal{R}(R_i, R_j)$. This operator comes from propositional proof theory: in propositional logic, an expression $(x \wedge C) \vee (\bar{x} \wedge C)$ is equivalent to C. Two rules R_i, R_j $(i < j)$ can be combined and replaced by a new rule $R_{i,j}$ (in place of R_i) if: (1) filters F_i and F_j coincide in all bit indices except k, and the k-th bit is not $*$ in F_i or F_j (2) the set $\mathcal{A}'_i = \mathcal{A}_i \cap \mathcal{A}_j \cap \bigcap_{t:i<t<j \text{ and } R_t \text{ intersects with } R_j} \mathcal{A}_t$ (intersection of action sets over all intersecting rules) is nonempty; after applying resolution we assign $\mathcal{A}'_i$ to $R_{i,j}$. In

Figure 8b, filters of R_1 and R_4 differ only in the last bit; we can apply resolution on R_1 and R_4 since $\mathcal{A}_1 \cap \mathcal{A}_3 \cap \mathcal{A}_4 \neq \emptyset$

In general, we can choose a single action for every rule in an approximate classifier, and each such combination will be an exact classifier. In this way we could exploit basic operators minimizing exact classifiers. The problem is not only the exponential number of different exact classifiers but also that generalized versions of the basic operators extend their applicability. We show the following two approximate classifiers, where generalized operators can achieve better results than even brute-force search over all exact classifiers:

Rule	Filter	$\mathcal{A}$
R_1	010*	$\{A_1, A_2\}$
R_2	0**0	$\{A_1\}$
R_3	**0*	$\{A_2\}$
R_4	01**	$\{A_1, A_2\}$

Rule	Filter	$\mathcal{A}$
R_1	00*1	$\{A_1\}$
R_2	11*0	$\{A_2\}$
R_3	**0*	$\{A_1, A_2\}$
R_4	10*0	$\{A_2\}$
R_5	01*1	$\{A_1\}$

In the first classifier, R_1 can be removed by a backward subsumption: R_4 covers R_1, and all necessary conditions hold. On the other hand, in neither of the four possible exact specializations of a first classifier we can reduce R_1 or any other rule. In the second classifier two resolutions $\mathcal{R}(R_1, R_5)$ and $\mathcal{R}(R_2, R_4)$ can be applied, but in any exact specialization only one of them remains.

Properties of operation sequences. An *optimization process* consists of a set of introduced above basic operators and a *heuristic* that chooses an *operation sequence* consisting of applied operations. We study properties of operation sequences. By definition, each applied operation removes exactly one rule, so after applying the operation sequence $\mathcal{S}$ on a classifier $\mathcal{K}$, the optimized classifier has $|\mathcal{K}| - |\mathcal{S}|$ rules. Therefore, considered heuristics maximizes a length of $\mathcal{S}$. A combinatorial search of longest operation sequences is a hard problem even in the case of exact classifier. We show that this problem becomes even more complex in the case of approximate classifiers.

Firstly, we consider a set of operations $\mathbf{O}$ applicable to the initial classifier that does not contain *conflicting* operations using the same rule. The longest operation sequence containing only operations in $\mathbf{O}$ provide us a lower bound on the length of the optimal operation sequence. Operations in $\mathbf{O}$ implicitly depend on each other; for example, a resolution operation can create a new possible rule that will intersect with another rule in a backward subsumption. The following theorem shows that in the case of exact classifier, we can apply all operations in $\mathbf{O}$ regardless of these implicit dependencies. Despite the popularity of Boolean minimization, to the best of our knowledge we know of no prior work where this result was proven.

Theorem. *In the case of exact classifiers, there is an operation sequence containing all operations in* $\mathbf{O}$.

Corollary. *The length of an optimal operation sequence in the case of exact classifiers is at least the size of biggest* $\mathbf{O}$*; moreover, this lower bound can be computed in time* $O(N^2 \cdot l + C_{\mathcal{R}}^{1.5})$*, where* $C_{\mathcal{R}}$ *is the number of resolutions applicable in the original classifier.*

In the approximate case, $\mathbf{O}$ can increase compared to the exact case since generalized operations are applicable more often. But unlike the exact case, an operation sequence of length $|\mathbf{O}|$ is not guaranteed to exist. Consider the following approximate classifier:

Rule	Filter	$\mathcal{A}$
R_1	011*	$\{A_1\}$
R_2	11*1	$\{A_1, A_2\}$
R_3	*11*	$\{A_1, A_2\}$
R_4	01*1	$\{A_2\}$

In this classifier we can apply resolution $\mathcal{R}(R_2, R_4)$ or backward subsumption $\mathcal{B}(R_1)$, but we cannot apply both. In general, the complexity of constructing the longest operation sequence on **O** arises from backward subsumptions.

Theorem. *In the approximate case, there is an operational sequence containing all forward subsumptions and resolutions from* **O**.

If **O** has backward subsumptions then finding the longest sequence of operations from **O** turns into an intractable problem.

Theorem. *In the approximate case, finding a longest sequence of applicable backward subsumptions from* **O** *is NP-complete problem, and it is not approximable up to a constant factor in polynomial time.*

Now we consider general operation sequences containing operations that are non necessary in **O**. Any operation sequence (including longest) can be modified according to the following lemma.

Lemma. *For an operation sequence $\mathcal{S}$, there exists another operation sequence $\mathcal{S}'$ that satisfies the following: (1) $\mathcal{S}'$ consist of the same operations, but a backward subsumption $\mathcal{B}(R)$ can be changed to a forward subsumption $\mathcal{F}(R)$; (2) resolutions and forward subsumptions are applied before backward subsumptions; (3) for each pair of backward subsumptions $\mathcal{B}(R_i)$,$\mathcal{B}(R_j)$ $(i < j)$, $\mathcal{B}(R_j)$ is applied before $\mathcal{B}(R_i)$; (4) for each rule R_i, its final action set $\mathcal{A}_i$ obtained after applying $\mathcal{S}$ is a subset of the action set $\mathcal{A}'_i$ obtained after applying $\mathcal{S}'$.*

This lemma can be used for the development of heuristics finding longest operation sequences. Also, after the proposed transformation of the operation sequence $\mathcal{S}$ additional operations may be applied even if $\mathcal{S}$ before the transformation can not be extended by any other operation. Hence, this transformation can be also used as a post optimization (see Example 6 in Appendix B).

Evaluation. We evaluate the heuristic `aBM` applying introduced above minimization operators. Firstly, `aBM` applies forward subsumptions and resolutions while possible, then backward subsumptions. The `aBM` algorithm has been evaluated on on five IPv4 FIB classifiers used in real systems and the classifiers generated by *Classbench* [40] framework. The action sets of rules in the considered classifiers were generated synthetically according to packet forwarding and QoS use cases. Generally, evaluations fully support our basic idea and theoretical results: allowing for a small error in the actions of a classifier can lead to significant space reductions, often comparable to the reduction in the best case when all rules have the same action. See Section 7 in Appendix B for more details.

4.2 Robust Distributed Monitoring of Traffic Flows

In this section we describe an approach proposed in the paper "Robust Distributed Monitoring of Traffic Flows" (see Appendix C and [51]). To reduce memory requirements in a single network element, this paper proposes methods supporting distributed flow packet counters.

Flow f enters the network in source switch S and exits in destination switch D. At switch S, the flow consists of $|f|$ packets $p_0, p_1, \ldots, p_{|f|-2}, p_{|f|-1}$. The corresponding packet counter c calculates $|f|$. In the proposed distributed approach, c is divided into two chunks c_1 and c_2 maintained by S and D, respectively; c_1 consists of n bits. Network noise might drop or reorder f packets before they arrive to D. In the following we propose methods that support c_1 and c_2 in consistent states under network noise. After the termination of f, the proposed algorithms recover $|f|$ by c_1 and c_2; while f is in a progress, c_2 allows to obtain a real-time lower bound on the number of packets in f that source S has already send. Communication between counter chunks is in-band via packets of the monitored unidirectional flow. Source S piggybacks at most t control bits on each packet.

Partial network noise. Firstly we consider two extreme cases of network noise: (1) packet reordering without loss and; (2) packet loss without reordering. In the first case, the following simple solution robustly computes $|f|$: the source switch counts packets in f and, only when local counter chunk c_1 overflows, marks a control bit in the sent packet; the destination switch increments its counter chunk c_2 only upon receiving a packet with the marked control bit. This solution is fully resilient to packet reordering and greatly vulnerable to packet loss: it might massively underestimate $|f|$ upon losing a packet with the marked control bit. In the case of packet loss without reordering, the source switch can set a control bit in each sent packet to the most significant bit of local counter chunk c_1 and then increment c_1. The destination switch increments its counter chunk c_2 only upon receiving a packet where the control bit is set differently than in the previously received packet. This solution is resilient to loss of up to $2^{n-1} - 1$ consecutive packets, but is highly vulnerable to packet reordering, e.g., when packets $2^{n-1} - 1$ and 2^{n-1} of the flow arrive to D in reverse order, D incorrectly increments its counter chunk c_2 thrice instead of once: upon receiving packets 2^{n-1}, $2^{n-1} - 1$, and $2^{n-1} + 1$.

General network noise. In Appendix C we have proposed Algorithm 1 supporting robust computation when network noise comprises both reordering and loss of packets. This algorithm employs *sync bits*, a common t-bit portion of counter chunks in S and D, to synchronize the two counter chunks. The sync bits correspond to the t most significant bits in c_1 and t least significant bits in c_2, which means that they make up the middle bits in the resulting merged two-chunk counter c. When switch S receives packet p, the switch writes the sync bits from c_1 into header $h[p]$ of packet p and increments its n-bit counter chunk c_1. When packet p arrives to D, the destination switch computes the difference between $h[p]$ and the t sync bits in counter chunk c_2 by modulo 2^t. If this difference is between 1 and 2^{t-1}, switch D adds it to c_2. To recover $|f|$ after termination of f, Algorithm 1 sets the n most significant bits of the resulting counter c to c_2, and then adds to c the difference between c_1 and the n least significant bits of c calculated by modulo 2^n.

Resilience of any distributed solution is subject to fundamental feasibility limits, e.g., no such solution is able to handle loss of all packets. We employ two parameters to constrain packet reordering and packet loss: *reordering parameter* R captures the maximal distance of packet reordering, i.e., the destination switch can receive packet p_j before packet p_i only if $j \leq i + R$, and *loss parameter* L is the limit on *consecutive* packet losses, i.e., the destination switch receives at least one packet from any interval $p_i, \ldots, p_{i+L}$. The following theorem states the tolerance of Algorithm 1 to the network noise.

Theorem. *Algorithm 1 computes $|f|$ correctly if: $L + R < 2^{n-1}$ packets and $R \leq 2^{n-1} - 2^{n-t}$ packets.*

Parameter t controls a trade-off between the reordering resilience and per-packet communication overhead. Setting t to n provides the largest reordering resilience, which is only 14% larger than with $t = 4$ bits. Setting t to 1 bit reduces Algorithm 1 to the solution proposed for the network noise without packet reordering. Hence, we recommend setting t to 2, 3 or 4 bits in practice. Algorithm 1 is highly resilient to network noise in practical settings. For example, by allocating only $n = 8$ bits for the counter chunk in switch S and using $t = 2$ of them as sync bits, Algorithm 1 guarantees its correct computation of $|f|$ under any reordering $R < 64$ packets and $L + R < 128$ packets, which constitute significant levels of network noise. Moreover, doubling the value of t from 2 to 4 bits raises the value of R to 112 packets.

Impact of noise representation. Here we propose an alternative representation that bounds not only stretch but also frequency of network noise: we partition the flow at switch S into groups of 2^k consecutive packets and introduce $\mathrm{span}(\gamma, k)$ as a packet sequence where each packet is followed by a packet from the same group or γ subsequent groups ($\gamma > 0$). Formally, $\mathrm{span}(\gamma, k)$ is defined as a sequence of packets p_{z_i} from flow f with the following properties: (1) indices z_i form an increasing sequence; (2) switch D receives all packets p_{z_i}; (3) D receives p_{z_i} before p_{z_j} if $i < j$; (4) $\left\lfloor \frac{z_{i+1}}{2^k} \right\rfloor - \left\lfloor \frac{z_i}{2^k} \right\rfloor \leq \gamma$ for every i, i.e., $p_{z_{i+1}}$ is at most γ groups of 2^k packets away from p_{z_i} at switch S; (5) $\left\lfloor \frac{z_0}{2^k} \right\rfloor \leq \gamma$ and $\left\lfloor \frac{|f|-1}{2^k} \right\rfloor - \left\lfloor \frac{z_{r-1}}{2^k} \right\rfloor \leq \gamma$, which represent boundary conditions for the first and the last packets in the span. If $\mathrm{span}(\gamma, k)$ exists, condition 4 ensures that packet reordering and loss cannot affect all $\gamma \cdot 2^k$ packets from γ consecutive groups. With this representation of network noise, the maximum extent of packet reordering is still R, and $\mathrm{span}(\gamma, k)$ bounds packet loss so that to imply $L \leq (2^\gamma + 1) \cdot 2^k - 1$.

This network noise representation enables Algorithm 2 (see Appendix C), that operates correctly under more relaxed conditions compared to Algorithm 1. Algorithm 2 sets k to $n - t$ and differs from Algorithm 2 only in the procedure updating c_2; now switch D updates c_2 only when the difference between $h[p]$ and the t sync bits in counter chunk c_2 does not exceed γ by modulo 2^t.

Theorem. *`Algorithm 2` computes $|f|$ correctly if:*

$$R \leq 2^n - (\gamma + 1) \cdot 2^{n-t} \text{ packets and } \exists\, \mathrm{span}(\gamma, n - t). \tag{1}$$

This theorem exposes a variety of trade-offs. As with our original representation of network noise, a larger t value increases not only the maximum extent of reordering resilience but also the overlap between c_1 and c_2. Moreover Algorithm 2 accommodates more relaxed traffic patterns under the same reordering constraint for a larger value of t.

An increase in γ tightens the maximum extent of packet reordering and relaxes the existence conditions for $\mathrm{span}(\gamma, k)$. Hence, parameter γ implements a trade-off between the constraints on packet reordering and loss. Setting γ to 1 leads to maximal reordering $R = 2^n - 2^{n-t+1}$ packets. With $\gamma = 2^{t-1}$, Algorithm 1 is identical to Algorithm 2, and the following theorem shows that our first representation of network noise is a special case of the second one.

Theorem. *When $\gamma = 2^{t-1}$, conditions 1 imply conditions 2.*

Settings with $\gamma \geq 2^{t-1}$ are useful when loss is more frequent than reordering. When $\mathrm{span}(\gamma, n - t)$

does not exist, Algorithm 2 underestimates $|f|$, and we derive a bound on this underestimation:

Theorem. *When there is no* $\mathrm{span}(\gamma, n-t)$*, and the total of X packets are lost, Algorithm 2 computes* $|f|$ *with an underestimation of at most* $\frac{2^t X}{2\gamma+1-2^t}$ *packets if:*

$$R \leq 2^n - (\gamma+1)\cdot 2^{n-t} \text{ packets and } \gamma \geq 2^{t-1} \text{ groups.} \tag{2}$$

When the constraint on R is satisfied, and there is no $\mathrm{span}(\gamma, n-t)$, Algorithm 2 with $t = 2$ sync bits and $\gamma = 2$ groups underestimates the flow size by at most $4X$ packets. With 3 sync bits and $\gamma = 6$ groups, the underestimation is at most by $1.6X$. When $\gamma = 2^t - 1$ groups, Algorithm 2 does not tolerate any reordering and increases its loss resilience to $L < 2^n - 2^{n-t}$ packets. When loss is unbounded, and there is no reordering, Algorithm 2 with such values of γ underestimates the flow size by at most $\frac{X}{1-2^{-t}}$ packets.

From the practical perspective, conditions 2 make Algorithm 2 more robust to reordering and loss than Algorithm 1. For example, when the source chunk contains 8 bits and $\gamma = 1$ group, the settings with 2 and 3 sync bits support $R \leq 128$ packets and $R \leq 224$ packets if D's packet sequence contains $\mathrm{span}(1, 64)$ and $\mathrm{span}(1, 32)$, respectively. For $n = 8$ bits, $t = 3$ bits, and $\gamma = 6$ groups, Algorithm 2 computes the flow size correctly when there exists $\mathrm{span}(1, 192)$ and $R \leq 32$ packets.

Evaluation. Evaluations follow the same approach as those for VL2 [42], pFabric [43], and pHost [44]. Specifically, we perform simulations on realistic traffic traces generated from the data-mining distribution of flow sizes [42], where the number of flows is 10^6, and the maximum number of packets in a flow equals $\frac{2}{3}\cdot 10^6$ packets, which requires a 20-bit counter. In the evaluated scenarios, a 7-bit (or sometimes even a 6-bit) per-flow source-counter chunk is sufficient to maintain distributed packet counter correctly for each flow. Moreover, even in the case of 5-bit source-counter chunks, Algorithm 2 and Algorithm 1 correctly compute the number of packets in 99.3% and 95.8% of the flows that contain at least 2^5 packets. Besides, when Algorithm 2 computes $|f|$ imprecisely, the computed value almost always lies in interval $[|f| - X, 1.02\cdot|f|]$, where X is a number of lost packets. See Section 7 in Appendix C for more details.

5 Conclusion

This dissertation explores fundamental problems in the design of network elements, following the specific goals and objectives introduced in Section 1.2. We have made important contributions in efficient representations of packet processing programs and robust distributed traffic monitoring.

First, to minimize the encoding size of range-based classifier representations, we introduced a notion of subranges allowing to operate on ranges with per-bit resolution and overcome constraints of representations constructed with per-field resolution. A subrange is a basic element of the proposed equivalent RR representations preserving rule disjointness on a subset of bits. We have shown that RR representations provide substantial saving in required TCAM memory.

Second, we have proposed novel combined representations for multiple classifiers sharing the same classification patterns. We have studied the case when each classification pattern can be stored only once regardless of the number of classifiers containing it. In the general case we have proposed algorithms

minimizing the number of duplicated filters in the constructed combined representations. The effectiveness of these algorithms has analytical guarantees and is supported by the evaluation study.

Third, we have generalized the classical packet classification problem, introducing a new abstraction of approximate classifiers, where users control the accuracy by labeling in advance which filters can be assigned to which actions. We have designed optimization methods exploiting this additional flexibility in actions to optimize the classifier size even more. The approximate classifiers exploit an interesting and unexplored tradeoff between required resources and accuracy of results.

Fourth, to reduce the memory requirements in a single switch we propose a distributed approach improving scalability of per-flow traffic monitoring by exploiting resources of the whole network. Instead of giving up the monitoring accuracy, the approach enabled scalable exact reconstruction of packet flow counters by involving network elements having spare resources. To make distributed execution of a stateful monitoring task robust against network noise, we adhered to the open-loop paradigm that introduced no extra packets and communicated flow state in-band by piggybacking few control bits on packets of the monitored flows. We develop algorithms for exact robust distributed computation of the number of packets in a flow with analytically established conditions guaranteeing their correctness. This approach can be further extended to maintain real-time values of telemetry functions (e.g., packet loss).

In this thesis we develop methods improving the efficiency of packet processing programs and reducing memory requirements in a single switch. Our results can significantly improve computational capabilities of the entire network and each involved element. More advanced services and network-wide behaviors require implementation of more expressive and efficient packet processing engines. This dissertation shows the feasibility of such implementations even on existing network infrastructure.

References

[1] OpenFlow 1.3 specification, 2012. — http://www.openflow.org/ wk/index.php/OpenFlow_1_3_proposal. — 2012.

[2] $P4_{16}$ Language Specification. — https://p4.org/p4-spec/docs/P4-16-v1.2.0.html. — 2019.

[3] Synopsys. DesignWare Ternary and Binary Content-Addressable Memory Compilers. — https://www.synopsys.com/dw/ipdir.php?ds=dwc_tcam_memory_compilers.

[4] Cvetkovski A. An Algorithm for Approximate Counting Using Limited Memory Resources // SIGMETRICS. — 2007. — P. 181–190.

[5] Discount Counting for Fast Flow Statistics on Flow Size and Flow Volume / Chengchen Hu [et al.] // IEEE/ACM Trans. Netw. — 2014. — Jun.. — Vol. 22, no. 3. — P. 970–981.

[6] Stanojevic R. Small Active Counters // INFOCOM. — 2007. — P. 2153–2161.

[7] Tsidon E., Hanniel I., Keslassy I. Estimators Also Need Shared Values to Grow Together. // INFOCOM. — 2012. — P. 1889–1897.

[8] Cormode G., Muthukrishnan S. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications // J. Algorithms. — 2005. — Apr.. — Vol. 55, no. 1. — P. 58–75.

[9] Estan C., Varghese G. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice // ACM Trans. Comput. Syst. — 2003. — Aug.. — Vol. 21, no. 3. — P. 270–313.

[10] Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams / Y. Tong [et al.] // Proc. VLDB Endow. — 2017. — Aug.. — Vol. 10, no. 11. — P. 1442–1453.

[11] One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon / Z. Liu [et al.] // SIGCOMM. — 2016. — P. 101–114.

[12] Elastic Sketch: Adaptive and Fast Network-Wide Measurements / T. Yang [et al.] // SIGCOMM. — 2018. — P. 561–575.

[13] Fast and Scalable Layer Four Switching / Venkatachary Srinivasan [et al.] // SIGCOMM 1992. — 1998. — P. 191–202.

[14] Lakshminarayanan Karthik, Rangarajan Anand, Venkatachary Srinivasan. Algorithms for advanced packet classification with ternary CAMs // SIGCOMM. — 2005. — P. 193–204.

[15] Bremler-Barr Anat, Hendler Danny. Space-Efficient TCAM-Based Classification Using Gray Coding // Trans. Computers. — 2012. — Vol. 61, no. 1. — P. 18–30.

[16] Exact Worst Case TCAM Rule Expansion / Ori Rottenstreich [et al.] // IEEE Trans. Computers. — 2013. — Vol. 62, no. 6. — P. 1127–1140.

[17] Chang Yeim-Kuan, Su Cheng-Chien. Efficient TCAM Encoding Schemes for Packet Classification Using Gray Code // GLOBECOM. — 2007. — P. 1834–1839.

[18] DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors / Hao Che [et al.] // Trans. Computers. — 2008. — Vol. 57, no. 7. — P. 902–915.

[19] Bremler-Barr Anat, Hay David, Hendler Danny. Layered interval codes for TCAM-based classification // Computer Networks. — 2012. — Vol. 56, no. 13. — P. 3023–3039.

[20] SAX-PAC (Scalable And eXpressive PAcket Classification) / Kirill Kogan [et al.] // SIGCOMM. — 2014. — P. 15–26.

[21] FIB efficiency in distributed platforms / Kirill Kogan [et al.] // ICNP. — 2016. — P. 1–10.

[22] How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes / Sergey I. Nikolenko [et al.] // INFOCOM Workshops. — 2016. — P. 521–526.

[23] Chuprikov Pavel, Kogan Kirill, Nikolenko Sergey I. General ternary bit strings on commodity longest-prefix-match infrastructures // ICNP. — 2017. — P. 1–10.

[24] QoS: Modular QoS Command-Line Interface Configuration Guide, Cisco IOS. — https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_mqc/configuration/15-mt/qos-mqc-15-mt-book/qos-mqc.html.

[25] Class of Service Configuration Guide. — https://www.juniper.net/documentation/en_US/junos11.1/information-products/topic-collections/security/software-all/class-of-service/index.html.

[26] Configuring Modular QoS Packet Classification. — http://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-2/qos/configuration/guide/qc42clas.html.

[27] Mittal S. A Survey of Techniques for Approximate Computing // ACM Comput. Surv. — 2016. — Vol. 48, no. 4. — P. 62:1–62:33.

[28] Liu A., Meiners C., Torng E. TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMs // Trans. Networking. — 2010. — Vol. 18, no. 2. — P. 490–500.

[29] Compressing rectilinear pictures and minimizing access control lists / David A. Applegate [et al.] // SODA. — 2007. — P. 1066–1075.

[30] Packet classifiers in ternary CAMs can be smaller / Qunfeng Dong [et al.] // SIGMETRICS. — 2006. — P. 311–322.

[31] Umans C. The Minimum Equivalent DNF Problem and Shortest Implicants // J. Comput. Syst. Sci. — 2001. — Vol. 63, no. 4. — P. 597–611.

[32] Khot S., Saket R. Hardness of Minimizing and Learning DNF Expressions // FOCS. — 2008. — P. 231–240.

[33] Demianiuk Vitalii, Kogan Kirill. How to Deal with Range-Based Packet Classifiers // SOSR. — 2019. — P. 29–35.

[34] New Alternatives to Optimize Policy Classifiers / Vitalii Demianiuk [et al.] // ICNP. — 2018. — Sep. — P. 121–131.

[35] New Alternatives to Optimize Policy Classifiers / V. Demianiuk [et al.] // IEEE/ACM Transactions on Networking. — 2020. — Vol. 28, no. 3. — P. 1088–1101.

[36] Garey Michael R., Johnson David S. Computers and Intractability: A Guide to the Theory of NP-Completeness. — NY : W. H. Freeman & Co., 1979. — ISBN: 0716710447.

[37] Demetrescu Camil, Finocchi Irene. Combinatorial Algorithms for Feedback Problems in Directed Graphs // Inf. Process. Lett. — 2003. — Vol. 86, no. 3. — P. 129–136.

[38] Approximating Minimum Feedback Sets and Multicuts in Directed Graphs / G. Even [et al.] // Algorithmica. — 1998. — Feb. — Vol. 20, no. 2. — P. 151–174.

[39] Demianiuk V., Kogan K., Nikolenko S. I. Approximate Classifiers with Controlled Accuracy // INFOCOM. — 2019. — P. 2044–2052.

[40] Taylor David E., Turner Jonathan S. ClassBench: A Packet Classification Benchmark // IEEE/ACM Trans. Netw. — 2007. — Jun.. — Vol. 15, no. 3. — P. 499–511.

[41] YAPS: Yet Another Packet Simulator. — `http://wiki.github.com/NetSys/simulator/`.

[42] VL2: A Scalable and Flexible Data Center Network / A. Greenberg [et al.] // SIGCOMM. — 2009. — P. 51–62.

[43] pFabric: Minimal Near-optimal Datacenter Transport / Mohammad Alizadeh [et al.] // SIGCOMM. — 2013. — P. 435–446.

[44] pHost: Distributed Near-Optimal Datacenter Transport over Commodity Network Fabric / Peter X. Gao [et al.] // CoNEXT. — 2015. — P. 1:1–1:12.

[45] Karp Richard M. Reducibility among Combinatorial Problems. — 1972. — P. 85–103.

[46] Foulser David E., Li Ming, Yang Qiang. Theory and Algorithms for Plan Merging // Artif. Intell. — 1992. — Vol. 57, no. 2-3. — P. 143–181.

[47] A computational framework for optimal masking in the synthesis of oligonucleotide microarrays / Simon Kasif [et al.] // Nucleic Acids Research. — 2002. — Vol. 30, no. 20. — P. e106.

[48] Ning Kang, Leong Hon. Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm // BMC Bioinformatics. — 2006. — Vol. 7, no. Suppl 4. — P. S12.

[49] Elmokashfi A., Dhamdhere A. Revisiting BGP churn growth // CCR. — 2014. — Vol. 44, no. 1. — P. 5–12.

[50] Strategies for Mitigating TCAM Space Bottlenecks / Kirill Kogan [et al.] // HOTI. — 2014. — P. 25–32.

[51] Robust Distributed Monitoring of Traffic Flows / V. Demianiuk [et al.] // 2019 IEEE 27th International Conference on Network Protocols (ICNP). — 2019. — P. 1–11.

Appendices

A Paper “New Alternatives to Optimize Policy Classifiers”

Authors. Vitalii Demianiuk, Sergey Nikolenko, Pavel Chuprikov and Kirill Kogan

Abstract. Growing expressiveness of services increases the size of a manageable state at the network data plane. A service policy is an ordered set of classification patterns (classes) with actions; the same class can appear in multiple policies. Previous studies mostly concentrated on efficient representations of a single policy instance. In this work, we study space efficiency of multiple policies, cutting down a classifier size by sharing instances of classes between policies that contain them. In this paper we identify conditions for such sharing, propose efficient algorithms and analyze them analytically. The proposed representations can be deployed transparently on existing packet processing engines. Our results are supported by extensive evaluations.

New Alternatives to Optimize Policy Classifiers

Vitalii Demianiuk, Sergey Nikolenko, Pavel Chuprikov, Kirill Kogan

***Abstract*—Growing expressiveness of services increases the size of a manageable state at the network data plane. A service policy is an ordered set of classification patterns (classes) with actions; the same class can appear in multiple policies. Previous studies mostly concentrated on efficient representations of a single policy instance. In this work, we study space efficiency of multiple policies, cutting down a classifier size by sharing instances of classes between policies that contain them. In this paper we identify conditions for such sharing, propose efficient algorithms and analyze them analytically. The proposed representations can be deployed transparently on existing packet processing engines. Our results are supported by extensive evaluations.**

I. Introduction

Transport networks satisfy requests to forward data in a given topology. To guarantee desired data properties during forwarding, network operators impose economic models implementing various policies such as security or quality-of-service. As network infrastructure becomes more intelligent, the complexity of these policies is constantly growing.

Unfortunately, increasing manageable state on the data plane has its limitations. Traditionally, service policies are represented by packet classifiers whose implementations are usually expensive (e.g., *ternary content-addressable* memories, or TCAMs). Most existing works optimize each policy instance separately (see Section VIII). In this work, we exploit other alternatives to achieve additional efficiency of policy state represented on the data plane. Our ideas hinge on the fact that similar "classification patterns" (*classes*) are reused in different policies, where each class consists of ternary-bit filters determining a set of matched packet headers. Various vendors already support the notion of classes in policy declarations [2], [3] allowing to abstract and manage classification patterns more efficiently. For instance, Cisco IOS supports up to 256 different QoS policies and up to 4096 classes per box [4]. In real deployments, the number of classes per policy ranges from tens to hundreds depending on the application model [4]. The size of a class depends on the complexity of represented pattern.

Traditionally, a separate class instance is allocated for each policy instance that contains it (see policies P_1, P_2, and P_3 in Fig. 1a). Each allocated class instance has an attached action

The preliminary version of this paper entitled "New Alternatives to Optimize Policy Classifiers" has been appeared at ICNP 2018 [1]. The work V. Demianiuk and K. Kogan was partially supported by the Ariel Cyber Innovation Center in cooperation with the Israel National Cyber Directorate in the Prime Minister's Office, and by the Data Science and Artificial Intelligence Research Center at Ariel University. The work of Sergey Nikolenko shown in Sections III, IV, and V (in particular, Theorems 2, 8, 9, 10) has been supported by the Russian Science Foundation grant no. 17-11-01276.

V. Demianiuk and K. Kogan are with Ariel University, 40700 Ariel, Israel (e-mail: vitalii@ariel.ac.il, kirillk@ariel.ac.il). S. Nikolenko is with National Research University Higher School of Economics, 199034 St. Petersburg, Russia, and also with Steklov Institute of Mathematics at St. Petersburg, 191023 St. Petersburg, Russia (e-mail: snikolenko@gmail.com). P. Chuprikov is with IMDEA Networks Institute, 28918, Leganes, Spain, and also with USI Lugano, 6900 Lugano, Switzerland (email: pavel.chuprikov@usi.ch).

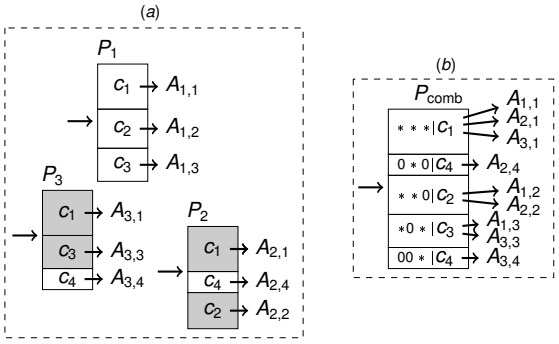


Fig. 1: (a) separate policies P_1, P_2, P_3; (b) a representation P_{comb} that emulates the policies; class instances that have been cut in P_{comb} are shown in gray.

specified by the corresponding policy (in Fig. 1a, an action $A_{i,j}$ is attached to the instance of c_j in the policy P_i). Since classes are used in different policies, it allows us to look at combined service policy representations, where ideally each class appears only once, providing substantial savings in representations of underlying classifiers in expensive memory such as TCAM. Usually, the complexity of structural properties of classifiers can be alleviated with additional classification lookups, but this is a shareable resource for the overall processing. The number of classification lookups per packet is one of the major constraints limiting line-rate characteristics. For instance, Cisco C12000 [5] supports at most six TCAM lookups per packet at line-rate for all services. As a result, in this work we prefer to consider combined policy representations that do not increase the number of classification lookups. Informally, proposed combined policy representations "emulate" the behaviours of represented policies.

Figure 1 illustrates major differences between the traditional attachment model, where a class instance is allocated per policy containing it, and the proposed combined representation P_{comb}. Note that P_{comb} stores a single instance of classes c_1, c_2, c_3. Class c_4 is duplicated since c_4 should be applied before c_2 in policy P_2 and after c_3 in policy P_3.

To emulate the classification of an incoming packet header by a policy P_i, the lookup process in P_{comb} should be performed only on class instances corresponding to the classes in P_i. For instance, in Fig. 1b, during the classification by P_3 the first instance of c_4 and c_2 in P_{comb} should be ignored. To achieve this, we prepend filters of every class instance in P_{comb} and incoming headers by special extra bits as described in Section III; e.g., on Fig. 1b the filters in c_3 are prepended by *0*, which indicates that c_3 belongs to P_1 and P_3.

In this work we propose equivalent combined representations for a given set of policies. We show the condition for the existence of *ideal representations* that contain only a single instance for every class of all policies and methods for constructing these representations. For the general case, we propose methods for minimizing the total number of filters in

duplicated class instances. All proposed representations do not increase the number of classification lookups, that is, we say that they satisfy the *single lookup constraint*; in other words, the lookup time complexity in such a representation is the same as in a single policy of the same size.

The paper is organized as follows. In Section III we explore ideal representations containing a single instance of every class in combined policy representations. Section IV proves that the proposed problem is intractable in the general case and shows how to deal with non-ideal representations. In Section V we propose two approximation algorithms and study them analytically for the offline case. Although the proposed algorithms can be extended for dynamic updates, in Section VI we propose a new algorithm that captures the right balance between time complexity and optimization results with dynamic updates. All proposed algorithms are evaluated in various settings in Section VII.

II. Model Description

In this section we first define the entities involved in the packet classification process and introduce our notation. A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits $h_i \in H$, $h_i \in \{0, 1\}$, $1 \leq i \leq w$; e.g., $(1\ 0\ 0\ 0)$ is a 4-bit header. We denote by $\mathcal{H}$ the set of all possible headers. A filter $F = (f_1, \ldots, f_w)$ is a sequence of w values corresponding to the header bits, but with possible values 0, 1, or $*$ ("don't care"). A header H *matches* a filter F if for every bit of H the corresponding bit of F has either the same value or $*$. Two filters are *disjoint* if there is no header that matches both filters.

Classes represent an intermediate level of abstraction: a *class* c is a set of filters. We denote by $w(c)$ the number of filters in c. A header H *matches* a class c if H matches at least one filter in c. Two classes (sets of filters) c and c' are *disjoint*, denoted by $c \perp c'$, if there are no headers matching both c and c' (all filters of c and c' are pairwise disjoint), otherwise, they *intersect*.

To define a *policy* P over a given set of classes $\mathcal{C}$, one needs to select a set of classes $\mathcal{C}_P \subseteq \mathcal{C}$ belonging to P, specify a sequence $\mathbb{S}(P)$ containing each class from $\mathcal{C}_P$ only once, and associate an action with each class in $\mathcal{C}_P$. For an incoming header, the action of a first matched class in $\mathbb{S}(P)$ is returned. If an incoming header is not matched by any class in $\mathbb{S}(P)$ then the policy *default* action is returned. Since classes can intersect, a policy is defined by a sequence rather than a set. Originally, classes were introduced to define common classification patterns [2], [3] that can significantly simplify policy management. In this way a single classification pattern should not be redefined during the declaration of another policy.

Two policies P_1 and P_2 are *equivalent* if for every given header both yield the same action. Note that different sequences on the same set of classes $\mathcal{C}_P$ can lead to several equivalent policies due to possible pairwise disjointness of classes in $\mathcal{C}_p$. For instance, Figure 2a shows a policy P defined by the sequence $\mathbb{S}(P) = c_3, c_2, c_4, c_1$, which is equivalent to P' defined by $\mathbb{S}(P') = c_3, c_2, c_1, c_4$ since c_1 and c_4 are disjoint.

To define policies that are equivalent to P on $\mathcal{C}_P$, we introduce the following partial order $\prec_P$ of classes in $\mathcal{C}_P$.

(a)

Class	Filter	#1	#2	#3	#4	Action
c_3	F_1	0	1	*	*	A_1
	F_2	*	1	*	0	A_1
c_2	F_3	*	*	0	0	A_2
c_1	F_4	1	0	*	*	A_3
c_4	F_5	0	0	*	*	A_4

(b)

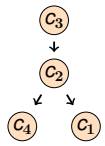


Fig. 2: (a) definition of the policy $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) partial order $\prec_P$: $c_3 \perp c_1, c_3 \perp c_4$, and $c_1 \perp c_4$.

We say that $c_i \prec_P c_j$ if at least one of the following conditions is satisfied:

- c_i intersects with c_j, c_i appears before c_j in $\mathbb{S}(P)$, and the actions assigned to c_i and c_j in P are different (disjointness constraint);
- there is a class $c_k \in P$ such that $c_i \prec_P c_k$ and $c_k \prec_P c_j$ (transitivity of the partial order).

For instance, Fig. 2a defines a policy P whose corresponding partial order $\prec_P$ is illustrated on Fig. 2b. In all illustrations of partial orders we omit arrows showing that $c_i \prec_P c_j$ if c_i precedes c_j in $\prec_P$ that follow from transitivity.

Observation 1. *The policy P' is equivalent to a policy P if $\prec_{P'}$ coincide with $\prec_P$ and each class in P' has the same action as in P (the default actions in P and P' should coincide).*

Informally, if we represent a policy P as a graph G with vertices corresponding to classes of P and edges corresponding to partial order constraints of $\prec_P$, then any topological order on the vertices of G forms a sequence of classes in the policy that is equivalent to P.

We denote by $\mathcal{P} = \{P_1, P_2, \ldots P_{|\mathcal{P}|}\}$ a set of $|\mathcal{P}|$ policies over the same set of classes $\mathcal{C}$; by $|\mathcal{C}|$, the number of classes in $\mathcal{C}$. Note that for an incoming header H, the corresponding policy P is retrieved from internal switch data structures, where the classification of H should be performed; in this case, we say that H is coming in the *context* of the policy P. A *combined policy* P_{comb} representing a group of policies $\mathcal{P}' \subseteq \mathcal{P}$ *emulates* $\mathcal{P}'$ if for any header H and any policy $P_i \in \mathcal{P}'$, the lookup of H in P_i and the lookup of H in P_{comb} in the context of P_i yield the same action. Informally, this means that P_{comb} mimics the behaviour of any policy in $\mathcal{P}'$.

In general, a *representation* $\mathcal{P}_{comb}$ implementing the classification in $\mathcal{P}$ can consist of more than one combined policy. Due to the single lookup constraint, a policy in $\mathcal{P}$ should be represented only by a single combined policy in $\mathcal{P}_{comb}$. Therefore, in order to construct $\mathcal{P}_{comb}$, the policies in $\mathcal{P}$ should be assigned into multiple disjoint groups $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$, where each group $\mathcal{P}^i$ is represented by a separate combined policy P^i_{comb}. By m we denote the number of groups. For every header incoming in the context of a policy $P \in \mathcal{P}$, the lookup is done in P^i_{comb} corresponding to the group $\mathcal{P}^i$ containing P. We say that $\mathcal{P}_{comb}$ *emulates* $\mathcal{P}$ if each $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ emulates the corresponding group of policies $\mathcal{P}^i$.

III. Ideal Representations

In traditional policy representations, if a single class (classification pattern) participates in multiple policies, per-policy instances of the class are allocated for every policy. Intuitively,

structural properties of induced policy classifiers should have a significant impact on memory requirements. We say that a combined representation $\mathcal{P}_{comb}$ of multiple policies $\mathcal{P}$ is ideal if $\mathcal{P}_{comb}$ contains a single instance of every class from $\mathcal{C}$. For a given $\mathcal{P}$, we propose a criteria of the existence of ideal representations (satisfying the single lookup constraint) and explain how to construct them. At this point, we assume that original policies from $\mathcal{P}$ are represented by $\mathcal{P}_{comb}$ consisting of a single combined policy P_{comb}; we will reconsider this assumption in Section III-D.

A. Disjoint classes

We begin with the simplest structural property, *class disjointness*, where any two different classes in $\mathcal{C}$ do not match the same headers. In this case we can construct an ideal policy P_{comb} that contains all classes from $\mathcal{C}$ in any order. A header H can be looked up in P_{comb} instead of a configured policy P_i, and if the first matched class c belongs to P_i (this can be verified with any set membership data structure), the action of c in P_i is returned. Otherwise, the classification result is the default action in P_i since only a single class in P_{comb} can match H.

B. Price of generalization

We have seen that class disjointness guarantees the existence of ideal representations. In Section III-C we will show that this structural property is not a necessary condition for the existence of ideal representations. Unlike the previous case, if the first class in P_{comb} matching a header H incoming in the context of a policy $P_i \in \mathcal{P}$ does not belong to P_i, there is no guarantee that P_{comb} does not contain a class from P_i matching H. Hence, to deal with more general structural properties, where classes in $\mathcal{C}$ are not pairwise disjoint, we must guarantee that the matched class $c \in P_{\text{comb}}$ belongs to P_i. To implement this requirement, we prepend each filter of a class c in P_{comb} with the ternary *policy prefix* $\text{pp}(c)$, and each header H incoming in the context of P_i with the binary *header prefix* pp_i satisfying the following property: $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$. Note that both policy and header prefixes have the same length. Such representations allow to match a header incoming in the context of P_i only against the classes in P_{comb} belonging to the original policy P_i. One of the possible variants of $\text{pp}(c)$ and pp_i satisfying the defined above property can be the following: $\text{pp}(c)$ is a ternary string of length $|\mathcal{P}|$ such that $\text{pp}(c)_i = *$ if $c \in P_i$ and $\text{pp}(c)_i = 0$ otherwise; pp_i is a bit string $0 \dots 010 \dots 0$ of length $|\mathcal{P}|$ that has a 1 only at position i.

Fig. 3a shows a sample lookup of a header H to P_1 in P_{comb} representing $\mathcal{P} = \{P_1, P_2\}$, where $\mathcal{C}$ contains non-disjoint classes (e.g., c_1 and c_2 are not disjoint in P_1). Observe that P_{comb} with policy prefixes emulates $\mathcal{P}$ and is ideal. The values in policy prefixes guarantee that only classes from P_1 participate in the lookup. Theorem 2 shows that adding $|\mathcal{P}|$ extra bits per filter in P_{comb} is unavoidable.

Theorem 2. *For any $l > 2$, there exists a set $\mathcal{P}$ of l policies such that $\mathcal{P}$ can be represented by an ideal P_{comb} and the length of prefixes should be at least l in any possible variant $\text{pp}(c)$ and pp_i satisfying that $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$.*

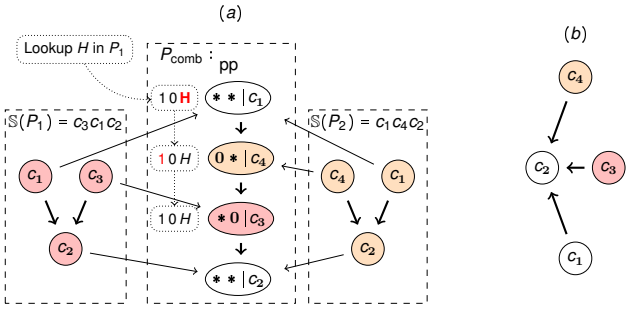


Fig. 3: (a) $\mathcal{P} = \{P_1, P_2\}$ with $\mathcal{C}$ containing non-disjoint classes, ideal P_{comb} and policy prefixes; (b) $G^{\text{jnt}}(\mathcal{P})$.

Proof. Consider a set $\mathcal{P}$ consisting of l policies over a set $\mathcal{C}$ of $2^l - 1$ different classes. A class $c_i \in \mathcal{C}$ belongs to the policy $P_j \in \mathcal{P}$ iff the jth bit in the binary representation of i is 1. E.g., $c_{5=00101_2}$ belongs to the policies P_1 and P_3. In each policy $P \in \mathcal{P}$, a class $c_i \in P$ precedes the class $c_j \in P$ ($c_i \prec_P c_j$) if $i < j$. Hence, for such $\mathcal{P}$ the order of classes in the ideal P_{comb} is unique and is defined by the class indices.

Note that each class c_i in $\mathcal{C}$ belongs to a unique set of policies, hence, there are should be at least 2^l different policy prefixes. Thus, the length of $\text{pp}(c)$ should be at least $\frac{l}{log_2 3} = O(l)$. In the remaining part of the proof, we are to show that the lower bound on the policy prefix length is exactly l.

We say that the kth bit of a policy prefix $\text{pp}(c_i)$ is *aggregating* if $\text{pp}(c_i)_k$ is '$*$' and at least two header prefixes with two different values in the kth bit position are matched by $\text{pp}(c_i)$. We denote by $|\text{pp}(c_i)|_*$ the number of aggregating bits in c_i. To prove the theorem, we are to show that for the set of policies introduced above, $|\text{pp}(c_{2^l-1})|_* \geq l$ in any valid assignment of header and policy prefixes.

Consider a sequence of classes $c_7, c_{15}, \dots, c_{2^i-1}, \dots, c_{2^l-1}$. The set of header prefixes matched by $\text{pp}(c_{2^i-1})$ consists of the header prefixes matched by $\text{pp}(c_{2^{i-1}-1})$ and the header prefix pp_i. Thus, $\text{pp}(c_{2^i-1})$ contains aggregating bits at the same bit positions as $\text{pp}(c_{2^{i-1}-1})$ and in at least one additional position. Hence, $|\text{pp}(c_{2^i-1})|_* > |\text{pp}(c_{2^{i-1}-1})|_*$ and $|\text{pp}(c_{2^l-1})|_* \geq |\text{pp}(c_7)|_* + l - 3$.

To finish the proof, it suffices to show that $|\text{pp}(c_7)|_* \geq 3$. Consider the policy prefixes $\text{pp}(c_1), \text{pp}(c_2), \text{pp}(c_4)$ of the classes $c_1 \in P_1, c_2 \in P_2, c_4 \in P_3$, where each one of them belongs to a single policy. These prefixes do not contain aggregating bits and at least two of them differs in at least two positions not containing $*$. W.l.o.g., we can assume that $\text{pp}(c_1), \text{pp}(c_4)$ differ in at least two bit positions not containing '$*$', hence, $\text{pp}(c_5)$ ($c_5 \in P_1, P_2$) has two aggregating bits at these positions. The policy prefix $\text{pp}(c_7)$ matches the same header prefixes as $\text{pp}(c_5)$ and the prefix pp_2. Thus, $|\text{pp}(c_7)|_* \geq |\text{pp}(c_5)|_* + 1 \geq 3$ and $|\text{pp}(c_{2^l-1})|_* \geq l$. □

C. When are representations ideal?

In this part we formulate the condition that still guarantee the existence of ideal representations and show how to build

them. For this purpose, we introduce the notion of a *joint graph* G^{jnt} for a set of policies $\mathcal{P}$ over classes $\mathcal{C}$; this is a directed graph $G^{\text{jnt}}(\mathcal{P}) = (\mathcal{C}, E^{\text{jnt}})$, where E^{jnt} contains an edge from c_i to c_j for $c_i, c_j \in \mathcal{C}$ if and only if $c_i \prec c_j$ in at least one policy in $\mathcal{P}$ (see Fig. 4b for an example).

Theorem 3. *For a given set of policies $\mathcal{P}$, there exists an ideal P_{comb} if the corresponding G^{jnt} is acyclic.*

Proof. If G^{jnt} is acyclic, we can construct an ideal representation from any topological order of the vertices of G^{jnt}: we put classes into P_{comb} in this order and prepend them by policy prefixes. This representation is correct since for every P_i and every $c, c' \in P_i$ such that $c \prec_{P_i} c'$ the class c appears in P_{comb} before c'. □

If G^{jnt} contains a cycle, then any possible linear ordering of classes in P_{comb} contradicts the order of classes in some policy P_i, i.e., for each P_{comb} there exists $c, c' \in P_i$ such that $c \prec_{P_i} c'$ but c' appears before c in P_{comb}. In this case, the ideal representation P_{comb} is correct only if for each pair of classes c, c' in P_{comb} contradicting $\prec_{P_i}$, each header belonging to the intersection of c and c' is matched by a class preceding c and c' in $S(P_i)$. Such property of P_{comb} appears rarely on practice and can not be verified in a polynomial time. Hence, we suggest to use the acyclicity of G^{jnt} as a criteria of the ideal representation existence.

The proof of Theorem 3 implies an algorithm that constructs an ideal representation if it exists. The time complexity of this algorithm equals to the time $T_{G^{\text{jnt}}}(\mathcal{P}) = O(\sum_{P \in \mathcal{P}} \left(|P|^2 + |P| \cdot D(P) \right))$ needed to construct G^{jnt}, where $D(P)$ is the number of intersecting class pairs from P that have different attached actions.

D. Multiple combined policies

So far we have assumed that a given set of policies $\mathcal{P}$ is represented by $\mathcal{P}_{comb}$ consisting of a single P_{comb}. In this subsection, we consider $\mathcal{P}_{comb}$ consisting of m combined policies $P^1_{\text{comb}}, P^2_{\text{comb}}, \ldots, P^m_{\text{comb}}$.

Theorem 4. *For a given set of policies $\mathcal{P}$, if there exists an ideal representation $\mathcal{P}_{comb}$ consisting of multiple combined policies, then there exists an ideal representation of $\mathcal{P}$ consisting of a single combined policy P_{comb}.*

Proof. To prove the theorem, we construct an ideal P_{comb} from the ideal $\mathcal{P}_{comb}$ in the following way: to obtain the sequence $\mathbb{S}$ of classes in P_{comb} we concatenate all class sequences defining combined policies in $\mathcal{P}_{comb}$; then we prepend class instances in $\mathbb{S}$ by policy prefixes as described in Section III-B. Since each class in $\mathcal{C}$ appears only in a single combined policy of $\mathcal{P}_{comb}$, the constructed P_{comb} is an ideal representation. □

Multiple combined policies in $\mathcal{P}_{comb}$ can reduce two different characteristics:

- $plen(\mathcal{P}_{comb})$, the total length of all extra prefixes in $\mathcal{P}_{comb}$ (in particular, if all class instances in $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ are disjoint then extra prefixes for P^i_{comb} are unnecessary);

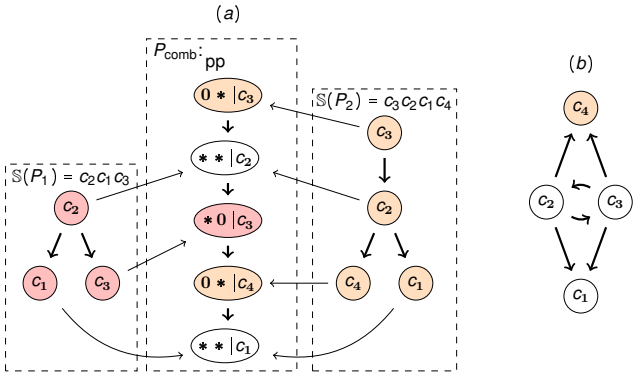


Fig. 4: (a) $\mathcal{P} = \{P_1, P_2\}$ and P_{comb} with duplicated c_3; (b) $G^{\text{jnt}}(\mathcal{P})$.

- $lsize(\mathcal{P}_{comb})$, the maximum number of filters involved in a single lookup (which can optimize lookup energy and time).

By the definition of an ideal $\mathcal{P}_{comb}$, the corresponding partition $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$ of $\mathcal{P}$ into m disjoint groups satisfies the following *sharing* condition: if P_i, P_j share a class then P_i and P_j belong to the same group. The following observation immediately follows from Theorems 3 and 4.

Observation 5. *If there exists an ideal representation of a given set of policies $\mathcal{P}$, then any partition of $\mathcal{P}$ satisfying the sharing condition defines an ideal $\mathcal{P}_{comb}$ emulating $\mathcal{P}$.*

Let $\mathbf{P}_{max}$ be a partition of $\mathcal{P}$ satisfying the sharing condition with a maximal number of groups.

Theorem 6. *If there exists an ideal representation of $\mathcal{P}$, then $\mathbf{P}_{max}$ defines an ideal $\mathcal{P}_{comb}$ with the minimum values of $plen(\mathcal{P}_{comb})$ and $lsize(\mathcal{P}_{comb})$ among all ideal representations.*

Proof. By definition of the sharing condition, $\mathbf{P}_{max}$ is unique and any partition of $\mathcal{P}$ satisfying the sharing condition can be obtained from $\mathbf{P}_{max}$ by merging the groups in $\mathbf{P}_{max}$. Since merging groups can only increase the values of $plen(\mathcal{P}_{comb})$ and $lsize(\mathcal{P}_{comb})$, the theorem immediately follows. □

IV. Non-Ideal Representations

In this section we discuss how to deal with a given set of policies whose representations cannot be ideal.

A. Conflict resolution among partial orders

We begin with an example. Fig. 4a illustrates two policies P_1 and P_2. Since $c_2 \prec_{P_1} c_3$ and $c_3 \prec_{P_2} c_2$, there is no ideal P_{comb} satisfying partial orders of classes in both P_1 and P_2. Fig. 4b shows the corresponding joint graph, and it indeed contains a cycle. To satisfy the partial orders of P_1 and P_2 at the same time, we can add an additional instance of c_3 to P_{comb} with the corresponding bits of the policy prefix. In particular, the sequence of classes in P_{comb} shown on Fig. 4a is $\mathbb{S} = c_3c_2c_3c_4c_1$; its subsequence $c_2c_3c_1$ is compatible with partial order $\prec_{P_1}$, and another subsequence $c_3c_2c_4c_1$ is compatible with partial order $\prec_{P_2}$. Now P_{comb} contains two instances of

c_3: the first is used during classification in P_2, and its policy prefix is $0*$; the second instance is used during classification in P_1, and its policy prefix is $*0$. In this case P_{comb} is non-ideal but still emulates P_1 and P_2.

In general, to deal with incompatible partial orders in policies we duplicate some instances of classes. Formally, a sequence $\mathbb{S}$ defining the order of class instances in P_{comb} is *compatible* with a policy P_i if there exists a subsequence $\mathbb{S}'$ of $\mathbb{S}$ that consists of a single instance of every class in P_i and for any two classes c_j, c_t in P_i, if $c_j \prec_{P_i} c_t$ then c_j appears before c_t in $\mathbb{S}'$. Only instances of classes from this subsequence participate in the classification by policy P_i, i.e., in the corresponding P_{comb} only for them the ith bit of the policy prefix is set to $*$, while for all other instances the ith bit of the policy prefix is set to zero. Header prefixes are exactly the same as in the case of ideal policies. The following observation immediately follows.

Observation 7. *There exists a P_{comb} emulating a given $\mathcal{P}$ if duplications of classes from $\mathcal{C}$ are allowed.*

B. Problem statement

Clearly, the number of filters in classes should be taken into account during class duplications. We denote by $\mathrm{W}^{+}(\mathbb{S})$ the total overhead in filters from duplicated class instances in the resulting sequence of classes $\mathbb{S}$, i.e., the difference between the total number of filters in all class instances from $\mathbb{S}$ and the total number of filters in original classes without duplications.

Problem 1 (Policy Sequence Packing, PSP)**.** *Given a set of policies $\mathcal{P}$, find a sequence of classes $\mathbb{S}$ compatible with all policies in $\mathcal{P}$ that minimizes $\mathrm{W}^{+}(\mathbb{S})$.*

Theorem 8. *PSP is NP-hard even for two policies, $|\mathcal{P}| = 2$.*

The proof can be found in the appendix. In the following, we denote by $|\mathbb{S}|$ a number of class instances in $\mathbb{S}$.

C. Optimal solution for PSP problem

If the partial orders of all policies in $\mathcal{P}$ are linear, the PSP problem is a weighted version of the classical Shortest Common Supersequence (SCS) problem [6]. For a set of strings, SCS finds a string with a minimal total length containing all these strings as subsequences. The algorithm in [6] finds an optimal solution for the weighted version of SCS in $O(|\mathcal{C}|^{|\mathcal{P}|})$ time, hence, this algorithm can find an optimal solution of the PSP problem but only if the number of policies is relatively small and the partial orders of all policies in $\mathcal{P}$ are linear.

D. Multiple combined policies again

Similar to ideal representations, $\mathcal{P}_{comb}$ consisting of multiple combined policies does not reduce the number of maintained filters compared to a representation consisting of a single P_{comb}. Note that the benefits from having multiple combined policies $\mathcal{P}_{comb}$ described in Section III-D remain the same for non-ideal representations. For instance, the lengths of policy prefixes can affect the memory requirements of a resulting $\mathcal{P}_{comb}$; to incorporate them into the final objective, we can minimize the total number of ternary bits in all maintained filters (including policy prefix bits). Also, representations consisting of multiple policies allow to reduce lookup complexity that can be incorporated by bounding the number of filters in each combined policy in $\mathcal{P}_{comb}$. The partition of $\mathcal{P}$ into multiple groups addresses two fundamental tradeoffs: (1) the total length of policy prefixes versus the number of filters in duplicated class instances; and (2) the maximum number of filters in each combined policy versus the number of filters in duplicated class instances.

Recall that any $\mathcal{P}_{comb}$ is obtained from a partition of $\mathcal{P}$ into multiple groups, where each group is represented in $\mathcal{P}_{comb}$ by a single combined policy. For a fixed partition of $\mathcal{P}$ into groups, the minimization of memory requirements (in bits) is obtained from the minimization of the number of entries in each combined policy since the lengths of the proposed policy prefixes depends only on the number of policies in every group. In the case of lookup complexity, the situation is similar. Hence, the construction of a combined policy for each group can be done by the proposed methods described below. We leave for the future study the development of methods partitioning $\mathcal{P}$ into multiple groups addressing both tradeoffs.

V. Approximation algorithms

In this section, we introduce several approximation algorithms for PSP and study them analytically.

A. Feedback Vertex Set as a tool

Our algorithms for PSP will use algorithms for the *Weighted Feedback Vertex Set* (WFVS) problem [7], which is NP-complete. The *feedback vertex set* is a set of vertices in a directed graph $G = (V, E)$ with weighted vertices such that removing them forms an acyclic graph, and the WFVS problem is to find a feedback vertex set of minimal total weight. For instance, the work [8] proposes an algorithm for WFVS with approximation factor $O(\log |V| \log\log |V|)$, but there are other alternatives [9]. In what follows we denote by $\alpha(G)$ the approximation factor of an algorithm for the $WFVS$ problem on a graph G. Note that WFVS is not harder than PSP (see the proof of Theorem 8). This is why the algorithms proposed below exploit WFVS as a building block.

B. Algorithm AllOrOne

By Theorem 3 the main reason for class duplications are cycles in the joint graph. The algorithm AllOrOne (AO) constructs G^{jnt} and transforms it into an acyclic graph G^{*} whose topological order produces a valid sequence of classes $\mathbb{S}$ for P_{comb}.

AO finds a feedback vertex set V^{wfvs} in G^{jnt} with minimal total weight, where vertex weight equals the number of filters in the corresponding class. By $W(V)$ we denote the total weight of vertices in V. An induced subgraph on vertices that are not in V^{wfvs} is acyclic, therefore, the corresponding classes appear only once in $\mathbb{S}$. For a class $c \in V^{\mathsf{wfvs}}$, the sequence $\mathbb{S}$ contains a separate c instance for each policy containing c.

To transform G^{jnt} into an acyclic graph G^{*}, the algorithm AO first removes all classes that are in V^{wfvs} (line 3 in

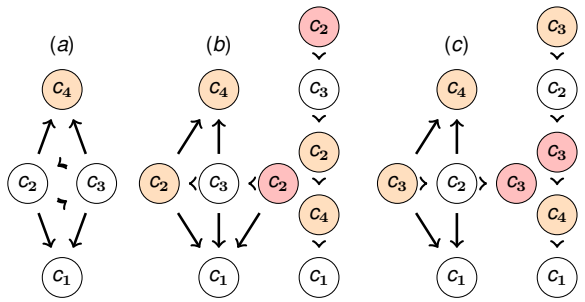


Fig. 5: (a) G^{jnt}; (b)-(c) two solutions depending on the values of $w(c_3)$ and $w(c_2)$.

Algorithm 1 AO($\mathcal{P}$)

1: construct a graph $G^{\text{jnt}}(P_1, \ldots, P_l)$;
2: $V^{\text{wfvs}} \leftarrow WFVS(G^{\text{jnt}})$, with vertex weights $w(c) = |c|$;
3: initialize G^* as a subgraph of G^{jnt} induced by $V \setminus V^{\text{wfvs}}$;
4: **for** each $c \in V^{\text{wfvs}}$ **do**
5: **for** each P_i containing c **do**
6: add to G^* an instance $\tilde{c}_i$ of c;
7: **for** each P_i **do**
8: **for** each $c \prec_{P_i} c'$ s.t. c or c' are in V^{wfvs} **do**
9: add edge $(\tilde{c}_i, \tilde{c}'_i)$ to G^*; ▷ here $\tilde{c}_i = c$ if $c \notin V^{\text{wfvs}}$
10: let $\mathbb{S}$ be a topological ordering of the vertices of G^*;
11: **return** $\mathbb{S}$.

Algorithm 1). Then for every class $c \in V^{\text{wfvs}}$ and every policy P_i containing c, a vertex $\tilde{c}_i$ is added into G^* (lines 4-6); other vertices in G^* will be connected with $\tilde{c}_i$ by edges induced by the partial order on $\prec_{P_i}$ (lines 7-9). A topological order of the vertices of G^* (line 10) forms a correct solution for the PSP problem (see Theorem 9). Since G^* can be constructed in at most $T_{G^{\text{jnt}}}(\mathcal{P})$ time, the running time of AO is $T_{\text{FVS}}(G^{\text{jnt}}) + T_{G^{\text{jnt}}}(\mathcal{P})$, where $T_{\text{FVS}}(G)$ is the running time of the algorithm for the WFVS problem.

Theorem 9. AO *correctly solves the PSP problem.*

Proof. If a graph G^* is acyclic, its topological order of vertices forms a correct $\mathbb{S}$ since all constraints introduced by partial orders of policies are represented by edges in G^*. So it is sufficient to show acyclicity of G^*. The first step of AO removes V^{wfvs} from V, making the graph G^* acyclic. Note that after adding a single vertex $\tilde{c}_i$ corresponding to the instance of c in P_i with incident edges, the graph G^* remains acyclic. This invariant holds since adding $\tilde{c}_i$ does not connect any new pair of vertices due to transitivity of $\prec_{P_i}$. Therefore, after adding $\tilde{c}_i$ a new cycle in G^* cannot appear. □

As we have already mentioned, a joint graph G^{jnt} contains edges induced by partial orders of originally given policies. To test whether G^{jnt} is acyclic, it suffices to maintain only edges for non-disjoint pairs of classes since other edges result from a transitive closure of policy partial orders and cannot introduce a cycle to G^{jnt}. On the other hand, for the correctness of AO it is necessary to consider all edges of G^{jnt}, otherwise the resulting feedback vertex set can lead to incorrect solutions.

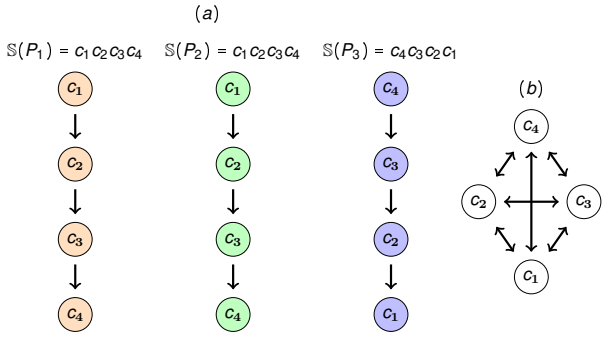


Fig. 6: PSP instance with $|\mathcal{P}| = 3$ policies showing the lower bound of AO: (a) the input $\mathcal{P} = P_1, P_2, P_3$; (b) the graph G^{jnt}.

Example 1. The following example illustrates AO running on two policies from Fig. 4. The joint graph for these policies has a cycle (see Fig. 5a); its FVS can be either $V^{\text{wfvs}} = \{c_2\}$ or $V^{\text{wfvs}} = \{c_3\}$. If $w(c_2) \leq w(c_3)$ then $V^{\text{wfvs}} = \{c_2\}$ and c_2 is duplicated (Fig. 5b shows the corresponding G^* and $\mathbb{S}$). Otherwise, AO duplicates c_3 (see Fig. 5c).

Theorem 10. AO *has an approximation factor at most* $(|\mathcal{P}| - 1) \cdot \alpha(G^{\text{jnt}})$.

Proof. Recall that in AO the found V^{wfvs} defines classes appearing more than once in the resulting $\mathbb{S}$, therefore, $\text{W}^+(\mathbb{S}) \leq (|\mathcal{P}| - 1) \cdot W(V^{\text{wfvs}})$. Note that $W(V^{\text{wfvs}}) \leq \alpha(G^{\text{jnt}})W(V^{\text{wfvs}}_{\text{OPT}})$, where $V^{\text{wfvs}}_{\text{OPT}}$ is FVS with the minimal total weight. Thus, $\text{W}^+(\mathbb{S}) \leq (|\mathcal{P}| - 1) \cdot \alpha(G^{\text{jnt}})W(V^{\text{wfvs}}_{\text{OPT}})$.

To finish the proof it suffices to show that $W(V^{\text{wfvs}}_{\text{OPT}})$ is less than $\text{W}^+(\mathbb{S}_{opt})$ for an optimal solution $\mathbb{S}_{opt}$ of the PSP problem. In any solution of the PSP problem, classes appearing more than once form a FVS in the graph G^{jnt}. The value of $W(V^{\text{wfvs}}_{\text{OPT}})$ is less than $W(V_{\mathbb{S}_{\text{OPT}}})$, where $V_{\mathbb{S}_{\text{OPT}}}$ is a FVS corresponding to $\mathbb{S}_{opt}$. Thus, $W(V^{\text{wfvs}}_{\text{OPT}}) \leq W(V_{\mathbb{S}_{\text{OPT}}}) \leq \text{W}^+(\mathbb{S}_{\text{OPT}})$. □

Theorem 11. *The approximation factor of the* AO *algorithm is at least* $|\mathcal{P}| - 1$.

Proof. The proof is by showing a hard example, where $|P| = l$ policies are constructed from n different classes; each class contains exactly one filter. The partial order of the first $l - 1$ policies is linear $c_1c_2 \ldots c_n$; the partial order of the last policy is also linear but contains the same classes in the reversed order $c_nc_{n-1} \ldots c_1$ (see Fig. 6a). In the corresponding graph G^{jnt} each pair of vertices is connected by two edges with different directions (see Fig. 6b). Any feedback vertex set of G^{jnt} consists of $(n - 1)$ vertices, therefore, the total overhead $\text{W}^+(\mathbb{S})$ incurred by AO is equal to $(n - 1)(l - 1)$. For an optimal solution $\mathbb{S}_{\text{OPT}} = c_1c_2 \ldots c_n \ldots c_2c_1$, the overhead is equal to $\text{W}^+(\mathbb{S}_{\text{OPT}}) = n - 1$. □

Note that AO either creates a separate instance of a class c in $\mathbb{S}$ for every policy or has a common instance of c in $\mathbb{S}$ for all policies limiting the optimization capabilities of the algorithm. In the proof of Theorem 11, AO finds a suboptimal $\mathbb{S}$ due to this limitation. One possible way to fix this is to apply additional optimization described in Section V-D. For the PSP instance in the proof of Theorem 11 these optimizations

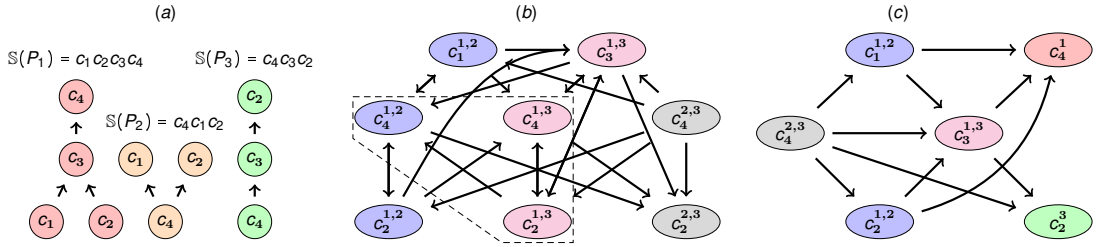


Fig. 7: Illustration of the CS algorithm: (a) the input $\mathcal{P} = \{P_1, P_2, P_3\}$; (b)$G^{\text{pair}}$; the dashed line encloses V^{wfvs}; (c) G^*.

Algorithm 2 CS($\mathcal{P}$)

1: construct the graph $G^{\text{pair}}(P_1, \ldots, P_l)$;
2: $V^{\text{wfvs}} \leftarrow WFVS(G^{\text{pair}})$, with vertex weights $w(c) = |c|$;
3: **for** every $c \in \mathcal{C}$ **do**
4: $\quad \mathbb{P}_c \leftarrow$ min. size partition of $\mathcal{P}_c$ into admissible subsets;
5: construct G^* from $\bigcup_{c \in \mathcal{C}} \mathbb{P}_c$ and $\mathcal{P}$;
6: let $\mathbb{S}$ be a topological ordering of the vertices of G^*;
7: **return** $\mathbb{S}$.

allow to produce an optimal $\mathbb{S}$, but in the general case they do not provide guarantees on $\mathrm{W}^+(\mathbb{S})$. In Section V-C we will introduce algorithms based on alternative principles that do not require unnecessary constraints.

In the following theorem we show the inapproximability of PSP by reduction from the SCS problem.

Theorem 12. *Unless* P = NP*, there is no polynomial algorithm for the PSP problem with a constant approximation factor on* $\mathrm{W}^+(\mathbb{S})$.

Proof. We reduce SCS on alphabet Σ to PSP of the same size by setting $\mathcal{C} = \Sigma$, and assigning a unit weight $w(c_i) = 1$ to every class $c_i \in \mathcal{C}$. Also, we interpret each string $s \in \Sigma^*$ as a separate policy in $\mathcal{P}$ whose partial order is linear and coincides with s.

It is known that there is no algorithm for SCS with a constant factor on the length of SCS unless P = NP [10]. The reduction described above is correct only for SCS instances where all letters in the same input string are different, which corresponds to the natural constraint for classifiers that classes are not repeated in the same input policy. However, the instance of SCS used in [10] to show inapproximability of SCS never uses a letter twice in the same input string. Thus, there is no algorithm for the PSP problem with a constant approximation ratio on the total weight $W(\mathbb{S})$ of class instances in $\mathbb{S}$ and on $\mathrm{W}^+(\mathbb{S})$ since $\mathrm{W}^+(\mathbb{S}) \le W(\mathbb{S})$ (unless P = NP). □

Existence of sublinear approximation algorithms with respect to $|\mathcal{P}|$ for the PSP problem is unclear; due to the reduction in Theorem 12, such an algorithm would solve a special case of the SCS problem with a sublinear approximation factor. To the best of our knowledge, even for this special case the existence of sublinear approximation algorithms for SCS is an open problem.

C. Algorithm CLIQUESHARE

The efficiency of an algorithm based on WFVS heavily depends on the information about $\mathbb{S}$ provided by FVS. In the AO algorithm this information is very limited: FVS only provides the set of classes appearing in $\mathbb{S}$ more than once. To overcome this limitation, we propose another algorithm CLIQUESHARE (CS): for each class c, FVS in CS provides the pairs of policies containing c that are not sharing the same instance of c in $\mathbb{S}$.

In the CS algorithm we construct another graph G^{pair} allowing to operate with a finer resolution. Denote by $\mathcal{P}_c$ the set of policies containing a class c. For each class c and each subset A of two policies in $\mathcal{P}_c$, G^{pair} contains a vertex c^A. For each P_i and any two classes $c_1 \prec_{P_i} c_2$, G^{pair} has an edge $(c_1^A, c_2^{A'})$ for all pairs of policies $A, A' \subset \mathcal{P}$ containing P_i (e.g., Fig. 7b shows a G^{pair} graph for the input $\mathcal{P}$ shown on Fig. 7a).

At the beginning, CS finds a feedback vertex set V^{wfvs} in G^{pair} with minimal total weight, where the weight of a vertex is equal to the number of filters in the corresponding class (line 2 in Algorithm 2). If c^A is in V^{wfvs} then the resulting $\mathbb{S}$ contains different instances of c for the policies $P_i, P_j \in A$. A set of policies can share the same instance of a class c if for any two policies from this set, the corresponding vertex for a class c in G^{pair} is not in V^{wfvs}, we call such sets *admissible* subsets of $\mathcal{P}_c$.

For each class c, CS computes a partition $\mathbb{P}_c$ of $\mathcal{P}_c$ into admissible subsets, minimizing the total number of sets in $\mathbb{P}_c$ (line 4 in Algorithm 2). For a class appearing only in a single policy in $\mathcal{P}$, the partition consists of a single admissible subset containing this policy. Each set in $\mathbb{P}_c$ corresponds to a separate instance of c in $\mathbb{S}$. After that CS constructs an acyclic G^*, for which a topological order of vertices forms a valid $\mathbb{S}$. For each admissible subset $B \in \mathbb{P}_c$, the graph G^* has a vertex c^B. The edges of G^* are defined similarly to G^{pair}: there is an edge $(c_1^B, c_2^{B'})$ for all $c_1, c_2 \in \mathcal{C}$, $B \in \mathbb{P}_{c_1}, B' \in \mathbb{P}_{c_2}$ such that there exists a policy $P \in B \cap B'$ for which $c_1 \prec_P c_2$.

To find a partition into admissible subsets, CS can use the algorithm that greedily constructs admissible subsets with running time $O(|\mathcal{P}_c|^2)$. Alternatively, it can use an algorithm based on dynamic programming that finds a partition with minimal number of subsets in time $O(3^{|\mathcal{P}_c|}|\mathcal{P}_c|^2)$. For both algorithms, CS has the same approximation factor but the first one has better time complexity, while the second algorithm finds an optimal partition into admissible subsets.

Example 2. The following example illustrates CS running on three policies (see Fig. 7a). The weights of all classes are the same. At first CS constructs G^{pair} (see Fig. 7b), which

has $\binom{3}{2} = 3$ vertices for c_2 and c_4 and one vertex for c_1 and c_3. G^{pair} has many cycles; one of its feedback vertex sets with minimal total weight is $V^{\text{wfvs}} = \{c_2^{1,3}, c_4^{1,2}, c_4^{1,3}\}$. The partitions of $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_3}$ consist of a single set since the vertices for c_1 and c_3 in G^{pair} do not appear in V^{wfvs}. For c_2 and c_4 optimal partitions into admissible subsets can be $\mathbb{P}_{c_2} = \{\{P_1, P_2\}, \{P_3\}\}$ and $\mathbb{P}_{c_4} = \{\{P_1\}, \{P_2, P_3\}\}$. The resulting G^* is acyclic (see Fig. 7c). Every topological order on G^* yields a valid $\mathbb{S}$, e.g., $c_4^{2,3} c_1^{1,2} c_2^{1,2} c_3^{1,3} c_4^{1} c_2^{3}$.

Note that CS and AO coincide in the case of two policies. Observe that CS finds an optimal $\mathbb{S}$ for the example in the proof of Theorem 11. In the following we prove that CS works correctly and estimate its approximation factor.

Theorem 13. CS *correctly solves the PSP problem.*

Proof. Similar to Theorem 9, we only need to show that G^* is acyclic. The construction of G^* from G^{pair} is equivalent to the following three-step procedure: (1) initialize G^* as a subgraph of G^* induced by all vertices c^A such that the policies $P_i, P_j \in A$ belong to the same admissible subset of $\mathbb{P}_c$; (2) for each $c \in \mathcal{C}$ add vertices into G^* for all admissible subsets of $\mathbb{P}_c$ consisting of a single policy; (3) for each $c \in \mathcal{C}$, "shrink" vertices corresponding to policies belonging to the same admissible subset.

A graph G^* is acyclic after the first step since at least all vertices in found FVS of G^{pair} are not included in G^*. After the second step G^* remains acyclic due to transitivity of partial orders, which is similar to the proof of Theorem 9. To prove that G^* will remain acyclic after the third step, it is sufficient to show that G^* remains acyclic after every shrink. A shrink produces a cycle in G^* if and only if before this shrink G^* had a path between two vertices corresponding to class with policies in the same admissible subset. Assume that there is such path w for a class c. W.l.o.g. let P_1 be a policy whose partial order defines the first edge of w, and P_2 be a policy whose partial order defines the last edge of w. The vertex c^A, where $P_1, P_2 \in A$ has an outgoing edge to the second vertex of w and has an incoming edge from penultimate vertex of w. Hence, there is a cycle in G^* containing a vertex c^A which is a contradiction to the assumption that G^* has no cycles before the current shrink. □

Theorem 14. CS *has an approximation factor of at most* $\alpha(G^{pair})\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

Proof. First, we are to show that for a produced sequence $\mathbb{S}$ by CS, $W^+(\mathbb{S})$ does not exceed the weight of the corresponding V^{wfvs}. Each class c appearing t times in $\mathbb{S}$ increases the value of $\mathrm{W}^+(\mathbb{S})$ by $(t-1)w(c)$. On the other hand, the found FVS in G^{pair} should contain at least $t-1$ vertices for c. Otherwise, at least two admissible subsets corresponding to the instances of c in $\mathbb{S}$ can be merged into a bigger admissible subset, which is not possible for partitions constructed by CS. Therefore, $\mathrm{W}^+(\mathbb{S}) \le W(V^{\text{wfvs}})$.

By any solution $\mathbb{S}'$ of the PSP problem we can construct an FVS in G^{pair} in the following way: if P_i and P_j do not share an instance of c in $\mathbb{S}'$, $c \in P_i, P_j$, then $c^{\{P_i, P_j\}}$ is in FVS.

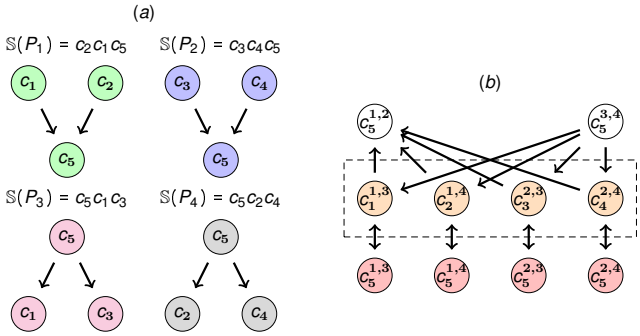


Fig. 8: The PSP instance with $|\mathcal{P}| = 4$ policies showing the lower bound of CS : (a) the input $\mathcal{P} = P_1, P_2, P_3, P_4$; (b) the graph G^{pair}; the dashed line encloses V^{wfvs}.

Hence, as in Theorem 10, $W(V^{\text{wfvs}}) \le \alpha(G^{pair})W(V^{\text{wfvs}}_{\text{OPT}}) \le \alpha(G^{pair})W(V_{\mathbb{S}_{\text{OPT}}})$, where $V^{\text{wfvs}}_{\text{OPT}}$ is FVS in G^{pair} with the minimal total weight and $V_{\mathbb{S}_{\text{OPT}}}$ is FVS in G^{pair} by which CS produces an optimal sequence $\mathbb{S}_{opt}$.

To finish the proof, we need to show that $W(V_{\mathbb{S}_{\text{OPT}}}) \le \lfloor \frac{l^2}{4} \rfloor W^+(\mathbb{S})$. Consider a class c belonging to l_c policies in $\mathcal{P}$ and appearing in $\mathbb{S}_{\text{OPT}}$ t times. Denote by $s_{c,i}$ (where $1 \le i \le t$), a number of policies in an i-th admissible subset. A set $V_{\mathbb{S}_{\text{OPT}}}$ can contain only c^A vertices such that $P_i, P_j \in A$ belong to different admissible subsets; the number of such vertices is at most $N_c = \frac{1}{2}\left(l_c^2 - \sum_{i=1}^{t} s_{c,i}^2\right)$. It can be shown that for any partition on admissible subsets, $N_c \le \left\lfloor \frac{|\mathcal{P}|^2}{4} \right\rfloor \cdot (t-1)$ [1]. Hence, a class c increasing $W^+(\mathbb{S}_{\text{OPT}})$ by $(t-1) \cdot w(c)$ can also increase $W(V_{\mathbb{S}_{\text{OPT}}})$ by the value not exceeding $\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor \cdot (t-1) \cdot w(c)$; summing this over all classes $c \in \mathcal{C}$, we find that $W(V_{\mathbb{S}_{\text{OPT}}}) \le \lfloor \frac{|\mathcal{P}|^2}{4} \rfloor W^+(\mathbb{S})$. □

Theorem 15. *The approximation factor of* CS *is at least* $\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

Proof. We provide an example with $|P| = l$ policies and $n = \lfloor \frac{l^2}{4} \rfloor + 1$ different classes. We add a class c_n to all policies. Also we enumerate all pairs of policies (P_i, P_j) such that $i \le \lfloor \frac{l}{2} \rfloor$ and $j > \lfloor \frac{l}{2} \rfloor$, there are $n-1$ such pairs. For each enumerated pair (P_i, P_j) we add a class c_k to policies P_i and P_j, where k is a number of this pair. The class $c_k \prec_{P_i} c_n$ and $c_k \succ_{P_j} c_n$. For instance, on Fig. 8a the class c_3 corresponding to the third pair of policies (P_2, P_3) precedes c_5 in P_2 and succeeds c_5 in P_3. The number of filters in classes is as follows: $|c_i| = x, i = 1 \ldots n-1, |c_n| = x+1$.

A graph G^{pair} consists of three categories of vertices (see Fig. 8b): (1) the $n-1$ vertices corresponding to a class c_n in the enumerated pairs of policies (in Fig. 8b, these vertices are shown in purple); (2) the $n-1$ vertices for all other classes c_i, where $i < n$; (in Fig. 8b, these vertices are orange); (3) the vertices corresponding to c_n in non-enumerated pairs of policies which do not affect acyclicity of G^{pair}. (in Fig. 8b, these vertices are white). The G^{pair} graph contains $n-1$ bidirectional edges

[1] The proof of this inequality consists of only cumbersome calculations so we omit it.

between vertices of the first two types, which form a maximal matching.

An FVS of G^{pair} with the minimal total weight V^{wfvs} consists of all vertices of the second category. Therefore, CS produces $\mathbb{S}$ containing one copy of c_n and two copies for each other class; the total overhead is equal $\mathrm{W}^+(\mathbb{S}) = (n-1)\cdot x$. The optimal solution for this example is $\mathbb{S}_{\mathsf{OPT}} = c_n c_1 c_2 \dots c_n$ with $\mathrm{W}^+(\mathbb{S}) = x+1$; taking x arbitrarily big, we show the stated lower bound.

To obtain an example with an arbitrarily large number of classes, we take multiple instances of the proposed example and combine them into one joint input: we merge policies P_i with the same index i, and classes from different instances of the example are different. □

The graph G^{pair} can be constructed in time $T_{G^{\mathsf{jnt}}}(\mathcal{P}) + O(|\mathcal{P}|^2 \cdot \sum_{P\in\mathcal{P}} |P|^2)$. Hence, the running time of CS is $T_{G^{\mathsf{jnt}}}(\mathcal{P}) + O(|\mathcal{P}|^2 \cdot \sum_{P\in\mathcal{P}} |P|^2) + T_{FVS}(G^{\mathsf{pair}}) + O(|\mathcal{C}| \cdot T_{\mathrm{part}}(|\mathcal{P}|))$, where T_{part} is the time complexity of the algorithm finding partitions into admissible subsets. The approximation factor of CS is quadratic on $|\mathcal{P}|$ and worse than for AO for all $|\mathcal{P}| > 3$. Nevertheless, we will see in Section VII that CS performs better on average since it operates with a better resolution.

Note that AO and CS find optimal solutions in 'simplest' PSP instances. For example, AO and CS always find optimal solutions if there exists an ideal representation or if an optimal solution for two policies contains at most one duplicated class instance. Various heuristics that are not based on FVS cannot guarantee solution optimality even in these simplest PSP instances. Unfortunately, AO has its own constraints limiting optimization capabilities. The proposed CS overcomes these limitations preserving AO advantages.

D. Additional optimizations

Both AO and CS algorithms can be further improved by additional optimizations. First, we define GREEDYGLUING (GG) optimization that greedily shrink pairs of vertices of an acyclic graph G with the maximal possible sum of weights while G remains acyclic. GG can be added to both AO and CS as the penultimate step, to simplify G^* before taking its topological order. The time complexity of GG is $O(n^3)$, where n is the number of vertices in the corresponding graph.

Another optimization procedure comes from the fact that proposed algorithms do not usually guarantee that $\mathbb{S}$ will be a *local minimum* solution, i.e., it might happen that one can remove some class instances from the resulting $\mathbb{S}$ and still get a valid sequence for P_{comb}. The LOCALDESCENT (LD) procedure is defined in the following way: given $\mathbb{S}$, try to remove classes from $\mathbb{S}$ one by one, while $\mathbb{S}$ remains compatible with all policies from $\mathcal{P}$. LD can be implemented in time $O(|\mathbb{S}| + \sum_{P\in\mathcal{P}}(|P| + D(P)))$. We will see in Section VII that LD does bring improvements in practice, although it has no effect on the worst case bounds.

VI. DYNAMIC UPDATES

Although economic models rarely change, support of dynamic updates in represented policies can become important in some deployment scenarios. We support two basic operations on policies in $\mathcal{P}$: (1) $\mathrm{delete}(P, c)$, remove a class c from a policy P; (2) $\mathrm{insert}(P, c, c_{succ})$, add a class c to a policy P such that c appears in $\mathbb{S}(P)$ just before c_{succ}. Each insert/delete operation modifies a sequence of classes $\mathbb{S}$ that represents the corresponding P_{comb}. Note that after each operation policy prefixes should be updated assuring that P_{comb} emulates $\mathcal{P}$.

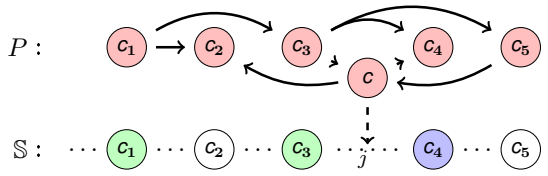


Fig. 9: Insert of an instance for a new class $c \in P$ into the j-th position of $\mathbb{S}$; the white instances are not in $\mathbb{L}_j$ or $\mathbb{R}_j$

Hypothetically, we can generalize the proposed algorithms in Section V to support dynamic operations by maintaining dynamically graphs $G^{\mathsf{jnt}}, G^{\mathsf{pair}}, G^*$ and the sequence $\mathbb{S}$. But running dynamic versions of AO and CS may be very time-consuming. They are better suited for environments where updates happen in batches. In this section, we propose another algorithm implementing the right balance between time complexity and optimization efficiency in a dynamic environment.

When we delete a class c from a policy P, we remove an instance of c in $\mathbb{S}$ if this instance corresponds only to P. After a delete operation $\mathbb{S}$ remains correct: if necessary, we can further optimize $\mathbb{S}$ by LD optimization to remove redundant class instances (DELETE() in Algorithm 3).

The case of an insert operation is more complicated. Let c be the class that is inserted into policy P and let $\mathcal{C}_{\mathrm{prec}}$ be the set of classes preceding c in $\prec_P$, and $\mathcal{C}_{\mathrm{succ}}$ be the set of classes succeeding c in $\prec_P$. If $\mathbb{S}$ is already compatible with the new P the insertion is done. Otherwise, to make $\mathbb{S}$ compatible with the new P, we insert to the jth position inside $\mathbb{S}$ an instance of c and instances of some classes in $\mathcal{C}_{\mathrm{prec}}$ and $\mathcal{C}_{\mathrm{succ}}$ (INSERT_AT() in Algorithm 3), where the jth position is selected to minimize the total number of filters in inserted instances.

Let $\mathbb{L}_j$ be the longest subsequence of $\mathbb{S}[0 \dots j-1]$ satisfying the following conditions:

(1) $\mathbb{L}_j$ contains at most one instance of every class from P;
(2) an instance of c_1 is in $\mathbb{L}_j$ only if for each $c_2 \prec_P c_1$ the instance of c_2 appears in $\mathbb{L}_j$ before c_1.

Similarly, we define a subsequence $\mathbb{R}_j$ in the suffix of $\mathbb{S}$ starting from the j-th position. But in this case condition (2) is reversed: an instance of c_1 is in $\mathbb{R}_j$ only if for each c_2, $c_1 \prec_P c_2$, the instance of c_2 appears in $\mathbb{R}_j$ after c_1. We insert into the jth position an c instance, class instances in $\mathcal{C}_{\mathrm{prec}}$ that are not in $\mathbb{L}_j$ and class instances in $\mathcal{C}_{\mathrm{succ}}$ that are not in $\mathbb{R}_j$ in order satisfying $\prec_P$. Figure 9 illustrates this insertion procedure for a policy P: $\mathbb{L}_j = c_1 c_3$, $\mathbb{R}_j = c_4$, and class instances c_2, c_5 are inserted together with the instance of c.

Theorem 16. *All class instances inserted to the jth position of $\mathbb{S}$ make $\mathbb{S}$ compatible with a new P.*

Proof. Let $\mathbb{I}_j$ be an inserted sequence of class instances to

Algorithm 3 Dynamic Updates

1: **procedure** DELETE($\mathbb{S}, P, c$)
2: remove from $\mathbb{S}$ an instance of c corresponding only to P
3: remove c from P
4: $\mathbb{S} \leftarrow \text{LD}(\mathbb{S})$
5: **end procedure**

6: **procedure** INSERT_AT($\mathbb{S}, P, c, j$)
7: $insert_set \leftarrow (\mathcal{C}_{\text{prec}} \setminus \mathbb{L}_j) \cup \{c\} \cup (\mathcal{C}_{\text{succ}} \setminus \mathbb{R}_j)$
8: $\mathbb{I} \leftarrow$ sequence of classes from $insert_set$ ordered by $\prec_P$
9: insert $\mathbb{I}$ into $\mathbb{S}$ at jth position
10: **if** $j > 0$ **and** $\mathbb{S}[j-1] = c$ **then**
11: remove $j-1$th element of $\mathbb{S}$
12: **end procedure**

13: **procedure** CALCULATE_$\mathcal{L}(P)$
14: $\mathcal{L}[0] = tempL = 0$
15: $\mathbb{L} = \{\}$
16: **for** each $c \in P$ **do**
17: $d(c) \leftarrow$ number of c_1 such that $c_1 \prec_P c$
18: **for** $j \in [0, |\mathbb{S}| - 1]$ **do**
19: **if** $d(\mathbb{S}[j]) = 0$ **and** $\mathbb{S}[j] \notin P$ **then**
20: $\mathbb{L} = \mathbb{L} \cup \{\mathbb{S}[j]\}$
21: **if** $c \in \mathcal{C}_{\text{prec}}$ **then** $tempL = tempL + |\mathbb{S}[j]|$
22: **for** all c_1 such that $\mathbb{S}[j] \prec_P c_1$ **do**
23: $d(c_1) = d(c_1) - 1$
24: $\mathcal{L}[j+1] = tempL$
▷ After jth iteration $\mathbb{L}$ denotes $\mathbb{L}_j$
25: **return** $\mathcal{L}$
26: **end procedure**

27: **procedure** INSERT(P, c, c_{succ})
28: update $\mathbb{S}(P)$ by adding c immediately before c_{succ}
29: **if** $\mathbb{S}$ is compatible to P **then exit**
30: $\mathcal{L} \leftarrow$ CALCULATE_$\mathcal{L}(P)$
31: $\mathcal{R} \leftarrow$ CALCULATE_$\mathcal{R}(P)$
▷ CALCULATE_$\mathcal{R}$ is symmetric to CALCULATE_$\mathcal{L}$
32: $j \leftarrow \arg\max_{0 \le j \le n}(\mathcal{L}[j] + \mathcal{R}[j] + \delta(\mathbb{S}, j, c) \cdot |c|)$
33: INSERT_AT($\mathbb{S}, P, c, j$)
34: $\mathbb{S} \leftarrow \text{LD}(\mathbb{S})$
35: **end procedure**

the j-th position in $\mathbb{S}$ (line 8 in Algorithm 3). Let $\mathbb{R}'_j$ be a subsequence of $\mathbb{R}_j$ that does not contain class instances from $\mathbb{L}_j$. Note that condition (2) from the definition of $\mathbb{R}_j$ is also satisfied for $\mathbb{R}'_j$. Consider a subsequence of $\mathbb{S}$ constructed by the concatenation of $\mathbb{L}_j, \mathbb{I}_j, \mathbb{R}'_j$. This subsequence contains one instance of each class from the new P and satisfies $\prec_P$. □

Corollary 1. *An insert operation is correct.*

When $j > 0$ and $\mathbb{S}[j-1]$ is an instance of an inserted class c, we can remove the $(j-1)$th element of $\mathbb{S}$ since a new copy of c is inserted (line 11 in Algorithm 3).

Denote by $\mathcal{I}$ an array of $|\mathbb{S}| + 1$ integeres where $\mathcal{I}[j]$ is a total number of filters in all inserted class instances in case when c is inserted into the jth position of $\mathbb{S}$. Let $\delta(\mathbb{S}, j, c)$ be a function such that $\delta(\mathbb{S}, j, c) = 1$ if $j > 0$ and $\mathbb{S}[j-1] = c$, otherwise, $\delta(\mathbb{S}, j, c) = 0$. Then:

$$\mathcal{I}_j = (1 - \delta(\mathbb{S}, j, c)) \cdot |c| + \sum_{c' \in \mathcal{C}_{\text{prec}} \setminus \mathbb{L}_j} |c'| + \sum_{c' \in \mathcal{C}_{\text{succ}} \setminus \mathbb{R}_j} |c'|$$

Among all potential positions to insert a class c, we choose one minimizing $\mathcal{I}[j]$. Let $\mathcal{L}$ and $\mathcal{R}$ be arrays such that $\mathcal{L}[j] = \sum_{c' \in \mathcal{C}_{\text{prec}} \cap \mathbb{L}_j} |c'|$ and $\mathcal{R}[j] = \sum_{c' \in \mathcal{C}_{\text{succ}} \cap \mathbb{R}_j} |c'|$; then the expression for $\mathcal{I}[j]$ can be rewritten as

$$\mathcal{I}_j = |c| + \sum_{c' \in \mathcal{C}_{\text{prec}} \cup \mathcal{C}_{\text{succ}}} |c'| - (\mathcal{L}_j + \mathcal{R}_j + \delta(\mathbb{S}, j, c) \cdot |c|)$$

We compute elements of $\mathcal{L}, \mathcal{R}$ and then select insertion position j maximizing value of $\mathcal{L}[j] + \mathcal{R}[j] + \delta(\mathbb{S}, j, c) \cdot |c|$ (INSERT() in Algorithm 3).

We calculate elements of $\mathcal{L}$ in the order of increasing j (CALCULATE_$\mathcal{L}$() in Algorithm 3). During this computation, we maintain the sets $\mathbb{L}_j$ and $\mathbb{L}_j \cap \mathcal{C}_{\text{prec}}$. If $\mathbb{S}[j] \notin \mathbb{L}_j$ and all classes preceding $\mathbb{S}[j]$ in P belong to $\mathbb{L}_j$ then $\mathbb{L}_{j+1} = \mathbb{L}_j \cup \{\mathbb{S}[j]\}$, otherwise $\mathbb{L}_{j+1} = \mathbb{L}_j$. Therefore, we can compute $\mathbb{L}_{j+1}$ (and $\mathbb{L}_{j+1} \cap \mathcal{C}_{\text{prec}}$ with $\mathcal{L}[j+1]$ respectively) from $\mathbb{L}_j$ in time proportional to the number of classes preceding $\mathbb{S}[j]$ in P. To speed up this process, when adding $\mathbb{S}[j]$ to $\mathbb{L}_{j+1}$ we look at all classes that succeed $\mathbb{S}[j]$ and mark those for which all classes preceding them in P are in $\mathbb{L}_{j+1}$ (lines 22-23 in Algorithm 3, class c_1 becomes marked if $d(c_1) = 0$). For a subsequent position j', we add $\mathbb{S}[j']$ into $\mathbb{L}_{j'+1}$ if and only if $\mathbb{S}[j']$ is not in $\mathbb{L}_{j'}$ and $\mathbb{S}[j']$ corresponds to a marked class (line 19 in Algorithm 3). This implementation allows to compute all elements of $\mathcal{L}$ in time $O(|\mathbb{S}| + |P| + D(P))$.

The values of $\mathcal{R}$ elements can be computed in the reverse order of j in a similar way. Therefore, the total time complexity of the insert operation equals $O(|\mathbb{S}| + |P| + D(P))$.

We can achieve additional memory savings by running LD on the resulting $\mathbb{S}$. Since the time complexity of LD is $O(|\mathbb{S}| + \sum_{P \in \mathcal{P}}(|P| + D(P)))$, we can run it after each insert or delete operation.

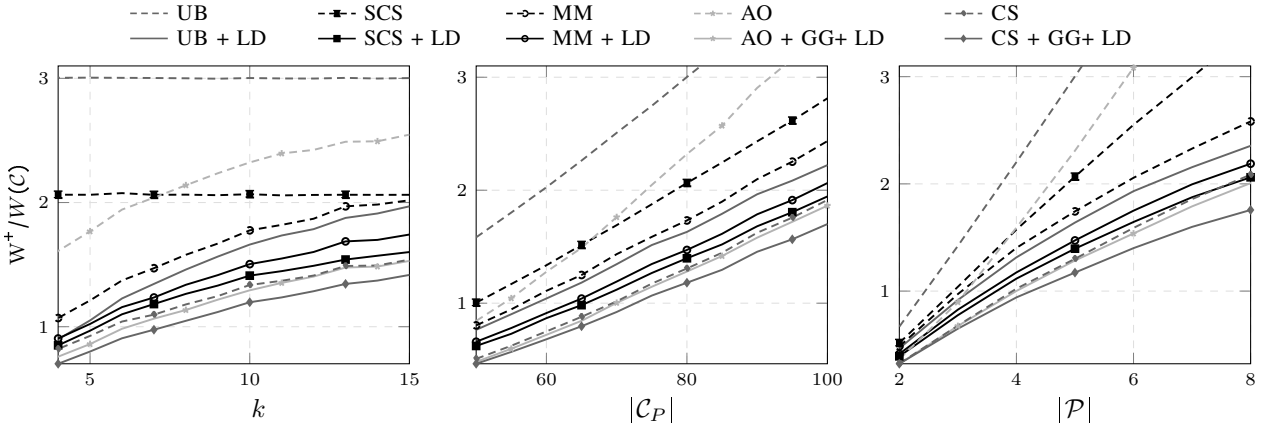
VII. Experimental evaluation

A. Combined representations

Algorithms. We compare the algorithms AO, CS, weighted SCS, UpperBound (UB) that simply concatenate all $\mathbb{S}(P_i)$, $P_i \in \mathcal{P}$, into a single $\mathbb{S}$, and MajorityMerge (MM) proposed in [6] with $(3,1)$-look-ahead extensions [11], [12] (see Algorithm 4). We have also extended the algorithms with the GG heuristic and LD. For each considered algorithm we also evaluate its version extended by additional optimizations. In particular, we extend all algorithms by LD, and additionally extend graph-based algorithms AO and CS by GG.

Methodology. Unfortunately, de-facto standard frameworks to generate filter-based classifiers such as ClassBench [13] do not have class-based level of abstraction and, hence, cannot be used for the declaration of multiple policies based on the same set of classes $\mathcal{C}$. Hence, we experimented on synthetic data produced in a way similar to intended usage:

(1) generate sizes of classes from $\mathcal{C}$;
(2) pick which classes are non-disjoint;
(3) generate a set of policies $\mathcal{P}$ on classes from $\mathcal{C}$, with each policy consisting of the same number of classes



(a) $|\mathcal{P}| = 5$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$ (b) $|\mathcal{P}| = 5$, $k = 10$, $|\mathcal{C}| = 100$ (c) $k = 10$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$

Fig. 10: Relative overhead $\mathrm{W}^+/W(\mathcal{C})$ as a function of: (a) average degree of intersection k; (b) number of classes in each policy $\mathcal{P}$; (c) number of policies $|\mathcal{P}|$

Algorithm 4 MM($\mathcal{P}$)

- $\mathcal{F}(c, \mathcal{P})$ – set of policies containing every policy P s.t. $c \in P$ and there is no class c' preceeding c in $\prec_P$
- $\mathcal{P}/c$ – instance of the PSP problem with c removed from policies in $\mathcal{F}(c, \mathcal{P})$

find (c_1, c_2, c_3) maximizing
$$(|\mathcal{F}(c_1, \mathcal{P})| - 1) \cdot w(\mathcal{P}, c_1) + (|\mathcal{F}(c_2, \mathcal{P} \setminus c_1)| - 1) \cdot w(c_2) + (|\mathcal{F}(c_3, (\mathcal{P} \setminus c_1) \setminus c_2| - 1) \cdot w(c_3),$$
breaking ties by $(|\mathcal{F}(c_1, \mathcal{P})| - 1) \cdot w(c_1)$;
if $\mathcal{P} \setminus c_1 = \varnothing$ **then return** c_1 ▷ return c_1 if $\mathcal{P}/c_1$ empty
$\mathbb{S} = c_1 \mathrm{MM}(\mathcal{P} \setminus c_1)$ ▷ concat c_1 with the recursive result
return $\mathbb{S}$

For simplicity, we assume that each class in every policy is associated with a different action.

For every setting, we performed 100 experiments with random instances and different random seeds (virtually all algorithms are randomized because the topological order on G^* is not unique in most cases); we show averaged results. Implementation of our experiments is available at [14], and the results are summarized on Fig. 10. The Y-axis in all plots shows the relative overhead $\mathrm{W}^+(\mathbb{S})/W(\mathcal{C})$, where $W(\mathcal{C})$ is the total size of all classes from $\mathcal{C}$; we show relative values of the overhead because absolute values change a lot from instance to instance.

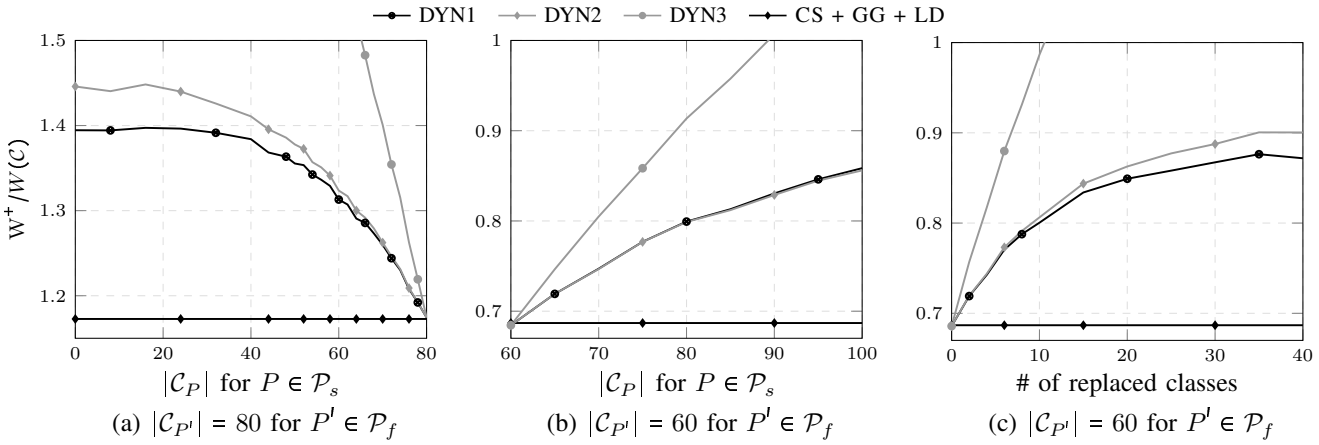
The number of filters in a combined representation significantly depends on the structure of given policies. There are three main characteristics of the input structure: (1) number of intersecting classes, (2) average number of policies that contain a class, (3) total number of policies. We generate inputs with different values of these characteristics. Fig 10a shows how the relative overhead grows as the average number of classes k intersecting with each $c \in \mathcal{C}$ increases. In Fig. 10b we vary the number of classes in each policy $|\mathcal{C}_P|$. In these experiments each class belongs to $\frac{|\mathcal{C}_P|}{|\mathcal{C}|} \cdot |\mathcal{P}|$ policies on average. The inputs in Fig. 10c consist of various numbers of policies.

The algorithm CS with GG and LD postprocessing (the strongest combination in our experiments) outperforms other algorithms regardless of input characteristics; this confirms our hypothesis that this algorithm is the best choice for a vast majority of inputs with different policy structures. Evaluations also show that CS with GG and LD constructs a representation with only 20-50% of the filters in duplicated instances of classes compared to representations where policies are stored separately. In what follows we describe our evaluation results for all algorithms in detail.

Additional optimizations: The evaluations show that class sharing introduces substantial savings, and changes the linear behavior to nearly logarithmic in Fig. 10.c; even UB with LD reduces the overhead for additional instances of classes (e.g., by 23-63% in Fig. 10a); still it is worse than the other considered algorithms. Additional optimizations are especially effective for UB, SCS, and AO algorithms saving up to 52%, 38%, and 47% respectively in the second set of experiments (see Fig. 10b); CS can be also improved by additional optimizations, but in this case its effect is not substantial (at most 17% in all experiments) since produced results are close to local minimum. Comparing SCS with and without LD in Fig. 10a where the former remains constant, one can see how exploiting partial orders can significantly affect optimization results, and how the effect diminishes as classes start to intersect more (k increases in Fig. 10a); this is due to optimality of SCS in the case of linear orders.

AO: In the third set of experiments (see Fig. 10.c), AO without optimizations outperforms UB without optimizations by up to 45% (see Fig. 10.b) but performs up to 1.7 times worse than CS without optimizations. In the first and second set of experiments, AO without optimization always performs worse than MM without optimizations (see Fig. 10.a-b). The main reason for this is a low resolution of AO adding for every class c either a single instance of c or a separate instance for every policy containing c. CS is proposed specifically to overcome this limitation.

CS: This algorithm significantly outperforms other evaluated algorithms. Moreover, CS without optimizations constructs



(a) $|\mathcal{C}_{P'}| = 80$ for $P' \in \mathcal{P}_f$ (b) $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$ (c) $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$

Fig. 11: Experiments with dynamic updates: (a) insertion scenario; (b) removal scenario; (c) replacement scenario.

class sequences with almost the same total number of filters in duplicated class instances as AO with GG and LD. For inputs generated with parameters ($k = 10$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$, $|\mathcal{P}| = 5$), CS with GG and LD requires only 40% of the filters in the duplicated instances of classes versus UB on average (see Fig. 10c).

B. Dynamic updates

Algorithms. In this part we evaluate insert/remove operations introduced in Section VI and compare the following three variations:

(1) DYN1 applies LD after each insert/remove operation;
(2) DYN2 applies LD only after all operations;
(3) DYN3 never apply LD.

Denote by $\mathcal{P}_f$ a set of policies after all applied dynamic operations. We compare $\mathbb{S}$ obtained after all operations with offline results of CS with GG and LD calculated on $\mathcal{P}_f$.

Methodology. We denote by $\mathcal{P}_s$ a set of policies just before dynamic operations. In all our experiments we first choose a final set of policies $\mathcal{P}_f$ in the same way as for the algorithms in the offline case. Then we generate an initial set of policies $\mathcal{P}_s$ from $\mathcal{P}_f$, and for $\mathcal{P}_s$ we construct $\mathbb{S}$ by CS with GG and LD, then we apply insert/remove operations one by one in a random order transforming $\mathcal{P}_s$ into $\mathcal{P}_f$.

We evaluate three different scenarios, each scenario defined by the method constructing $\mathcal{P}_s$ from $\mathcal{P}_f$:

(1) insertion scenario (Fig. 11a): generate $\mathcal{P}_s$ by removing classes from $\mathcal{P}_f$, and then run dynamic updates to insert back the removed classes until we get $\mathcal{P}_f$.
(2) removal scenario (Fig. 11b): generate $\mathcal{P}_s$ by inserting new classes to $\mathcal{P}_f$, and then run dynamic updates to remove classes until we get $\mathcal{P}_f$;
(3) replacement scenario (Fig. 11c): obtain $\mathcal{P}_s$ by replacing a certain number of classes in each policy of $\mathcal{P}_f$, and run dynamic updates to remove classes that are in $\mathcal{P}_s$ and not in $\mathcal{P}_f$ and insert classes that are in $\mathcal{P}_f$ and not in $\mathcal{P}_s$.

All policies in $\mathcal{P}_f$ (or in $\mathcal{P}_s$) have the same number of classes. In all experiments $|\mathcal{C}| = 100, |\mathcal{P}| = 5, |k| = 10$.

As we can see in Fig. 11, the usage of LD is extremely important after modification operations. In the case of class removals (Fig. 11b), one can apply LD after all operations, and the result will stay the same. In the insertion scenario (Fig. 11a), if no more than 25% of new classes are added (i.e. the $|\mathcal{C}_P|$ is at least 60), running LD after every operation (DYN1) do not introduce additional gains versus running LD after the whole batch (DYN2). Otherwise, the results become worse if we apply LD after all inserts, e.g., the difference goes up to 4% as we decrease $|\mathcal{C}_P|$ (see, again, Fig. 11a); the algorithm that never uses LD (DYN3) looses in all scenarios, e.g., by 17% in the removal scenario in Fig. 11b, $|\mathcal{C}_P| = 85$, and much more as we increase $|\mathcal{C}_P|$.

In the scenario with 8 replaced classes in each policy (see Fig. 11c) the number of filters in duplicated class instances in the output is at most 14% larger than in the case of CS with GG and LD. Finally, the algorithm DS constructing P_f with dynamic inserts from scratch ($|\mathcal{C}_P| = 0$ in Fig. 11a) builds a combined representation that has 19% more filters in duplicated class instances then CS with GG and LD. On the other hand, the time complexity of DS is significantly smaller than the time complexity of CS with GG and LD; hence, DS can be used as the offline algorithm constructing $\mathbb{S}$ in cases when time complexity of constructing the representation is a bottleneck. Note that DS does not provide any guarantees on the resulting $\mathbb{S}$ even for $\mathcal{P}$ that does have an ideal representation.

Our evaluation study confirms the usefulness of class-based abstractions in optimization of policy classifiers both based on sequences and partial orders of classes, but the former perform much better. In the offline case, CS with GG and LD significantly outperform all other evaluated algorithms. The proposed algorithm for dynamic updates implements the fundamental trade-off between the number of filters in duplicated class instances and the time spent on the construction of resulting representations.

VIII. Previous Work

Finding efficient representations of a single instance of a packet classifier is a well-known problem. Approaches to this problem fall into two major categories: software-based and hardware-based solutions. These solutions mainly span three techniques: decision trees, hashing, and coding-based compression [15]–[20]. In decision trees, finding a matching filter is based on tracing a path in a decision tree (e.g., [15]); however, there is an inherent tradeoff between space and time

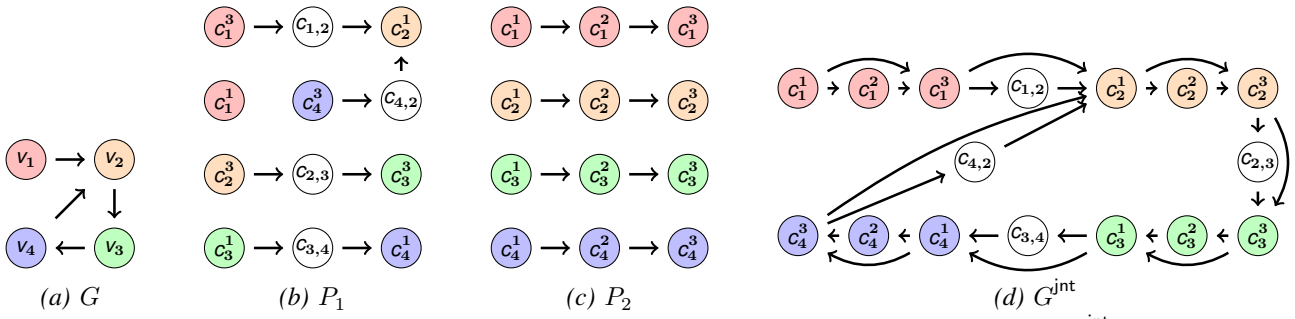


Fig. 12: Illustration for the proof of Theorem 8: (a) a graph G; (b) - (c) P_1, P_2 constructed from G; (d) G^{jnt} for $\mathcal{P} = \{P_1, P_2\}$.

complexity in these approaches. Hash-based solutions that match a packet to its possible matching filters have also been considered [17]. Other works discuss efficient hardware implementations that have no native representation. E.g., TCAMs have no native support for ranges, so one has to translate ranges into TCAM-friendly prefix matching representations [18]–[22]. Unfortunately, in most cases these methods apply only to filters with a limited number of fields or perform poorly as it increases. The works [23]–[28] exploits structural properties of classifiers, in particular order-independence, to create equivalent classifiers with a fewer number of fields. Representations of order-independent classifiers as Boolean expressions and the relation to the MinDNF problemis studied in [29]. In [30], per-flow per-policy class state is implemented when the same policy can be attached to multiple flows. In general, the previous works mostly concentrate on optimizing a single instance of policy classifier, whereas we concentrate on combined optimizations of multiple different policies. The problem of splitting a policy into several lookup tables while minimizing the maximal local table size has been broadly studied in [31], [32] and found to be an intractable optimization problem. The main contribution of [33] is an optimal algorithm with linear time complexity that can handle dynamic fields at the price of a single bit of metadata prepended to every packet. The work [34] introduces the notion of classification with a controlled error allowing to trade the classification accuracy for the additional efficiency of classifier representations. Note that all these proposals are orthogonal to our combined policy representations and can be used alongside with it.

The FVS and especially the SCS problems are among classical NP-complete problems, with plenty of research devoted to them. In particular, the work [35] presents general ideas of "splitting" vertices that we extend here, [36] proves hardness results for SCS and similar problems, though [37] studies parameterized complexity and shows that SCS problem is already $W[1]$-hard. The work [38] presents approximation results for $\mathrm{SCS}(2,k)$ and $\mathrm{SCS}(2,3)$ that also employ a reduction to the FVS problem. The work [39] presents `Reduce-Expand` techinuqe that is effective only for SCS instances in which the same characters often appear in the same strings. [12] surveys practical SCS heuristics.

IX. Conclusion

In this work, we exploit new alternatives to optimize policy classifiers, introducing novel techniques that operate on the inter-policy level. We show how to share classes among policies and analyze the proposed algorithms analytically. Our evaluation study has shown significant gains from sharing classes and using partial policy orders on a single network element, varying the structural properties of represented policies on a single network element.

Appendix

Proof of Theorem 8. The proof is by reduction from the WFVS problem [7]. For a directed weighted on vertices graph $G = (V, E)$ we construct $\mathcal{P} = \{P_1, P_2\}$ such that $\mathbb{S}$ with a minimal overhead corresponds to FVS in G with the minimal total weight.

For each vertex $v_i \in V$ we create three classes: an *input* class c_i^1, a *middle* class c_i^2, and an *output* class c_i^3. For each edge $(v_i, v_j) \in E$ we create a class $c_{i,j}$. The size of input classes is equal to the weights of the corresponding vertices in G, the size of all other classes is equal to the total weight of all vertices in V plus 1. For each $v_i \in V$, the class c_i^2 intersects with both c_i^1 and c_i^3. For each $(v_i, v_j) \in E$, the class $c_{i,j}$ intersects with both c_i^3 and c_j^1. All other classes are pairwise disjoint. Each class in every policy is associated with a different action. The policy P_1 contains the classes c_i^1, c_i^3 for each $v_i \in V$ and classes $c_{j,k}$ for each $(v_j, v_k) \in E$. The partial order of P_1 is defined as follows: for each edge $(v_i, v_j) \in E : c_i^3 \prec_{P_1} c_{i,j} \prec_{P_1} c_j^1$. The policy P_2 consists of all classes c_i^1, c_i^2, c_i^3 for each $v_i \in V$. The partial order of P_2 is defined as follows: for each $v_i \in V : c_i^1 \prec_{P_2} c_i^2 \prec_{P_2} c_i^3$.s Figure 12 illustrates the reduction from FVS to PSP for a graph G consisting of four vertices.

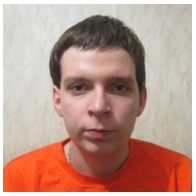
Each feedback vertex set V_G in a graph G corresponds to FVS $V_{G^{\text{jnt}}}$ in a joint graph G^{jnt} for P_1, P_2: $c_i^1 \in V_{G^{\text{jnt}}} \equiv v_i \in V_G$. Since the size of each non-input class is bigger than the weight of all vertices in G, an optimal FVS in G corresponds to the optimal FVS in G^{jnt}. Each valid $\mathbb{S}$ corresponds to FVS in G^{jnt} consisting of the classes appearing twice in $\mathbb{S}$. On the other hand, each FVS in G^{jnt} produces a valid $\mathbb{S}$ by AO. In the case of two policies $\mathrm{W}^+(\mathbb{S})$ is equal to the weight of the corresponding FVS in G^{jnt}. Thus, the algorithm for PSP on two policies finds an optimal FVS in G^{jnt} and G. □

References

[1] V. Demianiuk, S. Nikolenko, P. Chuprikov, and K. Kogan, "New alternatives to optimize policy classifiers," in *ICNP*, Sept 2018, pp. 121–131.

[2] "Qos: Modular qos command-line interface configuration guide, cisco ios," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_mqc/configuration/15-mt/qos-mqc-15-mt-book/qos-mqc.html.

[3] "Class of service configuration guide," https://www.juniper.net/documentation/en_US/junos11.1/information-products/topic-collections/security/software-all/class-of-service/index.html.
[4] "Configuring modular QoS packet classification," http://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-2/qos/configuration/guide/qc42clas.html.
[5] "Cisco xr 12000 series gigabit ethernet line cards," https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html.
[6] D. E. Foulser, M. Li, and Q. Yang, "Theory and algorithms for plan merging," *Artif. Intell.*, vol. 57, no. 2-3, pp. 143–181, 1992.
[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. NY: W. H. Freeman & Co., 1979.
[8] G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, Feb 1998.
[9] C. Demetrescu and I. Finocchi, "Combinatorial algorithms for feedback problems in directed graphs," *Inf. Process. Lett.*, vol. 86, no. 3, pp. 129–136, 2003.
[10] T. Jiang and M. Li, *On the approximation of shortest common supersequences and longest common subsequences*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 191–202.
[11] S. Kasif, Z. Weng, A. Derti, R. Beigel, and C. DeLisi1, "A computational framework for optimal masking in the synthesis of oligonucleotide microarrays," *Nucleic Acids Research*, vol. 30, no. 20, p. e106, 2002.
[12] K. Ning and H. Leong, "Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm," *BMC Bioinformatics*, vol. 7, no. Suppl 4, p. S12, 2006.
[13] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.
[14] "New alternatives to optimize policy classifiers," https://github.com/PolicyCompressor/PolicyCompr.
[15] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput," in *SIGCOMM*, 2010, pp. 207–218.
[16] H. Song and J. S. Turner, "ABC: Adaptive binary cuttings for multidimensional packet classification," *IEEE/ACM Trans. Netw.*, vol. 21, no. 1, pp. 98–109, 2013.
[17] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *ANCS*, 2006.
[18] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM*, 1998.
[19] A. Bremler-Barr and D. Hendler, "Space-efficient tcam-based classification using gray coding," *IEEE Trans. Computers*, vol. 61, no. 1, pp. 18–30, 2012.
[20] O. Rottenstreich and I. Keslassy, "Worst-case TCAM rule expansion," in *INFOCOM*, 2010.
[21] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *INFOCOM*, 2013.
[22] V. Demianiuk and K. Kogan, "How to deal with range-based packet classifiers," in *SOSR*, 2019, pp. 29–35.
[23] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards Efficient Implementation of Packet Classifiers in SDN/OpenFlow," in *HotSDN*, 2013.
[24] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (Scalable And eXpressive PAcket Classification)," in *SIGCOMM*, 2014.
[25] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich, "FIB efficiency in distributed platforms," in *ICNP*, 2016, pp. 1–10.
[26] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Bérczi-Kovács, and A. Shalimov, "How to represent ipv6 forwarding tables on ipv4 or MPLS dataplanes," in *INFOCOM Workshops*, 2016, pp. 521–526.
[27] P. Chuprikov, K. Kogan, and S. I. Nikolenko, "General ternary bit strings on commodity longest-prefix-match infrastructures," in *ICNP*. IEEE Computer Society, 2017, pp. 1–10.
[28] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and speed tradeoffs in TCAM hierarchical packet classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.
[29] C. Umans, "The minimum equivalent DNF problem and shortest implicants," *J. Comput. Syst. Sci.*, vol. 63, no. 4, pp. 597–611, 2001.
[30] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan, "Strategies for Mitigating TCAM Space Bottlenecks," in *HOTI*, 2014.
[31] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM*, 2013, pp. 545–549.
[32] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *CoNEXT*, 2013, pp. 13–24.
[33] P. Chuprikov, K. Kogan, and S. I. Nikolenko, "How to implement complex policies on existing network infrastructure," in *SOSR*, 2018, pp. 9:1–9:7.
[34] V. Demianiuk, K. Kogan, and S. I. Nikolenko, "Approximate classifiers with controlled accuracy," in *INFOCOM*, April 2019, pp. 2044–2052.
[35] V. G. Timkovsky, "Ten etudes on shortest common nonsubsequence and supersequence approximations," Unpublished manuscript, 1993.
[36] V. G. Timkovskii, "Complexity of common subsequence and supersequence problems and related problems," *Cybernetics*, vol. 25, no. 5, pp. 565–580, Sep 1989.
[37] M. Fellows, M. Hallett, and U. Stege, "Analogs & duals of the mast problem for sequences & trees," *Journal of Algorithms*, vol. 49, no. 1, pp. 192 – 216, 2003.
[38] Z. Gotthilf and M. Lewenstein, "Improved approximation results on the shortest common supersequence problem," in *SPIRE*, 2009, pp. 277–284.
[39] P. Barone, P. Bonizzoni, G. Della Vedova, and G. Mauri, "An approximation algorithm for the shortest common supersequence problem: An experimental analysis," in *SAC1 2001*, 01 2001, pp. 56–60.



Vitalii Demianiuk Vitalii Demianiuk is a postdoctoral fellow at the Ariel University. He completed his Ph.D. studies at the Steklov Institute of Mathematics at St.Petersburg in 2019. While pursuing his PhD he worked as a Research Assistant at IMDEA Networks Institute, Madrid. He obtained his M.Sc. from the ITMO University, St. Petersburg in 2016. His research interests include packet classification, software defined networks, network function virtualization, and combinatorial optimization.



Sergey Nikolenko Sergey Nikolenko is an Associate Professor at the Higher School of Economics at St. Petersburg, Russia, and a Laboratory Head at the Steklov Institute of Mathematics at St. Petersburg, Russia. He is doing research in machine learning (deep learning, Bayesian methods, natural language processing, and more), analysis of algorithms (algorithms for networking, competitive analysis, theoretical computer science), and mathematics. He obtained his Ph.D. degree from the Steklov Institute of Mathematics in 2009 and his M.Sc. from the St. Petersburg State University in 2005.



Pavel Chuprikov Pavel Chuprikov received his Ph.D. degree from the Higher School of Economics, Moscow after completing Ph.D. studies at the Steklov Institute of Mathematics at St.Petersburg. While pursuing his PhD he worked as a Research Assistant at IMDEA Networks Institute, Madrid. Currently, Pavel is a Postdoctoral researcher at the Università della Svizzera Italiana. His research interests include software defined networking, online algorithm design, and dependent types.



Kirill Kogan Kirill Kogan is a Senior Lecturer at Ariel University. He received his PhD from Ben-Gurion University (Israel) at 2012. He is a former Technical Leader at Cisco Systems, where he worked during 2000-2012. He was a Postdoctoral Fellow at University of Waterloo and Purdue University during 2012-2014. His current research interests are in design, analysis, and implementation of networked systems, broadly defined.

B Paper “Approximate Classifiers with Controlled Accuracy”

Authors. Vitalii Demianiuk, Kirill Kogan, and Sergey Nikolenko

Abstract. Performing exact computations can require significant resources. Approximate computing allows to alleviate resource constraints, sacrificing the accuracy of results. In this work, we consider a generalization of the classical packet classification problem. Our major contribution is to introduce various representations for approximate packet classifiers with controlled accuracy and optimization techniques to reduce classifier sizes exploiting this new level of flexibility. We validate our theoretical results with a comprehensive evaluation study.

Approximate Classifiers with Controlled Accuracy

Vitalii Demianiuk*†, Kirill Kogan*, Sergey I. Nikolenko†,‡
*IMDEA Networks Institute, Madrid
†Steklov Institute of Mathematics at St. Petersburg
‡NRU Higher School of Economics, St. Petersburg

***Abstract*—Performing exact computations can require significant resources. *Approximate computing* allows to alleviate resource constraints, sacrificing the accuracy of results. In this work, we consider a generalization of the classical *packet classification problem*. Our major contribution is to introduce various representations for *approximate packet classifiers* with controlled accuracy and optimization techniques to reduce classifier sizes exploiting this new level of flexibility. We validate our theoretical results with a comprehensive evaluation study.**

I. Introduction

Exact computations may require excessive resources. *Approximate computing* deals with potentially inaccurate computations helping alleviate resource constraints [1]. In this paper we generalize a classical *packet classification problem* (the exact case), where each rule is composed from a ternary bit string and an action; since rules can overlap they have priorities and the first *matched* rule is returned for an incoming packet. There is a long line of research exploring various optimization methods to find *semantically equivalent* packet classifiers, where each header matches the same action in an originally given and optimized classifiers [2], [3], [4]. In this work we consider the case when semantically equivalent classifiers fail to achieve desired optimization results and introduce approximate representations of packet classifiers allowing to "multiplex" multiple actions. This additional level of flexibility allows to improve resource requirements by still keeping desired level of accuracy.

Majority of proprietary heuristics optimizing packet classifiers can be reduced to well-known operators minimizing the size of Boolean expressions [5], [6]. Unfortunately, using these operators for the approximate case can lead to exponential number of the considered variants. To avoid this complexity, we generalize the basic operators (used in the exact case) to the approximate case and study the properties of different operation sequences that significantly simplify optimization process and achieve better optimization results. In the case of prefix classifiers with LPM priorities, we show how to generalize the algorithm eLP [7], [8] that constructs a classifier with minimal number of entries to the approximate case. As a byproduct, we improve its time complexity by a factor of a, where a is the number of different actions in an input classifier.

II. Motivating Examples

We begin with two motivating examples that introduce the idea of approximate representations. First, consider a common situation when the rules of a classifier map headers to a quantitative characteristic, e.g., desired latency, which is further aggregated into service classes which the packets

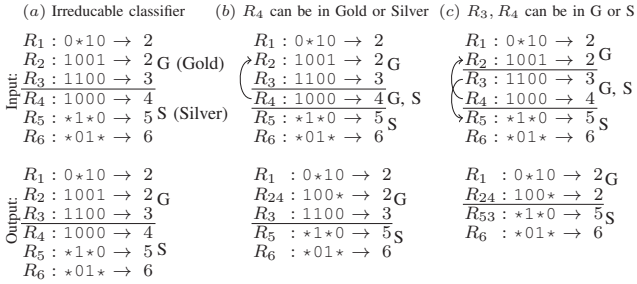


Fig. 1. Tradeoff between accuracy and required memory: (a) exact case; (b-c) approximate case.

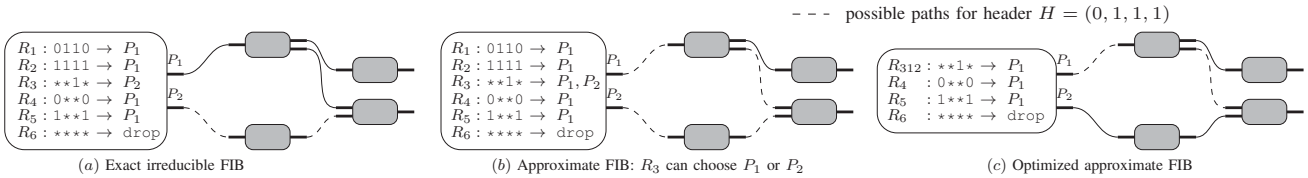
are classified into. E.g., on Fig. 1a values 1-3 are mapped to the *Gold* class; 4-6, to *Silver*. The classifier in Fig. 1a is irreducible with respect to *semantic equivalence*: no smaller classifier maps exactly the same headers to the same classes.

However, let us consider "borderline" rules between service classes. Suppose that we can allow packets falling under R_4, which currently maps to *Silver* but has value 4, very close to the lower bound of the *G*old class, to be associated with either *Gold* or *Silver* classes. In this case we can perform further optimizations, associating R_4 with *Gold* or *Silver* to better reduce the classifier; Fig. 1b shows how R_4 can be merged with R_2 and replaced by R_{24} that maps to *Gold*.

We can take the approximation one step forward. Since R_3 has value 3, which is very close to the upper bound of the *Silver* class, we can allow R_3 to associate with *Silver* (Fig. 1c), and classifier optimization can now exploit this additional flexibility, merging R_3 with R_5 to get R_{53}. In general, Fig. 1 illustrates the fundamental tradeoff between accuracy and efficiency in approximate classifier representations.

Note that even in this specific example, not only borderline rules between different classes may be extended with additional actions. Although R_6 maps to 6, very far from the lower bound of the *Gold* class, it might still be allowed to extend R_6 with the *Gold* option due to some additional considerations. For instance, if we know *a priori* that required bandwidth for the traffic classified by R_6 is relatively small compared to the total volume of *Gold* traffic, we might be able to afford this approximation anyway.

The second motivating example is of a different nature, showing that approximate classifier representations are a much more general tool. The scalability problem of forwarding tables (FIBs) is a largely unsolved problem to date [9]. One can run a third-party process that estimates the "quality" of different paths for the header space covered by a given traffic matrix. It turns out that by exploiting approximate



(a) Exact irreducible FIB (b) Approximate FIB: R_3 can choose P_1 or P_2 (c) Optimized approximate FIB

Fig. 2. Approximate computing in routing.

representations and extending the actions of already computed FIBs with acceptable alternatives (based on the "quality" of estimated paths), we can often reduce the classifier size.

The exact classifier in Fig. 2a is irreducible and consists of 6 entries. For instance, the header $H = (0, 1, 1, 1)$ is classified by R_3 and follows path of length 3. If we allow R_3 to redirect traffic through the port P_1, the optimization process will be able to reduce two entries. Now headers classified by R_3 will be transmitted through the port P_1, and other headers will not be affected. In particular, the header H in Fig. 2c now follows another path with the same length.

III. Model Description

We begin with the basic notions. A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits $h_i \in H$, $h_i \in \{0, 1\}$, $1 \leq i \leq w$; e.g., $(1\ 0\ 1\ 0)$ is a 4-bit header. A filter $F_i = (f_1^i, \ldots, f_w^i)$ is a sequence of w values corresponding to bits in the headers, but the possible values are now 0, 1, or $*$ ("don't care"). A *rule* $R_i = (F_i, \{\mathcal{A}_i\})$ consists of a *filter* F_i and a pointer to the corresponding *action set* $\mathcal{A}_i$. A header H is *matched* by a filter F (or by a rule $R = (F, \mathcal{A})$) if for every bit of H the corresponding bit of F has either the same value or $*$.

A *classifier* $\mathcal{K} = \{R_1, \ldots, R_N\}_{\prec}$ is an ordered (by $\prec$) set of rules. If multiple rules match a header, the rule R with highest priority is preferred, and we say that H is *classified* by R (or its filter). The classifier finds the classifying action for a given header. We assume that every classifier contains a "catch-all" rule $R_\perp$ at the end, matching all headers unmatched by other rules; $R_\perp$ has a unique action D that no other rule has.

Example 1. In the following classifier with $w = 4$, $H = (0\ 1\ 1\ 0)$ matches R_1, R_4, $R_\perp$, so H is classified by R_1.

$\mathcal{K}$	#1	#2	#3	#4	Exact	Approx
R_1	0	1	1	0	A_1	A_1, A_4
R_2	1	$*$	0	$*$	A_2	A_2
R_3	1	0	1	$*$	A_3	A_3
R_4	0	$*$	1	$*$	A_4	A_4
R_5	$*$	0	0	$*$	A_5	A_5
$R_\perp$	$*$	$*$	$*$	$*$	D	D

We say that a rule R_i *covers* a rule R_j if all headers that match R_j also match R_i. $R_\perp$ always covers all other rules. Two rules R_i, R_j *intersect* ($R_i \cap R_j \neq \emptyset$) if there exists a header matched by both rules. In Example 1, R_4 covers R_1, while R_2 and R_5 intersect.

The notion of *exact* and *approximate* classifiers is central for this work. We say that a classifier $\mathcal{K}$ is *exact* if for each rule $R_i \in \mathcal{K}$ $\mathcal{A}_i$ consists of a single element; otherwise, $\mathcal{K}$ is *approximate*. Exact classifiers are a traditional way to represent packet classifiers. Approximate classifiers are a generalization where $\mathcal{A}_i$ represents multiple options that can be taken for the matched rule. In Example 1, the filters of the classifier combined with the action sets in Exact and Approx columns form the exact $\mathcal{K}_e$ and approximate $\mathcal{K}_a$ classifiers, respectively. E.g., if classifiers represent forwarding tables, action set $\mathcal{A}_1 = \{A_1, A_4\}$ means that a packet can be transmitted through either A_1 or A_4. Note that in this case $\mathcal{A}_1 = \{A_1, A_4\}$ does not represent possibilities for load balancing but adds options for an *optimization process* whose output is always an exact classifier. We say that a classifier $\mathcal{K}'$ is *approximately equivalent* to a classifier $\mathcal{K}$ if for every header H, the classifying action set $\mathcal{A}' \in \mathcal{K}'$ is a subset of the classifying action set $\mathcal{A} \in \mathcal{K}$. In Example 1, the classifier $\mathcal{K}_e$ is approximately equivalent to $\mathcal{K}_a$.

IV. Basic Operators

In this section, we show basic operators reducing the number of entries in exact and approximate classifiers.

A. Classifiers as Boolean Expressions

Optimization methods that minimize the required space for classifiers [2], [3], [4] usually can be reduced to methods minimizing size of Boolean expressions. There are two main cases here. If no pair of rules intersect then the classifier represents a Boolean formula in DNF, with every filter as a clause. The *MinDNF* problem (minimizing DNF size) has been extensively studied in complexity theory. In theory, it is $\mathrm{NP}^{\mathrm{NP}}$-complete ($\Sigma_2^P$-complete), with some inapproximability results as well [5], [6]. In practice, classical heuristics for *MinDNF* are based on Karnaugh maps.

In the order-dependent case, rule priorities can be encoded with circuits of depth 3 with OR–AND–OR alternation, i.e., AC^0 circuits of depth 3. However, even in the order-independent case there is no tractable algorithm minimizing Boolean expressions, so we cannot hope for polynomial optimal algorithms minimizing size of classifiers. Existing heuristics include a set of basic operators applied to a classifier while possible. They consist of two major blocks: *basic operators* and an *optimization process* that constructs a sequence of basic operations to reduce the classifier.

B. Generalized basic operators

Usually, three basic operators are considered in the scope of packet classifiers [4] since they have the right balance between operational complexity and applicability. These basic operators decide when a rule can be removed or replaced by another rule. Here, we further generalize them for the approximate case.

If action sets consist of a single action, they can be directly applied to exact classifiers.

Example 2. Through this section we will use the following classifier as a running example. Actions in the Exact column correspond to the exact classifier $\mathcal{K}_e$; the Approx column corresponds to the approximate classifier $\mathcal{K}_a$:

$\mathcal{K}$	#1	#2	#3	#4	#5	Exact	Approx
R_1	0	1	1	$*$	0	A_1	A_1
R_2	$*$	1	$*$	0	$*$	A_1	A_1, A_2
R_3	1	1	$*$	1	1	A_2	A_1, A_2
R_4	$*$	1	1	$*$	$*$	A_1	A_1, A_2
R_5	0	1	$*$	1	1	A_2	A_1, A_2
R_6	$*$	0	$*$	1	1	A_2	A_2
R_7	0	0	0	1	1	A_3	A_3
$R_\perp$	$*$	$*$	$*$	$*$	$*$	D	$\{D\}$

Forward Subsumption $\mathcal{F}(R_i)$. This operator removes all unreachable rules and does not depend on the structure of action sets. Formally, a rule $R_i \in \mathcal{K}$ can be removed if there exists a rule $R_j \prec R_i$ that covers R_i. Since forward subsumptions do not depend on actions, the resulting classifier is approximately equivalent to the original. In Example 2, R_7 can be removed since R_6 covers it.

Backward Subsumption $\mathcal{F}(R_i)$. Sometimes a rule can still be removed if it is covered by a rule with lower priority. Formally, $R_i \in \mathcal{K}$ can be removed if the following conditions hold:

1) there is a rule $R_j \succ R_i$ that covers R_i such that their action sets $\mathcal{A}_i$ and $\mathcal{A}_j$ *intersect*;
2) for every rule $R_t \in \mathcal{K}$ such that $R_i \prec R_t \prec R_j$ and $R_t \cap R_i \neq \emptyset$, the *intersection* of $\mathcal{A}_t$ and $\mathcal{A}_i$ is also nonempty; after applying backward subsumption, we set $\mathcal{A}_t := \mathcal{A}_t \cap \mathcal{A}_i$ for each such rule.

If several R_j are possible, we choose R_j with the largest priority.

In Example 2 (approximate case), R_1 can be removed by backward subsumption since it is covered by R_4, and even though R_1 intersects with R_2 they have intersecting action sets. Note that the header (0 1 1 0 0) that was previously classified by R_1 will now be classified by R_2, so we replace $\mathcal{A}_2, \mathcal{A}_4$ by the sets $\mathcal{A}_2 \cap \mathcal{A}_1 = \{A_1\}$ and $\mathcal{A}_4 \cap \mathcal{A}_1 = \{A_1\}$. The exact case is very similar but now action sets of rules R_1, R_2, R_4 are equal, therefore, this operation is allowed.

Resolution $R'_i = \mathcal{R}(R_i, R_j)$. This operator comes from propositional proof theory: in propositional logic, expressions $(x \wedge C) \vee (\bar{x} \wedge C)$ and $(x \vee C) \wedge (\bar{x} \vee C)$ are both obviously equivalent to C and can be simplified to C.

Two rules $R_i \prec R_j$ can be combined and replaced by a new rule R'_i (in place of R_i in the list) if:

1) filters F_i and F_j coincide in all bit indices except k, and the k-th bit is not $*$ in F_i or F_j (when it is, subsumption applies);
2) the set $\mathcal{A}'_i = \mathcal{A}_i \cap \mathcal{A}_j \cap \bigcap_{t: R_i \prec R_t \prec R_j \text{ and } R_t \cap R_j \neq \emptyset} \mathcal{A}_t$ (intersection of action sets over all intersecting rules) is nonempty; after applying resolution we set $\mathcal{A}_i := \mathcal{A}'_i$.

In Example 2, F_3 and F_5 differ only in the first bit. In the approximate classifier, we can apply $R'_3 := \mathcal{R}(R_3, R_5)$ since action sets of R_3, R_4, R_5 are all equal to $\{A_1, A_2\}$; in the exact case, however, we would not be able to apply $\mathcal{R}(R_3, R_5)$ since actions for R_4, R_5 differ.

Both subsumption operators are unary operators (removing one rule), and resolution is a binary operator (replacing two rules with one). Applying the introduced operators to exact and approximate classifiers in Example 2 leads to classifiers with six and four rules, respectively.

C. Why to generalize

In general, we can choose a single action for every rule in an approximate classifier, and each such combination is an exact classifier. In this way we could exploit already existing heuristics applicable to previously studied basic operators in the exact case. The problem is not only the exponential number of different exact classifiers but also that generalized versions of the basic operators extend their applicability.

Example 3. We show two approximate classifiers, where generalized operators can achieve better results than even brute-force search over all exact classifiers.

$\mathcal{K}$	#1	#2	#3	#4	$\mathcal{A}$
R_1	0	1	0	$*$	A_1, A_2
R_2	0	$*$	$*$	0	A_1
R_3	$*$	$*$	0	$*$	A_2
R_4	0	1	$*$	$*$	A_1, A_2
$R_\perp$	$*$	$*$	$*$	$*$	D

Applying approximate operators in this classifier, R_1 can be removed by backward subsumption: R_4 covers R_1, and all necessary conditions hold. On the other hand, in neither of the four possible exact specializations of $\mathcal{K}$ we can reduce R_1 or any other rule.

Similarly, for the following classifier two resolution operations can be applied, $\mathcal{R}(R_1, R_5)$ and $\mathcal{R}(R_2, R_4)$, but in any exact case only one of them remains:

$\mathcal{K}$	#1	#2	#3	#4	$\mathcal{A}$
R_1	0	0	$*$	1	A_1
R_2	1	1	$*$	0	A_2
R_3	$*$	$*$	0	$*$	A_1, A_2
R_4	1	0	$*$	0	A_2
R_5	0	1	$*$	1	A_1
$R_\perp$	$*$	$*$	$*$	$*$	D

V. Properties of Heuristics

An *optimization process* consists of a set of *basic operators* (that we have introduced above) and a *heuristic* that chooses a sequence of applied operations (*operation sequence*). In this section we study properties of operation sequences to improve optimization results and reduce the time complexity of considered heuristics.

By definition, each applied operation removes exactly one rule, so after an operation sequence $\mathcal{S}$ on a classifier $\mathcal{K}$ the optimized classifier has $|\mathcal{K}| - |\mathcal{S}|$ rules. Therefore, considered heuristics can shift from minimizing number of rules to maximizing length of a constructed operation sequence. A sequence of applied operations is *optimal* if its length is maximal among all other sequences. Similarly, an operation sequence $\mathcal{S}$ is *extensible*, if $\mathcal{S}$ can be appended with additional

operations; otherwise, $\mathcal{S}$ is *inextensible*. An optimal sequence is always inextensible, but there are suboptimal inextensible.

Example 4. The following example shows that different operation sequences even in the exact case can lead to different results.

P	#1	#2	#3	#4	#5	*Action*
R_1	1	1	1	$*$	$*$	A_1
R_2	1	0	$*$	$*$	0	A_1
R_3	1	1	0	$*$	0	A_1
R_4	1	1	1	$*$	0	A_1
$R_\perp$	$*$	$*$	$*$	$*$	$*$	D

Here, we can apply either forward subsumption $\mathcal{F}(R_4)$ or resolution $R = \mathcal{R}(R_3, R_4)$ followed by another resolution $\mathcal{R}(R_2, R)$. Both operation sequences are inextensible, but only one of them is optimal, and forward subsumption cannot lead to new operations.

Example 4 shows that a combinatorial search for optimal operation sequences is a hard problem even in the exact case. Therefore, in what follows we will study a specific fixed set of operations applicable to the original classifier. If we can find the longest operation sequence among a reasonably wide class of such sets, it will provide us with a lower bound on the length of the optimal operation sequence.

We denote by $\mathbf{O}$ a set of operations applicable to the initial classifier that does not contain *conflicting* operations using the same rule; that is, if there are multiple operations applicable to the same rule we choose one of them. Note that there can be many different $\mathbf{O}$ for the same classifier $\mathcal{K}$; at this point we fix $\mathcal{K}$ and one specific $\mathbf{O}$. Operations in $\mathbf{O}$ implicitly depend on each other; for example, a resolution operation can create a new possible rule that will intersect with another rule in backward subsumption. We make one additional assumption: if for some rule R_i both $\mathcal{F}(R_i)$ and $\mathcal{B}(R_i)$ are applicable in the original classifier, we assume that $\mathbf{O}$ contains $\mathcal{F}(R_i)$ since it makes no sense to apply backward instead of forward subsumption.

A. Operation sequences in the exact case

The following theorem shows that in the exact case, we can apply all operations in $\mathbf{O}$ regardless of these implicit dependencies. Later we will see that in the approximate case maximal sequences on $\mathbf{O}$ have a more complex structure. Despite the popularity of Boolean minimization in the exact case, to the best of our knowledge we know of no prior work where this result was proven.

Theorem 1. *If all action sets in $\mathcal{K}$ contain exactly 1 action (the exact case), there is an applicable sequence consisting of all operations from $\mathbf{O}$.*

Proof. First we apply all forward subsumptions, then all backward subsumptions. We can apply all forward subsumptions because for any applicable $\mathcal{F}(R_i)$ classifier will contain a rule $R_j \prec R_i$ that covers R_i regardless of already applied forward subsumptions. Also, forward subsumptions cannot change the covering rule R_j for a backward subsumption $\mathcal{B}(R_i)$ since otherwise either R_i were subject to forward subsumption in the original classifier or the rule that covers R_j is covering in $\mathcal{B}(R_i)$. Therefore, after forward subsumptions we can apply $\mathcal{B}(R_i)$ in the order of priority of R_i since applied $\mathcal{B}(R_i)$ does not affect backward subsumptions further down the list. Subsumptions only remove rules that are not used by resolutions in $\mathbf{O}$; therefore, we can apply all resolutions after all subsumptions. We apply $\mathcal{R}(R_i, R_j)$ in the order of priority on R_i. Again, no conflicts arise in this order since $\mathcal{R}(R_i, R_j)$ can only remove a rule between R'_i and R'_j for some further applicable resolution $\mathcal{R}(R'_i, R'_j)$. □

The optimal sequence of operations does not always apply all operations from $\mathbf{O}$: some rules can find better use in new operations that were not possible in $\mathbf{O}$. But there is a lower bound on the length of the optimal sequence.

Corollary 1. *The length of an optimal operation sequence in the exact case is at least the maximal length of all possible $\mathbf{O}$; moreover, this lower bound can be computed in time $O(N^2 \cdot w + C_{\mathcal{R}}^{1.5})$, where $C_{\mathcal{R}}$ is the number of resolutions applicable in the original classifier.*

Proof. To construct a maximal $\mathbf{O}$, first we take all subsumptions on different rules and then choose a maximal number of non-conflicting resolutions. This is an optimal algorithm since we look only on operations applicable to the original classifier, and it makes to sense to apply resolution instead of subsumption if both apply. We ignore all resolutions that conflict with a chosen subsumption rule. On the other applicable resolutions, we construct a bipartite graph $G = (V_0, V_1, E)$: the set of vertices V_0 contains rules whose filters have even number of bits with value 1, V_1 consists of rules with odd number of such bits, and edges consist of possible resolutions, i.e., each resolution connects a vertex from V_0 with a vertex from V_1 in G. Edges in a maximal matching in G form a maximal set of non-conflicting resolutions that can be added to $\mathbf{O}$. A maximal matching can be found in time $O(N + C_{\mathcal{R}} \cdot min(N, C_{\mathcal{R}})^{0.5})$. □

So far, we have seen that while finding the best possible operation sequence is always hard, in the exact case it is relatively easy to construct a maximal set of non-conflicting operations $\mathbf{O}$ that can be applied to the original classifier. Next we turn to the approximate case.

B. Operation sequences in the approximate case

In the approximate case, $\mathbf{O}$ can increase compared to the exact case since generalized operations are applicable more often (recall Example 3). But unlike the exact case, Theorem 1 does not hold for the approximate case, and an operation sequence of length $|\mathbf{O}|$ is not guaranteed to exist.

Example 5. Operations from $\mathbf{O}$ are not always applicable together.

$\mathcal{K}$	#1	#2	#3	#4	$\mathcal{A}$
R_1	0	1	1	$*$	A_1
R_2	1	1	$*$	1	A_1, A_2
R_3	$*$	1	1	$*$	A_1, A_2
R_4	0	1	$*$	1	A_2
$R_\perp$	$*$	$*$	$*$	$*$	D

For instance here, we can apply resolution $\mathcal{R}(R_2, R_4)$ or backward subsumption $\mathcal{B}(R_1)$, but we cannot apply both since otherwise $\mathcal{A}_2$ would become empty.

The complexity of constructing the longest operation sequence on $\mathbf{O}$ arises from backward subsumptions.

Theorem 2. *A maximal sequence of applicable operations from* $\mathbf{O}$ *can be found in polynomial time if* $\mathbf{O}$ *consists of forward subsumptions and resolutions.*

Proof. Apply all operations from $\mathbf{O}$ as follows: first all forward subsumptions, then all resolutions applied in the order of decreasing priority of the top rule. □

But if $\mathbf{O}$ has backward subsumptions then finding the longest sequence turns into an intractable problem.

Theorem 3. *Finding a maximal sequence of applicable backward subsumptions from* $\mathbf{O}$ *is an NP-complete problem; moreover, it is not approximable up to a constant factor in polynomial time.*

Proof. We reduce the Maximal Independent Set problem: by a graph $G = (V, E)$ we construct a classifier where every possible sequence of applied backward subsumptions will correspond to an independent set of G. We denote $n = |V|$ and $m = |E|$. The classifier will consist of $2n + m$ rules. The first n rules $R_1, \ldots, R_n$ and last n rules $R_{n+m+1}, \ldots, R_{2n+m}$ correspond to vertices, and the m rules in the middle correspond to edges. A rule R_i for $1 \leq i \leq n$ is covered by a rule R_{n+m+i} and additionally it intersects exactly those rules that correspond to the edges adjacent to v. For all rules R_i that correspond to vertices their $\mathcal{A}_i$ will contain a special action A_0. For each edge $e = (u, v)$, we create two actions $A_{e,u}$ and $A_{e,v}$: action sets for rules corresponding to u contain $A_{e,u}$; to v, $A_{e,v}$; to e, both. In this construction, for any edge $e = (u, v)$ backward subsumptions on the rules corresponding to u and v are allowed separately but not together, and this implies that the set of applied backward subsumptions corresponds to an independent set in G, and finding the largest such set is equivalent to finding the largest independent set. □

C. How to extend operation sequences

So far we have studied properties of operation sequences constructed on a set $\mathbf{O}$ of initially non-conflicting operations for both exact and approximate cases. In this part, we study the properties of optimal operation sequences in the approximate case. The following structural property of operation sequences allows us to reduce the space of operation sequences considered to find the longest (optimal) one.

Lemma 1. *For an operation sequence* $\mathcal{S}$*, there exists another operation sequence* $\mathcal{S}'$ *that satisfies the following:*

1) $\mathcal{S}'$ *consist of the same operations, but a backward subsumption* $\mathcal{B}(R)$ *can be changed to a forward subsumption* $\mathcal{F}(R)$*;*
2) *resolutions and forward subsumptions are applied before backward subsumptions;*
3) *backward subsumptions* $\mathcal{B}(R)$ *are applied in the reverse order of the priorities of* R*;*
4) *for each rule* R_i*, its final action set* $\mathcal{A}_i$ *obtained after* S *is applied is a subset of the action set* $\mathcal{A}'_i$ *after* S' *is applied.*

Due to the space limits, the proof of Lemma 1 is omitted. Lemma 1 immediately implies the following corollary.

Corollary 2. *There exists an optimal operation sequence where all resolutions and forward subsumptions are applied before backward subsumptions, and all backward subsumptions are applied in the increasing order of priorities of their rules.*

The impact of this transformation is that after changing an operation sequence $\mathcal{S}$ to such an $\mathcal{S}'$, inextensible sequences may become extensible, as the following example shows.

Example 6. Suppose that in Example 2, we apply $\mathcal{F}(R_7)$ followed by $\mathcal{B}(R_1)$ and then $R'_3 = \mathcal{R}(R_3, R_5)$, getting the following classifier:

$\mathcal{K}$	#1	#2	#3	#4	#5	$\mathcal{A}$
R_2	$*$	1	$*$	0	$*$	$\{A_1\}$
R'_3	$*$	1	$*$	1	1	$\{A_1\}$
R_4	$*$	1	1	$*$	$*$	$\{A_1\}$
R_6	$*$	0	$*$	1	1	$\{A_2\}$
$R_\perp$	$*$	$*$	$*$	$*$	$*$	$\{D\}$

But if we applied $R'_3 = \mathcal{R}(R_3, R_5)$ before $\mathcal{B}(R_1)$, we would have $\mathcal{A}'_3 = \{A_1, A_2\}$. Therefore, we can apply $\mathcal{R}(R'_3, R_6)$ and get the following classifier:

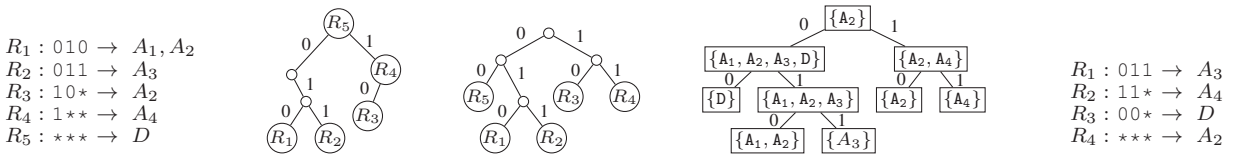
$\mathcal{K}$	#1	#2	#3	#4	#5	$\mathcal{A}$
R_2	$*$	1	$*$	0	$*$	$\{A_1\}$
R''_3	$*$	$*$	$*$	1	1	$\{A_2\}$
R_4	$*$	1	1	$*$	$*$	$\{A_1\}$
$R_\perp$	$*$	$*$	$*$	$*$	$*$	$\{D\}$

This observation allows us to improve optimization results: if we have obtained an inextensible operation sequence $\mathcal{S}$ that does not satisfy the conditions of Lemma 1, we can transform it into a new sequence $\mathcal{S}'$, as explained in the constructive proof of Lemma 1, and $\mathcal{S}'$ may become extensible.

In this section, we have seen algorithms that try to construct an optimal operation sequence from a set of applicable operations $\mathbf{O}$. They are guaranteed to find a maximal sequence only in special cases, and it is a hard computational problem in general. Still, in many practical situations algorithms presented here work well, and we have also introduced a heuristic for further optimization that reorders operations.

VI. Approximate LPM Classifiers

So far we considered a general case where each filter is a ternary bit string. Since minimizing a given classifier is an intractable problem, we defined our optimization process as constructing a maximal sequence over a predefined set of operators. There is, however, an important special case when all rules are prefixes ordered with longest-prefix-match (LPM) priorities; now the alphabet is binary, and the first $*$ is just a delimiter that defines prefix length.



(a) Original classifier *(b)* T_o trie *(c)* T_e *(d)* T_e with calculated $L(x)$ *(e)* Optimized classifier

Fig. 3. Example of running the aLP algorithm.

In the exact case, the works [7], [8] introduce the algorithm eLP that computes a minimal LPM classifier in time $O(N \cdot w \cdot a)$, where N is the number of rules in the input, w is the classification width (in bits), and a is the number of different actions. In this section, we show how to generalize eLP to the approximate case while preserving optimality; we call the generalized version aLP. As a side effect, we improve the time complexity of eLP to $O(N \cdot w)$.

A. The eLP Algorithm

We begin by briefly explaining the basic intuition behind eLP [7], [8]. First, the filters of all rules from the input classifier $\mathcal{K}$ are organized in a binary prefix trie T_o (Fig 3a-b). Note that prefixes of rules in $\mathcal{K}$ do not necessarily end in leaves, and not every internal node in T_o has two sons. To avoid prefixes in internal nodes of T_o, eLP pushes them to leaves, extending the trie to T_e as follows: each node with a single child is complemented with another one; for each added leaf x its corresponding rule inherits the action from x's closest ancestor with rule (see Fig 3c). As a result, in T_e only leaves correspond to rules. We denote by $T_o(x)$ a subtree of T_o rooted in x; by $T_e(x)$, the corresponding subtree of the extended trie. Both $T_o(x)$ and $T_e(x)$ represent equivalent classifiers. $T_e(x)$ can have more corresponding rules than $T_o(x)$ but one can show that optimal classifiers for $T_e(x)$ and $T_o(x)$ are the same.

We denote by R_x a rule that will be created for a node $x \in T_e$ with an action A_x. The algorithm eLP is based on the following process. Let l and r be the left and right children of a node $x \in T_e$. Consider already computed optimal classifiers $\mathcal{K}_l$ and $\mathcal{K}_r$ corresponding to $T_e(l), T_e(r)$ respectively. Now:

- $\mathcal{K}_x = \mathcal{K}_l \cup \mathcal{K}_r \cup \{R_x\}$;
- if $A_x = A_l$, remove R_l from $\mathcal{K}_x$;
- if $A_x = A_r$, remove R_r from $\mathcal{K}_x$.

It can be shown that the resulting classifier $\mathcal{K}_x$ is optimal.

Denoting by $f(x, A_x)$ the size of the optimal classifier $\mathcal{K}_x$, we get the following recurrent expression:

$$f(x, A_x) = \min_{A_l, A_r} f(l, A_l) + f(r, A_r) + 1 - [A_l = A_x] - [A_r = A_x]$$

To implement this process, eLP maintains in each node $x \in T_e$ an explicitly computed list $L(x)$ of actions A that minimizes $f(x, A)$. The list $L(x)$ can be calculated from $L(x_l)$ and $L(x_r)$: if $L(x_l) \cap L(x_r) \neq \varnothing$ then $L(x) = L(x_l) \cap L(x_r)$ (in this case, the corresponding rules in classifiers for x_l and x_r have the same action, and this action will be chosen for the last rule of a classifier at node x), otherwise $L(x) = L(x_l) \cup L(x_r)$ (in this case the rules for x_l, x_r have different actions, and one of them will be chosen for x).

Note that originally, the work [7] only defined the eLP algorithm; subsequent complexity analysis in [8] showed that it runs in time $O(N \cdot w \cdot a)$. In the following, we will show how to generalize eLP to the approximate case and remove the a factor from the complexity bounds for the exact case.

B. Optimality in the approximate case

Let us generalize eLP to the approximate case; we call the new algorithm aLP. Suppose that all $\mathcal{A}_i$ are given as lists of actions, and actions appear in these lists in the same order. The above process for deciding which rules to add or remove remains unchanged. In the approximate case, the recurrence relation for $f(x, A)$ still equals the size of the optimal classifier $\mathcal{K}_x$. Now actions can be recovered up to sets $\mathcal{A}_i$s. This modification does not change $f(x, A)$ and does not affect reconstruction rules for $L(x)$ in internal nodes of T_e. Only the definition of $L(x)$ for leaf nodes changes: $L(x) = \mathcal{A}_{i_x}$, where R_{i_x} is the rule corresponding to a leaf x in T_e. Figure 3 illustrates the aLP algorithm.

Theorem 4. *For the approximate case, time complexity of aLP is $O(N \cdot w \cdot l_{\max})$, where $l_{\max} = \max_i |\mathcal{A}_i|$.*

Proof. We denote by $V(T)$ the set of nodes in a trie (or subtree) $T \subseteq T_e$. The total time complexity of the proposed method is $O(\sum_{x \in V(T_e)} |L(x)|)$ since union and intersection of sets can be done in linear time. Summing $|L(x)|$ over all leaves, we get $O(N \cdot w \cdot l_{\max})$ since T_e has $O(N \cdot w)$ leaves.

Each internal node x of T_e also occurs in T_o. For rules corresponding to a node in $T_o(x)$, all action sets $\mathcal{A}_i$ can participate in the construction of $L(x)$; among other rules, only one rule can participate in $L(x)$ – the one that ends in the nearest ancestor of x. Therefore, $|L(x)| \leq l_{\max} \cdot (s(x)+1)$, where $s(x)$ is number of nodes in $T_o(x)$ where some rules end.

Each node in T_o corresponding to a rule increments $s(x)$ for at most w internal nodes, namely its ancestors. Therefore, $\sum_{x \in T_o} s(x) \leq N \cdot w$. This immediately implies that the sum of $|L(x)|$ over all internal nodes is $O(N \cdot w \cdot l_{\max})$. □

Since approximate case degrades to the exact case when all $|\mathcal{A}_i| = 1$, by Theorem 4 we have improved the time complexity of eLP compared to [8, p. 294] by a factor of a, the number of different actions in the original classifier.

VII. Evaluation

In this section we evaluate the efficiency of proposed approaches on LPM and general approximate classifiers. Optimization results depend not only on the quality of optimization

methods but also on classifier structure; we validate the fundamental tradeoff between memory requirements and desired level of accuracy.

A. LPM classifiers

Heuristics. We compare two algorithms for minimizing approximate LPM classifiers: aLP from Section VI and aBM for the general case (ternary bit strings with general priorities) which applies the basic operators we introduced. In the general case, by Lemma 1 the heuristic applies forward subsumptions and resolutions while possible, followed by backward subsumptions in reverse priority order.

Methodology. We have evaluated our approach on five IPv4 FIB classifiers; Table I shows their most important characteristics. We have made the source code of our simulations available at [10]; unfortunately, due to NDA restrictions we cannot make the classifiers available online. To make an approximate classifier, we extend actions of an exact instance with alternative options (paths), with actions more common in the exact version appearing more frequently in the approximate instance as well.

Each approximate instance is characterized by two parameters: p_{A}, the share of rules with alternative paths, and l_A, the number of alternative actions for each rule. For each rule in the selected share p_{A}, the action set $\mathcal{A}_i$ is extended with randomly selected alternatives until the size of $\mathcal{A}_i$ reaches $l_A + 1$. We choose each next action for $\mathcal{A}_i$ with a probability proportional to its number of occurrences in the exact classifier (except for actions already in $\mathcal{A}_i$).

We denote by $\mathcal{K}'_{\mathrm{aLP}}$ and $\mathcal{K}'_{\mathrm{aBM}}$ the optimized classifiers obtained by running aLP and aBM, respectively, on the original classifier $\mathcal{K}$. The third and fourth columns in Table I show the number of rules in $\mathcal{K}'_{\mathrm{aLP}}$ and $\mathcal{K}'_{\mathrm{aBM}}$ for all five FIBs. For each FIB, we have also constructed a classifier where all possible actions except the default can be chosen as alternative. This is the best case for optimization for a given set of filters; we denote by $\mathcal{K}''_{\mathrm{aLP}}$ and $\mathcal{K}''_{\mathrm{aBM}}$ classifiers optimized with the corresponding algorithms. For all five FIBs, the sizes of $\mathcal{K}''_{\mathrm{aLP}}$ and $\mathcal{K}''_{\mathrm{aBM}}$ are given in the fifth and sixth columns of Table I. During the generation of approximate classifiers, we varied the parameter p_A from 0 to 1 and l_A from 1 to 3, running aLPM and aBM on each generated classifier.

Size of optimized classifiers. Figs. 4(a,d,g) show the number of rules in optimized classifier $\mathcal{K}_{\mathrm{A}}$ as a function of p_{A} (plots with $l_A > 1$ do not show fib1 curves because the fib1 classifier has only two actions). We see that classifiers optimized by aLP are 10-20% smaller than the same classifiers optimized by aBM. The aLP algorithm wins because it is guaranteed to construct the optimal solution in the class of LPM classifiers. The main advantage of Boolean minimization techniques is their flexibility: they can be used for more general classifiers.

Savings in comparison to exact optimization. Plots on Figs. 4(b,e,h) show additional savings $W(\mathcal{K}_{\mathrm{A}})$ for the optimized approximate classifier compared to the corresponding optimized exact version, i.e., $W(\mathcal{K}_{\mathrm{A}}) = \frac{|\mathcal{K}'_{\mathrm{aLP}}|-|\mathcal{K}_{\mathrm{A}}|}{|\mathcal{K}_{\mathrm{aLP}'}|}$ if $\mathcal{K}_{\mathrm{A}}$ was optimized by the aLP algorithm and $W(\mathcal{K}_{\mathrm{A}}) = \frac{|\mathcal{K}'_{\mathrm{aBM}}|-|\mathcal{K}_{\mathrm{A}}|}{|\mathcal{K}'_{\mathrm{aBM}}|}$ in case of aBM. The plots support our theoretical results: the number of rules in optimized approximate classifiers may be significantly smaller, up to 20-50%, than in the optimized exact classifiers. Note that the slope of the curves on the plots for W is different for different datasets, which is a feature of our procedure for generating alternative actions. Classifiers where a small subset of actions covers a large share of the rules are easier to optimize because alternative actions in $\mathcal{A}$ will be likely chosen from this small subset. The largest slope is for fib1 because it has only two actions, and all rules with alternative actions will have the same pair of actions. We have chosen this generation procedure specifically to show off the effect of different action coverage.

TABLE I
FIB DATASETS

Name	$\lvert P\rvert$	$\lvert\mathcal{K}'_{\mathrm{aLP}}\rvert$	$\lvert\mathcal{K}'_{\mathrm{aBM}}\rvert$	$\lvert\mathcal{K}''_{\mathrm{aLP}}\rvert$	$\lvert\mathcal{K}''_{\mathrm{aBM}}\rvert$
fib1	137150	94232	109594	50411	70394
fib2	202671	160836	176446	49838	76718
fib3	180655	140482	154049	50384	75497
fib4	149432	107740	121974	49624	70707
fib5	180429	134580	149974	55512	81918

Comparison to the hypothetical perfect reduction. Figs. 4(c,f,i) show the savings in the optimized classifier compared to the hypothetical reduction in case when every action was allowed for every rule; in this hypothetical case, the classifier simply serves as a "match/no match" binary filter, distinguishing rules that match at least one rule and rules that are matched by $R_\perp$. Obviously, this is the case when the largest reductions are possible; we denote this fraction by $W^+(\mathcal{K}_{\mathrm{A}})$, defined as $W^+(\mathcal{K}_{\mathrm{A}}) = \frac{|\mathcal{K}'_{\mathrm{aLP}}|-|\mathcal{K}_{\mathrm{A}}|}{|\mathcal{K}'_{\mathrm{aLP}}|-|\mathcal{K}''_{\mathrm{aLP}}|}$ for the aLPM algorithm and $W^+(\mathcal{K}_{\mathrm{A}}) = \frac{|\mathcal{K}'_{\mathrm{aBM}}|-|\mathcal{K}'_{\mathrm{A}}|}{|\mathcal{K}'_{\mathrm{aBM}}|-|\mathcal{K}''_{\mathrm{aBM}}|}$ for aBM algorithm. The plots show how W^+ grows with $\mathcal{K}_A$. For fib1, this curve goes from $(0, 0)$ to $(1, 1)$ since if all rules in fib1 have an alternative action then $\mathcal{A}$ for all rules will have the same pair of actions, and the optimized classifier will be the same as $\mathcal{K}''_{\mathrm{aLP}}$ or $\mathcal{K}''_{\mathrm{aBM}}$, i.e., $W^+ = 1$. A similar situation occurs with fib4 for $l_A = 2$ since 88% of its rules are covered by only two actions, so almost all rules contain these two most frequent actions. Values of W and W^+ are much smaller for fib2 since the distribution of actions in fib2 is more uniform. Note how W and W^+ grow in fib2 as l_A increases; e.g., for $p_A = 0.5$ W is 2.5 times higher for $l_A = 3$ than for $l_A = 1$.

B. General classifier experiments

Heuristics. The same aBM heuristic as in Section VII-A is used to reduce size of general ternary approximate classifiers.

Methodology. To evaluate our methods on general classifiers, we used 4 classifiers generated by ClassBench [11]. Filters of generated rules have width 112 bits. For every classifier, we have removed all rules with at least 69 bits with value $*$ since such short filters cover many more rules and if we leave them in, the resulting size will mostly depend on what common actions these general rules have with more specific rules. For each of the 4 classifiers, we have considered three

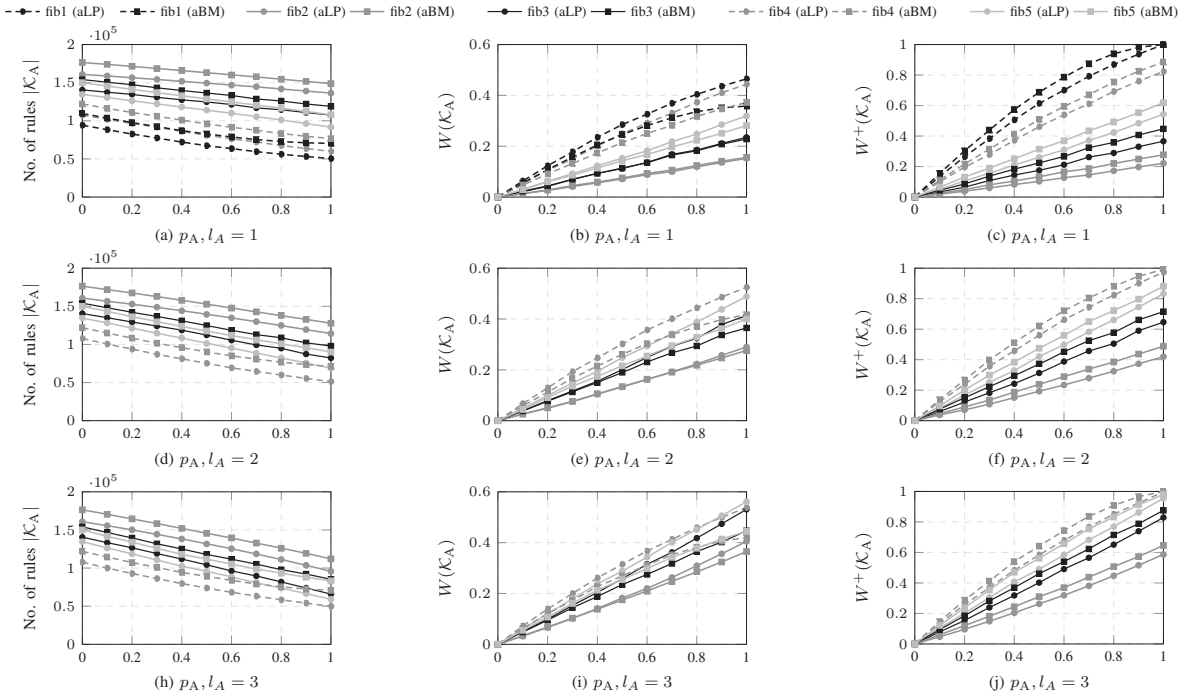


Fig. 4. FIB classifier optimization; columns show the number of rules, $W(\mathcal{K}_A)$, and $W^+(\mathcal{K}_A)$ metrics as functions of the p_A fraction of rules with: (a-c) one additional alternative action, $l_A = 1$; (d-f) $l_A = 2$; (h-j) $l_A = 3$.

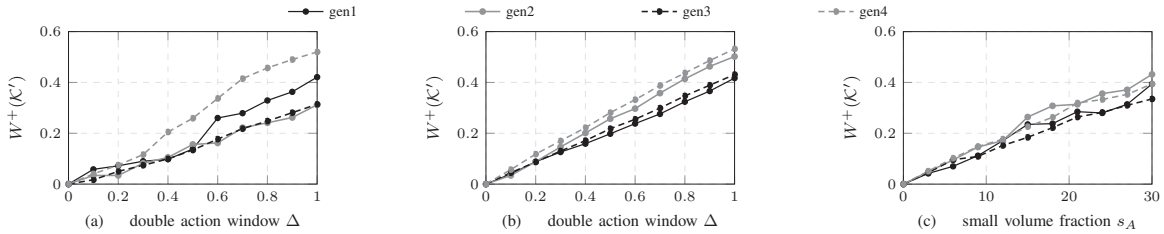


Fig. 5. General classifier optimization: $W^+(\mathcal{K}')$ as a function of (a) window size Δ for the same rule priority and action ordering; (b) window size Δ for different rule priority and action orderings; (c) fraction of "small volume" rules s_A.

kinds of instances of approximate classifiers that differ in the rules for constructing the $\mathcal{A}$ sets.

Action sets from monotone action values. The first type of approximate classifiers is constructed similarly to the motivational example from Fig. 1. We assume that the classifier defines a value of latency that decreases with the rule's priority. The first third of the rules corresponds to the Gold service class; the second third, Silver; the last third, Bronze. Action set $\mathcal{A}$ of each rule contains the corresponding action. Parameter Δ reflects the size of the window where rules have two possible actions: for a rule $R_i \in \mathcal{K}$ with $(\frac{1}{3} - \frac{\Delta}{6}) \cdot |\mathcal{K}| \leq i \leq (\frac{1}{3} - \frac{\Delta}{6}) \cdot |\mathcal{K}|$, we set $\mathcal{A}_i = \{\text{Gold}, \text{Silver}\}$; for $(\frac{2}{3} - \frac{\Delta}{6}) \cdot |\mathcal{K}| \leq i \leq (\frac{2}{3} - \frac{\Delta}{6}) \cdot |\mathcal{K}|$, $\mathcal{A}_i = \{\text{Silver}, \text{Bronze}\}$. Figure 5(a) shows the values of W^+ for the optimized classifiers with Δ varying from 0 to 1. Note how, as Δ increases, the value of W^+ increases as well and may reach 0.5 (for gen4), i.e., the number of rules deleted by optimization in the approximate case is comparable with the number of rules that could be deleted if all $\mathcal{A}$ contained all three actions.

Action sets from reordered monotone action values. The second type is the same as the first except that the order of decreasing latency now differs from the order of decreasing rule priority. We randomly shuffle the rules and then generate $\mathcal{A}$ sets in the same way as in the first type. Then we optimize the classifier with $\mathcal{A}$ labels generated by the new order and rule priorities in the original order. In this case, W^+ for reduced classifiers is shown on Fig. 5(b); similar to the previous case, it can also reach 0.5 (for gen4 and gen2).

Full action sets for small traffic volumes. In the third type of classifiers, we still assume that the first third of the rules corresponds to the Gold service class, second to Silver, and last to Bronze. But now we assume that fraction s_A of the rules have a very small volume, so the quality of service can be arbitrary and their $\mathcal{A}_i$ contain all three actions. For different s_A from 0 to 0.3 the values of W^+ for reduced classifiers of the third type are shown on Fig. 5(c); note that it reaches 0.4 for $s_A = 0.3$.

General classifiers in this experiment have 50-100 thousand rules, and Boolean minimization techiniques cannot remove more than 6000 rules even when all actions are the same. This means that ClassBench has generated classifiers close to irreducible; that is why we only show the W^+ metric on

Fig. 5. Notice, however, that we are not increasing the error for a large number of rules to reduce the classifier only a little: for rules left untouched by our heuristics (in this case, an overwhelming majority of them) one can easily cut $\mathcal{A}$ back to the original action.

Generally, practical evaluations fully support our basic idea and theoretical results: allowing for a small error in the actions of a classifier can lead to very significant space reductions, often comparable to the largest theoretical reduction in the trivial case when all rules have the same action.

VIII. Related Work

Majority of efficient implementations of packet classifiers in the exact case can be classified into two major categories: *software-based* and TCAM-based; The comprehensive surveys can be found in [12], [13]. Softwared-based solutions mainly rely on decision trees, hashing, or coding-based compression. The works [14], [15] suggest how to partition the multi-dimensional rule space, finding possible matching rules by tracing a path in a decision tree. Techniques to balance the partition in each node exist, but rule replication often is unavoided [16]. There is a fundamental tradeoff between space and time complexities in these approaches. The ABC algorithm for filter distribution offers higher throughput with lower memory overhead [17]. The works [18], [19] discuss hash-based solutions to match a packet to its possible matching rules. Efficient coding-based representations are shown in [20], [21]. Different approaches have been described to reduce number of entries: [22], [23], [24]. They include removing redundancies [2], [3], [4], applying block permutations in the header space [25], transformations [22], [23], [26], [27], [28], [29]. In particular [30], [31], [32] considered representations based on rule disjointness composed with prefix-reorderability to cut down classification width.

IX. Conclusion

In this work, we have generalized the classical *packet classification problem* (exact case), introducing a new abstraction of approximate classifiers, where we control the accuracy by labeling in advance which filters can be assigned to which actions. We have designed optimization methods exploiting this additional flexibility in actions to optimize the classifier size. We believe that approximate classifiers exploit an interesting and unexplored tradeoff between required resources and accuracy of results.

Acknowledgments The work of Vitalii Demianiuk and Kirill Kogan was partially supported by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation. The work of Sergey Nikolenko shown in Section VI has been supported by the Russian Science Foundation grant no. 17-11-01276.

References

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[2] A. Liu, C. Meiners, and E. Torng, "TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs," *Trans. Networking*, vol. 18, no. 2, pp. 490–500, 2010.

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

[4] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *SIGMETRICS*, 2006, pp. 311–322.

[5] C. Umans, "The minimum equivalent DNF problem and shortest implicants," *J. Comput. Syst. Sci.*, vol. 63, no. 4, pp. 597–611, 2001.

[6] S. Khot and R. Saket, "Hardness of minimizing and learning DNF expressions," in *FOCS*, 2008, pp. 231–240.

[7] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill, "Constructing optimal IP routing tables," in *INFOCOM*, 1999, pp. 88–97.

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

[9] A. Elmokashfi and A. Dhamdhere, "Revisiting BGP churn growth," *CCR*, vol. 44, no. 1, pp. 5–12, 2014.

[10] Anonymous, "Code for simulations," 2018. [Online]. Available: https://github.com/ACCAAPPROX/Acca

[11] D. Taylor and J. Turner, "Classbench: A packet classification benchmark," *Trans. Networking*, vol. 15, no. 3, pp. 499–511, June 2007.

[12] D. Taylor, "Survey and taxonomy of packet classification techniques," *Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[13] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2005.

[14] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *Micro*, vol. 20, no. 1, pp. 34–41, 2000.

[15] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput," in *SIGCOMM*, 2010, pp. 207–218.

[16] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *Infocom*, 2010, pp. 848–856.

[17] H. Song and J. Turner, "ABC: Adaptive binary cuttings for multidimensional packet classification," *Trans. Networking*, vol. 21, no. 1, pp. 98–109, 2013.

[18] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast packet classification using Bloom filters," in *ANCS*, 2006, pp. 61–70.

[19] Y. Kanizo, D. Hay, and I. Keslassy, "Optimal fast hashing," in *Infocom*, 2009, pp. 2500–2508.

[20] A. Korsi, J. Tapolcai, B. Mihlka, G. Mszros, and G. Rtvri, "Compressing IP forwarding tables: Realizing information-theoretical space bounds and fast lookups simultaneously," in *ICNP*, 2014, pp. 332–343.

[21] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables for datacenter scalability," *JSAC*, vol. 32, no. 1, pp. 138 – 151, 2014.

[22] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and speed tradeoffs in TCAM hierarchical packet classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.

[23] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan, "Strategies for mitigating TCAM space bottlenecks," in *HOTI*, 2014, pp. 111–121.

[24] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal in/out TCAM encodings of ranges," *Trans. Networking*, vol. 24, no. 1, pp. 555–568, 2016.

[25] R. Wei, Y. Xu, and H. J. Chao, "Block permutations in Boolean space to minimize TCAM for packet classification," in *INFOCOM*, 2012, pp. 2561–2565.

[26] C. Meiners, A. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *Trans. Networking*, vol. 19, no. 1, pp. 237–250, 2011.

[27] P. Chuprikov, K. Kogan, and S. Nikolenko, "General ternary bit strings on commodity longest-prefix-match infrastructures," in *ICNP*, 2017, pp. 1–10.

[28] ——, "How to implement complex policies on existing network infrastructure," in *SOSR*, 2018, pp. 9:1–9:7.

[29] V. Demianiuk, S. I. Nikolenko, P. Chuprikov, and K. Kogan, "New alternatives to optimize policy classifiers," in *ICNP*, 2018, pp. 121–131.

[30] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *Trans. Networking*, vol. 24, no. 2, pp. 1251–1264, 2016.

[31] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich, "FIB efficiency in distributed platforms," in *ICNP*, 2016, pp. 1–10.

[32] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Kovács, and A. Shalimov, "How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes," in *Infocom Workshops*, 2016.

C Paper "Robust Distributed Monitoring of Traffic Flows"

Authors. Vitalii Demianiuk, Sergey Gorinsky, Sergey Nikolenko and Kirill Kogan

Abstract. Scalable monitoring of traffic flows faces challenges posed by unrelenting traffic growth, device heterogeneity, and load unevenness. We explore an approach that tackles these challenges by shifting a portion of the monitoring-task execution from an overloaded network element to another element that has spare resources. Moving the entire execution of the task to a lightly loaded element might be infeasible because execution on multiple elements is inherent in the task or requires at least partial participation by the particular overloaded element (e.g., flow-size computation at the ingress element for billing purposes). Distributed execution of a stateful traffic-monitoring task has to be robust against packet reordering or loss, i.e., network noise. This paper designs robust traffic monitoring where the goal is to determine a flow metric for each flow exactly in spite of network noise. We follow the open-loop paradigm that does not add any control packets, communicates flow state in-band by appending few (on the order of 2 or 4) control bits to packets of the monitored flows, and keeps latency low. We consider the task of flow-size computation, analytically derive conditions assuring correct operation of the designed algorithms, and evaluate the algorithms on realistic traffic traces. The algorithms successfully distribute the monitoring-task load without imposing significant computation or storage overhead.

Robust Distributed Monitoring of Traffic Flows

Vitalii Demianiuk[1,2], Sergey Gorinsky[1], Sergey Nikolenko[2], and Kirill Kogan[1]
[1]IMDEA Networks Institute
[2]Steklov Institute of Mathematics at St. Petersburg
Email: {vitalii.demianiuk, sergey.gorinsky, kirill.kogan}@imdea.org, snikolenko@gmail.com

***Abstract*—Scalable monitoring of traffic flows faces challenges posed by unrelenting traffic growth, device heterogeneity, and load unevenness. We explore an approach that tackles these challenges by shifting a portion of the monitoring-task execution from an overloaded network element to another element that has spare resources. Moving the entire execution of the task to a lightly loaded element might be infeasible because execution on multiple elements is inherent in the task or requires at least partial participation by the particular overloaded element (e.g., flow-size computation at the ingress element for billing purposes). Distributed execution of a stateful traffic-monitoring task has to be robust against packet reordering or loss, i.e., network noise. This paper designs robust traffic monitoring where the goal is to determine a flow metric for each flow exactly in spite of network noise. We follow the open-loop paradigm that does not add any control packets, communicates flow state in-band by appending few (on the order of 2 or 4) control bits to packets of the monitored flows, and keeps latency low. We consider the task of flow-size computation, analytically derive conditions assuring correct operation of the designed algorithms, and evaluate the algorithms on realistic traffic traces. The algorithms successfully distribute the monitoring-task load without imposing significant computation or storage overhead.**

I. Introduction

Monitoring of network traffic is crucial for efficient, reliable, and secure operation of any network [1], [2]. Knowledge of traffic properties helps the network operator in capacity planning, QoS (Quality of Service) assurance, service differentiation, attack mitigation, etc. In some applications, the operator needs to know metrics of traffic flows exactly. For example, a billing application has to know exactly the size of a traffic flow at its ingress element to accurately bill the sender for delivering the traffic flow through the network.

Scalable monitoring of traffic flows is challenging due to unrelenting traffic growth, device heterogeneity, and load unevenness. First, while traffic keeps growing in both volume and number of flows, the processing and storage needed for traffic monitoring in network elements grow as well. Second, networks comprise elements of increasing heterogeneity ranging from basic IoT (Internet of Things) access devices with greatly limited capabilities to high-end core routers that forward millions of concurrent flows. Third, the traffic-monitoring load on different network elements is uneven, and one element might get overloaded even when other elements have spare resources.

978-1-7281-2700-2/19/$31.00 2019 © IEEE

A promising direction for meeting these challenges is to monitor traffic flows by utilizing resources in multiple network elements as a shared pool. If a traffic-monitoring task overloads an element, then – regardless of whether the task runs on this element alone or inherently executes on multiple elements – a part of the task execution can be shifted from the overloaded element to another element that has ample resources. This empowers the network to leverage its global processing and storage resources to effectively cope with local traffic-monitoring overloads.

Shift of a traffic-monitoring load to an element with abundant resources might require both distributed execution of the traffic-monitoring task and communication of flow state between the involved network elements. Moving the entire execution of a task from a particular overloaded element to another element might be infeasible even when the task does not inherently require multiple elements for its execution. For instance, let us revisit the billing application that monitors the sizes of the traffic flows entering the network: the specific ingress element of each flow has to participate in computing the flow size at least to some extent because otherwise packet losses inside the network would render the computation of the flow size inaccurate. Furthermore, when the execution of a stateful task is shifted from the overloaded element only partially, the distributed execution necessitates communication of flow state between the involved elements.

In practice, it is impossible to fully avoid packet reordering or loss, and distributed traffic monitoring has to be robust to such *network noise*. Unlike distribution of static policies [3]–[7], distributed execution of a stateful task needs additional means to acquire such robustness. Open-loop and closed-loop control constitute two general approaches to dealing with network noise. While a closed-loop design can adapt its operation to the current level of network noise, the feedback-driven robustness increases latency, which is undesirable for real-time monitoring. Also, asymmetric routing, restrictions on generation of new packets in a network element, and other factors might make it infeasible – or at least very difficult – to provide feedback to a previous element on the path of a unidirectional traffic flow [8]. Hence, the problem of robust distributed monitoring of traffic flows is more amenable to the open-loop approach that can communicate flow state in-band and keep latency low.

In this paper, we study robust distributed open-loop monitoring of traffic flows where the objective is to compute a

flow metric for all flows exactly despite network noise. The explored approach does not introduce any control packets and communicates flow state by piggybacking few (on the order of 2 or 4) control bits on packets of the monitored flows. Our solution methodology relies on the following two design principles:

1) The elements involved in distributed monitoring manage flow state in chunks that overlap to keep the state representation consistent despite network noise.
2) We partition each flow into groups of consecutive packets so that performance and overhead of designed algorithms can be expressed with respect to the group size.

Our evaluation combines theoretical analysis with experimentation. For a given level of maximum network noise, we analytically characterize conditions under which the proposed algorithms are guaranteed to operate correctly in spite of network noise. We analytically derive conditions assuring correct operation of the designed algorithms and also evaluate the algorithms on realistic traffic traces. The algorithms successfully distribute the monitoring-task load without imposing significant computation or storage overhead.

While our paper focuses on exact reconstruction of flow metrics, a large body of related prominent work explore approximate solutions for scalable traffic monitoring. Even the relatively simple problem of computing the flow sizes for all flows turns greatly challenging when the number of flows becomes large. Although most flows are mice, i.e., have a small size, the lack of a priori knowledge about flow sizes forces a counting element to allocate space for all flows, and the element has insufficient memory on its data path to compute the flow sizes exactly for a large number of flows. Previous approximate solutions for the flow-size computation include estimators [1], [9]–[11] and sketches: CM [12], CU [13], Pyramid Sketch [14], UnivMon [15], and Elastic Sketch [2]. A trace-driven evaluation of CEDAR [1], SAC [11], and DISCO [10] shows their average relative errors in excess of 12% for 8-bit per-flow estimators [1]. Even Elastic Sketch, one of the most advanced current proposals, underestimates the sizes of mice flows by a factor of 4 when using 0.2 MB to represent 110K flows, i.e., around 15 bits per flow [2]. Such accuracy is insufficiently low for billing and other traffic-monitoring applications. For examples, if the network overcharges the sender of a flow by 10%, such inaccuracy is unacceptable in practice. Our paper pioneers an alternative distributed approach: instead of sacrificing the accuracy, we support scalable exact reconstruction of flow metrics by involving additional network elements that have spare resources.

The rest of the paper is organized as follows. Section II formulates the problem of robust traffic monitoring in the context of flow-size computation. Section III discusses implications of network noise. Section IV examines a split of distributed monitoring between two network elements. Section V considers alternative representations of network noise and assesses their design ramifications. Section VI extends the solution for a split of distributed monitoring between any number of network elements on the flow path as well as for multi-path flows. Section VII provides an evaluation study. Section VIII presents related work. Section IX concludes the paper by summing up its contributions.

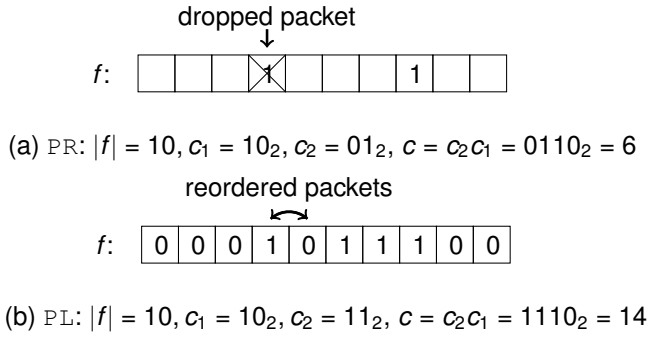


Fig. 1: Vulnerability of the PR and PL approaches respectively to: (a) packet loss and (b) packet reordering.

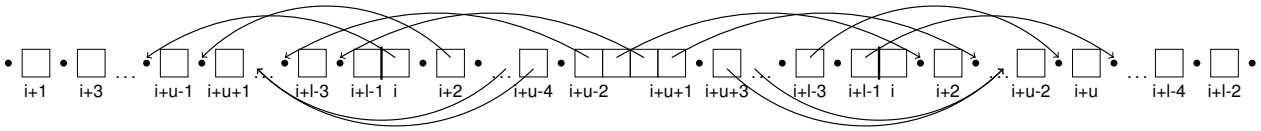
II. Model

We formulate the problem of robust traffic monitoring in the context of computing the flow sizes for all flows. For ease of exposition, this paper interchangeably refers to network elements as switches (even though the formulation is equally applicable to other types of network elements). Flow f enters the network in source switch S and exits in destination switch D. At switch S, the flow consists of $|f|$ packets p_0, p_1, ..., $p_{|f|-2}$, $p_{|f|-1}$. While the objective is to compute flow size $|f|$ for all flows, switch S does not have enough resources to accomplish this task on its own and needs assistance from another switch. Switch i that participates in such distributed flow-size computation maintains counter chunk c_i consisting of n_i bits. Communication of flow state is in-band via packets of the monitored unidirectional flow. Switch i piggybacks at most t_i control bits on each packet. We refer to n_1 and t_1 as simply n and t respectively. Network noise might reorder or drop packets. When the flow terminates, the distributed algorithm should reconstruct $|f|$ exactly. While the flow is in progress, the estimates provided for the flow size by the algorithm should be non-decreasing and may not exceed the actual flow size.

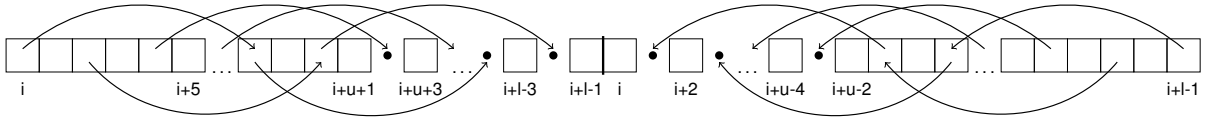
III. Implications of network noise

This section exposes ramifications of network noise for robust flow-size computation that is split between switches S and D. We do this for two extreme cases of network noise: (PR) Packet Reordering without loss and (PL) Packet Loss without reordering.

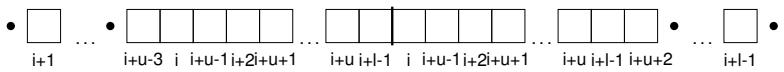
(PR) The following simple solution robustly computes the flow sizes under the PR kind of network noise: the source switch counts packets of the flow and, only when local counter chunk c_1 overflows, marks a control bit in the sent packet;



(a) Alteration of a 3-cycle state sequence by packet reordering and loss for even l and u



(b) Alteration of a 2-cycle state sequence by packet reordering and loss for even l and u



(c) The same packet sequence produced by the above sequence alterations

Fig. 2: State sequences in the proof of theorem 1.

the subsequent participating switch increments its counter chunk c_2 only upon receiving a packet with the marked control bit. This PR solution is fully resilient to packet reordering and greatly vulnerable to packet loss: it might massively underestimate the flow size upon losing a packet with the marked control bit. Figure 1a illustrates this vulnerability when a single packet loss causes the PR method to incorrectly compute the flow size as 6 instead of 10 packets.

(PL) For robust flow-size computation under the PL kind of network noise, the source switch can set a control bit in each sent packet to the most significant bit of local counter chunk c_1 and then increment c_1. The subsequent participating switch increments its counter chunk c_2 only upon receiving a packet where the control bit is set differently than in the previously received packet. This PL solution is resilient to loss of up to $2^{n-1}-1$ consecutive packets, where n is the size of counter chunk c_1. However, the PL method is highly vulnerable to packet reordering, e.g., when packets $2^{n-1}-1$ and 2^{n-1} of the flow arrive to the subsequent participating switch in reverse order, this switch incorrectly increments its counter chunk c_2 thrice instead of once: upon receiving packets 2^{n-1}, $2^{n-1}-1$, and $2^{n-1}+1$. For $n=3$ bits, figure 1b illustrates the resulting overestimation: the PL method incorrectly computes the flow size as 14 instead of 10 packets.

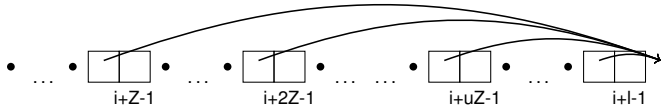
IV. Two-switch split

We now build on the above observations to robustly compute the flow size when network noise comprises both reordering and loss of packets. First, we consider a split of the distributed computation between two switches or, specifically, source and destination of the flow. Note that the flow traverses these two switches regardless of network routing. Resilience of any distributed solution is subject to fundamental feasibility limits, e.g., no such solution is able to handle loss of all packets. We employ two parameters to constrain packet reordering and packet loss: *reordering parameter* R captures the maximal distance of packet reordering, i.e., the destination switch can receive packet p_j before packet p_i only if $j \leq i+R$, and *loss parameter* L is the limit on *consecutive* packet losses, i.e., the destination switch receives at least one packet from any interval $p_i, \ldots, p_{i+L}$. For this $R\&L$ representation of maximum reordering and loss, we narrow down the feasibility limits for distributed flow-size computation as follows:

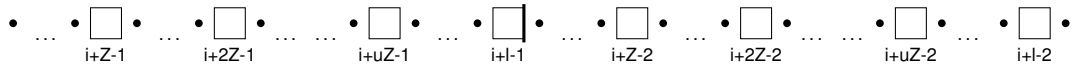
Theorem 1. *With a two-switch split where R and L bound packet reordering and loss respectively, no deterministic algorithm can guarantee correct distributed flow-size computation if $R > 2^{n-1}$ and $L > 0$.*

Proof. Let A be a distributed algorithm that computes the flow size on switches S and D. Algorithm A has n bits to store its counting state x_k for flow f on switch S, including the counter value and any auxiliary information. Upon receiving packet p, switch S updates its counting state to x_{k+1} and may modify p to communicate some state to switch D. We assume that A can record the entire state x_k into packet p. Consider a sequence of consecutive states $x_0, x_1, x_2, \ldots$ of algorithm A. Due to the n-bit representation, there are at most 2^n states. Hence, starting from some state, the counting follows a cycle of length $l \leq 2^n$, i.e., $x_{k+l} = x_k$ for any k.

First, assume that l is divisible by 4 and denote $u = \frac{l}{2}$. Figures 2a and 2b respectively show 3 and 2 cycles of length l in this sequence that are affected by packet reordering and loss. Switch D receives all other packets of the flow without any disruption. We use i to label the packets corresponding to state x_i, denote lost packets as black dots, and display packet reordering with arched arrows. In figure 2a, reordering shifts one half of the second-cycle packets to the first cycle and the other half to the third cycle. In figure 2b, reordering moves even-numbered packets from the first cycle forward by $u-2$ positions and odd-numbered packets from the second cycle backward by $u-2$ positions. These two alterations of two



(a) 1-cycle state sequence (reordering, loss)



(b) 2-cycle state sequence (loss)

Fig. 3: State sequences in the proof of theorem 3.

Algorithm 1 Two-switch computation of flow sizes

```
1: function DIFFERENCE(a,b,t)
2:     return (a + 2^t − b mod 2^t) mod 2^t
3: procedure SOURCEUPDATE(p)
4:     h[p] ← ⌊c_1 / 2^{n−t}⌋
5:     c_1 ← (c_1 + 1) mod 2^n
6: procedure DESTINATIONUPDATE(p)
7:     diff ← Difference(h[p], c_2, t)
8:     If 1 ≤ diff ≤ 2^{t−1} then c_2 ← c_2 + diff
9: procedure TOTALCOUNT(c_1, c_2)
10:    c ← c_2 · 2^{n−t}
11:    c ← c + Difference(c_1, c, n)
```

different sequences produce the same packet sequence shown in figure 2c. Upon receiving this sequence, switch D is unable to distinguish whether it originated from the 3-cycle or 2-cycle state sequence in switch S. Thus, there is no guarantee that algorithm A counts the flow size correctly. Since both alterations of the sequence involve packet reordering with $R = u + 1 \leq 2^{n-1} + 1$ and loss with $L = 1$, the above proof establishes the theorem for even u and l that is divisible by 4. For all other combinations of l and u, the proof remains almost the same. □

While theorem 1 establishes limits on robustness of distributed flow-size computation when R and L bound packet reordering and loss respectively, we now present algorithm 1 that achieves these limits. The algorithm employs *sync bits*, a common t-bit portion of the counter chunks in switches S and D, to synchronize the two counter chunks. The sync bits correspond to the t most significant bits in S's chunk c_1 and t least significant bits in D's chunk c_2, which means that they make up the middle bits in the resulting merged two-chunk counter c that counts up to 2^{n+n_2-t}. When switch S receives packet p, the switch writes the sync bits from c_1 into header $h[p]$ of packet p and increments its n-bit counter chunk c_1. When packet p arrives to switch D, the destination switch uses the DIFFERENCE procedure to compute, subject to modulo 2^t, the difference between $h[p]$ and the t sync bits in counter chunk c_2. If this difference is between 1 and 2^{t-1}, switch D adds it to c_2. To compute $|f|$, the TOTALCOUNT procedure of algorithm 1 sets the n most significant bits of counter c to c_2 and then adds to c the difference between c_1 and the n least significant bits of c.

Theorem 2. *Algorithm 1 correctly computes the flow size up to 2^{2n-t} packets under the following conditions:*

$$L + R < 2^{n-1} \quad \textit{and} \quad R \leq 2^{n-1} - 2^{n-t}. \tag{1}$$

Correctness of theorem 2 directly follows from theorems 4 and 6 which we state and prove later in the paper. Parameter t controls a trade-off between the constraint on R and the size of counter chunk c_2 in destination switch D. Note that constraint $L+R < 2^n - 1$ is independent of t. If $R = 0$, i.e., the network does not reorder packets, algorithm 1 counts the flow size correctly under loss of up to $L < 2^{n-1}$ consecutive packets. When $R > 2^{n-1}$, algorithm 1 offers no assurance of correct operation, which perfectly matches the infeasibility result of theorem 1.

In algorithm 1, counter chunk c_2 never decreases, and product $c_2 \cdot 2^{n-t}$ never exceeds the number of packets received by source switch S. Hence, $c_2 \cdot 2^{n-t}$ serves as a real-time lower bound on the number of packets that switch S has received so far.

Algorithm 1 is highly resilient to network noise in practical settings. For example, by allocating only $n = 8$ bits for the counter chunk in switch S and using $t = 2$ of them as sync bits, algorithm 1 guarantees its correct computation of the flow size under any reordering $R < 64$ packets and $L + R < 128$ packets, which constitute significant levels of network noise. Moreover, doubling the value of t from 2 to 4 bits raises the value of R to 112 packets.

V. IMPACT OF NOISE REPRESENTATION

Whereas section IV shows that robust distributed flow-size computation is feasible even under severe packet reordering and loss, we now study how the network-noise representation affects the feasibility. Specifically, we consider an alternative representation that bounds not only stretch but also frequency of network noise: we partition the flow at switch S into groups of 2^k consecutive packets and introduce $\mathrm{span}(\gamma, k)$ as a packet sequence where each packet is followed by a packet from the same group or γ subsequent groups. Formally, $\mathrm{span}(\gamma, k)$ is defined as a sequence of packets p_{z_i} from flow f with the following properties:

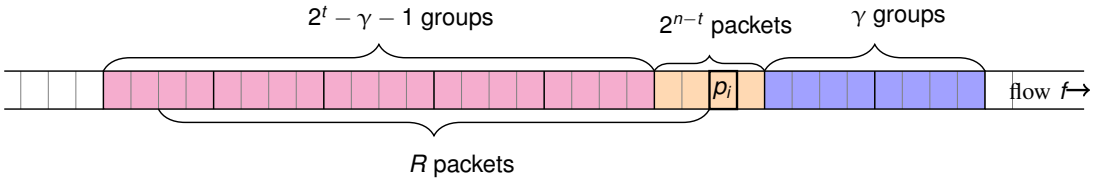


Fig. 4: Packet subsequences of the flow in its source switch S from the proof of correctness for algorithm 2.

(1) $k < n$, $1 \leq \gamma < 2^{n-k}$, and $i = 0, \ldots, r-1$;
(2) indices z_i form an increasing sequence;
(3) switch D receives all packets p_{z_i};
(4) D receives p_{z_i} before p_{z_j} if $i < j$;
(5) $\left\lfloor \frac{z_{i+1}}{2^k} \right\rfloor - \left\lfloor \frac{z_i}{2^k} \right\rfloor \leq \gamma$ for every i, i.e., $p_{z_{i+1}}$ is at most γ groups of 2^k packets away from p_{z_i} at switch S;
(6) $\left\lfloor \frac{z_0}{2^k} \right\rfloor \leq \gamma$ and $\left\lfloor \frac{f}{2^k} \right\rfloor - \left\lfloor \frac{z_{r-1}}{2^k} \right\rfloor < 2^t$.

If $\mathrm{span}(\gamma, k)$ exists, condition 5 ensures that packet reordering and loss cannot affect all $\gamma \cdot 2^k$ packets from γ consecutive groups. With this representation of network noise, the maximum reordering extent is still R, and packet loss is arbitrary as long as switch D still receives $\mathrm{span}(\gamma,\ k)$. Hence, packet reordering and loss may exceed the network-noise limits examined in section IV. Also, parameter γ in this new network-noise representation controls a different trade-off between packet reordering and loss.

Theorem 3. *With a two-switch split where R bounds packet reordering, and packet loss preserves $\mathrm{span}(\gamma,\ n-t)$, no deterministic algorithm can guarantee correct distributed flow-size computation if $R > 2^n - \gamma \cdot 2^{n-t}$.*

Proof. We pursue a similar approach as in the proof of theorem 1 and construct two state-alteration sequences that produce the same packet sequence. For $Z = \gamma \cdot 2^{n-t}$ and $u = \left\lfloor \frac{l}{Z} \right\rfloor$ where $Z \leq l$, figures 3a and 3b respectively show 1 and 2 cycles of length l that are affected by different kinds of network noise. Upon receiving this packet sequence, switch D cannot distinguish whether it originated from the 1-cycle or 2-cycle state sequence in switch S. Thus, no deterministic algorithm provides a guarantee of counting the flow size correctly. Since the two alterations have $R = l - Z + 1 \leq 2^n - \gamma \cdot 2^{n-t} + 1$, keep a gap of at most $Z - 1$ between two consecutive packets in switch D, and thus conserve $\mathrm{span}(\gamma, n-t)$, we establish the theorem. For $l \mod Z = 1$, packets $i + uZ - 1$ and $i + l - 1$ are consecutive, and we have to relocate the latter packet to prove the theorem in this case. □

For $\gamma = 2^{t-1}$, the bound in theorem 3 is $R > 2^{n-1}$, i.e., the same as in theorem 1. Nevertheless, theorem 3 is not a generalization of theorem 1: the latter exacts more restrictive constraints. Algorithm 2 meets the limits stated in theorem 3. This algorithm sets k to $n - t$ and differs from algorithm 1 only in its DESTINATIONUPDATE procedure; the procedure now updates counter chunk c_2 only when switch D receives a packet from the next γ groups.

Algorithm 2 Robust two-switch computation of the flow size under the span constraint

```
1: procedure DESTINATIONUPDATE(p)
2:     diff ← Difference(h[p], c2, t)
3:     If 0 < diff ≤ γ then c2 ← c2 + diff
```

Theorem 4. *Algorithm 2 correctly computes $|f|$ under the following conditions:*

$$R \leq 2^n - (\gamma + 1) \cdot 2^{n-t} \quad \textit{and} \quad \exists\ \mathrm{span}(\gamma, n-t). \tag{2}$$

Proof. We assert that packet p_i received by switch D updates counter chunk c_2 correctly if DESTINATIONUPDATE modifies c_2 to become $\left\lfloor \frac{i}{2^{n-t}} \right\rfloor$. By induction, we show that each update of c_2 is correct. By definition of $\mathrm{span}(\gamma,\ n-t)$, there exists $z_0 \leq \gamma \cdot 2^{n-t}$ such that packet p_{z_0} arrives to switch D. The index of any packet received by D before p_{z_0} does not exceed $z_0 + R \leq \gamma \cdot 2^{n-t} + 2^n - (\gamma+1) \cdot 2^{n-t} < 2^n$, meaning that all such packets are from the first 2^t groups of size 2^{n-t} packets. Then, either these packets update c_2 correctly, or DESTINATIONUPDATE ignores them. Hence, the first update is correct.

For the induction step, we consider the last packet p_i that updates c_2 correctly and show that the next update of c_2 is correct. Since there exists $\mathrm{span}(\gamma, n-t)$, at least one packet from the subsequent γ groups arrives to D after p_i (the blue region in figure 4); we denote by p_j the first such packet. Due to the reordering constraint, any packet p_t received by D before p_j and after p_i is either from the $2^t - (\gamma+1)$ groups that precede the group containing p_i (the pink region in figure 4) or from the $2^t - (\gamma + 1)$ groups succeeding p_j's group. Also, p_t cannot be from the γ groups following p_i's group since p_j is the first such packet. Therefore, DESTINATIONUPDATE ignores all such packets p_t and updates c_2 correctly when switch D receives packet p_j.

To finish the proof, we show that the final update of c_2 is done by a packet from a group with a larger number than $\left\lfloor \frac{|f|}{2^{n-t}} \right\rfloor - 2^t$. We use induction to prove that the following situation does not happen: for some packet p_{z_j} from $\mathrm{span}(\gamma, n-t)$, packet p_i with $i < z_j$ arrives to D after p_{z_j} and updates c_2. Thus, when the last packet $p_{z_{r-1}}$ from this span arrives to D, either $c_2 \geq \left\lfloor \frac{z_{r-1}}{2^{n-t}} \right\rfloor$, or $p_{z_{r-1}}$ updates c_2. □

Theorem 4 exposes a variety of trade-offs. As with our first representation of network noise, a larger t value increases the size of c_2 and raises the maximal tolerated reordering R.

The following theorem demonstrates that under the same reordering constraint, algorithm 2 accommodates less restrictive traffic patterns for a larger value of t.

Theorem 5. *If a flow satisfies conditions 2 in theorem 4 for n, t, and γ, the flow also satisfies conditions 2 for n, $t+1$, and $2\gamma+1$.*

Proof. Because $R \leq 2^n - (\gamma+1)\cdot 2^{n-t} = 2^n - ((2\gamma+1)+1)\cdot 2^{n-t-1}$, an increase of t does not change the packet-reordering constraint. The largest possible difference between the indices of two consecutive packets from $\text{span}(\gamma,\ n-t)$ is less than $(\gamma+1)\cdot 2^{n-t} = ((2\gamma+1)+1)\cdot 2^{n-t-1}$. This implies that these packets form $\text{span}(2\gamma+1,\ n-t-1)$. □

Furthermore, larger values of t provide us with a better leverage over the trade-off between the conditions of theorem 4. Setting $\gamma = 1$ leads to maximal reordering $R = 2^n - 2^{n-t+1}$. With $\gamma = 2^{t-1}$, algorithms 1 and 2 are identical, and the following theorem shows that our first representation of network noise is a special case of the second one.

Theorem 6. *When $\gamma = 2^{t-1}$, conditions 1 imply conditions 2.*

Proof. Since $L+R < 2^{n-1} = \gamma 2^{n-t}$, the sequence of packets that updates the counter chunk in switch D in algorithm 1 forms $\text{span}(2^{t-1},\ n-t)$. □

Settings with $\gamma \geq 2^{t-1}$ are useful in situations where loss is more frequent than reordering. Such settings imply $R < \gamma 2^{n-t}$, i.e., they shift the restricted zone for packet p_j towards later packets. This means that packet loss can be a bigger threat for the existence of $\text{span}(\gamma,\ n-t)$ than packet reordering. When $\text{span}(\gamma, n-t)$ does not exist, algorithm 2 underestimates $|f|$, and we derive a bound on this underestimation:

Theorem 7. *When there is no $\text{span}(\gamma,\ n-t)$, and the total of X packets are lost, algorithm 2 computes $|f|$ with an underestimation of at most $\frac{2^t X}{2\gamma+1-2^t}$ packets under the following conditions:*

$$R \leq 2^n - (\gamma+1)\cdot 2^{n-t} \quad \textit{and} \quad \gamma \geq 2^{t-1}. \tag{3}$$

Proof. Let p_i and p_j be two packets that consecutively update c_2 upon their arrival to switch D. The condition on R enforces $j > i$, implying that algorithm 2 never overestimates $|f|$.

If p_j's group is $a2^t + b$ groups after p_i's group, i.e., $\left\lfloor \frac{j}{2^{n-t}} \right\rfloor - \left\lfloor \frac{i}{2^{n-t}} \right\rfloor = a2^t + b$, switch D adds b to its counter c_2. Then, algorithm 2 can miss up to $a2^n$ packets in total. Conditions 3 imply that at least $a\gamma 2^{n-t}$ out of these $a2^n$ packets are lost or arrive before p_i. At most $2^n - (\gamma+1)2^{n-t}$ packets with larger indices than i can arrive before p_i. Thus, at least $(a-1)\gamma 2^{n-t} + 2\gamma + 1 - 2^t$ packets are lost, and the missed-to-lost packet ratio is at most $\frac{a2^n}{(a-1)\gamma 2^{n-t}+2\gamma+1-2^t}$, which is at most $\frac{2^t}{2\gamma+1-2^t}$. This characterization holds for flow f's entire duration, including before the first update and after the last update of counter chunk c_2. Therefore, algorithm 2 underestimates $|f|$ by at most $\frac{2^t X}{2\gamma+1-2^t}$ packets. □

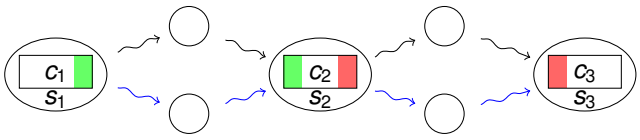


Fig. 5: Split of the size-counting state for flow f between three switches s_1, s_2, and s_3: counter chunks c_1 and c_2 overlap in their green portions; counter chunks c_2 and c_3 overlap in their red portions; packets can flow along either black or blue path segments.

When the constraint on R is satisfied, and there exists no $\text{span}(\gamma,\ n-t)$, algorithm 2 with $t=2$ sync bits and $\gamma = 2$ underestimates the flow size by at most $4X$ packets. With 3 sync bits and $\gamma = 6$, the underestimation is at most by $1.6X$ packets.

When $\gamma = 2^t - 1$, algorithm 2 does not tolerate any reordering and increases its loss resilience to $L < 2^n - 2^{n-t}$. When loss is unbounded, and there is no reordering, algorithm 2 with such values of γ underestimates the flow size by at most $\frac{X}{1-2^{-t}}$ packets.

From a practical perspective, conditions 2 make algorithm 2 more robust to reordering and loss than under conditions 1. For example, when the source-switch chunk contains 8 bits, the settings with 2 and 3 sync bits and $\gamma = 1$ support $R \leq 128$ packets and $R \leq 224$ packets and leniently require D's packet sequence to form $\text{span}(1,\ 64)$ and $\text{span}(1,\ 32)$ respectively. For $n = 8$ bits, $t = 3$ bits, and $\gamma = 6$, this flow-accounting solution calculates $|f|$ correctly when there exists $\text{span}(1,\ 192)$ and $R \leq 32$ packets.

VI. Flow accounting on 3+ switches

While sections IV and V split traffic monitoring between two switches only, we now relax this constraint and suppose that a flow utilizes multiple paths between its source and destination and that all these paths contain a fixed subsequence of switches $S = s_1, s_2, \ldots, s_{m-1}, s_m = D$ that maintains a distributed flow counter.

To characterize the maximum network noise affecting the packet sequence that arrives to switch s_i, we define parameters R_i and γ_i similarly to R and γ respectively. Packet p_j arrives to s_i before packet p_k only if $j \leq k + R_i$, and the packet sequence forms $\text{span}(\gamma_i, N_{i-1} - t_{i-1})$ where $N_i = n_i + \sum_{j=1}^{i-1} n_j - t_j$.

We then develop algorithm 3 for distributed computation of the flow size in such settings. Switch s_i stores the n_i least significant bits of $\left\lfloor \frac{c}{2^{N_i - n_i}} \right\rfloor$ in its counter chunk c_i. The above-defined N_i is a partial joint counter maintained by the first i switches, and 2^{N_m} bounds the size of the total joint counter, i.e., maximal supported flow size.

Algorithm 3 reuses the SourceUpdate and DestinationUpdate procedures of algorithm 2 to update counter chunks c_1 and c_m respectively. The InternalUpdate procedure updates the t_{i-1} least significant bits of each internal

counter chunk c_i using $h[p]$ according to algorithm 2. Then, INTERNALUPDATE stores in packet header $h[p]$ the t_i most significant bits of c_i.

To compute $|f|$, the extended TOTALCOUNT procedure iteratively assembles all the c_i counters. After c_{i-1} is processed, c becomes $|f| \mod 2^{N_{i-1}}$. When the algorithm processes c_i, INTERNALUPDATE sets c equal to $c_i \cdot 2^{N_{i-1}-t_{i-1}}$ and increases this counter to make its N_{i-1} least significant bits equal to c_{prev}.

There is an important difference between the SOURCEUPDATE and INTERNALUPDATE procedures: SOURCEUPDATE first stores the bits in $h[p]$ and then updates counter chunk c_i, while INTERNALUPDATE first updates counter chunk c_i and only then sets the header. This is because invariants for c_1 and another counter chunk c_i with $i > 1$ are different: c_1 stores the n least significant bits of the *next* packet index, while another c_i uses its bits for the index of the last packet received by source switch S.

Theorem 8. *Algorithm 3 correctly computes the flow size up to 2^{N_m} packets when the following conditions hold for every non-source switch s_i with $i > 1$:*

$$\begin{cases} R_i < 2^{N_{i-1}} - (\gamma_i + 1) \cdot 2^{N_{i-1}-t_{i-1}} \text{ and} \\ \exists\ \text{span}(\gamma_i, N_{i-1} - t_{i-1}) \text{ in the } i\text{-th switch.} \end{cases} \quad (4)$$

Proof. To prove that counter updates in switch s_i are correct, we view s_i as the destination switch for a two-switch counter with $R = R_i$, $\gamma = \gamma_i$, $t = t_i$, and the N_{i-1} least significant bits of the two-switch counter are supported by an imaginary source switch that combines counter chunks $c_1, \ldots, c_{i-1}$. We use the proof of theorem 4 as the induction step. The settings differ from theorem 4 in the following detail: while packet header $h[p_j]$ in theorem 4 is constructed from the bits of p_j, header $h[p_j]$ in this proof can correspond to the bits of a packet with a larger index than j. In this case, c_{i-1} is not updated, $h[p]$ does not update c_i, and counting is not affected. After TOTALCOUNT processes c_i, counter c becomes $|f| \mod 2^{N_i}$. By iterating the above two-switch proof technique for each $i > 1$, we prove the theorem. □

Many parameters characterize a split of flow-size computation between m switches. For the n_i-bit counter chunks c_i, we need to specify $t_1, \ldots, t_{m-1}$ and $\gamma_2, \ldots, \gamma_m$. In practice, it makes sense to use $t_i = 2$ for $i > 1$ and $\gamma_i = 2$ for $i > 2$. The most important decision is the choice of t_1 and γ_2 because the most frequent state updates go from S to s_2, and the update frequency declines exponentially on each subsequent hop.

For a numerical illustration, consider a split between 3 switches, where the first two switches maintain 7-bit counter chunks. We use $t_1 = 3$ sync bits, which implies $R_2 + 16 \cdot \gamma_2 \leq 112$ packets. Increasing t_1 does not significantly improves the situation because theorem 3 rules out existence of a deterministic algorithm for $R_2 + 2^{n-t}\gamma_2 > 128$ packets. We set $\gamma_2 = 5$, which leads to $R_2 = 40$ packets and the constraint that at least one out of 5 consecutive sets of 16 packets is not completely changed by packet reordering and loss. This is reasonable in practical settings. For the first non-source switch s_2, even 2 sync bits and $\gamma_3 = 2$ produce $R_3 = 512$ packets and $2^{N_2-t_2}\gamma_3 = 1024$ packets.

Algorithm 3 Computation of the flow size on 3+ switches

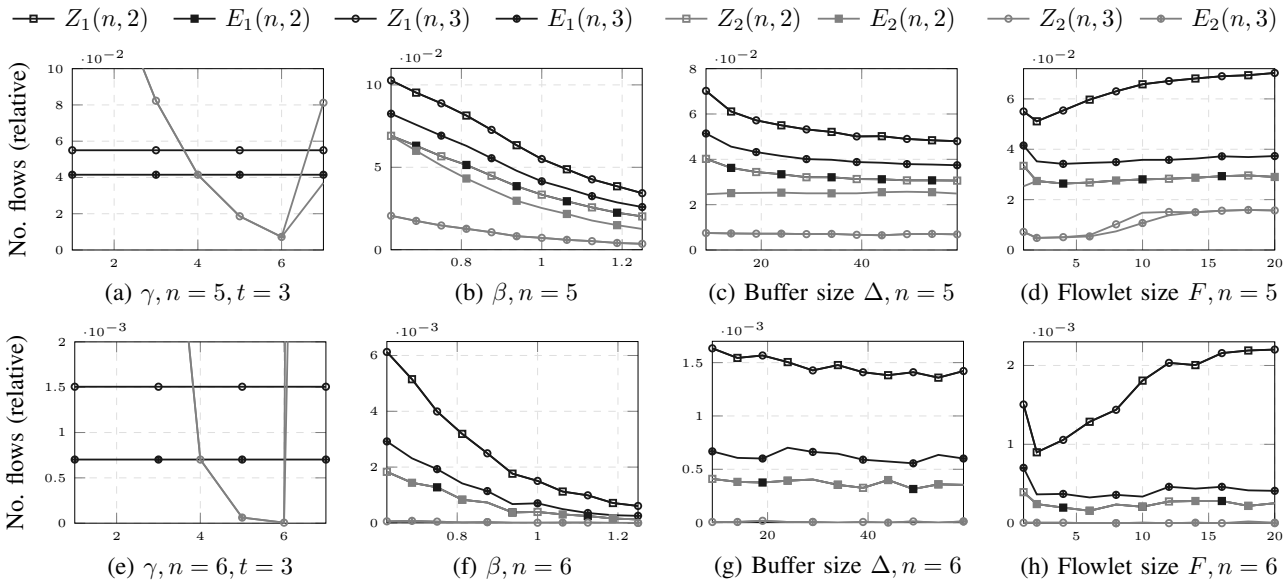
```
1: procedure SOURCEUPDATE(p)
2:     h[p] ← ⌊c_1 / 2^(n_1 − t_1)⌋
3:     c_1 ← (c_1 + 1) mod 2^n
4: procedure INTERNALUPDATE(p)
5:     diff ← Difference(h[p], c_i, t_{i−1})
6:     If 0 < diff ≤ γ_i then c_i ← (c_i + diff) mod 2^{n_i}
7:     h[p] ← ⌊c_i / 2^{n_i − t_i}⌋
8: procedure DESTINATIONUPDATE(p)
9:     diff ← Difference(h[p], c_m, t_{m−1})
10:    If 0 < diff ≤ γ_m then c_m ← c_m + diff
11: procedure TOTALCOUNT(c_1, c_2, ..., c_m)
12:    For i ← 2 to m with step 1
13:        c_prev ← c
14:        c ← c_i · 2^{N_{i−1} − t_{i−1}}
15:        c ← (c + Difference(c_prev, c, N_{i−1})) mod 2^{N_i}
```

This example shows that communication quality on the first hop is the most important, whereas subsequent switches are amenable to more economical use of its resources because they typically do not face substantial network noise. This property holds in general, implying that the source switch should get the largest counter chunk, and the counter chunks on further switches can be smaller. For example, if we increase S' chunk to 9 bits and decrease s_2's chunk to 5 bits, the network-noise constraints for switch s_3 remain the same but the network-noise constraints on the first hop are significantly and usefully relaxed to $R_2 + \gamma_2 \cdot 64 \leq 448$ packets.

VII. EXPERIMENTAL EVALUATION

Methodology. Our evaluation follows the same approach as those for VL2 [16], pFabric [17], and pHost [18]. Specifically, we perform simulations driven by realistic traffic traces generated from the data-mining distribution of flow sizes [16], where the number of flows is 10^6, and the maximum possible flow size of $\frac{2}{3} \cdot 10^6$ packets, which requires a 20-bit counter. We utilize the YAPS packet simulator in its unreliable transport configuration [19]. The network has a two-tier multi-rooted tree topology where four switches constitute the root. 90% of all flows traverse three internal switches on their way from the source to the destination. By default, YAPS sprays packets of each flow by probabilistically sending the packets to different internal switches in accordance with a chosen load-balancing strategy. The packet spraying causes packet reordering. The simulations examine the task of exact flow-size computation and have their source code publicly available [20].

We repeat the simulations for different congestion levels by varying parameter β that scales the expected time between the transmissions of two consecutive packets from the same source. Smaller values of β lead to larger congestion. We also evaluate different values of buffer size Δ in the switches



(a) $\gamma, n = 5, t = 3$ (b) $\beta, n = 5$ (c) Buffer size $\Delta, n = 5$ (d) Flowlet size $F, n = 5$

(e) $\gamma, n = 6, t = 3$ (f) $\beta, n = 6$ (g) Buffer size $\Delta, n = 6$ (h) Flowlet size $F, n = 6$

Fig. 6: Numbers Z_1 and Z_2 of flows violating conditions 1 and 2 respectively, and numbers E_1 and E_2 of flows with sizes calculated incorrectly by algorithms 1 and 2 respectively, as functions of γ, β, buffer size Δ, and flowlet size F; the source counter chunks are sized to: (a)-(d) $n = 5$ bits and (e)-(f) $n = 6$ bits.

along the flow paths. Flowlet size F parameterizes the packet-spraying strategy: for every group of F consecutive packets, the internal switch is chosen according to the round-robin strategy. Our *standard experiment* uses the following default parameter values: $\beta = 1$, $\Delta = 24$ packets, and $F = 1$ packet.

Number of bits in the source switch. To understand the size required for the counter chunks in source switches, we experiment with different values of β, Δ, and F. In these experiments, 7 bits for the source-counter chunk are sufficient, reducing the source counter-chunk size by 65% and 78% for 20-bit and 32-bit counters respectively. The experiments track the following metrics:

1) $Z_1(n,t)$ is the number of flows that violate conditions 1 with n-bit source counter chunks and t sync bits;
2) $E_1(n,t)$ refers to the number of flows for which algorithm 1 computes the flow size incorrectly;
3) $Z_2(n,t)$ denotes the number of flows that violate conditions 2;
4) $E_2(n,t)$ is the number of flows for which algorithm 2 miscalculates the flow size.

These metrics are normalized to the number of flows sized to at least 2^n packets. For other flows, the counter fits in the source switch entirely. Since 7-bit counter chunks on source switches are already sufficient, we compute the metrics for n of 5 and 6 bits. The number of sync bits t is 2 or 3. In the experiments where γ is not specified explicitly, we choose such a γ value that leads to the minimal value of the corresponding metric.

Dependency on γ. Figures 6a and 6e show how Z_2 and E_2 depend on γ for $t = 3$ in the standard experiment. The effect of packet loss is more pronounced than packet reordering, and Z_2 and E_2 decrease when γ increases up to 6. For $\gamma = 2^{t-1} = 7$, Z_2 and E_2 increase drastically because any packet reordering violates conditions 2. For $\gamma = 7$, we have $Z_2 > E_2$ since conditions 2 are usually violated by a packet reordering, which does not necessarily lead to an invalid counter value. For $\gamma \leq 6$, we have $Z_2 = E_2$ because conditions 2 are violated only by absence of $\mathrm{span}(\gamma, n - t)$, which always results in incorrect counting by algorithm 2.

For $n = 5$ and optimal γ, we have $Z_2(5,3) = E_2(5,3) = 7 \cdot 10^{-3}$, meaning that it is sufficient to maintain a 5-bit source-counter chunk for 99.3% of flows with at least 32 packets. For $n = 6$ and optimal γ, the outcome improves further as only one flow does not satisfy conditions 2.

In figures 6a and 6e, Z_1 and E_1 appear as horizontal lines because they do not depend on γ. For $n = 5$, we have $Z_1(5,3) = 5.5 \cdot 10^{-2}$ and $E_1(5,3) = 4.2 \cdot 10^{-2}$, i.e., 94.5% of the flows with at least 32 packets satisfy conditions 1, and algorithm 1 computes the flow sizes correctly for 95.8% of the flows. Hence, the number of flows for which algorithm 2 computes sizes incorrectly is 6+ times lower than for algorithm 1. For $n = 6$, this gap widens further as algorithm 1 errs for few hundreds of flows while algorithm 2 fails for only one flow. The difference observed for $Z_1(5,3)$ vs. $Z_2(5,3)$ with $\gamma = 4$ corroborates theorem 6 by showing that our second representation of network noise is more expressive than the first one: only 76% of the flows violating conditions 1 also violate conditions 2.

Dependency on β**.** Figures 6b and 6f exhibit how the examined metrics depend on β for buffer size $\Delta = 24$ packets and flowlet size $F = 1$ packet. As β increases, network congestion and all the metrics decrease. Specifically, as β grows to 1.25, $Z_2(5,2)$ decreases by 40%, $Z_2(5,3)$ plummets by a factor of two, $E_2(5,2)$ drops by 50%, and $Z_2(6,2)$ and

$E_2(6,2)$ lower by 65%. For each evaluated setting of β, the number of flows that violate conditions 2 for $n = 6$ and $t = 3$ does not exceed 13. With $\beta \geq 1.1875$, the number of such flows is strictly zero, implying that 6-bit source-counter chunks are sufficient for all flows.

For any β value, equalities $Z_2(6,3) = E_2(6,3)$, $Z_2(6,2) = E_2(6,2)$, and $Z_2(5,3) = E_2(5,3)$ hold, and $\gamma = 2^t - 2$ is optimal for these metrics. This is because traffic patterns that violate existence of $\mathrm{span}(\gamma, n-t)$ in conditions 2 always lead to incorrect flow-size computation by algorithm 2. $E_1(5,2) = Z_2(5,2)$ and $E_1(6,2) = Z_2(6,2)$ arise because algorithms 1 and 2 are identical for $\gamma = 2^{t-1}$, which is the optimal γ value for $Z_2(5,2)$ and $Z_2(6,2)$.

On the other hand, $E_2(5,2) < Z_2(5,2)$ since the optimal γ for $E_2(5,2)$ is 3 rather than 2. Under any packet reordering with this γ value, all flows violate the packet-reordering constraint in conditions 2, which does not necessarily cause incorrect computation of flow sizes by algorithm 2. Also, as t increases, the number of flows that violate conditions 1 does not change since the packet-reordering constraint holds for all flows even for $t = 2$, and the constraint on $L + R$ does not depend on t.

By comparing $E_1(5,2)$ and $E_1(5,3)$, we can see advantages provided by our second representation of network noise. Although conditions 1 loosen as t grows, the number of flows for which algorithm 1 computes incorrect sizes increases as t steps up from 2 to 3, i.e., $E_1(5,3) > E_1(5,2)$. This happens because flows that satisfy conditions 2 for $\gamma = 2^{t-1}$ and violate conditions 1 may violate conditions 2 for $t+1$ sync bits and $\gamma = 2^t$. According to theorem 5, all such flows satisfy conditions 2 for $t+1$ sync bits when γ is $2^t + 1$.

Dependency on buffer size Δ. Figures 6c and 6g plot the dependencies of the examined metrics on buffer size Δ for $\beta = 1$ and $F = 1$ packet. As the buffer size decreases, all the metrics rise but at lower rates than in response to changes in β. For $\Delta = 9$, $Z_1(5,2)$ increases by 27%, $Z_2(5,3)$ and $E_2(5,3)$ grow by less than 1.5%, and $Z_1(6,2)$ rises by 15%. For each evaluated β value, the number of flows that violate conditions 2 with $n = 6$ and $t = 3$ is smaller than 4.

In general, the plots in figure 6c are much smoother than in figure 6g because the absolute metric values are much higher for $n = 5$ bits, and the randomness in the number of flows that violate the corresponding conditions affects the curves less. A similar observation holds for the plots in figures 6b and 6d vs. figures 6f and 6h.

Dependence on flowlet size F. Figures 6d and 6h reveal effects of flowlet size F on the assessed metrics when β and Δ are 1 and 24 respectively. As F grows from 2 to 20 packets, the metrics increase, reflecting the increased network noise. When F reaches 20 packets, $Z_2(5,3)$ and $E_2(5,3)$ rise by more than thrice, $Z_1(6,2)$ increases by about 2.5 times, and $Z_2(5,2)$ and $E_2(5,2)$ grow by only 5%. However, as F steps down from 2 packets to 1, the metrics surge abruptly, and $Z_2(5,2)$ and $E_2(5,2)$ reach their maximum values with $F = 1$ packet. This effect can be due to the increased probability of traffic loss when packets of the same flow follow a single path rather than multiple.

Also, as F increases, the impact of packet reordering becomes more perceptible. Unlike what we have seen before, $\gamma = 2$ with $F \geq 2$ is also optimal for $Z_2(5,2)$ and $E_2(5,2)$ because packet reordering becomes so common that algorithm 2 with parameters $n = 5$, $t = 2$, and $\gamma = 3$ starts to compute incorrect sizes for many flows. For the same reason, $Z_2(5,3)$ and $E_2(5,3)$ begin to grow quickly with $F = 8$ packets and have the optimal γ value of 5 (rather than 6) with $F = 14$ packets.

Accuracy of algorithm 2. When algorithm 2 computes the size of flow f incorrectly because $\mathrm{span}(\gamma, n-t)$ does not exist, theorem 7 states that the computed flow size with $\gamma \geq 2^{t-1}$ lies in interval $[|f| - \frac{2^t X}{2\gamma+1-2^t}, |f|]$, where X is the number of lost packets. In the standard experiment with n of 5 or 6, t of 2 or 3, and $2^{t-1} \leq \gamma < 2^t - 1$, algorithm 2 can err only due to lack of $\mathrm{span}(\gamma, n-t)$. In such cases, the computed flow size for any flow belongs to interval $[|f| - X, |f|]$ with $n = 6$ and falls outside interval $[|f| - X, |f|]$ for at most 9 flows with $n = 5$. For $\gamma = 2^{t-1}$, algorithm 2 fails mostly due to excessive packet reordering and overestimates the flow size. With n of 5 or 6, t of 2 or 3, and $\gamma = 2^{t-1}$, the overestimation for 98% of all such flows is less than 2%.

The above evaluation suggests that the proposed distributed execution of traffic-monitoring tasks can significantly help in effective utilization of resources available in a network. In practice, a 7-bit (or even a 6-bit) per-flow source-counter chunk is sufficient, compared to the 32-bit counters used for exact computation of flow sizes in a single element. Our assessment under various settings of n, γ and t also shows that even in the case of 5-bit source-counter chunks, the distributed method correctly computes flow sizes for 99.3% of the flows that contain at least 2^5 packets. Besides, when algorithm 2 computes the size of flow f imprecisely, the computed flow size almost always lies in interval $[|f| - X, 1.02 \cdot |f|]$, where X is a number of lost packets. Finally, the empirical evaluation confirms the analytical advantages of our second representation of network noise.

VIII. Related work

Flow-size computation in a single switch. A long line of research deals with efficient state representation for flow-size computation in a single network element. To utilize SRAM memory effectively, [21]–[23] propose hybrid SRAM/DRAM counting architectures. These methods allocate in SRAM a small counter only for frequent updates and maintain the entire counter in slower but significantly bigger DRAM memory. In contrast, our approach does not require a big pool of additional cheaper memory and distributes the computation to leverage resources available elsewhere in the network.

SAC [11], DISCO [10], and CEDAR [1] calculate an approximate number of per-flow packets by probabilistically incrementing a counter, which allows reducing the number of counter bits for long flows. In SAC, the counter is split between the exponent and estimation parts. To increment a counter, SAC probabilistically increments the estimation part,

and the increment probability depends on the exponent part. When the estimation part overflows, SAC increments the exponent. In DISCO and CEDAR, the entire counter corresponds to a counter value. For n-bit counters, CEDAR constructs variables as an array mapping values of a counter variable to real counter values, the increment probability is inversely proportional to the difference between two corresponding consecutive values in this array. Among all possible arrays, CEDAR finds one that minimizes the relative error.

Methods such as CounterBraids [24] and CounterTree [25] store counters for all flows in hash-based data structures. The idea of CounterBraids is based on sparse random graph codes. The scheme maintains all variables in a tree-like architecture where leaves correspond to less significant bits of the counter variable, and internal nodes correspond to most significant bits.

Flow-size computation is frequently tackled by sketch-based solutions. One of the first such solution is CounterMin (CM) [12]. A CM sketch is a table with r rows and w columns, where each row has a corresponding hash function mapping a flow to a cell that stores a corresponding variable. For an arriving packet, CM increments values of the corresponding variables in all rows. To estimate a counter, CM takes the minimum of the corresponding variables among all rows. Pyramid Sketch [14] combines the ideas of the CounterTree and CM sketches, reducing the number of bits in each cell of the sketch table. UnivMon [15] exploits a sketch hierarchy for different measurement tasks, such as heavy-hitter detection or moment estimation. Elastic Sketch [2] separates mice and elephant flows: mice flows are stored in a CM sketch, and elephant flows are stored in a hash table. Elastic Sketch uses the Ostracism principle to move counter variables between the CM sketch and hash table.

Network-wide flow-size computation. Focusing on the objective of minimizing the communication complexity, [26] detects network-wide heavy hitters in a model where switches report their local counters to a coordinator. FlowRadar [27] maintains a small efficient hash-based data structure in each switch to support storage of encoded flow information, including counters, and a controller can leverage its network-wide view on these data structures to decode the flow information precisely. The given paper extends our preliminary ideas reported in [28].

Distributed flow state that changes routing. DIFANE [29] and vCRIB [30] exploit switches in the network to enforce endpoint ties. They both route traffic through intermediate switches, deviating from the routing policy given by the users.

Distributed flow state that obeys routing. Distribution of static policy state over the flow path is already considered in [3]–[5]. These schemes are static and hence immune to network noise.

Telemetry. Telemetry of various characteristics such as real-time packet loss inherently involves distributed flow state based on the number of packets in the flow source and destination. Prior telemetry solutions [31], [32] require an additional out-of-band communication channel, depend on the quality of time synchronization, and do not provide any analytical robustness guarantees. One can address these drawbacks by extending and applying our approach to real-time telemetry.

Relation to CRDT. Conflict-free replicated data type (CRDT) supports replicas in multiple network nodes without coordination and concurrency on updates [33], resolving inconsistencies by its mathematical properties. Since state-based CRDT (CvRDT) functions that merge states from the replicas must be commutative, associative, and idempotent, CvRDT is stable to reorderings in the sequence of update operations. A flow counter in S and D can be interpreted as a special case of a vector grow-only counter CvRDT [33], but our framework requires only a partial counter at S and D; unlike CRDT, network constraints now have to be incorporated into feasibility analysis.

IX. Conclusion

This paper pioneered a distributed approach to improving scalability of per-flow traffic monitoring. Instead of sacrificing the accuracy of traffic monitoring, the approach enabled scalable exact reconstruction of flow metrics by involving network elements that had spare resources. To make distributed execution of a stateful traffic-monitoring task robust against network noise, we adhered to the open-loop paradigm that introduced no extra packets, communicated flow state in-band by piggybacking few control bits on packets of the monitored flows, and kept latency low. Our methodology relied on two main design principles: use of state overlap in different elements to consistently represent distributed state despite network noise, and partitioning of each flow into groups of consecutive packets to express performance and overhead of designed algorithms in regard to the group size. We applied these design principles for robust exact computation of flow sizes for all flows. We analytically established conditions guaranteeing correct operation of the designed algorithms and complemented the analysis with simulations driven by realistic traffic traces. Our evaluation suggested that the distributed execution of traffic-monitoring tasks could successfully balance the monitoring load on the network without imposing significant storage or computation overhead. In the future work we plan to study applicability of the proposed design principles for telemetry tasks.

Acknowledgments

This work was made possible by grants from various organizations. The Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation, supported Vitalii Demianiuk and Kirill Kogan. The Regional Government of Madrid enabled the research by Sergey Gorinsky and Kirill Kogan with grant P2018/TCS-4499, EdgeData-CM. The Russian Science Foundation provided grant no. 17-11-01276 to support the contributions by Sergey Nikolenko and, in particular, Theorems 1, 3, 5 and Section VI. We also thank the anonymous reviewers for their detailed comments and insightful suggestions.

REFERENCES

[1] E. Tsidon, I. Hanniel, and I. Keslassy, "Estimators Also Need Shared Values to Grow Together." in *INFOCOM*, 2012, pp. 1889–1897.

[2] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *SIGCOMM*, 2018, pp. 561–575.

[3] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One Big Switch" Abstraction in Software-Defined Networks," in *CoNEXT*, 2013, pp. 13–24.

[4] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing Tables in Software-Defined Networks," in *INFOCOM*, 2013, pp. 545–549.

[5] P. Chuprikov, K. Kogan, and S. Nikolenko, "How to Implement Complex Policies on Existing Network Infrastructure," in *SOSR*, 2018, pp. 9:1–9:7.

[6] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and Speed Tradeoffs in TCAM Hierarchical Packet Classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.

[7] K. Kogan, S. I. Nikolenko, P. T. Eugster, and E. Ruan, "Strategies for Mitigating TCAM Space Bottlenecks," in *HOTI*, 2014, pp. 25–32.

[8] S. Gorinsky, S. Jain, H. M. Vin, and Y. Zhang, "Design of Multicast Protocols Robust Against Inflated Subscription," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 249–262, 2006.

[9] A. Cvetkovski, "An Algorithm for Approximate Counting Using Limited Memory Resources," in *SIGMETRICS*, 2007, pp. 181–190.

[10] C. Hu, B. Liu, H. Zhao, K. Chen, Y. Chen, Y. Cheng, and H. Wu, "Discount Counting for Fast Flow Statistics on Flow Size and Flow Volume," *IEEE/ACM Trans. Netw.*, vol. 22, no. 3, pp. 970–981, Jun. 2014.

[11] R. Stanojevic, "Small Active Counters," in *INFOCOM*, 2007, pp. 2153–2161.

[12] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[13] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, Aug. 2003.

[14] Y. Tong, Z. Yang, J. Hao, C. Shigang, and L. Xiaoming, "Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1442–1453, Aug. 2017.

[15] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *SIGCOMM*, 2016, pp. 101–114.

[16] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *SIGCOMM*, 2009, pp. 51–62.

[17] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-Optimal Datacenter Transport," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, 2013.

[18] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "pHost: Distributed Near-Optimal Datacenter Transport over Commodity Network Fabric," in *CoNEXT*, 2015, pp. 1:1–1:12.

[19] "YAPS: Yet Another Packet Simulator," http://wiki.github.com/NetSys/simulator/.

[20] "Robust Distributed Monitoring of Traffic Flows." https://github.com/RobustCounter/RoubstMonitoring.

[21] D. Shah, S. Iyer, B. Prabhakar, and N. McKeown, "Analysis of a Statistics Counter Architecture," in *HOTI*, 2001, pp. 107–111.

[22] ——, "Maintaining Statistics Counters in Router Line Cards," *IEEE Micro*, vol. 22, no. 1, pp. 76–81, 2002.

[23] Q. Zhao, J. J. Xu, and Z. Liu, "Design of a Novel Statistics Counter Architecture with Optimal Space and Time Efficiency," in *SIGMETRICS/Performance*, 2006, pp. 323–334.

[24] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," in *SIGMETRICS*, 2008, pp. 121–132.

[25] M. Chen, S. Chen, and Z. Cai, "Counter Tree: A Scalable Counter Architecture for Per-Flow Traffic Measurement," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1249–1262, April 2017.

[26] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-Wide Heavy Hitter Detection with Commodity Switches," in *SOSR*, 2018, pp. 8:1–8:7.

[27] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," in *NSDI*, 2016, pp. 311–324.

[28] V. Demianiuk, S. Gorinsky, S. I. Nikolenko, and K. Kogan, "Distributed counting along lossy paths without feedback," in *SIROCCO*, 2018, pp. 30–33.

[29] M. Yu, J. Rexford, M. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," in *SIGCOMM*, 2010, pp. 351–362.

[30] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "vCRIB: Virtualized Rule Management in the Cloud," in *HotCloud*, 2012.

[31] G. Fioccola, A. Capello, M. Cociglio, L. Castaldelli, M. Chen, L. Zheng, G. Mirsky, and T. Mizrahi, "Alternate-Marking Method for Passive and Hybrid Performance Monitoring," RFC 8321, January 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8321

[32] T. Mizrahi, C. Arad, G. Fioccola, M. Cociglio, M. Chen, L. Zheng, and G. Mirsky, "Compact Alternate Marking Methods for Passive and Hybrid Performance Monitoring," IETF, October 2018. [Online]. Available: https://tools.ietf.org/html/draft-mizrahi-ippm-compact-alternate-marking-03

[33] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

D Paper “How to deal with range-based packet classifiers”

Authors. Vitalii Demianiuk and Kirill Kogan

Efficient representations of multi-field packet classifiers with fields represented by ranges is a core mechanism to express services on data plane. To implement classifiers in ternary- addressable memory (TCAM), each range should be encoded into multiple ternary bit strings whose number is at most lin- ear to the width (in bits) of a represented field independently from range encoding method. In this paper we introduce a notion of a subrange allowing to represent a field range on any chosen subset of bit indices that significantly improve efficiency of classifier representations. Our analytic results are confirmed with a comprehensive evaluation study show- ing applicability of our approach to implement desired levels of expressiveness and scalability in packet classifiers.

How to deal with range-based packet classifiers

Vitalii Demianiuk
IMDEA Networks Institute
Steklov Institute of Mathematics at St. Petersburg

Kirill Kogan
IMDEA Networks Institute

ABSTRACT

Efficient representations of multi-field packet classifiers with fields represented by ranges is a core mechanism to express services on data plane. To implement classifiers in *ternary-addressable* memory (TCAM), each range should be encoded into multiple ternary bit strings whose number is at most linear to the width (in bits) of a represented field independently from range encoding method. In this paper we introduce a notion of a *subrange* allowing to represent a field range on any chosen subset of bit indices that significantly improve efficiency of classifier representations. Our analytic results are confirmed with a comprehensive evaluation study showing applicability of our approach to implement desired levels of expressiveness and scalability in packet classifiers.

CCS CONCEPTS

• **Networks** → **Packet classification**; *Network algorithms*;

1 INTRODUCTION AND MOTIVATION

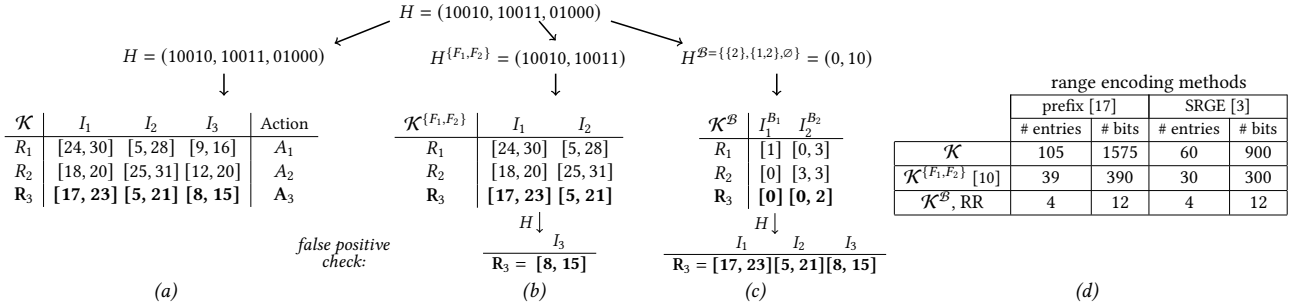
Packet classification is a core building block implementing packet processing programs on data plane. With the adoption of OpenFlow [14] and P4 [15], packet classification has become even more prominent. Each *packet classifier* is an ordered set of rules, where a *rule* consists of the filter (matching packet headers) and the associated action to be applied on matched packets. A *filter* is a concatenation of field representations participating during classification; in the simplest form each field in a filter is represented by exact values. In this case packet classifiers can be implemented with a constant lookup time but the number of rules can be infeasible to fit in memory. In the other extreme case, to address the scalability constraint, fields can be represented by ranges of values. In this case the number of rules is significantly reduced but the lookup complexity increases. As a result, some intermediate forms of field representations were introduced as *prefixes* or more general ternary bit strings, where every bit has three values: zero, one, or don't care. For ternary bit strings, a specialized *ternary content addressable* (TCAM) memory was introduced that is actually a coprocessor running multiple searches in parallel [12].

Related work. There are two major directions dealing with range-base packet classifiers. The first one is independent from structural properties of classifiers encoding every field range by multiple prefixes or ternary bit strings whose number is at most linear to the field size (in bits) [3, 4, 11, 16, 17]. Hence, each rule whose fields are represented by ranges is actually a classifier with exponential number of rules based on ternary bit strings. Recently, [1] proposes efficient encoding for the special case of short ranges. Other range encoding methods exploiting structural properties can achieve more compact representations [2, 5] but usually they perform well only when the number of encoded ranges is relatively small. Note that both these lines of research consider transformations to *equivalent* classifiers.

Recently, [6, 9, 10, 13] proposed representations of packet classifiers exploiting their structural properties like *rule disjointness*; observe that these representations become equivalent to originally given classifiers only when the lookup table based on a subset of fields is complemented by the false-positive check on a single matched rule. This allows to balance which fields are required to implement desired structural properties in the lookup table and which are going to the false-positive check where range encoding is unnecessary. It allows to reduce the size of classifier representations. To store classifier in TCAM, a participating subset of ranges in the lookup table should be encoded by one of the previously mentioned methods [3, 4, 11, 16, 17].

Our contributions. In this paper we are going beyond [10] and propose *range reduction* (RR) methods allowing to implement structural properties of classifiers with per-bit resolution (and not with per-field as in [10]), when fields are represented by ranges. We introduce an interesting notion of a *subrange* on a predefined set of range bit-indices allowing to construct equivalent representations of packet classifiers with per-bit resolution avoiding intermediate range expansions. To illustrate this, consider a classifier $\mathcal{K}$ in Fig. 1a based on three range fields. For an incoming header H, the R_3 rule is matched. All three rules are *disjoint* (do not match the same header) on three fields. Note that two fields are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '19, April 3–4, 2019, San Jose, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6710-3/19/04…$15.00
https://doi.org/10.1145/3314148.3314346



$\mathcal{K}$	I_1	I_2	I_3	Action
R_1	[24, 30]	[5, 28]	[9, 16]	A_1
R_2	[18, 20]	[25, 31]	[12, 20]	A_2
$\mathbf{R_3}$	**[17, 23]**	**[5, 21]**	**[8, 15]**	$\mathbf{A_3}$

$\mathcal{K}^{\{F_1,F_2\}}$	I_1	I_2
R_1	[24, 30]	[5, 28]
R_2	[18, 20]	[25, 31]
$\mathbf{R_3}$	**[17, 23]**	**[5, 21]**

$H\downarrow$

	I_3
$\mathbf{R_3}$ =	**[8, 15]**

$\mathcal{K}^{\mathcal{B}}$	$I_1^{B_1}$	$I_2^{B_2}$
R_1	[1]	[0, 3]
R_2	[0]	[3, 3]
$\mathbf{R_3}$	**[0]**	**[0, 2]**

$H\downarrow$

	I_1	I_2	I_3
$\mathbf{R_3}$ =	**[17, 23]**	**[5, 21]**	**[8, 15]**

range encoding methods

	prefix [17]		SRGE [3]	
	# entries	# bits	# entries	# bits
$\mathcal{K}$	105	1575	60	900
$\mathcal{K}^{\{F_1,F_2\}}$ [10]	39	390	30	300
$\mathcal{K}^{\mathcal{B}}$, RR	4	12	4	12

Figure 1: Our RR method vs. previous works: (a) original classifier $\mathcal{K}$ whose rules consist of three 5-bit fields; (b) the representation of $\mathcal{K}$ as in [10] exploiting rule disjointness based on two ranges; (c) the proposed RR implementing rule disjointness on three bits; (d) comparison of three different representations.

sufficient to keep rule disjointness as it is done in [10]. Still these two ranges in every rule are to be encoded by one of the range encoding methods (e.g., prefix expansion [17] or SRGE based on Gray encoding [3]). Intuitively, representations narrowing down the covered field ranges can significantly improve memory requirements. This is a reason why the notion of subranges introduced in this paper allow to implement rule disjointedness only on 3 bit indices (see Fig. 1c). Fig. 1d demonstrates advantages of per-bit resolution leading to significant reduction in TCAM memory requirements both in total bits and ternary entries (e.g., RR allows to reduce the number of ternary entries maintained in TCAM from 60 to 4 for SRGE range encoding).

Based on properties of subranges, we introduce heuristics to find efficient classifier representations in TCAMs. We demonstrate the viability of our approach through comprehensive evaluation where in the extreme case all six fields are based on ranges. Such classifiers cannot be implemented in TCAM with the conventional methods dealing with equivalent classifiers. The evaluation results together with analytic observations confirm that subranges could be an interesting direction to implement a fundamental tradeoff between expressiveness and scalability.

2 MODEL DESCRIPTION

A packet *header* $H = (H_1, \ldots, H_k)$ is a concatenation of k header fields, where each header field H_i is a binary string on w_i bits; headers are matched by classifiers. A *classifier* $\mathcal{K} = \{R_1, \ldots, R_N\}_{\prec}$ is an ordered (by $\prec$) set of rules, where each *rule* $R_i = (\mathcal{F}_i, A_i)$ consists of a *filter* $\mathcal{F}_i$ and the associated *action* A_i. A filter $\mathcal{F} = (F_1, \ldots, F_k)$ is a sequence of k field representations, where each *field representation* F_i is on w_i bits corresponding to the header field H_i. In this paper we consider two types of field representations: (1) a range of values; and (2) a *ternary* bit string, where each bit can have one of the three values 0, 1, or $*$ ("don't care"). We say that a range I is *matching* a value x if $x \in I$. We say that a ternary bit string T *matching* a binary string H of the same length l if for each bit index $1 \leq i \leq l$ either $H[i] = T[i]$ or $T[i] = *$. In the case of ternary field representations, a filter is a ternary bit string produced by concatenation of the ternary bit strings of the corresponding field representations. A filter $\mathcal{F}$ *matches* a header H if every F_i of $\mathcal{F}$ matches the corresponding header field H_i. We say that two filters $\mathcal{F}_1$ and $\mathcal{F}_2$ are *disjoint* if no single header matches both of them. Otherwise, $\mathcal{F}_1$ and $\mathcal{F}_2$ *intersect*. Two rules intersect (are disjoint) if their filters intersect (are disjoint). The main purpose of the classification process is to find the action corresponding to the highest priority rule matching a given header that we call a *classifying* rule. If there is no rule matching H in $\mathcal{K}$, the classification result is a default action that differs from all actions in $\mathcal{K}$. Two classifiers (or their representations) are *equivalent* if the classification result coincides in both classifiers (representations) for all headers.

Example 2.1. A sample 2-field ($k = 2$) range-based classifier $\mathcal{K}$ with 3-bit fields ($w_1 = w_2 = 3$).

$\mathcal{K}$	I_1	I_2	Action
R_1	[0, 6]	[2, 5]	A_1
R_2	[0, 3]	[1, 3]	A_2

A header $H = (010, 001)$ is classified by R_2 since it is matched only by R_2; a header $H' = (010, 011)$ is matched by both R_1 and R_2 but H' is classified by R_1.

3 RANGE ON A SUBSET OF BITS

Recall that each range is encoded into multiple ternary entries whose number is at most linear to the field width (in bits) independently from range encoding methods [16]. Intuitively, taking values on a subset of bit indices of a given range leads to consideration of a smaller set of values and potentially more efficient representations in ternary entries than the originally given range. But this is not enough, for correctness, we need that a matched value in the original range continues to be matched the corresponding entity on the representing subset of bit indices. In this section we introduce such entities implementing both efficiency and correctness that we call *subrange*s and study their properties.

Consider a non-empty subset of bit indices $B \subseteq \{1, 2 \ldots, w\}$ of a w-bit range I. For a value x, denote by x^B a value obtained from x by taking the values of bits at the positions in B. Also we denote by I^B a set of values on B bit indices obtained from the values in a range I.

Example 3.1. Consider a 5-bit range $I = [23, 25]$. The following table consists of values $x \in I$ and $x^B \in I^B$ for a subset of bit-indices $B = \{1, 3, 5\}$; $I^B = \{4, 5, 7\}$.

x	23	24	25
x^B	7 (111_2)	4 (100_2)	5 (101_2)

In the following we explore structural properties of I^B in details. First, note that I^B is not necessarily a range, in Example 3.1, $I^B = \{4, 5, 7\}$. Second, for a range containing a single value $I = [l, l]$, the corresponding $I^B = [l^B, l^B]$ contains also a single value. Next we explore I^B properties on ranges containing at least two values.

For a range $I = [l, r]$, let $sim(I)$ be a first bit index whose value differs in the binary representations of l and r; in Example 3.1 $sim(I) = 2$. We say that a w-bit range $I = [l, r]$ is a *left border range* if l **mod** $2^{w-sim(I)} = 0$; and I is a *right border range* if $(r + 1)$ **mod** $2^{w-sim(I)} = 0$. For instance, $I = [24(11000_2), 28(11100_2)]$ is a left border range; and $I = [19(10011_2), 23(10111_2)]$ is a right border range.

Observation 1. *A range $I = [l, r]$ can be represented as a union of a right border range $I_1 = [l, l$ **or** $(2^{w-sim(I)} - 1)]$[1] and a left border range $I_2 = [r - r$ **mod** $2^{w-sim(I)}, r]$.*

For instance, $[9, 14]$ is a union of $I_1 = [9, 11]$ and $I_2 = [12, 14]$. To understand the properties of I^B, we are starting with a special case of a range I when I is a left (right) border range.

Lemma 3.2. *For a subset of bit-indices $B \subseteq \{1, \ldots, w\}$ and a w-bit range $I = [l, r]$, if I is a left (right) border range, the corresponding I^B is a $|B|$-bit left (right) border range.*

Proof. We show the lemma only for the case when I is a left border range. For the right border range the proof is symmetric. Note that it is sufficient to show the lemma only for the case when $sim(I) = 1$ (i.e., $l = 0$) since in other cases the bits at the first $sim(I) - 1$ positions coincide for all values belonging to I. For a range $I = [0, r]$, consider two values x, y such that $x < r$ and $y = x + 1$; then, either $y^B = x^B + 1$ or $y^B \leq x^B$. Thus, I^B is a range $[0, m^B]$, where m^B is a maximal value belonging to I^B. □

Now we are ready to show that the structure of I^B is similar to the structure of a regular range I; namely, both I^B and I can be represented as a union of the left and right border ranges despite the fact that I^B is not necessary a range.

[1] **or**, **and** are bitwise operations

Algorithm 1 Construction of I^B for w-bit range $I = [l, r]$

```
1:  procedure IS_DIFFERENT_BIT(l, r, w, x)
2:      if (l and 2^(w−x)) ≠ (r and 2^(w−x)) then
3:          return true
4:      else
5:          return false
6:  procedure REDUCE_LEFT(I=[l, r], B)
7:      p = {x : x ∉ B, IS_DIFFERENT_BIT(l, r, w, x)}
8:      if p = ∅ then return [l^B, r^B]
9:      m = r or (2^(w−min(p)) − 1) − 2^(w−min(p))
            ▷ set (w-min(p))th bit of r to 0 and bits after (w − min(p))th to 1
10:     return [l^B, m^B]
11: procedure REDUCE_RIGHT(I=[l, r], B)
12:     p = {x : x ∉ B, IS_DIFFERENT_BIT(l, r, w, x)}
13:     if p = ∅ then return [l^B, r^B]
14:     m = l − l mod (2^(w−min(p))) + 2^(w−min(p))
            ▷ set (w-min(p))th bit of l to 1 and bits after (w − min(p))th to 0
15:     return [m^B, r^B]
16: procedure REDUCE(I=[l, r], B)
17:     if l = r then return [l^B, l^B]
18:     I_1 = [l, l or (2^(w−sim(I)) − 1)]
19:     I_2 = [r − r mod 2^(w−sim(I)), r]
                                ▷ Split I into two border ranges I_1 and I_2
20:     return REDUCE_RIGHT(I_1, B) ∪ REDUCE_LEFT(I_2, B)
```

Theorem 3.3. *For a subset of bit-indices $B \subseteq \{1, \ldots, w\}$ and a w-bit range I, the set I^B can be represented as a union of the $|B|$-bit left border range and $|B|$-bit right border range.*

Proof. By Observation 1 a range I can be splitted into the left and right border ranges. Applying Lemma 3.2 for these border ranges, the theorem immediately follows. □

By definition we knew that in general I^B is a set of values, Theorem 3.3 sheds a light on internals of I^B. From now on, we call a set I^B of values as a *sub-range* of a w-bit range I on bit-indices $B \subseteq \{1, \ldots, w\}$. Algorithm 1 shows how to construct I^B for a given range I and a subset of bit-indices B.

Lemma 3.4. *For a set of indices $B \subseteq \{1, \ldots, w\}$ and a left (right) border range I, the procedure REDUCE_LEFT (REDUCE_RIGHT) in Algorithm 1 correctly constructs I^B.*

Proof. We show the lemma only for the case when I is a left border range; for the right border range the proof is symmetric. Since I is a left border range, $I^B = [l^B, m^B]$ for some $m \in I$. Denote by t a minimal bit index such that $t \notin B$ and the binary representations of l and r differ at the position t. Since I is a left border range values of bit of l and r at the position t equals 0 and 1, respectively. For r and m, the values of bits at the bit positions in B preceding t coincide; otherwise, there exists a bit index $t' < t$ such that $t' \notin B, l$ **and** $2^{w-t'} \neq r$ **and** $2^{w-t'}$ contradicting to minimality of t. Since a bit at the tth position is not considered during construction of I^B, we can obtain m from r by setting in a binary representation of r the bit value at the tth position to 0 and the bits at the positions succeding t to 1. If there is no such position t then $m = r$. □

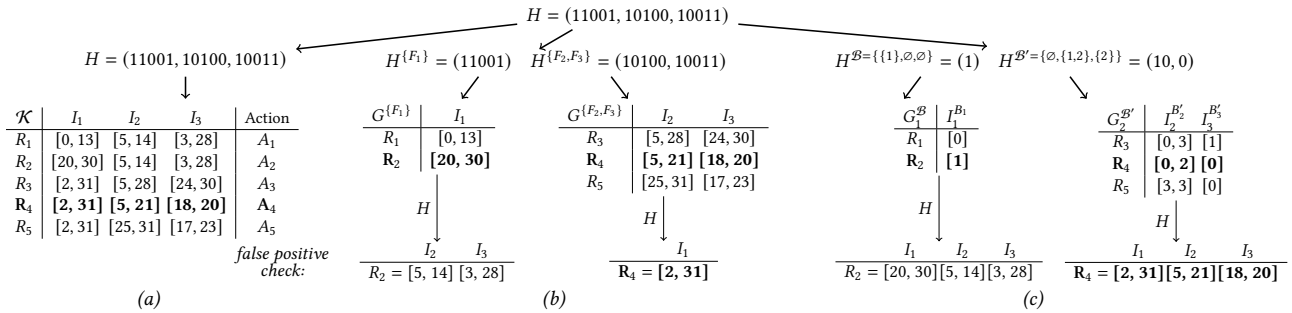


Figure 2: Multigroup representations our RR method vs. SAX-PAC [10]: (a) an original classifier $\mathcal{K}$ with 3-field rules, each field is on 5 bits; (b) SAX-PAC representation of $\mathcal{K}$ on two groups: the first group rules are disjoint on a single field; the second group rules are disjoint on two fields; (c) the proposed RR implementing rule disjointness on a single bit in the first group and on three bits in the second group.

Theorem 3.5. *For a given subset of indices $B \subseteq \{1, \ldots, w\}$ and a w-bit range $I = [l, r]$, the procedure* REDUCE *in Algorithm 1 correctly calculates I^B in $O(w)$ time.*

Proof. If a range I contains a single value, $I^B = [l^B, l^B]$, otherwise I can be splitted into the left and right border ranges I_1, I_2 and then their sub-ranges are constructed correctly by Lemma 3.4. The running time immediately follows by construction of Algorithm 1. □

Recall that encoding of a w-bit regular range consists of $2 \cdot w - 2$ ternary bit strings for the prefix expansion and $2 \cdot w - 4$ for SRGE in the worst case. The following theorem shows that while a subrange I^B is not necessary a range, its ternary encoding consists of at most $2 \cdot |B| - 2$ ternary bit strings for both prefix expansion and SRGE encoding.

Theorem 3.6. *A subrange I^B can be encoded by at most $2 \cdot |B| - 2$ ternary bit strings using prefix expansion or SRGE encoding.*

Proof. The correctness of the theorem immediately follows from the fact that prefix expansion or SRGE encoding of a single w-bit left (right) border range consists of at most $w - 1$ ternary bit strings. □

Note that we can operate on subranges as on regular ranges since the intersection of two subranges can be verified in a constant time. Now we are ready to move on to equivalent representations of multi-field range-based classifiers.

4 EQUIVALENT REPRESENTATIONS

In difference from the previous works considering equivalent classifiers by encoding all ranges of every field [3, 17] (to list a few), [10] proposes equivalent multigroup representations (not classifiers). These representations consist of at most β lookup tables implementing rule disjointness on a subset of fields followed by false-positive checks, where β is a constant

	range encoding methods			
	prefix [17]		SRGE [3]	
	# entries	# bits	# entries	# bits
$\mathcal{K}$	358	5370	188	2820
$G^{\{F_1\}} + G^{\{F_2,F_3\}}$ [10]	44	405	35	320
$G_1^{\mathcal{B}} + G_2^{\mathcal{B}'}$, RR	6	14	6	14

Figure 3: Encoding sizes of the reprsentations in Fig. 2.

corresponding to a number of "pseudo-parallel" lookups that can be issued at line-rate. Representations in [10] allow to encode only a subset of ranges participating in lookup tables in difference from equivalent classifiers requiring encoding all range fields of every rule. Because of rule disjointness only a single rule can be matched at each lookup table; hence, for an incoming header, the false-positive check is done only on matched rules whose ranges can be verified without encoding. Multiple lookup tables (a *multi-group representation*) are necessary to deal with general classifiers whose rules can intersect on all fields and to improve representation efficiency. Note that different lookup tables (groups) can use different subsets of fields to implement rule disjointness.

Formally, consider a classifier $\mathcal{K}$ with k fields represented by ranges. Let $\mathcal{G}$ be a set of β disjoint groups containing rules from $\mathcal{K}$, where every group implements rule disjointness on at most $k-1$ fields. Since not all rules can be covered by $\mathcal{G}$ for a given classifier $\mathcal{K}$, there is a portion of remaining rules C from $\mathcal{K}$ not belonging to $\mathcal{G}$. The classification process in [10] is the following: (1) find a classifying rule at every group (lookup table) and perform a false-positive check for every matched rule (at most β overall); (2) independently with (1) find a classifying rule in C; (3) from at most β matched rules passing a false-positive check and a classifying rule in C return the action of a rule with the highest priority or the default action if there is no a single matching rule for a given header. The classification process is depicted in Fig. 2b. In [10] the authors show that this representation is equivalent to an originally given k-field classifier whose fields are represented by ranges.

Algorithm 2 Heuristic for SAX-PAC [10] representation

```
1:  procedure GROUP_BUILD(𝒦, ℒ)
2:      G = {}
3:      for R ∈ 𝒦 do
4:          if R^ℒ is disjoint to all R'^ℒ : R' ∈ G then G = G ∪ {R}
5:      return G
6:  procedure FIND_MAX_GROUP(𝒦, l)
7:      ℒ = a set of l fields maximizing GROUP_BUILD(𝒦, ℒ)
8:      return (GROUP_BUILD(𝒦, ℒ), ℒ)
9:  procedure GREEDY_GROUP(𝒦, l)
10:     𝒢 = {}
11:     while 𝒦 ≠ ∅ do
12:         G, ℒ = FIND_MAX_GROUP(𝒦, l)
13:         𝒦 = 𝒦 \ G;
14:         𝒢 = 𝒢 ∪ {G^ℒ}
15:     return 𝒢
```

Since our goal is to demonstrate advantages of per-bit over per-field resolution, we follow the same classification process and consider the similar representation $(\mathcal{G}, C)$ but now each group implements rule disjointness on a subset of subranges (at most one per field) on $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$, where B_i is a subset of bit indices of a field F_i. Note that B_i can be empty meaning that the corresponding field does not participate in the lookup table. Observe that different groups in $\mathcal{G}$ can implement rule disjointness on different $\mathcal{B}$s. The classification process is illustrated in Fig. 2c. Similarly to the representation with per-field resolution this representation is equivalent to an originally given classifier with two differences: the matching property of subranges and the false-positive check that should be done on all k fields (and not only on the remaining fields as in [10]). The impact of per-bit versus per-field resolution can be seen in Fig. 3.

We assume that both $\mathcal{G}$ and C (if C exists) are implemented in TCAM. For a given range encoding method E and a classifier $\mathcal{K}^{\mathcal{B}}$ on $\mathcal{B}$ bit indices, denote by $E(\mathcal{K}^{\mathcal{B}})$ the total size in bits of all ternary rules constructed from $\mathcal{K}^{\mathcal{B}}$. Similarly, the size $E(\mathcal{G})$ is a total size of containing groups.

Problem 1. *For a given range encoding method E and a range-based classifier $\mathcal{K}$ find a multigroup representation $(\mathcal{G}, C)$ minimizing $E(\mathcal{G}) + E(C)$.*

Even in the special case of a $\beta = 1$ this problem is intractable that can be shown by reduction from SetCover [8]. Since SAX-PAC representations are included in the set of considered solutions of Problem 1, the resulting encoding of any SAX-PAC representation is no better than the optimal solution of Problem 1.

5 EVALUATION STUDY

In this section we explore the impact of subranges on efficiency of classifier representations.

Evaluated heuristics. Since we want to demonstrate the impact of range representations with per-bit versus per-field resolution, we use the same heuristic as in SAX-PAC [10] to build a multi-group representation; we consider two cases: at every group the rules are pairwise disjoint based on one or two fields ($l = 1$ or $l = 2$); see GREEDY_GROUP($\mathcal{K}, l$) in Algorithm 2. Since SAX-PAC implements per-field resolution for range-based fields, to be stored in TCAM, they are encoded into ternary entries by one of the encoding methods (in our case prefix [17] or SRGE [3]). The proposed RR starts with the same assignment of $\mathcal{K}$ rules into multiple groups as in SAX-PAC but now for every range-based field a subrange is found minimizing the total size in bits or in ternary entries; see GREEDY_GROUP_BIT($\mathcal{K}, l$) in Algorithm 3. For cases when $l \leq 2$ the running time of Algorithm 2 is $O(N^2 \cdot k^2 \cdot \beta)$ and the running time of Algorithm 3 is $O(N^2 \cdot k^2 \cdot \beta + N^2 \cdot w^2)$, where N is a number of rules in $\mathcal{K}$.

Algorithm 3 Heuristic for RR representation

```
1:  procedure REDUCE_GROUP(G)
2:      ℬ = set of sets of all bit-indices of all fields of G
3:      while it is possible do
4:          remove a bit-index from one of sets in ℬ minimizing E(G^ℬ)
            and preserving pairwise disjointness of all rules in G^ℬ.
5:      return G^ℬ
6:  procedure GREEDY_GROUP_BIT(𝒦, l)
7:      𝒢 = {}
8:      while 𝒦 ≠ ∅ do
9:          G, ℱ = FIND_MAX_GROUP(𝒦, l)          ▷ ℱ is ignored
10:         𝒦 = 𝒦 \ G;
11:         𝒢 = 𝒢 ∪ {REDUCE_GROUP(G)}
12:     return 𝒢
```

Methodology. Since the goal of this paper is to understand design principals to represent desired levels of expressiveness and scalability, we assume the extreme case when all classifier fields are represented by ranges. Each synthetically generated classifier consists of 10000 rules that are generated independently. Each field range $I = [l, r]$ is generated independently according to the following distribution: (1) the value of $sim(I)$ is chosen uniformly at random; (2) the range bounds l and r are generated uniformly such that the binary representations of l and r coincide on the first $sim(I) - 1$ bits. Though other distributions are possible, we choose this one to simulate sparseness of covered range values. In our experiments we vary a number of range-based fields in a rule and a range width in bits. For both SRGE and prefix range encodings, we compare the total size in bits and entries among (1) conventional equivalent ternary classifiers, (2) multi-group representations with per-field resolution as in SAX-PAC (the columns SAX-PAC in Fig. 4 and Fig. 5), and (3) multigroup representations with per-bit resolution constructed by GREEDY_GROUP_BIT (the columns RR in Fig. 4 and Fig. 5). We release the code for our evaluation study as an open source [7].

Impact of range width. In the following denote by $\mathcal{G}_1$ and $\mathcal{G}_2$ multi-group representations implementing rule disjointness on one and two fields, respectively. For classifier

Input classifier $\mathcal{K}$				Multigroup: rule disjointness on 1 field						Multigroup: rule disjointness on 2 fields					
characteristics		ternary, prefix		# groups		SAX-PAC [10]		RR		# groups		SAX-PAC [10]		RR	
# ranges	range width	# entries	# bits	95% of $\mathcal{K}$	100% of $\mathcal{K}$	# entries	# bits	# entries	# bits	95% of $\mathcal{K}$	100% of $\mathcal{K}$	# entries	# bits	# entries	# bits
4	16	$3.4 \cdot 10^7$	$2.2 \cdot 10^9$	24	44	40528	648448	23310	505876	7	18	370852	$1.2 \cdot 10^7$	128712	$4.1 \cdot 10^6$
4	20	$8.4 \cdot 10^7$	$6.7 \cdot 10^9$	11	26	54375	$1.1 \cdot 10^6$	18157	417708	5	14	632983	$2.5 \cdot 10^7$	68043	$2.1 \cdot 10^6$
4	24	$1.8 \cdot 10^8$	$1.7 \cdot 10^{10}$	8	20	71220	$1.7 \cdot 10^6$	15286	363079	4	11	983914	$4.7 \cdot 10^7$	52304	$1.6 \cdot 10^6$
4	28	$3.4 \cdot 10^8$	$3.9 \cdot 10^{10}$	6	17	87787	$2.5 \cdot 10^6$	14177	328468	3	10	$1.4 \cdot 10^6$	$7.9 \cdot 10^7$	44735	$1.7 \cdot 10^6$
4	32	$5.9 \cdot 10^8$	$7.5 \cdot 10^{10}$	4	13	106248	$3.4 \cdot 10^6$	12688	317128	3	9	$1.9 \cdot 10^6$	$1.2 \cdot 10^8$	29981	$1.0 \cdot 10^6$
3	32	$3.8 \cdot 10^7$	$3.7 \cdot 10^9$	5	18	109115	$3.5 \cdot 10^6$	18121	451028	3	11	$2.1 \cdot 10^6$	$1.3 \cdot 10^8$	71387	$2.3 \cdot 10^6$
4	32	$5.8 \cdot 10^8$	$7.4 \cdot 10^{10}$	4	11	106315	$3.4 \cdot 10^6$	13998	371824	2	7	$1.9 \cdot 10^6$	$1.2 \cdot 10^8$	31027	$1.0 \cdot 10^6$
5	32	$9.1 \cdot 10^9$	$1.5 \cdot 10^{12}$	4	9	104343	$3.3 \cdot 10^6$	11521	301740	2	7	$1.9 \cdot 10^6$	$1.2 \cdot 10^8$	15523	464471
6	32	$1.4 \cdot 10^{11}$	$2.7 \cdot 10^{13}$	5	9	105026	$3.4 \cdot 10^6$	11160	288901	3	5	$1.9 \cdot 10^6$	$1.2 \cdot 10^8$	13882	464171

Figure 4: Total size of ternary rules in entries and in bits by prefix expansion.

Input classifier $\mathcal{K}$				Multigroup: rule disjointness on 1 field						Multigroup: rule disjointness on 2 fields					
characteristics		ternary, SRGE		# groups		SAX-PAC [10]		RR		# groups		SAX-PAC [10]		RR	
# ranges	range width	# entries	# bits	95% of $\mathcal{K}$	100% of $\mathcal{K}$	# entries	# bits	# entries	# bits	95% of $\mathcal{K}$	100% of $\mathcal{K}$	# entries	# bits	# entries	# bits
4	16	$2.1 \cdot 10^7$	$1.4 \cdot 10^9$	24	44	34620	553920	19053	405194	7	18	286856	$9.2 \cdot 10^6$	87677	$2.8 \cdot 10^6$
4	20	$5.8 \cdot 10^7$	$4.6 \cdot 10^9$	11	26	47328	946560	16084	374638	5	14	512044	$2.0 \cdot 10^7$	46351	$1.5 \cdot 10^6$
4	24	$1.3 \cdot 10^8$	$1.3 \cdot 10^{10}$	8	20	63513	$1.5 \cdot 10^6$	14055	341758	4	11	824121	$4.0 \cdot 10^7$	37976	$1.2 \cdot 10^6$
4	28	$2.6 \cdot 10^8$	$2.9 \cdot 10^{10}$	6	17	79694	$2.2 \cdot 10^6$	12903	305834	3	10	$1.2 \cdot 10^6$	$6.8 \cdot 10^7$	40733	$1.5 \cdot 10^6$
4	32	$4.6 \cdot 10^8$	$5.9 \cdot 10^{10}$	4	13	97917	$3.1 \cdot 10^6$	11899	292533	3	9	$1.7 \cdot 10^6$	$1.1 \cdot 10^8$	24192	775260
3	32	$3.2 \cdot 10^7$	$3.1 \cdot 10^9$	5	18	100641	$3.2 \cdot 10^6$	16286	399179	3	11	$1.8 \cdot 10^6$	$1.2 \cdot 10^8$	58765	$1.9 \cdot 10^6$
4	32	$4.5 \cdot 10^8$	$5.8 \cdot 10^{10}$	4	11	97835	$3.1 \cdot 10^6$	13263	350490	2	7	$1.7 \cdot 10^6$	$1.1 \cdot 10^8$	24976	805932
5	32	$6.7 \cdot 10^9$	$1.1 \cdot 10^{12}$	4	9	95834	$3.1 \cdot 10^6$	11229	285668	2	7	$1.7 \cdot 10^6$	$1.1 \cdot 10^8$	14516	444518
6	32	$1.0 \cdot 10^{11}$	$1.9 \cdot 10^{13}$	5	9	96610	$3.1 \cdot 10^6$	10830	284829	3	5	$1.6 \cdot 10^6$	$1.0 \cdot 10^8$	12355	380373

Figure 5: Total size of ternary rules in entries and in bits by SRGE.

rules based on four 16-bit ranges encoded by prefix expansion, the total size of $\mathcal{G}_1$ with per-field resolution is 1.73 times bigger than with per-bit; this ratio is rapidly growing when range width is increasing; e.g., for 32-bit ranges the same ratio is already equal to 10. The similar effect is seen on the total size in bits for the both range encoding methods. For $\mathcal{G}_2$ representations, the effect of per-bit resolution is even more pronounced; e.g., for classifier rules consisting of four 32-bits ranges encoded by SRGE, the ratio between per-field and per-bit resolution in ternary entries is already 70 times since the total size of $\mathcal{G}_2$ depends quadratically on range width.

Number of fields in a rule. Increasing the number of fields has no significant effect on the total size of multi-group representations with per-field resolution and this is a significant advantage of SAX-PAC versus equivalent classifiers encoding all ranges. The per-bit resolution provides additional memory saving since it allows to pickup multiple subranges minimizing the total size; e.g., for classifier rules on six 32-bit ranges, the size of SRGE encoding for $\mathcal{G}_1$ with per-bit resolution is 10830 ternary entries which is very close to the optimal case when every range-based rule is encoded by a single ternary entry; from the other hand, $\mathcal{G}_1$ with per-field resolution and SRGE encoding requires 96610 entries. In the worst case among all experiments, the average number of ternary entries in $\mathcal{G}_1$ with per-bit resolution does not exceed 2.4 entries per rule.

Number of groups. $\mathcal{G}_2$ requires smaller number of groups than $\mathcal{G}_1$ when per-field resolution is considered; from the other hand, the total size of encoded ternary entries is significantly bigger for $\mathcal{G}_2$ than for $\mathcal{G}_1$; e.g., for rules on five 32-bit ranges, the total size of $\mathcal{G}_2$ in entries for SRGE encdoing is 17 times bigger than for $\mathcal{G}_1$. For per-bit resolution, this effect is less pronounced and does not exceed 6 times in all experiments. Note that in all experiments for classifiers with at least 24-bits range width, the number of entries required for $\mathcal{G}_2$ with per-bit resolution is smaller even than for $\mathcal{G}_1$ with per-field resolution showing effectiveness of per-bit resolution for reduction not only of the total size (both in entries and bits) but also of the number of required groups.

6 CONCLUSION

Range-based field representation is an important abstraction to balance between scalability and expressiveness in packet classifiers; when a number of range-based fields is growing, equivalent classifiers encoding all ranges is not the right direction. Equivalent representations (not classifiers) as SAX-PAC [10] significantly improve memory requirements but still operating with per-field resolution can demand significant memory resources and the number of groups to implement desired structural properties. In this paper we introduce a notion of subranges allowing to operate on ranges with per-bit resolution and overcome constraints of per-field representations. Coexistence of subranges with other encoding methods as [1, 2, 5] and software-based classifier representations in regular memory we leave for the future study.

Acknowledgements. This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation. We thank the anonymous reviewers and our shepherd Ran Ben-Basat for their insightful comments.

REFERENCES

[1] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yacov Hel-Or. 2018. Encoding Short Ranges in TCAM Without Expansion: Efficient Algorithm and Applications. *IEEE/ACM Trans. Netw.* 26, 2 (2018), 835–850.

[2] Anat Bremler-Barr, David Hay, and Danny Hendler. 2012. Layered interval codes for TCAM-based classification. *Computer Networks* 56, 13 (2012), 3023–3039.

[3] Anat Bremler-Barr and Danny Hendler. 2012. Space-Efficient TCAM-Based Classification Using Gray Coding. *Trans. Computers* 61, 1 (2012), 18–30.

[4] Yeim-Kuan Chang and Cheng-Chien Su. 2007. Efficient TCAM Encoding Schemes for Packet Classification Using Gray Code. In *GLOBECOM*. 1834–1839.

[5] Hao Che, Zhijun Wang, Kai Zheng, and Bin Liu. 2008. DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors. *Trans. Computers* 57, 7 (2008), 902–915.

[6] Pavel Chuprikov, Kirill Kogan, and Sergey I. Nikolenko. 2017. General ternary bit strings on commodity longest-prefix-match infrastructures. In *ICNP*. 1–10.

[7] How to deal with range-based packet classifiers [n. d.]. How to deal with range-based packet classifiers. https://github.com/sosrranges/subranges.

[8] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems*. Springer US, Boston, MA, 85–103.

[9] Kirill Kogan, Sergey I. Nikolenko, Patrick Eugster, Alexander Shalimov, and Ori Rottenstreich. 2016. FIB efficiency in distributed platforms. In *ICNP*. 1–10.

[10] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Th. Eugster. 2016. Exploiting Order Independence for Scalable and Expressive Packet Classification. *Trans. Networking* 24, 2 (2016), 1251–1264.

[11] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for advanced packet classification with ternary CAMs. In *SIGCOMM*. 193–204.

[12] Netlogic Microsystems. Content addressable memory [n. d.]. Netlogic Microsystems. Content addressable memory. http://www.netlogicmicro.com.

[13] Sergey I. Nikolenko, Kirill Kogan, Gábor Rétvári, Erika R. Bérczi-Kovács, and Alexander Shalimov. 2016. How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes. In *INFOCOM Workshops*. 521–526.

[14] OpenFlow 1.3 specification, 2012 2012. OpenFlow 1.3 specification, 2012. http://www.openflow.org/wk/index.php/OpenFlow_1_3_proposal.

[15] $P4_{16}$ Language Specification 2017. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[16] Ori Rottenstreich, Rami Cohen, Danny Raz, and Isaac Keslassy. 2013. Exact Worst Case TCAM Rule Expansion. *IEEE Trans. Computers* 62, 6 (2013), 1127–1140.

[17] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. 1998. Fast and Scalable Layer Four Switching. In *SIGCOMM*. 191–202.