PREDICAT: Efficient Packet Classification via Prefix Disjointness

Pavel Chuprikov
USI Lugano, Switzerland
pavel.chuprikov@usi.ch

Vitalii Demianiuk
Ariel University, Israel
vitalii@ariel.ac.il

Sergey Gorinsky
IMDEA Networks Institute, Spain
sergey.gorinsky@imdea.org

***Abstract*—While secure efficient operation of computer networks requires cost-effective line-rate packet classification, network programmability strengthens this need. A promising approach is to transform a packet classifier to a semantically equivalent representation that supports more effective classification. This paper explores transformation of ternary classifiers to equivalent prefix representations so that classification can benefit from efficient Longest Prefix Match solutions. We propose the property of prefix disjointness and design PREDICAT, a method that leverages this new property in combination with a variety of existing techniques to convert an arbitrary ternary classifier to an equivalent prefix representation. The paper analyzes prefix disjointness and evaluates PREDICAT against state-of-the-art transformation alternatives on a packet classification benchmark in regard to the number of lookups. The evaluation shows that PREDICAT outperforms a ternary-to-binary method by an order of magnitude, improves on another ternary-to-prefix solution by a factor from 2 to 5, and preforms similarly to a ternary-to-ternary approach that requires costly power-hungry Ternary Content-Addressable Memories to efficiently handle the resulting ternary representation.**

***Index Terms*—packet classification, filter representation, ternary classifier, prefix classifier, equivalent transformation, prefix disjointness**

I. Introduction

Packet classification determines which action the network element takes upon receiving a packet [1,2]. A packet classifier comprises a sequence of rules where each rule contains a filter and corresponding action. A packet matches a rule if the packet header conforms to the filter of the rule. When the packet matches one or more rules, the element performs the action of the matched rule that has the highest priority. When the packet matches no rule, the element executes a default action.

Cost-effective packet classification at line rate is critical for secure efficient operation of modern computer networks. Network elements classify packets to support firewalling, service differentiation, traffic accounting, etc. Pioneered by OpenFlow [3], network programmability vastly expands usage of packet classification. Domain-specific languages, such as P4 [4], provide means for programming the network-element operation as a series of packet classifiers.

The amount of storage and computation needed by packet classification depends on the classifier size, memory architecture, and filter representation. While classification typically considers multiple fields in the packet header, it is possible to represent each classifier so that the combined content of every filter is a character string. The filters of a binary classifier are bit strings. In a ternary classifier, every filter is a ternary string of characters 0, 1, and $*$ where wildcard character $*$ stands for both 0 and 1. Each filter of a prefix classifier is a prefix, a special kind of a ternary string where a substring of bits precedes a substring of wildcards $*$. Filter representations via prefixes and ternary strings require much less storage space compared to representing each filter as a bit string.

Ternary Content-Addressable Memories (TCAMs) provide effective hardware support for ternary classifiers [5]. Due to the constant-space storage and constant-time lookup offered for ternary-string filter representations, TCAMs align well with the need to classify packets at line rate. On the negative side, TCAMs are costly and consume a lot of power. Implementation of packet classification on conventional memories necessitates more complicated data structures and algorithms, resulting in slower operation [6,7].

One can visualize a packet classifier as a table where the rules form rows, and each character position in the filters constitutes a column. We refer to the number of these columns as the *filter width*. In practice, both filter width and number of rules keep growing, e.g., due to the increasing network programmability and IPv4-to-IPv6 transition [8]. Hence, classification designs should be capable of scaling up to cope with the growth in both dimensions.

Semantically equivalent transformation. Transformation of a packet classifier to a semantically equivalent representation has a potential to substantially improve classification efficiency. This general approach exploits structural properties that classifiers have in practice. SAX-PAC deals with ternary classifiers and introduces the property of order independence as a basis for classifier transformation [9]. Given an order-independent classifier, SAX-PAC reduces its filter width via filter-column removal that preserves order independence. To ensure that the transformation does not affect the outcome of packet classification, SAX-PAC equips the transformed classifier with a true-positive check of the packet against the entire filter of the matched rule in the original classifier. When the input classifier is not order-independent, SAX-PAC partitions it into multiple order-independent groups of rules. The trend towards wider filters in real classifiers makes order independence more common.

Laying a different foundation for classifier transformation, prefix reorderability refers to the ability to convert a ternary classifier to a semantically equivalent prefix classifier via permutation of the filter columns [10]. Prefix reorderability is an appealing property because packet classification can leverage efficient Longest Prefix Match (LPM) solutions developed for packet forwarding on CPU, GPU, and FPGA platforms that do not use TCAMs [11–17]. On the other hand, the trend towards wider filters is detrimental for prefix reorderability.

In this paper, we propose *prefix disjointness* as a new basis for transformation of ternary classifiers to semantically equivalent prefix classifiers. The respective transformation relies on a technique of *wildcarding* that replaces character 0 or 1 in a ternary filter with wildcard $*$. When beneficial for clarity of exposition, we alternatively use $\circledast$ to denote the $*$ characters introduced by wildcarding, in order to distinguish them from the wildcards in the original classifier. A ternary classifier is *prefix-disjoint* if it is order-independent, and wildcarding can transform it to a prefix-reorderable classifier without violation of order independence. We design a test for prefix disjointness and an efficient algorithm for converting a prefix-disjoint classifier to its prefix-reorderable equivalent. The paper analyzes how the trend towards wider filters affects prefix disjointness.

We also develop PREDICAT, a novel method for transforming an arbitrary ternary classifier to an equivalent prefix representation. In addition to our novel procedure for transformation of prefix-disjoint classifiers to prefix-reorderable equivalents, PREDICAT utilizes various existing techniques such as classifier partitioning, filter-width reduction, filter-column permutation, and true-positive checking. Our evaluation on a packet classification benchmark compares PREDICAT with state-of-the-art ternary-to-prefix, ternary-to-binary, and ternary-to-ternary transformation methods in regard to the number of lookups.

Example 1. *PREDICAT transformation of prefix-disjoint classifier $\mathcal{K}$ to prefix classifier $\mathcal{K}'''$:*

$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
R_1	1	1	1	1	1	A_1
R_2	1	1	1	1	0	A_2
R_3	0	$*$	1	1	$*$	A_3
R_4	$*$	$*$	0	1	0	A_4
R_5	1	$*$	$*$	0	$*$	A_5

$\Rightarrow$

$\mathcal{K}'''$	f_4	f_3	f_1	f_5	*Action*
R_1'''	1	1	1	1	$A_1\|C$
R_2'''	1	1	1	0	$A_2\|C$
R_3'''	1	1	0	$*$	$A_3\|C$
R_4'''	1	0	$*$	$\circledast$	$A_4\|C$
R_5'''	0	$*$	$\circledast$	$*$	$A_5\|C$

Example 1 illustrates operation of PREDICAT on classifier $\mathcal{K}$. This classifier is not prefix-reorderable because there exists no permutation of its filter columns that transforms it to a prefix representation. Yet, $\mathcal{K}$ is prefix-disjoint. PREDICAT transforms $\mathcal{K}$ to $\mathcal{K}'''$, a prefix classifier with narrower filters. Specifically, PREDICAT transforms rules R_4 and R_5 to R_4''' and R_5''' respectively by wildcarding of their (R_4, f_5) and (R_5, f_1) characters. Then, PREDICAT removes filter column f_2 and permutes filter columns $f_1f_3f_4f_5$ to $f_4f_3f_1f_5$. Each rule R_i''' matched in $\mathcal{K}'''$ makes its action A_i conditional on true-positive check C that verifies whether the packet also matches respective rule R_i in $\mathcal{K}$. The classification of a packet with header $h_1h_2h_3h_4h_5 = 10010$ analogously converts the header to $h_4h_3h_1h_5 = 0110$, employs the converted header to identify R_4''' as the matched rule in $\mathcal{K}'''$, passes check C by using the original header to confirm R_4 as the matched rule in $\mathcal{K}$, and returns action A_4. The classification of a packet with header 10011 similarly identifies R_4''' as the matched rule in $\mathcal{K}'''$ but fails the same C check and returns the default action.

Our main contributions are as follows:

- We propose and analyze prefix disjointness as a novel foundation for transforming a ternary classifier to a semantically equivalent prefix classifier. The paper develops an efficient test for prefix disjointness as well as algorithm PDTRANSFORM that applies wildcarding and true-positive checking to convert a prefix-disjoint classifier to an order-independent prefix-reorderable equivalent in time linear in the classifier size. We also prove that the trend towards wider filters preserves prefix disjointness.
- The paper designs PREDICAT, a method that transforms an arbitrary ternary classifier to an equivalent prefix representation. PREDICAT combines PDTRANSFORM with a variety of existing techniques such as classifier partitioning into multiple groups of rules, filter-width reduction based on order independence, and permutation of filter columns.
- The evaluation on the ClassBench benchmark compares the number of lookups for PREDICAT vs. state-of-the-art transformation approaches: ternary-to-prefix LPM-PR, ternary-to-binary EXACT, and ternary-to-ternary SAX-PAC. The evaluation shows that PREDICAT outperforms LPM-PR by a factor from 2 to 5, EXACT by an order of magnitude, and preforms similarly to SAX-PAC that relies on costly power-hungry TCAMs to efficiently classify packets on the resulting ternary representations.

The rest of the paper has the following structure. Section II details relevant background in a consistent mathematical notation. Section III proposes and analyzes prefix disjointness. Section IV presents PREDICAT. Section V reports on the evaluation. Section VI discusses related work. Section VII concludes the paper with a summary of its results.

II. Background

This section defines our formal model and uses it to consistently present background on packet classification, including order independence [9] and prefix reorderability [10].

A. Packet classification

To model packet headers, we consider only those header bits that are relevant for packet classification. *Header H* of a packet is a w-bit string, with each bit h_j being either 0 or 1, e.g., 10110 is a 5-bit header. *Classifier $\mathcal{K}$* refers to a sequence of *rules* $R_i = \langle F_i \mapsto A_i \rangle$ where each rule consists of *filter* F_i and corresponding *action* A_i. Filter F_i contains a w-character ternary string where each character is either 0, 1, or *wildcard* $*$. The wildcard character stands for both 0 and 1. For instance, $*0*1*$ is a 5-character filter. We represent classifier $\mathcal{K}$ as a table where each rule R_i forms a separate row, and the jth most significant characters of all the filters compose *filter column* f_j. Hence, cell (R_i, f_j) of the table stores *filter character* c_{ij}, which is the jth most significant character of filter F_i. The classifier arranges the rules in the order of their decreasing priority. Packet P with header H *matches* rule R_i if $c_{ij} = * \vee c_{ij} = h_j$ for $j = 1, \ldots, w$, i.e., every character of filter F_i is either $*$ or the same as respective bit h_j. For fluency of discourse about rule matching, we use expressions "P matches" and "H matches" interchangeably. The classifier returns action A_i of the first matched rule, i.e., the matched rule with the highest priority. When the packet matches no rule, the classifier returns default action ✗. Two classifiers are *semantically equivalent*, or just *equivalent* for short, if either classifier returns the same action whenever presented with the same packet.

Example 2. *Classifier $\mathcal{K}'$ with 5 rules and 5-character filters:*

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
R'_1	1	1	1	1	1	A'_1
R'_2	1	1	1	1	0	A'_2
R'_3	0	$*$	1	1	$*$	A'_3
R'_4	$*$	$*$	0	1	$*$	A'_4
R'_5	$*$	$*$	$*$	0	$*$	A'_5

$H_1 = 11111 \Rightarrow A'_1$

$H_2 = 11010 \Rightarrow A'_4$

$H_3 = 10110 \Rightarrow$ ✗

Example 2 depicts classifier $\mathcal{K}'$ with 5 rules R'_1, R'_2, R'_3, R'_4, and R'_5 and 5 filter columns f_1, f_2, f_3, f_4, and f_5. For packets with headers $H_1 = 11111$, $H_2 = 11010$, and $H_3 = 10110$, the classifier returns actions A'_1, A'_4, and ✗ respectively.

B. Order independence and prefix reorderability

A classifier is *order-independent* if any packet matches at most one of its rules [9]. The property name reflects independence of the returned action from the order of the rules in the classifier. SAX-PAC exploits order independence to transform the classifier to an equivalent classifier with narrower filters.

A ternary classifier is a *prefix classifier* if all its filters are prefixes, e.g., classifier $\mathcal{K}'$ in Example 2 is not a prefix classifier. A prefix classifier is an *LPM classifier* if, whenever a packet matches multiple rules, the filter of the first matched rule has the longest binary substring, i.e., the shortest wildcard substring. LPM classifiers are attractive because packet classification can readily utilize efficient LPM solutions developed for packet forwarding. Because one can transform any prefix classifier to an equivalent LPM classifier [10], the rest of this paper interchangeably refers to an LPM classifier as a prefix classifier.

Ternary classifier $\mathcal{K}'$ is *prefix-reorderable* if there exists a permutation of its filter columns that transforms $\mathcal{K}'$ to an equivalent prefix classifier [10]. We denote this prefix classifier as $\mathcal{K}''$. When applied to the bits of header H', the same permutation produces header H''. The classifications of H' on $\mathcal{K}'$ and H'' on $\mathcal{K}''$ result in identical actions. The *chain criterion* specifies conditions for prefix reorderability [10,18]. The criterion characterizes each rule R_i of the ternary classifier with set $\mathrm{exact}(R_i) = \{j : c_{ij} = 0 \vee c_{ij} = 1\}$ that records the positions where the w-character filter of the rule contains 0 or 1. According to the chain criterion, classifier $\mathcal{K}'$ is prefix-reorderable iff the $\mathrm{exact}(R_i)$ sets of all rules R_i in $\mathcal{K}'$ form a chain of inclusion relations. This inclusion chain imposes a partial order on the filter columns in the permuted classifier. The partial order determines the permutations that transform $\mathcal{K}'$ to a prefix classifier.

Example 3. *Semantically equivalent transformation of prefix-reorderable classifier $\mathcal{K}'$ to prefix classifier $\mathcal{K}''$:*

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
R'_1	1	1	1	1	1	A'_1
R'_2	1	1	1	1	0	A'_2
R'_3	0	$*$	1	1	$*$	A'_3
R'_4	$*$	$*$	0	1	$*$	A'_4
R'_5	$*$	$*$	$*$	0	$*$	A'_5

$\Rightarrow$

$\mathcal{K}''$	f_4	f_3	f_1	f_2	f_5	*Action*
R''_1	1	1	1	1	1	A'_1
R''_2	1	1	1	1	0	A'_2
R''_3	1	1	0	$*$	$*$	A'_3
R''_4	1	0	$*$	$*$	$*$	A'_4
R''_5	0	$*$	$*$	$*$	$*$	A'_5

Borrowing classifier $\mathcal{K}'$ from Example 2, Example 3 illustrates its prefix reorderability by transforming it to prefix classifier $\mathcal{K}''$. $\mathcal{K}'$ is prefix-reorderable because there exists the following inclusion chain: $\mathrm{exact}(R'_1) = \{1,2,3,4,5\} \supseteq \mathrm{exact}(R'_2) = \{1,2,3,4,5\} \supset \mathrm{exact}(R'_3) = \{1,3,4\} \supset \mathrm{exact}(R'_4) = \{3,4\} \supset \mathrm{exact}(R'_5) = \{4\}$. The chain imposes the following partial order on the filter columns in the permuted classifier: filter columns f_1, f_3, and f_4 precede filter columns f_2 and f_5; filter columns f_3 and f_4 precede f_1; filter column f_4 precedes f_3. Filter-column permutation $f_4 f_3 f_1 f_2 f_5$ transforms $\mathcal{K}'$ to $\mathcal{K}''$.

III. Prefix disjointness

While prefix reorderability usefully enables efficient packet classification on LPM solutions, this property frequently does not hold, e.g., even when each filter consists of only two prefix fields for matching the source and destination IP addresses in the packet header. Besides, the trend towards wider filters makes prefix reorderability less likely.

This section explores a different approach to equivalent transformation of a ternary classifier to a prefix classifier. In accordance with the chain criterion, a classifier is not prefix-reorderable iff the $\mathrm{exact}(R_i)$ sets do not form an inclusion chain. Hence, alteration of some rules R_i to create an inclusion chain from the modified $\mathrm{exact}(R_i)$ sets constitutes a promising direction for transformation of the classifier to a prefix-reorderable representation and eventually a prefix classifier. One option for pursuing this direction is rule duplication: when a rule contains a wildcard that interferes with creating an inclusion chain, duplicate the rule and change the problematic $*$ to 0 and 1 in the first and second copies of the rule respectively. For instance, revisit Example 1 and duplicate rule R_3 into rules R_3^a and R_3^b, change their (R_3^a, f_5) and (R_3^b, f_5) characters from $*$ to 0 and 1 respectively, duplicate rule R_4 into rules R_4^a and R_4^b, and change their (R_4^a, f_1) and (R_4^b, f_1) characters from $*$ to 0 and 1 respectively. However, the practical utility of rule duplication is doubtful due to the exponential increase in storage space needed for the duplicated rules.

Our approach proposes the property of *prefix disjointness* and relies on the technique of *wildcarding* that replaces character 0 or 1 in the filter of a rule with wildcard $*$. We alternatively denote the $*$ character introduced by wildcarding as $\circledast$. Note that wildcarding does not affect the number of rules. The main challenge lies in assuring the semantic equivalence of original classifier $\mathcal{K}$ and transformed classifier $\mathcal{K}'$. We tackle this challenge by imposing the constraint of order independence on $\mathcal{K}$, meaning that any packet matches at most one of the rules in $\mathcal{K}$. By also requiring that the transformation preserves order independence, we guarantee that any packet matches at most one of the rules in $\mathcal{K}'$ as well. Due to wildcarding, the set of packets that conform to the only matched rule is larger for $\mathcal{K}'$ compared to $\mathcal{K}$. Thus, when the classification on $\mathcal{K}'$ results in a match, we apply a true-positive check to the respective matched rule in $\mathcal{K}$. If the true-positive check succeeds, the classification returns the action of this rule. If the check fails, or the packet does not match any rule in $\mathcal{K}'$, the classification returns the default action of $\mathcal{K}$. Based on the above, we define a ternary classifier to be *prefix-disjoint* if it is order-independent, and wildcarding can transform it to a prefix-reorderable classifier without violation of order independence. We refer to the resulting prefix-reorderable classifier, when it is equipped with the true-positive check, as a *prefix-reorderable equivalent* of the prefix-disjoint classifier. Note that the prefix-reorderable equivalent is both prefix-reorderable and order-independent.

Theorem 1. *A prefix-disjoint classifier and its prefix-reorderable equivalent are semantically equivalent.*

Proof. Let $\mathcal{K}'$ be a prefix-reorderable equivalent of prefix-disjoint classifier $\mathcal{K}$. If packet P matches rule R_i in $\mathcal{K}$, P also matches respective rule R'_i in $\mathcal{K}'$. Because both $\mathcal{K}$ and $\mathcal{K}'$ are order-independent, P matches on either $\mathcal{K}$ or $\mathcal{K}'$ only one rule (R_i or R'_i respectively). The true-positive check for P on rule R_i in $\mathcal{K}$ succeeds, and the classification of P on either $\mathcal{K}$ or $\mathcal{K}'$ returns the same action, the action of rule R_i.

If P does not match any rule in $\mathcal{K}$, P either does not match any rule in $\mathcal{K}'$ or, due to order independence of $\mathcal{K}'$, matches only one rule R'_i in $\mathcal{K}'$. In the case of the match, the true-positive check for P on respective rule R_i in $\mathcal{K}$ fails. In either case, the classification of P on either $\mathcal{K}$ or $\mathcal{K}'$ returns the same default action. Thus, the classification of any packet returns the same action on either $\mathcal{K}$ or $\mathcal{K}'$. □

Example 4. *Semantically equivalent transformation of prefix-disjoint classifier $\mathcal{K}$ to its prefix-reorderable equivalent $\mathcal{K}'$:*

$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
R_1	1	1	1	1	1	A_1
R_2	1	1	1	1	0	A_2
R_3	0	$*$	1	1	$*$	A_3
R_4	$*$	$*$	0	1	0	A_4
R_5	1	$*$	$*$	0	$*$	A_5

$\Rightarrow$

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
R'_1	1	1	1	1	1	$A_1\|C$
R'_2	1	1	1	1	0	$A_2\|C$
R'_3	0	$*$	1	1	$*$	$A_3\|C$
R'_4	$*$	$*$	0	1	$\circledast$	$A_4\|C$
R'_5	$\circledast$	$*$	$*$	0	$*$	$A_5\|C$

Example 4 shows how wildcarding of the (R_4, f_5) and (R_5, f_1) filter characters and addition of true-positive check C transform prefix-disjoint classifier $\mathcal{K}$ to its prefix-reorderable equivalent $\mathcal{K}'$. The classification of a packet with header 10000 on $\mathcal{K}'$ matches rule R'_5, succeeds in check C on rule R_5 in $\mathcal{K}$, and returns action A_5. While a packet with header 00000 also matches rule R'_5 in $\mathcal{K}'$, check C on rule R_5 in $\mathcal{K}$ fails, and the classification of this packet returns default action ✗.

While the trend towards wider filters is detrimental for prefix reorderability, the following theorem reveals that this trend does not undermine prefix disjointness.

Theorem 2. *Widening the filters of a prefix-disjoint classifier preserves prefix disjointness.*

Proof. Let $\mathcal{K}_2$ be a classifier obtained from prefix-disjoint classifier $\mathcal{K}_1$ by widening its filters with some number of extra columns. Since $\mathcal{K}_1$ is order-independent, any packet matches at most one rule in $\mathcal{K}_1$. Because widening the filters maintains the property of matching at most one rule for any packet, $\mathcal{K}_2$ is also order-independent. Consider a transformation of $\mathcal{K}_1$ to its prefix-reorderable equivalent. Application of the same transformation to $\mathcal{K}_2$ in combination with wildcarding of all the characters in the newly added filter columns transforms $\mathcal{K}_2$ to its prefix-reorderable equivalent. Hence, $\mathcal{K}_2$ is prefix-disjoint. □

Practical application of prefix disjointness requires an efficient test for the property and an efficient algorithm for transforming a prefix-disjoint classifier to its prefix-reorderable equivalent. Below, we formalize this need as problem PDCHECK and develop algorithm PDTRANSFORM for solving the problem.

Problem 1 (PDCHECK)**.** *Given a ternary classifier, determine whether it is prefix-disjoint. If it is, provide a prefix-reorderable equivalent of the classifier.*

We define *all-exact set* $\mathcal{E}(\mathcal{K})$ of classifier $\mathcal{K}$ as the set of the filter positions where all rules R_i in $\mathcal{K}$ contain character 0 or 1, i.e., $\mathcal{E}(\mathcal{K}) = \bigcap_{R_i \in \mathcal{K}} \text{exact}(R_i)$. For each rule R_i, we construct *counterpart rule* $R_i^{\mathcal{E}}$ by wildcarding of all its filter characters at positions not present in $\mathcal{E}(\mathcal{K})$ and adding true-positive check C. Then, we extract classifier $\mathcal{K}^{\mathcal{E}}$ from $\mathcal{K}$ by removing each rule R_i such that a packet matches both counterpart rule $R_i^{\mathcal{E}}$ and another rule in $\mathcal{K}$. In Example 4, all-exact set $\mathcal{E}(\mathcal{K})$ is $\{4\}$, rule $R_5^{\mathcal{E}}$ has filter $***0*$, and $\mathcal{K}^{\mathcal{E}}$ contains only rule R_5.

Lemma 1. *If classifier $\mathcal{K}$ is prefix-disjoint, there exists its prefix-reorderable equivalent $\mathcal{K}'$ where, for each rule R_i in $\mathcal{K}^{\mathcal{E}}$, counterpart rule $R_i^{\mathcal{E}}$ serves as rule R'_i.*

Proof. Let $\mathcal{K}$ be a prefix-disjoint classifier. Because the filter positions present in all-exact set $\mathcal{E}(\mathcal{K})$ pose no obstacles for prefix reorderability, one can transform $\mathcal{K}$ to its prefix-reorderable equivalent without wildcarding in the respective columns. Thus, we consider prefix-reorderable equivalent $\mathcal{K}'$ obtained through such transformation, meaning that all-exact set $\mathcal{E}(\mathcal{K}')$ remains the same as $\mathcal{E}(\mathcal{K})$. We extract $\mathcal{K}^{\mathcal{E}}$ from $\mathcal{K}$ and, for each rule R_i in $\mathcal{K}^{\mathcal{E}}$, replace rule R'_i in $\mathcal{K}'$ with counter-

Algorithm 1 PDTRANSFORM($\mathcal{K}$)

```
1:  $\mathcal{K}' \leftarrow ()$   // initialization of $\mathcal{K}'$ with an empty classifier
2:  $h[1,\ldots,N] \leftarrow 0,\ldots,0$   // character-tracking array
3:  for $j = 1,\ldots,w$ do
4:      $e[j] \leftarrow |\{R_i \in \mathcal{K} : c_{ij} = 0 \vee c_{ij} = 1\}|$
        // $e[j]$ tracks the total number of 0s and 1s in column $f_j$
5:  $\mathcal{E}_{\text{prev}} \leftarrow \{\}$   // initialization of the all-exact set
6:  while $\mathcal{K} \neq ()$ do
7:      $\mathcal{E}(\mathcal{K}) \leftarrow \{j : e[j] = |\mathcal{K}|\}$   // update of the all-exact set
8:      if $\mathcal{E}(\mathcal{K}) = \mathcal{E}_{\text{prev}}$
9:          return not prefix-disjoint
10:     $d[0,\ldots,N-1] \leftarrow 0,\ldots,0$
11:     for $R_i \in \mathcal{K}$ do
12:         $h[i] \leftarrow g(h[i], (c_{ij} : j \in \mathcal{E}(\mathcal{K}) \setminus \mathcal{E}_{\text{prev}}))$
13:         $d[h[i]] \leftarrow d[h[i]] + 1$
            // counting the rules that have the same $h[i]$ value
14:     $\mathcal{K}^{\mathcal{E}} \leftarrow (R_i \in \mathcal{K} : d[h[i]] = 1)$
15:     $\mathcal{K}' \leftarrow \mathcal{K}' \cup (R_i^{\mathcal{E}} : R_i \in \mathcal{K}^{\mathcal{E}})$
16:     for $j = 1,\ldots,w$ do
17:         $e[j] \leftarrow e[j] - |\{R_i \in \mathcal{K}^{\mathcal{E}} : c_{ij} = 0 \vee c_{ij} = 1\}|$
18:     $\mathcal{E}_{\text{prev}} \leftarrow \mathcal{E}(\mathcal{K})$
19:     $\mathcal{K} \leftarrow \mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}$
20: return prefix-disjoint, $\mathcal{K}'$
```

part rule $R_i^{\mathcal{E}}$. Due to preserving both order independence and prefix reorderability of $\mathcal{K}'$, the replacement produces a prefix-reorderable equivalent of $\mathcal{K}$ with the desired property. ☐

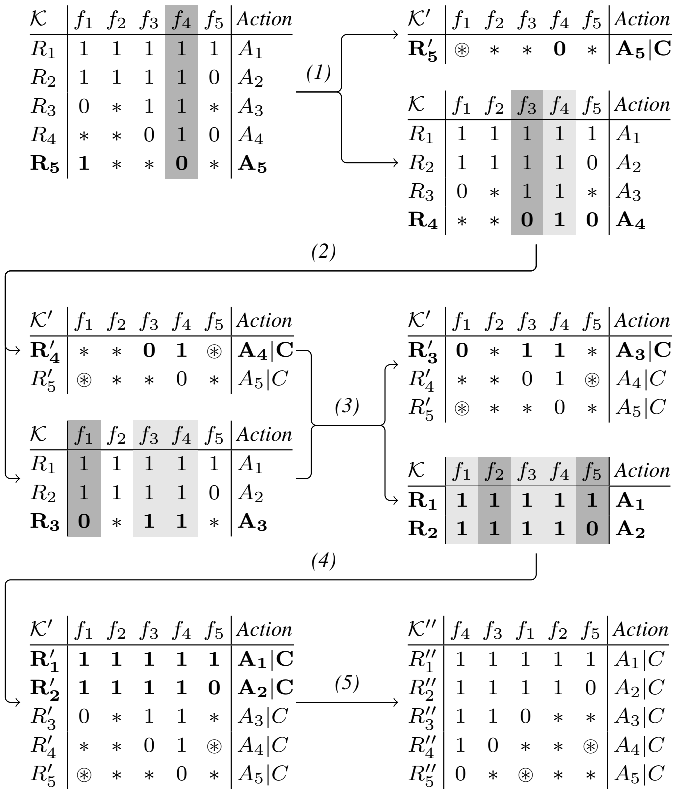
Lemma 2. *If $\mathcal{K}$ is a prefix-disjoint classifier such that $\mathcal{K}^{\mathcal{E}}$ differs from $\mathcal{K}$, set $\mathcal{E}(\mathcal{K})$ is a proper subset of $\mathcal{E}(\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}})$.*

Proof. As in the proof of Lemma 1, we consider such prefix-reorderable equivalent $\mathcal{K}'$ of prefix-disjoint classifier $\mathcal{K}$ that $\mathcal{E}(\mathcal{K}') = \mathcal{E}(\mathcal{K})$. Because $\mathcal{K}'$ is order-independent, $\mathcal{K}^{\mathcal{E}}$ includes each such rule R_i that set $\text{exact}(R_i')$ for respective rule R_i' in $\mathcal{K}'$ is the same as $\mathcal{E}(\mathcal{K}')$. When rule R_i of $\mathcal{K}$ is not in $\mathcal{K}^{\mathcal{E}}$, set $\mathcal{E}(\mathcal{K}')$ is a proper subset of $\text{exact}(R_i')$. Since $\mathcal{K}'$ is prefix-reorderable, and $\mathcal{K}^{\mathcal{E}}$ differs from $\mathcal{K}$, classifier $\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}$ contains rule R_j such that $\text{exact}(R_j')$ is a subset of $\text{exact}(R_i')$ for any R_i in $\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}$. Consequently, $\mathcal{E}(\mathcal{K}')$ is a proper subset of $\text{exact}(R_j') = \bigcap_{R_i \in \mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}} \text{exact}(R_i')$ since $R_j \notin \mathcal{K}^{\mathcal{E}}$. Because $\mathcal{E}(\mathcal{K})$ is the same as $\mathcal{E}(\mathcal{K}')$, which is a proper subset of $\bigcap_{R_i \in \mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}} \text{exact}(R_i')$, which is a subset of $\bigcap_{R_i \in \mathcal{K} \setminus \mathcal{K}^{\mathcal{E}}} \text{exact}(R_i)$, and the latter is $\mathcal{E}(\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}})$, we conclude that set $\mathcal{E}(\mathcal{K})$ is a proper subset of $\mathcal{E}(\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}})$. ☐

Lemmata 1 and 2 pave the way for solving the PDCHECK problem. Given ternary classifier $\mathcal{K}$, the solution initializes $\mathcal{K}'$ with an empty classifier and starts iterating. Following the insight of Lemma 1, each iteration: (a) extracts $\mathcal{K}^{\mathcal{E}}$ from $\mathcal{K}$, (b) inserts, for each rule R_i in $\mathcal{K}^{\mathcal{E}}$, counterpart rule $R_i^{\mathcal{E}}$ as rule R_i' into $\mathcal{K}'$, and (c) removes the rules of $\mathcal{K}^{\mathcal{E}}$ from $\mathcal{K}$. In accordance with Lemma 2, if any iteration encounters conditions where $\mathcal{K}^{\mathcal{E}}$ differs from $\mathcal{K}$ but all-exact sets $\mathcal{E}(\mathcal{K})$ and $\mathcal{E}(\mathcal{K} \setminus \mathcal{K}^{\mathcal{E}})$ are the same, the solution terminates by concluding that the original classifier is not prefix-disjoint. Otherwise, after iterating until $\mathcal{K}$ becomes empty, the solution terminates by providing $\mathcal{K}'$ as a prefix-reorderable equivalent of the original classifier.

Algorithm 1 presents the pseudocode of our PDTRANSFORM solution. The input to the algorithm is $\mathcal{K}$, a ternary classifier that has N rules R_i and w filter columns f_j. Lines 1–5 perform initializations. In particular, Line 2 initializes array h where element $h[i]$ contains the integer between 0 and $N-1$ that identifies the characters of rule R_i in the filter positions present in all-exact set $\mathcal{E}(\mathcal{K})$. The algorithm starts iteratively constructing $\mathcal{K}'$ in Line 6. Line 7 updates the all-exact set of $\mathcal{K}$. Lines 8 and 9 check the prefix-disjointness conditions and, if the conditions do not hold, terminate the algorithm with proclaiming that the input classifier is not prefix-disjoint. Lines 11–14 extract $\mathcal{K}^{\mathcal{E}}$ from $\mathcal{K}$. Note that rule R_i belongs to $\mathcal{K}^{\mathcal{E}}$ iff $\mathcal{K}$ does not contain another rule such that all characters in the $\mathcal{E}(\mathcal{K})$ positions of its filter are the same as the corresponding characters in R_i. Hence, Line 12 updates $h[i]$ by applying g, a perfect hash function [19], to the current $h[i]$ value and all characters c_{ij} in the filter positions present in the set difference of the updated $\mathcal{E}(\mathcal{K})$ set and its previous version. Line 14 inserts into $\mathcal{K}^{\mathcal{E}}$ each rule R_i such that its corresponding element $h[i]$ has a unique value in array h. Line 15 inserts counterpart rule $R_i^{\mathcal{E}}$ into $\mathcal{K}'$. Line 19 removes from $\mathcal{K}$ all the rules of $\mathcal{K}^{\mathcal{E}}$.

Example 5. *Transformation of prefix-disjoint classifier $\mathcal{K}$ to its prefix-reorderable equivalent $\mathcal{K}'$ by* PDTRANSFORM*:*



$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
R_1	1	1	1	1	1	A_1
R_2	1	1	1	1	0	A_2
R_3	0	$*$	1	1	$*$	A_3
R_4	$*$	$*$	0	1	0	A_4
$\mathbf{R_5}$	**1**	$*$	$*$	**0**	$*$	$\mathbf{A_5}$

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
$\mathbf{R_5'}$	$\circledast$	$*$	$*$	**0**	$*$	$\mathbf{A_5 \mid C}$

$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
R_1	1	1	1	1	1	A_1
R_2	1	1	1	1	0	A_2
R_3	0	$*$	1	1	$*$	A_3
$\mathbf{R_4}$	$*$	$*$	**0**	**1**	**0**	$\mathbf{A_4}$

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
$\mathbf{R_4'}$	$*$	$*$	**0**	**1**	$\circledast$	$\mathbf{A_4 \mid C}$
R_5'	$\circledast$	$*$	$*$	0	$*$	$A_5 \mid C$

$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
R_1	1	1	1	1	1	A_1
R_2	1	1	1	1	0	A_2
$\mathbf{R_3}$	**0**	$*$	**1**	**1**	$*$	$\mathbf{A_3}$

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
$\mathbf{R_3'}$	**0**	$*$	**1**	**1**	$*$	$\mathbf{A_3 \mid C}$
R_4'	$*$	$*$	0	1	$\circledast$	$A_4 \mid C$
R_5'	$\circledast$	$*$	$*$	0	$*$	$A_5 \mid C$

$\mathcal{K}$	f_1	f_2	f_3	f_4	f_5	*Action*
$\mathbf{R_1}$	**1**	**1**	**1**	**1**	**1**	$\mathbf{A_1}$
$\mathbf{R_2}$	**1**	**1**	**1**	**1**	**0**	$\mathbf{A_2}$

$\mathcal{K}'$	f_1	f_2	f_3	f_4	f_5	*Action*
$\mathbf{R_1'}$	**1**	**1**	**1**	**1**	**1**	$\mathbf{A_1 \mid C}$
$\mathbf{R_2'}$	**1**	**1**	**1**	**1**	**0**	$\mathbf{A_2 \mid C}$
R_3'	0	$*$	1	1	$*$	$A_3 \mid C$
R_4'	$*$	$*$	0	1	$\circledast$	$A_4 \mid C$
R_5'	$\circledast$	$*$	$*$	0	$*$	$A_5 \mid C$

$\mathcal{K}''$	f_4	f_3	f_1	f_2	f_5	*Action*
R_1''	1	1	1	1	1	$A_1 \mid C$
R_2''	1	1	1	1	0	$A_2 \mid C$
R_3''	1	1	0	$*$	$*$	$A_3 \mid C$
R_4''	1	0	$*$	$*$	$\circledast$	$A_4 \mid C$
R_5''	0	$*$	$\circledast$	$*$	$*$	$A_5 \mid C$

Elaborating on Example 4, Example 5 demonstrates how PDTRANSFORM transforms prefix-disjoint classifier $\mathcal{K}$ to its prefix-reorderable equivalent $\mathcal{K}'$. Each of Steps 1 – 4 in Example 5 illustrates one iteration where PDTRANSFORM extracts rules R_i from $\mathcal{K}$, applies wildcarding to the filter characters in these rules, equips them with true-positive check C, and inserts resulting rules R'_i into $\mathcal{K}'$. The step denotes extracted rules R_i and inserted rules R'_i in **bold**. After each of these steps, $\mathcal{K}$ contains two sets of darker ■ and lighter □ shaded filter columns that represent new positions in all-exact set $\mathcal{E}(\mathcal{K})$ and its previous version respectively. Step 5 in Example 5 shows the semantically equivalent transformation of $\mathcal{K}'$ to prefix classifier $\mathcal{K}''$, the same as already depicted in Example 3.

Theorem 3. *The* PDTRANSFORM *algorithm correctly solves the* PDCHECK *problem for any ternary classifier.*

Proof. The algorithm correctness follows from Lemmata 1 and 2. While Lemma 2 establishes conditions for prefix disjointness, Lines 8 and 9 of PDTRANSFORM check these conditions and correctly terminate the algorithm if the classifier violates the conditions. The fulfillment of the prefix-disjointness conditions in Line 8 ensures that the algorithm keeps iterating until $\mathcal{K}'$ becomes an N-rule classifier. Each iteration implements a transformation of $\mathcal{K}'$ that preserves its order independence and prefix reorderability in accordance with Lemma 1. Hence, if the input classifier is prefix-disjoint, PDTRANSFORM correctly proclaims its prefix disjointness and returns $\mathcal{K}'$ as a prefix-reorderable equivalent. □

The computational complexity of the PDTRANSFORM algorithm is $O(w \cdot N)$, i.e., linear in the number of characters that the input classifier has in its filters. This complexity arises in Lines 11 and 12 that update array h and consume per-iteration time $O(|\,\mathcal{E}(\mathcal{K}) \setminus \mathcal{E}_{\text{prev}}| \cdot N)$ as the algorithm iterates to cover all w columns of the filters.

IV. PREDICAT

While the PDTRANSFORM algorithm that converts a prefix-disjoint classifier to its prefix-reorderable equivalent constitutes the most innovative contribution of our paper, this section builds on PDTRANSFORM to design PREDICAT, a method that transforms an arbitrary ternary classifier to an equivalent prefix representation. PREDICAT combines PDTRANSFORM with various existing techniques.

Given a ternary classifier that is not prefix-disjoint, PREDICAT splits this classifier $\mathcal{K}$ into vector $\vec{\mathcal{K}}$ of M prefix-disjoint groups of rules, as Section IV-A discusses in more detail later. For each group $\mathcal{K}_z$ in $\vec{\mathcal{K}}$, PREDICAT uses PDTRANSFORM to construct its prefix-reorderable equivalent $\mathcal{K}'_z$. Then, PREDICAT explores and exploits the possibility that each $\mathcal{K}'_z$ contains more filter columns than necessary to preserve its order independence and prefix reorderability. At this stage, PREDICAT transforms every $\mathcal{K}'_z$ to equivalent classifier $\mathcal{K}''_z$ with narrower filters, as we discuss later in Section IV-B. Finally, PREDICAT converts each $\mathcal{K}''_z$ to equivalent prefix classifier $\mathcal{K}'''_z$ by permuting the filter columns in $\mathcal{K}''_z$.

Example 6. *PREDICAT transformation of ternary classifier $\mathcal{K}$, which is not prefix-disjoint, to an equivalent set of prefix classifiers:*

$\mathcal{K}$	f_1	f_2	f_3	f_4	*Action*
R_1	0	0	0	1	A_1
R_2	0	0	1	$*$	A_2
R_3	1	0	$*$	$*$	A_3
R_4	$*$	1	1	0	A_4
R_5	$*$	$*$	0	0	A_5
R_6	$*$	$*$	1	0	A_6
R_7	$*$	0	$*$	1	A_7

(1)

$\mathcal{K}_1$	f_1	f_2	f_3	f_4	*Action*
R_1	0	0	0	1	A_1
R_2	0	0	1	$*$	A_2
R_3	1	0	$*$	$*$	A_3
R_4	$*$	1	1	0	A_4

$\mathcal{K}_2$	f_1	f_2	f_3	f_4	*Action*
R_5	$*$	$*$	0	0	A_5
R_6	$*$	$*$	1	0	A_6
R_7	$*$	0	$*$	1	A_7

(2)

$\mathcal{K}'_1$	f_1	f_2	f_3	f_4	*Action*
R'_1	0	0	0	1	$A_1\|C$
R'_2	0	0	1	$*$	$A_2\|C$
R'_3	1	0	$*$	$*$	$A_3\|C$
R'_4	$*$	1	$\circledast$	$\circledast$	$A_4\|C$

$\mathcal{K}'_2$	f_1	f_2	f_3	f_4	*Action*
R'_5	$*$	$*$	0	0	$A_5\|C$
R'_6	$*$	$*$	1	0	$A_6\|C$
R'_7	$*$	$\circledast$	$*$	1	$A_7\|C$

(3)

$\mathcal{K}''_1$	f_1	f_2	f_3	*Action*
R''_1	0	0	0	$A_1\|C$
R''_2	0	0	1	$A_2\|C$
R''_3	1	0	$*$	$A_3\|C$
R''_4	$*$	1	$\circledast$	$A_4\|C$

$\mathcal{K}''_2$	f_3	f_4	*Action*
R''_5	0	0	$A_5\|C$
R''_6	1	0	$A_6\|C$
R''_7	$*$	1	$A_7\|C$

(4)

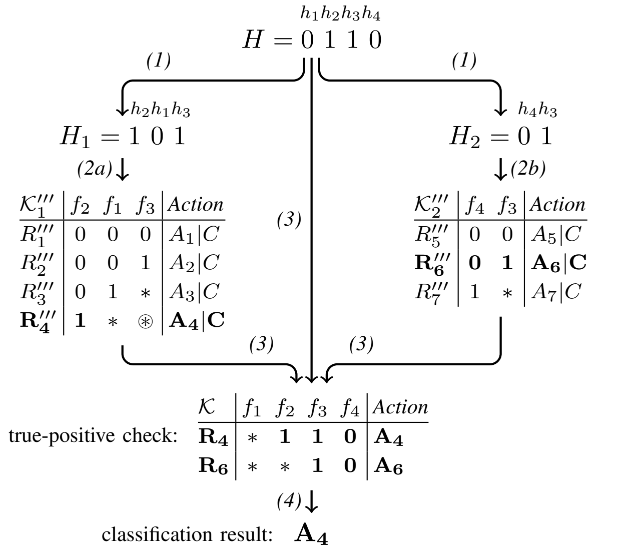
$\mathcal{K}'''_1$	f_2	f_1	f_3	*Action*
R'''_1	0	0	0	$A_1\|C$
R'''_2	0	0	1	$A_2\|C$
R'''_3	0	1	$*$	$A_3\|C$
R'''_4	1	$*$	$\circledast$	$A_4\|C$

$\mathcal{K}'''_2$	f_4	f_3	*Action*
R'''_5	0	0	$A_5\|C$
R'''_6	0	1	$A_6\|C$
R'''_7	1	$*$	$A_7\|C$

Example 6 illustrates how PREDICAT operates on ternary classifier $\mathcal{K}$ that is not prefix-disjoint. In Step 1, PREDICAT partitions $\mathcal{K}$ into prefix-disjoint groups $\mathcal{K}_1$ and $\mathcal{K}_2$. The darker ■ and lighter □ shaded areas highlight changes related to $\mathcal{K}_1$ and $\mathcal{K}_2$ respectively. In Step 2, our method converts $\mathcal{K}_1$ and $\mathcal{K}_2$ to prefix-reorderable equivalents $\mathcal{K}'_1$ and $\mathcal{K}'_2$ respectively. In Step 3, PREDICAT constructs prefix-reorderable order-independent classifiers $\mathcal{K}''_1$ and $\mathcal{K}''_2$ via respective removal of filter column f_4 from $\mathcal{K}'_1$ and columns f_1 and f_2 from $\mathcal{K}'_2$. In Step 4, the method transforms $\mathcal{K}''_1$ and $\mathcal{K}''_2$ to equivalent prefix classifiers $\mathcal{K}'''_1$ and $\mathcal{K}'''_2$ respectively by permuting the filter columns.

With classifier $\mathcal{K}$ represented as M prefix classifiers $\mathcal{K}'''_z$, the classification of a packet with header H on this multi-group representation uses the following procedure: (i) for each prefix classifier $\mathcal{K}'''_z$, construct header H_z that contains the bits of H corresponding to the filter columns in $\mathcal{K}'''_z$; (ii) classify each H_z on $\mathcal{K}'''_z$ and, for any of up to M matched rules overall, perform the true-positive check for H against respective rule R_i in $\mathcal{K}$; (iii) among all the rules that passed the true-positive check, return the action of the highest priority rule.

Example 7. *Classification of a packet with header H on $\mathcal{K}_1'''$ and $\mathcal{K}_2'''$, a multi-group representation of classifier $\mathcal{K}$:*



$\mathcal{K}_1'''$	f_2	f_1	f_3	*Action*
R_1'''	0	0	0	$A_1 \mid C$
R_2'''	0	0	1	$A_2 \mid C$
R_3'''	0	1	$*$	$A_3 \mid C$
$\mathbf{R_4'''}$	**1**	$*$	$\circledast$	$\mathbf{A_4 \mid C}$

$\mathcal{K}_2'''$	f_4	f_3	*Action*
R_5'''	0	0	$A_5 \mid C$
$\mathbf{R_6'''}$	**0**	**1**	$\mathbf{A_6 \mid C}$
R_7'''	1	$*$	$A_7 \mid C$

$\mathcal{K}$	f_1	f_2	f_3	f_4	*Action*
$\mathbf{R_4}$	$*$	**1**	**1**	**0**	$\mathbf{A_4}$
$\mathbf{R_6}$	$*$	$*$	**1**	**0**	$\mathbf{A_6}$

Example 7 depicts how PREDICAT classifies a packet with header $H = h_1h_2h_3h_4 = 0110$ on the multi-group prefix representation constructed for classifier $\mathcal{K}$ in Example 6. In Step 1, PREDICAT constructs headers $H_1 = h_2h_1h_3 = 101$ and $H_2 = h_4h_3 = 01$. In Steps 2a and 2b, H_1 and H_2 match rules R_4''' and R_6''' in prefix classifiers $\mathcal{K}_1'''$ and $\mathcal{K}_2'''$ respectively. In Step 3, the true-positive checks for H against both rules R_4 and R_6 in $\mathcal{K}$ succeed. In Step 4, PREDICAT returns action A_4 as the classification result because rule R_4 has a higher priority than rule R_6.

A. Partition into multiple prefix-disjoint classifiers

Because each prefix-disjoint group in the multi-group representation of a classifier requires a separate lookup during classification, it is desirable to minimize the number of prefix-disjoint groups in the partition.

Problem 2 (PDSPLIT)**.** *Given a ternary classifier, partition it into as few prefix-disjoint groups as possible.*

The following theorem reveals that PDSPLIT is a computationally difficult problem.

Theorem 4. *The* PDSPLIT *problem is NP-hard.*

Proof. To prove the theorem, we reduce the set cover problem, which is NP-hard [20], to PDSPLIT. Consider set $\mathcal{U} = \{1 \ldots N\}$ and collection $\mathcal{S} = \{S_1, \ldots, S_M\}$ of sets that contain only elements from $\mathcal{U}$. The *cover* of $\mathcal{U}$ is collection $\mathcal{S}' \subseteq \mathcal{S}$ such that the union of all sets in $\mathcal{S}'$ equals $\mathcal{U}$. The objective in this problem is to find the cover of $\mathcal{U}$ with the minimum number of sets. For a given instance of the set cover problem, we construct classifier $\mathcal{K}$ with N rules where rule R_i corresponds to $i \in U$. Filter F_i consists of $M \cdot l$ characters, where $l = \lceil \log_2(N) \rceil$. For each set $S_j \in \mathcal{S}$ that contains i, the characters of F_i in positions $(j-1) \cdot l + 1, \ldots, j \cdot l$ form the binary representation of $i - 1$, and the other characters of F_i equal $*$, e.g., if N and M are equal to 4, and element 2 from U appears only in S_1 and S_3, filter F_2 is 01**01**. Now, consider partition $\vec{\mathcal{K}}$ of $\mathcal{K}$ into prefix-disjoint groups. Since each group $\mathcal{K}_z \in \vec{\mathcal{K}}$ is prefix-disjoint, $\mathcal{E}(\mathcal{K}_z) \neq \emptyset$. Position x in $\mathcal{E}(\mathcal{K}_z)$ corresponds to set $S_{\lceil (x-1)/l \rceil}$. R_i can belong to $\mathcal{K}_z$ only if $i \in S_{\lceil (x-1)/l \rceil}$. Thus, $\mathcal{K}_z$ defines a subset of $S_{\lceil (x-1)/l \rceil}$ and, since every rule belongs to exactly one group, $\vec{\mathcal{K}}$ represents a cover of $\mathcal{U}$ of the same size. Conversely, each set $S_j \in \mathcal{S}$ defines prefix-disjoint group $\mathcal{K}_j$ that contains R_i iff $i \in S_j$; each such $\mathcal{K}_j$ is prefix-disjoint because the filter columns in $\mathcal{E}(\mathcal{K}_j)$ ensure order independence for all rules in $\mathcal{K}_j$. Thus, each cover of $\mathcal{U}$ defines a partition of $\mathcal{K}$ into prefix-disjoint groups. Hence, a partition of $\mathcal{K}$ into the minimal number of prefix-disjoint groups defines a minimal set cover of $\mathcal{U}$. ☐

Algorithm 2 PDPARTITION($\mathcal{K}$)

```
1: $\vec{\mathcal{K}} \leftarrow ()$
2: while $\mathcal{K} \neq ()$ do
3:     $\mathcal{K}_{prev} \leftarrow \mathcal{K}$
4:     $\mathcal{K}_z \leftarrow ()$
5:     while $\mathcal{K} \neq ()$ do
6:         $R \leftarrow \mathrm{argmax}_{R \in \mathcal{K}} \, |\mathcal{E}(\mathcal{K}_z) \cap \mathrm{exact}(R)|$
7:         if PDTRANSFORM($\mathcal{K}_z \cup \{R\}$) = prefix-disjoint, $\mathcal{K}_z'$
8:             $\mathcal{K}_z \leftarrow \mathcal{K}_z \cup \{R\}$
9:         $\mathcal{K} \leftarrow \mathcal{K} \setminus \{R\}$
10:    $\vec{\mathcal{K}} \leftarrow \vec{\mathcal{K}} \cup \{\mathcal{K}_z\}$
11:    $\mathcal{K} \leftarrow \mathcal{K}_{prev} \setminus \mathcal{K}_z$
12: return $\vec{\mathcal{K}}$
```

Due to the NP-hardness result in Theorem 4, PREDICAT uses heuristics to find group partitions. Algorithm 2 reports its iterative greedy algorithm PDPARTITION that constructs one prefix-disjoint group per iteration so that to maximize the number of rules in this group. A straightforward way to build a single prefix-disjoint $\mathcal{K}_z$ group with PDTRANSFORM is by trying each rule in $\mathcal{K}$ one by one, checking each time if the initially empty $\mathcal{K}_z$ group is still prefix-disjoint when this rule is added, and, if it is, updating $\mathcal{K}_z$. The choice of the next tried rule is the main control knob in this general method. Clearly, a right choice can even produce an optimal solution for PDSPLIT. In algorithm PDPARTITION, we choose rule R_j with the largest size of $\mathcal{E}(\mathcal{K}_z \cup \{R_j\})$ to maximize the number of the filter columns that PDTRANSFORM($\mathcal{K}_z \cup \{R_j\}$) considers in its first iteration. The computational complexity of PDPARTITION is $O(w \cdot N^2)$ because the total number of rules in all groups of the constructed $\vec{\mathcal{K}}$ partition equals N, and Algorithm 2 checks for prefix disjointness in Line 7 for each rule with every group at most once.

In some scenarios, the number of groups constructed by PDPARTITION (or even by an optimal algorithm) can exceed the number of prefix lookups that the underlying LPM infrastructure supports at line rate. In such cases, PREDICAT turns to a *mixed representation* that uses TCAM or other non-LPM infrastructures for a relatively small portion of the rules, e.g., at most 5% of them. TCAM capabilities are not strictly necessarily, and a linear pass over the rules might be acceptable if there are only few non-prefix rules. Regardless, it is desirable to keep the non-prefix rule group as small as possible, which brings us to the following problem formulation:

Problem 3 (PDMIX). *Given ternary classifier $\mathcal{K}$ and $\beta \in \mathbb{N}$, find a partition of subset $\mathcal{K}'$ in $\mathcal{K}$ into M groups $\mathcal{K}_z$ such that M is at most β, each group $\mathcal{K}_z$ is prefix-disjoint, and $\mathcal{K} \setminus \mathcal{K}'$ contains as few rules as possible.*

Expectedly, the PDMIX problem is also NP-hard because we can reduce PDSPLIT to it via a binary search over β. Given the one-group-at-a-time nature of the PDPARTITION algorithm, it is straightforward to apply the algorithm to PDMIX: simply return the first β constructed groups as the result.

B. Reduction of the filter width

When packet classification relies on LPM infrastructures, the latter constrain not only the filter representation but also the filter width. Generally, narrower filters lead to smaller resource footprints and/or faster lookups. In practice, there may exist a hard constraint of either 32 characters (length of an IPv4 address) or 128 characters (length of an IPv6 address) on the filter width in LPM classification infrastructures due to their original designation for destination-based forwarding.

To reduce the filter width in prefix-reorderable equivalent $\mathcal{K}'_z$ of prefix-disjoint group $\mathcal{K}_z$, PREDICAT removes filter columns from $\mathcal{K}'_z$ iteratively while preserving order independence, with prefix reorderability preserved as a by-product. During each iteration, if the filter width of $\mathcal{K}'_z$ still exceeds the filter width supported by the underlying LPM infrastructure, and it is impossible to remove a filter column from $\mathcal{K}'_z$ without violation of order independence, PREDICAT removes a filter column in such a way that maximizes the size of the largest order-independent subset of $\mathcal{K}'_z$; in the case of a tie, PREDICAT removes the filter column with the largest number of wildcards. After removing a filter column, PREDICAT removes from $\mathcal{K}'_z$ each rule R'_i that does not belong to the largest order-independent subset of $\mathcal{K}'_z$ and also removes respective rule R_i from $\mathcal{K}_z$. All rules removed from $\mathcal{K}_z$ are assigned to subsequent groups. Note that the process described above may lead to situations where some rule is inserted to and removed from multiple groups. Thus, when PREDICAT performs filter-width reduction, its computational complexity becomes $O(p \cdot w \cdot N^2 + w^2 \cdot N)$ where p refers to the number of groups in the resulting representation.

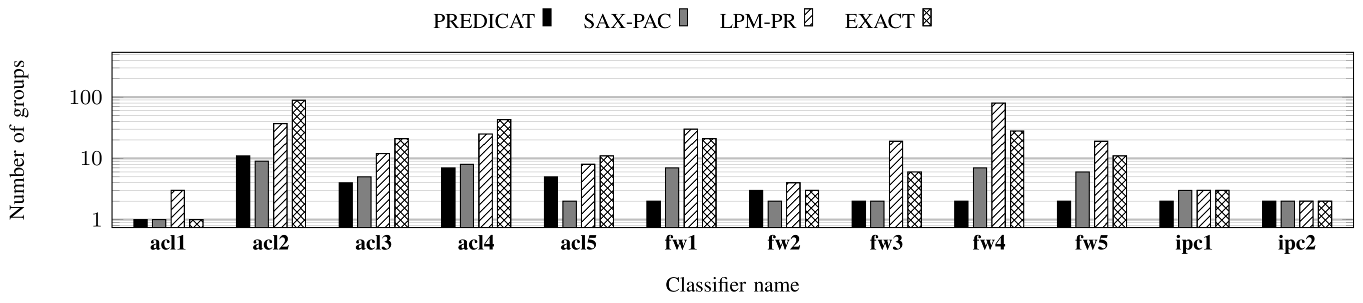
V. EVALUATION

To compare the proposed method with related work, we consider the following classifiers generated with realistic parameter values from the ClassBench benchmark: firewall classifiers **fw1** through **fw5**, access-control-list classifiers **acl1** through **acl5**, and IPchain classifiers **ipc1** and **ipc2** [21]. The classifier filters include fields for the source and destination ports represented as value ranges. We expand the ranges to ternary strings via the SRGE encoding scheme based on Gray coding [22]. Because the true-positive check can be performed efficiently for any original rule, we always consider the ternary rules from the same range expansion to be order-independent. Each range-expanded classifier in the set has from 50,000 to 280,000 rules with the filter width of 104 characters. Our code for the classifier transformation and evaluation is available at GitLab [23].

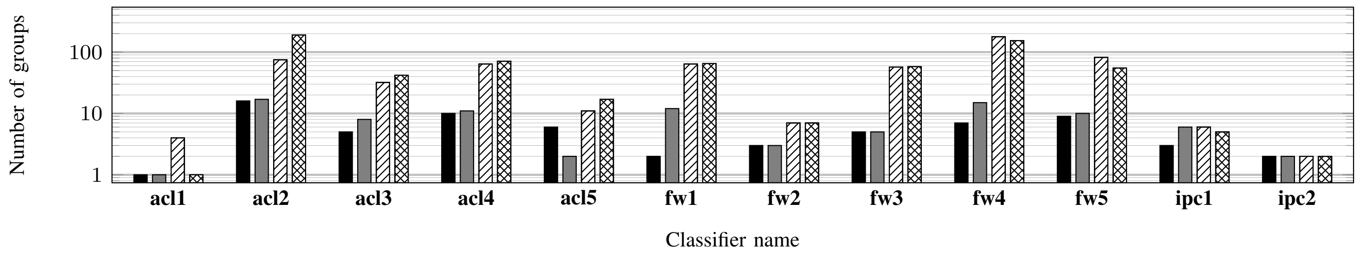
We evaluate PREDICAT against the following three alternatives: 1) LPM-PR that leverages prefix reorderability [10]; 2) EXACT that uses SAX-PAC restricted to binary representations without wildcards [8]; 3) SAX-PAC that leverages order independence and produces non-prefix ternary representations [9]. We evaluate PREDICAT, LPM-PR and SAX-PAC in two settings with: (a) filter width restricted to 32 characters in Figure 1 and (b) original filter width of 104 characters in Figure 2. For EXACT, we report only the full-width representations in both Figures 1 and 2 because exact binary classification is usually implemented with hash tables. The metric of interest is the number of groups necessary to represent the majority of rules in the classifier, namely 95%, 99%, and 100% of all rules. This metric ultimately reflects the number of lookups in mixed representations, where 5%, 1% and 0% of all rules are maintained by conventional methods and do not participate in partitions into groups.

Prefix disjointness vs. prefix reorderability. We start by evaluating the advantages of prefix disjointness over prefix reorderability, as represented by PREDICAT and LPM-PR respectively. For the filter width restricted to 32 characters, which is a more favorable case for prefix reorderability, Figure 1a shows that LPM-PR is hardly feasible already with 95% of the rules and requires more than 20 groups on average and 80 groups for **fw4**. In contrast, to represent the same 95%, PREDICAT never needs more than 11 groups in *all* cases, 3.5 groups on average (5× better than LPM-PR), and at most 2 groups for 7 of the 12 classifiers. Figure 1b demonstrates that PREDICAT is able to represent 99% of the rules using no more than 5 groups for 7 of the 12 classifiers and no more than 16 groups overall. In the same setting, LPM-PR requires more than 30 groups for a half of the classifiers and 178 groups in the worst case. Figure 1c reveals that while the 100% representation of the rules is challenging for both LPM-PR and PREDICAT, the results are again in favor of prefix disjointness, as LPM-PR needs 4× more groups than PREDICAT on average (140 groups vs. 29) and more than 100 groups for a majority of the classifiers.

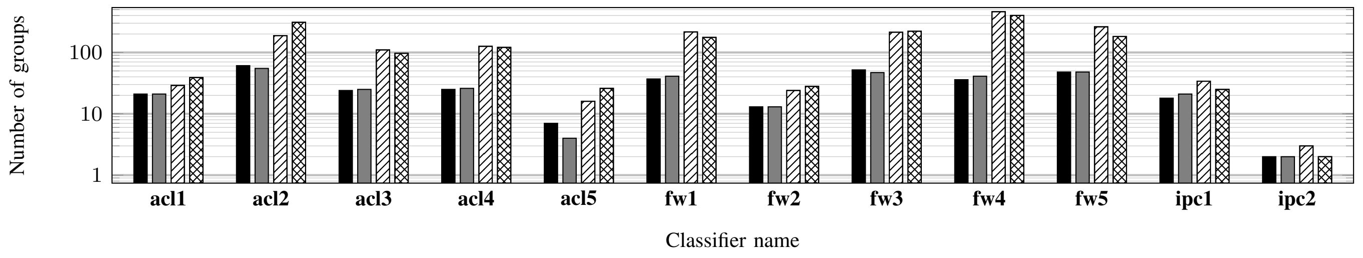
In the full-width settings, Figure 2 shows that the performance gap only increases. LPM-PR suffers a major performance degradation compared to the 32-character case: Figure 2a unveils a 2× increase in the number of groups from 20 to 40 on average for 95% of the rules. This is in line with our earlier observation that prefix reorderability deteriorates as the filter width increases. On the other hand, prefix disjointness only strengthens, and PREDICAT requires 20% less groups on average to represent both 95% and 99% of the rules, reducing the number to 2.8 and 4.8 groups respectively. Overall, we conclude that prefix disjointness is by far a more valuable property than prefix reorderability, and its advantage only



(a) Representing 95% of the rules in the classifier



(b) Representing 99% of the rules in the classifier



(c) Representing 100% of the rules in the classifier

Fig. 1: Number of groups needed by each of the approaches to represent a fixed fraction – 95%, 99%, or 100% – of the classifier rules when the filter width is restricted to 32 characters for PREDICAT, LPM-PR and SAX-PAC.

increases as the filter width grows.

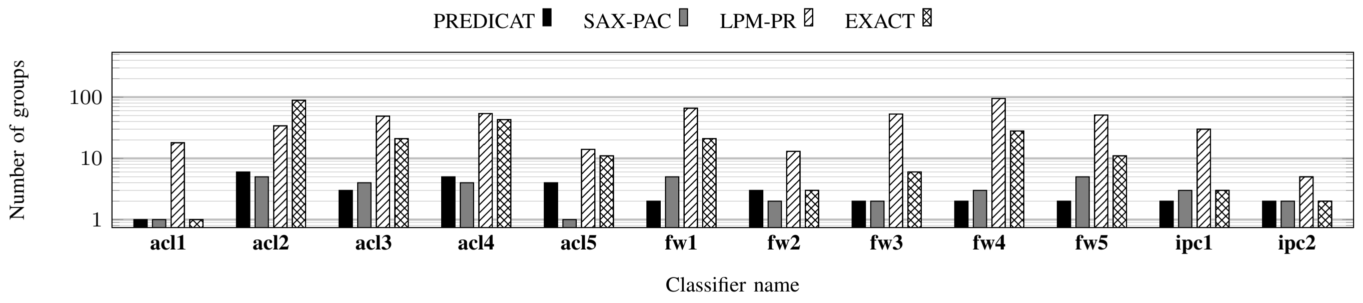
Prefix disjointness vs. exact representation. Next, we compare the prefix representations constructed by PREDICAT and binary representations provided by EXACT. The former are the only feasible ones for many classifiers. While we already showed that PREDICAT improves its performance as the filters become wider, Figure 1 depicts the performance of PREDICAT with 32-character filters vs. EXACT. Figure 1a reveals that EXACT needs more than 10 groups for 7 of the 12 classifiers and 90 groups for **acl2** in order to represent 95% of the rules. In relative terms, EXACT uses an *order of magnitude* more groups than PREDICAT for **acl2**, **acl3**, **fw1**, and **fw4**. For 99% of the rules, Figure 1b shows a performance gap of the same magnitude: on average, PREDICAT and EXACT require 5.8 groups and 55 groups respectively. For 100% of the rules, Figure 1c demonstrates that EXACT needs at least 100 groups for 7 of 12 the classifiers, which similar with LPM-PR, and 4× more groups than with PREDICAT. To sum up, although exact binary lookups might be easier to implement than prefix lookups, the observed order-of-magnitude gap in the number of groups gives PREDICAT a clear advantage over the ternary-to-binary transformation method.

Prefix disjointness vs. order independence. Lastly, we contrast prefix disjointness and order independence, as represented by PREDICAT with SAX-PAC respectively. Figures 1 and 2 show that, while the latter does not result in prefix-disjoint groups and is not suitable for prefix representations, the difference in the number of groups is marginal both with and without filter-width reduction: no more than 2 additional groups for PREDICAT on average. In several cases, e.g., **fw1** and **fw4** in Figure 2a, PREDICAT even needs slightly fewer groups than SAX-PAC, which is ultimately due to both approaches relying on greedy (suboptimal) heuristics.

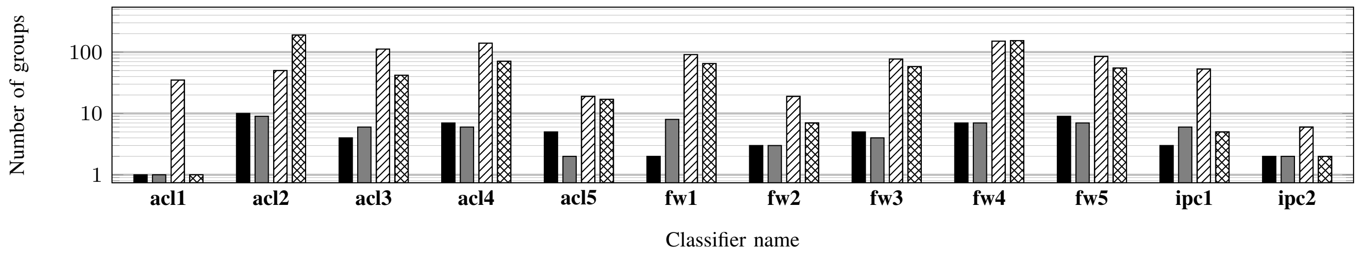
Overall, the evaluation shows that PREDICAT is able to produce prefix representations, while enjoying essentially the same performance as ternary-to-ternary SAX-PAC, and substantially outperforming LPM-PR and EXACT.

VI. Related Work

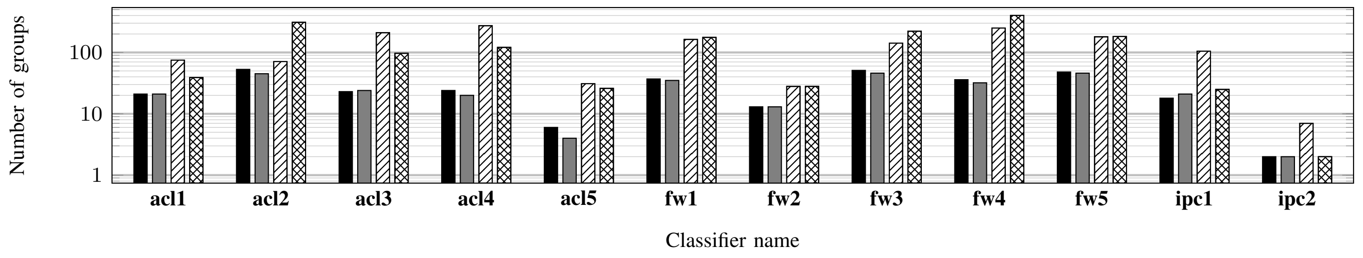
TCAM representations. To represent range-based fields of packet classifiers on TCAMs, ranges must be encoded as ternary strings. Several range-encoding methods have been



(a) Representing 95% of the rules in the classifier



(b) Representing 99% of the rules in the classifier



(c) Representing 100% of the rules in the classifier

Fig. 2: Number of groups needed by each of the approaches to represent a fixed fraction – 95%, 99%, or 100% – of the classifier rules when the filter width is unrestricted, i.e., at the maximum of 104 characters.

proposed for that purpose: prefix expansion encodes ranges as prefixes [24], SRGE enhances prefix expansion with Gray coding [22], DIRPE combines prefix expansion with fence encoding [25], RENE encodes a "short" range by a single ternary string [26] , and [27] provides optimal prefix encoding scheme for single-range classifiers. The number of rules in the resulting ternary classifiers can be further reduced by a variety of other methods: [28] removes redundant rules, [29] optimizes ternary representations of hierarchical packet classifiers, [30] applies block permutations to reduce TCAM consumptions, TCAM Razor [31] reduces the number of ternary rules using firewall decision diagrams, [9,32,33] exploit structural properties of packet classifiers to reduce the number of rules and filter width, [34,35] reduce approximate packet classifiers, [36] represents multiple classification policies by a single ternary classifier, and [37] constructs distributed classifier representations. All approaches for constructing and optimizing ternary packet classifiers can be combined with PREDICAT to represent their outputs on LPM infrastructure.

LPM classification. While PREDICAT transforms a ternary classifier to one or more prefix classifiers, many specialized methods deal with prefix classification: software solutions that exploit trie-based algorithms [38–41], Bloom filters [11,12], range representations [13,42], FPGA-based schemes [17,43,44], and schemes that accelerate IP lookups on GPUs [15,16]. In particular, SAIL [14] proposes fast IP classification mechanisms that can be implemented on different platforms including multicore CPU, FPGA, and GPU.

Structural properties of classifiers. The structural property of order independence is proposed by SAX-PAC to reduce the filter width [9]. [32] introduces the notions of action order independence and non-conflicting rules for the objective of reducing the number of filters and filter width even in classifiers that are not order-independent. [8] focuses on order-independent representations without wildcards, and [33] proposes an approach preserving order independence with the per-character resolution in range-based classifiers. [10,18] examine the chain criterion for prefix reorderability.

VII. Conclusion

This paper studied transformation of ternary classifiers to semantically equivalent prefix representations so that packet classification can benefit from efficient LPM solutions. We proposed the property of prefix disjointness and developed the PDTRANSFORM algorithm that tests for it. When the input classifier is prefix-disjoint, PDTRANSFORM uses wildcarding and true-positive checking to convert the classifier to an order-independent prefix-reorderable representation in time linear in the classifier size. Then, we designed PREDICAT, a method for transforming an arbitrary ternary classifier to an equivalent prefix representation. To achieve its goal, PREDICAT combines PDTRANSFORM with a variety of existing techniques such as classifier partitioning into multiple groups, filter-column permutation of prefix-reorderable classifiers, and filter-width reduction based on order independence. The evaluation on the ClassBench benchmark with respect to the number of lookups showed that PREDICAT compared favorably to existing transformation methods: PREDICAT outperformed a state-of-the-art ternary-to-prefix method by a factor from 2 to 5, improved on a ternary-to-binary solution by an order of magnitude, and preformed similarly to a ternary-to-ternary approach that required costly power-hungry TCAMs to efficiently handle the resulting ternary representation.

REFERENCES

[1] D. E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, 2005.

[2] G. Varghese, *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann, 2004.

[3] Open Networking Foundation, "OpenFlow Switch Specification, Version 1.5.1," 2015, https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[4] The P4 Language Consortium, "P4$_{16}$ Language Specification, Version 1.2.1," 2020, https://p4.org/p4-spec/docs/P4-16-v1.2.1.pdf.

[5] Synopsys, "DesignWare Ternary Content-Addressable Memory Compilers," 2021, https://www.synopsys.com/dw/ipdir.php?ds=dwc_tcam_memory_compilers.

[6] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *SIGCOMM*, 1999, pp. 147–160.

[7] M. H. Overmars and A. F. van der Stappen, "Range Searching and Point Location among Fat Objects," in *ESA*, 1994, pp. 240–253.

[8] 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.

[9] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (Scalable And eXpressive PAcket Classification)," in *SIGCOMM*, 2014, pp. 15–26.

[10] P. Chuprikov, K. Kogan, and S. I. Nikolenko, "General Ternary Bit Strings on Commodity Longest-Prefix-Match Infrastructures," in *ICNP*, 2017, pp. 1–10.

[11] H. Lim, K. Lim, N. Lee, and K. Park, "On Adding Bloom Filters to Longest Prefix Matching Algorithms," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 411–423, 2014.

[12] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *SIGCOMM*, 2003, p. 201–212.

[13] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a Billion Routing Lookups per Second in Software," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, p. 29–36, 2012.

[14] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP Lookup Performance with FIB Explosion," in *SIGCOMM*, 2014, p. 39–50.

[15] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-Accelerated Software Router," in *SIGCOMM*, 2010, p. 195–206.

[16] J. Zhao, X. Zhang, X. Wang, and X. Xue, "Achieving O(1) IP Lookup on GPU-Based Software Routers," in *SIGCOMM*, 2010, p. 429–430.

[17] H. Fadishei, M. S. Zamani, and M. Sabaei, "A Novel Reconfigurable Hardware Architecture for IP Address Lookup," in *ANCS*, 2005, pp. 81–90.

[18] C. R. Meiners, A. X. Liu, and E. Torng, "Bit Weaving: A Non-prefix Approach to Compressing Packet Classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 488–500, 2012.

[19] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a Sparse Table with 0(1) Worst Case Access Time," *Journal of the ACM*, vol. 31, no. 3, p. 538–544, 1984.

[20] R. M. Karp, *Reducibility among Combinatorial Problems*. Springer US, 1972, pp. 85–103.

[21] Applied Research Laboratory, "ClassBench: A Packet Classification Benchmark," http://www.arl.wustl.edu/classbench/.

[22] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, 2012.

[23] P. Chuprikov, "PREDICAT: Efficient Packet Classification via Prefix Disjointness (Code for Evaluation)," 2021, https://gitlab.com/pschuprikov/predicat.

[24] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," in *SIGCOMM*, 1998, pp. 191–202.

[25] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," in *SIGCOMM*, 2005, p. 193–204.

[26] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Hel-Or, "Encoding Short Ranges in TCAM Without Expansion: Efficient Algorithm and Applications," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 835–850, 2018.

[27] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On Finding an Optimal TCAM Encoding Scheme for Packet Classification," in *INFOCOM*, 2013, pp. 2049–2057.

[28] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-Match Based Complete Redundancy Removal for Packet Classifiers in TCAMs," in *INFOCOM*, 2008, pp. 574–582.

[29] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and Speed Tradeoffs in TCAM Hierarchical Packet Classification," *Journal of Computer and System Sciences*, vol. 79, no. 1, pp. 111–121, 2013.

[30] R. Wei, Y. Xu, and H. J. Chao, "Block Permutations in Boolean Space to Minimize TCAM for Packet Classification," in *INFOCOM Mini-Conference*, 2012, pp. 2561–2565.

[31] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 490–500, 2010.

[32] K. Kogan, S. I. Nikolenko, P. T. Eugster, A. Shalimov, and O. Rottenstreich, "FIB Efficiency in Distributed Platforms," in *ICNP*, 2016, pp. 1–10.

[33] V. Demianiuk and K. Kogan, "How to Deal with Range-Based Packet Classifiers," in *SOSR*, 2019, p. 29–35.

[34] V. Demianiuk, K. Kogan, and S. Nikolenko, "Approximate Packet Classifiers With Controlled Accuracy," *IEEE/ACM Transactions on Networking*, accepted for publication, 2021.

[35] O. Rottenstreich and J. Tapolcai, "Lossy Compression of Packet Classifiers," in *ANCS*, 2015, p. 39–50.

[36] V. Demianiuk, S. Nikolenko, P. Chuprikov, and K. Kogan, "New Alternatives to Optimize Policy Classifiers," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1088–1101, 2020.

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

[38] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, p. 97–122, 2004.

[39] H. Lim, C. Yim, and E. E. Swartzlander, "Priority Tries for IP Address Lookup," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 784–794, 2010.

[40] M. Bando and H. J. Chao, "FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps," in *INFOCOM*, 2010, pp. 1–9.

[41] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond," in *SIGCOMM*, 2013, p. 111–122.

[42] S. Suri, G. Varghese, and P. R. Warkhede, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," in *GLOBECOM*, 2001, pp. 1610–1614.

[43] N. Futamura, R. Sangireddy, S. Aluru, and A. K. Somani, "Scalable, Memory Efficient, High-speed Lookup and Update Algorithms for IP Routing," in *ICCCN*, 2003, pp. 257–263.

[44] D. Pao, Z. Lu, and Y. H. Poon, “IP Address Lookup using Bit-Shuffled Trie,” *Computer Communications*, vol. 47, pp. 51–64, 2014.