ClassBench-ng: Benchmarking Packet Classification Algorithms in the OpenFlow Era

Jiří Matoušek, Adam Lučanský, David Janeček, Jozef Sabo, Jan Kořenek, Gianni Antichi

***Abstract*—Packet classification, i.e., the process of categorizing packets into flows, is a first-class citizen in any networking device. Every time a new packet has to be processed, one or more header fields need to be compared against a set of pre-installed rules. This is done for basic forwarding operations, to apply security policies, application-specific processing, or quality-of-service guarantees. A lot of research efforts have identified better lookup techniques, i.e., finding the best match between packet headers and rules, by capitalizing on the rule sets characteristics. Here, ClassBench has greatly served the community by enabling the generation of IPv4 rule sets. In this paper, we present a new tool, ClassBench-ng, that creates synthetic IPv4, IPv6, and OpenFlow rules. We start from an analysis of classification rules deployed in-the-wild and we use the findings to craft our solution. ClassBench-ng can generate a user-defined number of rules as well as an associated header trace matching them. Compared to state-of-the-art solutions, the rule set generation process is usually more accurate and it is able to produce rules matching a number of different use cases, i.e., from an IPv4 router to an OpenFlow switch, which is unique among current rule set generation tools.**

***Index Terms*—ClassBench, Packet Classification, OpenFlow, IPv4, IPv6, Synthetic Rules.**

I. Introduction

Every networking device, no matter its purpose, capacity, or layer of operation, shares a common functionality: packet classification. As a new packet arrives, one or more header fields need to be compared against a set of pre-defined rules to assign a flow identifier. This is used for basic forwarding operations, to apply security policies, application-specific processing, or quality-of-service guarantees.

The continuous innovation in computer networks with the advent of IPv6 first, and Software Defined Networking (SDN)/OpenFlow [2] later, alongside a constant increase of link capacities, has repeatedly challenged state-of-the-art packet classification algorithms. In particular, the IPv6 protocol, which is not anymore an uninteresting rarity [3], [4], quadrupled the size of IP addresses, making the lookup process more complex than the IPv4 case. Moreover, SDN/OpenFlow have extendend the matching criteria to multiple fields from different layers. This has gathered the interest of many operators spanning from Internet eXchange Points (IXP) [5] to Wide Area Networks (WAN) [6], [7], [8] and data center networks [9]. All those aspects have contributed in renewing the interests of researchers and practitioners towards developing new packet classification algorithms [10], [11], [12], [13], [14] that could better cope with the stringent needs of *fast lookup* and *matching complexity*.

A number of research efforts have identified better packet classification techniques by leveraging the characteristics of real rule sets [15], [16], [17]. Additionally, it has also been demonstrated that the capacity and efficiency of the most prominent hardware-based packet classification solution, Ternary Content Addressable Memories (TCAM), are also subject to the characteristics of rule sets [10]. So far, the lack of publicly available rule sets has been mitigated by a number of synthetic rule generators [18], [19], [20]. However, they either focus on one specific case, i.e., IPv4 [18], IPv6 [20], or they have been designed to be generic at the cost of not following any specific real rule set characteristics [19].

In this paper, we present ClassBench-ng, a new open source tool that can generate synthetic IPv4, IPv6, and OpenFlow rule sets alongside an associated header trace matching them. ClassBench-ng accepts as an input a description of statistical properties for the rule set to be generated. In this context, we analyzed a number of classification rule sets taken from IPv4/IPv6 backbone routers and OpenFlow switches from a cloud data center provider. Our analysis is then used to build appropriate input configuration files for ClassBench-ng. Finally, to make this solution attractive in the long term and for a wide number of different use cases, we propose a mechanic to create input parameter files from real rule sets. We aim to use the tool's repository as a place where researchers and operators can continuously upload new parameter files that match a number of different environments or use cases, e.g., a data center, Internet Service Provider, or Internet eXchange Point. This will further increase the impact of ClassBench-ng on the research community.

The main contributions of the paper can be summarized as follows:

- An in-depth analysis of in-the-wild classification rule sets from IPv4/IPv6 backbone routers and OpenFlow switches.
- A new tool that is able to analyze and generate IPv4, IPv6, and OpenFlow rule sets with an associated header trace matching them.
- The tool is open and available to anyone at: `https://classbench-ng.github.io/`.

The rest of the paper is organized as follows. We first concentrate on research questions related to rule set and

J. Matoušek and J. Kořenek are with Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations, CZ

A. Lučanský is with Mendel University in Brno, Faculty of AgriSciences, CZ

D. Janeček and J. Sabo are with Brno University of Technology, Faculty of Information Technology, CZ

G. Antichi is with Queen Mary University of London, School of Electronic Engineering and Computer Science, UK

trace generation (Section II). We then present an analysis of real IPv4, IPv6, and OpenFlow data sets (Section III), alongside the ClassBench-ng architecture (Section IV) and the experimental evaluation (Section V). Finally, we cover related works (Section VI) and conclude the paper (Section VII).

II. Research Questions

In this section, we discuss the research questions driving the ClassBench-ng design.

A. Rule Generation

The synthetic rule generation process transforms input parameters[1] into a complete rule set. Available tools use as an input either statistic distributions of real sets [18] or user-defined characteristics [19]. It is clear that the former is better when it is needed an output whose attributes are as close as possible to a real case. This brings two specific challenges: (1) *what shall be included into a seed?* and (2) *how to produce a rule set that reliably follow the input seed?*

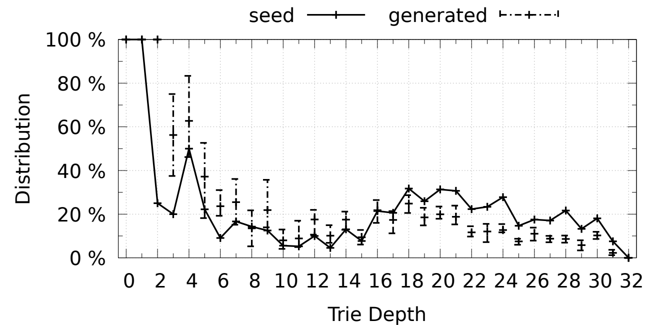
Past research, i.e., ClassBench [18], has already tackled both aspects in the context of IPv4 5-tuples. While the representation of layer four ports and protocol was designed as quite simple, source/destination IP prefixes required more sophisticated approach. The idea was to represent an IP prefix set as a binary prefix tree, i.e., trie, that can be characterized with four statistical parameters: a prefix length distribution, a branching probability distribution, an average skew distribution, and a prefix nesting threshold [18]. The prefix length distribution characterizes the span of prefixes. The branching probability distribution represents the probability, at each trie level, of having one-child or two-children nodes. Skew is instead defined in Equation 1.

$$skew = 1 - \frac{weight(lighter)}{weight(heavier)} \quad (1)$$

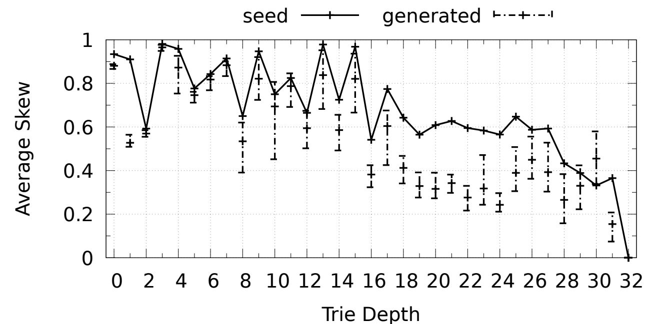
$weight()$ function returns the number of prefixes in a specified subtree and $lighter/heavier$ represent subtrees of a two-children node with smaller/higher number of prefixes, respectively. Finally, the prefix nesting threshold specifies the maximum number of prefix nodes that appear on an arbitrary path from the root to the leaves. We follow the same representation also in this paper and we look for a way ***how to extend ClassBench seeds in order to support IPv6 and OpenFlow rule set generation.***

Once this is done, it is possible to build a rule generator. In this regard, we performed a test campaign to better understand ClassBench internals and evaluate its fidelity. Since ClassBench has demonstrated an accurate generation of layer four ports and protocol, Figures 1 compare selected IP prefix set parameters extracted from an input IPv4 seed and from rules generated by original ClassBench, along with their error bars. While the prefix nesting threshold precisely follow the required distribution, the other parameters do not. Indeed, the generated branching probability meets the requirements only for 13 trie levels, while the average skew only for 5. We believe

[1] We call them seeds.



(a) Branching probability distribution (two-children nodes).



(b) Average skew distribution.

Fig. 1: Comparison of destination prefix set parameters from the `acl4` seed and rule sets generated from this seed using original ClassBench (target size was set according to the seed). The parameters of the generated sets are represented by average, minimum, and maximum values of 10 sets.

that such errors are caused by parameters interdependence: once a parameter with the highest priority has been fixed, the tool tries to meet the other requirements. The prefix nesting threshold has the highest priority, thus justifying its accuracy. In this paper, we build upon this and we look into ***how to improve rule generation fidelity while allowing the increased number of input parameters needed to support OpenFlow.***

B. Trace Generation

Just a rule set would be sufficient for a basic analysis of classification algorithm's memory footprint, i.e., the size of its memory representation, and performance, i.e., the number of memory accessess required per lookup. However, it does not allow to easily test its correctness. In practice, a single input might trigger a match of many rules, but only the one with the highest priority or with the most specified fields shall be selected. To perform this analysis, it is important to generate a trace associated to a given rule set. This brings one specific challenge: *given a rule set, how to efficiently generate a trace with the minimal number of headers, that will hit all the rules specified in the original set?*

Answering this question is trivial in case of non-overlapping rules: create one header per rule. However, in real scenarios, rules usually do overlap [21], [22], [23]. Therefore, the trace shall ideally contain headers that will match not only each rule individually, but also all their overlaps. Figure 2 shows

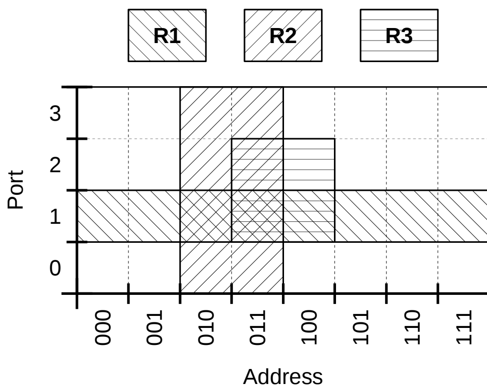


Fig. 2: Example set of 3 overlapping rules defined in two dimensions (3-bit Address and 2-bit Port).

one of many possible cases of overlaps within a set of three classification rules defined in dimensions Address (3 bits) and Port (2 bits). The full classification space can thus be divided into several, possibly discontinuous, subspaces that we call *regions*. For instance, the space in Figure 2 consists of 8 regions:

- 1 region of 0-rule overlap
- 3 regions of 1-rule overlap (R1, R2, R3)
- 3 regions of 2-rule overlap (R1+R2, R2+R3, R3+R1)
- 1 region of 3-rule overlap (R1+R2+R3)

An ideal trace generator should analyze the input rule set first and then generate a trace that will hit each region corresponding to a non-empty set of overlapping rules. Nevertheless, because of the number of classification dimensions, e.g., up to forty five in OpenFlow 1.5.1 [24], and their size, e.g., 2^{128} unique values in case of IPv6 address, a timely analysis might be unfeasible. According to our experiments, a real OpenFlow 1.0.0 rule set comprising approximately 20 k rules may define more than 2.14×10^{17} 12-dimensional elementary subspaces, which would the generator need to transform into regions before even starting the trace generation process itself. Header trace generators thus have to apply some heuristics, which allows to generate the trace reasonably fast and achieve a high coverage of non-empty regions at the same time [18]. Unfortunately, the heuristic utilized by the trace generator of original ClassBench allows to achieve a high coverage of individual rules only. As shown in Figure 3, its coverage of regions quickly decreases with an increasing complexity of classification rules. In an effort to overcome this limitation, in this paper, we answer the following question: ***how to guarantee a "good" coverage for regions regardless the complexity of an input rule set?***

III. Analysis of Real Classification Rules

This section provides an analysis of IP prefixes sets taken from core routers (Section III-A), classification rules obtained from access control lists (ACLs) applied at a university network's perimeter (Section III-B) and OpenFlow data sets coming from a set of Open vSwitches running in a cloud data center (Section III-C). Table I summarizes the data sets being used in the analysis.

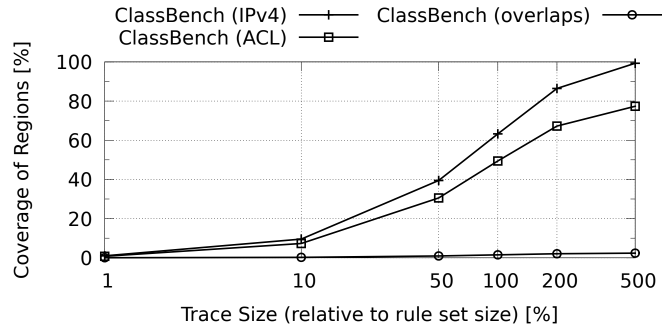


Fig. 3: Coverage of non-empty regions in rule sets of various type (IPv4 prefixes, ACL, a rule set with many overlaps) by header traces of different size generated using the trace generator of original ClassBench (average of 10 traces).

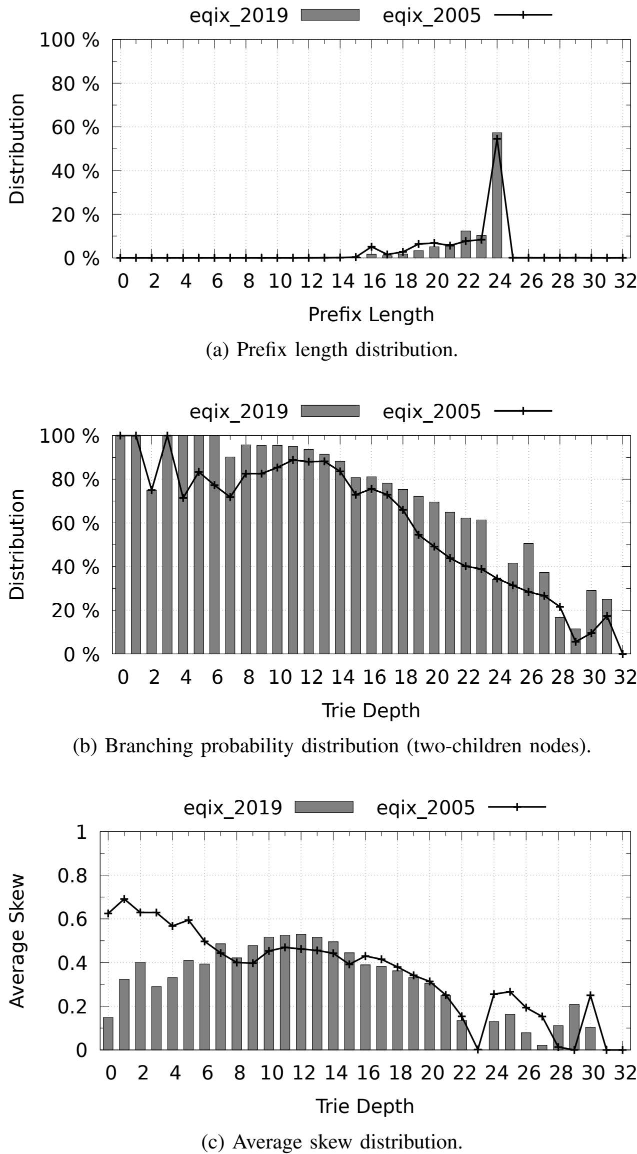
TABLE I: Utilized data sets. OpenFlow set `of3` exists in several instances, one for each day in the given interval.

Name	Prefixes or Rules	Source	Date
IPv4 Prefix Sets			
`eqix_2019`	787 885	Route Views [25]	2019-07-02
`eqix_2005`	164 455		2005-07-02
`rrc00_2019`	812 723	RIPE RIS [26]	2019-07-02
`rrc00_2005`	168 525		2005-07-02
IPv6 Prefix Sets			
`eqix_2019`	73 880		2019-07-02
`eqix_2017`	42 401	Route Views [25]	2017-07-02
`eqix_2005`	658		2005-07-02
`rrc00_2019`	75 008		2019-07-02
`rrc00_2017`	41 838	RIPE RIS [26]	2017-07-02
`rrc00_2005`	499		2005-07-02
ACL Rule Sets			
`uni_2010`	96	ACLs from	2010-08-30
`uni_2015`	122	university network	2015-01-14
OpenFlow Rule Sets			
`of1`	16 889		2015-05-29
`of2`	20 250	OpenFlow Switch	2015-05-29
	1 757	in a data center	2015-06-18
`of3`	to		to
	7 456		2015-07-14

A. IP Prefixes

1) IPv4: Figures 4 compare the same prefix set (`eqix`) in fourteen years time. While the prefix length distribution is almost the same between years 2005 and 2019 (Figure 4a), nowadays we are facing an increase of two-children nodes in the trie (Figure 4b) and the average skew is lower (Figure 4c). The prefix nesting threshold remained unchanged between 2005 and 2019. The same results are also confirmed in prefix sets `rrc00`. Growing number of two-children nodes and their smaller skew correlates with approximately 4.8 times higher number of prefixes after 14 years, as shown in Table I. Branching probability and average skew distributions follow the same trends and although the prefix sets grew in size, prefix length distribution is the same. These results are aligned with the path towards the saturation of IPv4 addresses [27].

2) IPv6: We propose for the IPv6 analysis the same statistical approach being used in the IPv4 context. Prefix sets are collected from the same core routers over a span of fourteen years.



(a) Prefix length distribution.

(b) Branching probability distribution (two-children nodes).

(c) Average skew distribution.

Fig. 4: Comparison between eqix IPv4 prefix sets in 2005 and 2019.

Figures 5 compare the selected parameters between the eqix prefix sets from years 2005 and 2019. Only the first 64 trie levels are shown as there were no IPv6 prefixes longer than 48 bits in 2005. Figure 5a shows that the prefix length distribution has changed significantly in the last 14 years. While prefix length 32 dominated the distribution in 2005, currently the most common prefix length is 48. This has affected both the branching probability distribution (Figure 5b) and the average skew distribution (Figure 5c). We believe that such a big difference in the prefix length distribution is related to the steady growth of IPv6 deployments, which is also indicated by an increased prefix nesting threshold. In 2005, most of the allocated prefixes belonged to Internet Service Providers (ISPs) or Regional Internet Registries (RIRs), while nowadays most of the prefixes belong to end users (organizations) [28]. Changes of branching probability and average skew between 2005 and 2019 have been also caused by the emergence of prefixes longer than 64 bits. Prefix sets rrc00 show similar behavior.

In 2005, both the eqix and rrc00 prefix sets contained only a few hundreds of IPv6 prefixes, while there are currently more than 73 thousands of prefixes in both sets (Table I). In this context, big changes over the parameter distributions, i.e., branching probability and average skew, are not surprising. However, if we compare parameter values over a shorter span (between 2017 and 2019), where the prefix length distribution is almost stable, the values of branching probability and average skew distributions follow similar trends. Note that the number of IPv6 prefixes in the eqix set almost doubled between 2017 and 2019.

TABLE II: Distribution of rules over protocol values.

Data Set	Protocol Values		
	wildcard	TCP	UDP
uni_2010	26.0 %	71.9 %	2.1 %
uni_2015	38.5 %	54.9 %	6.6 %

B. Ports and Protocol

The following analysis is performed using rule sets taken from ACLs in a university campus network (Table I). The data spans over a period of five years to enable a comparative analysis over time. We first concentrated on the distribution of rules over protocol values (Table II). The results show an increased number of rules specifying a wildcard or UDP, while the number of rules specifying TCP is decreasing. The ICMP protocol is not specified in the available rule sets at all.

Table III presents the distribution of rules over port classes, separately for source and destination port fields. The classes being used to describe port ranges are five [18]:

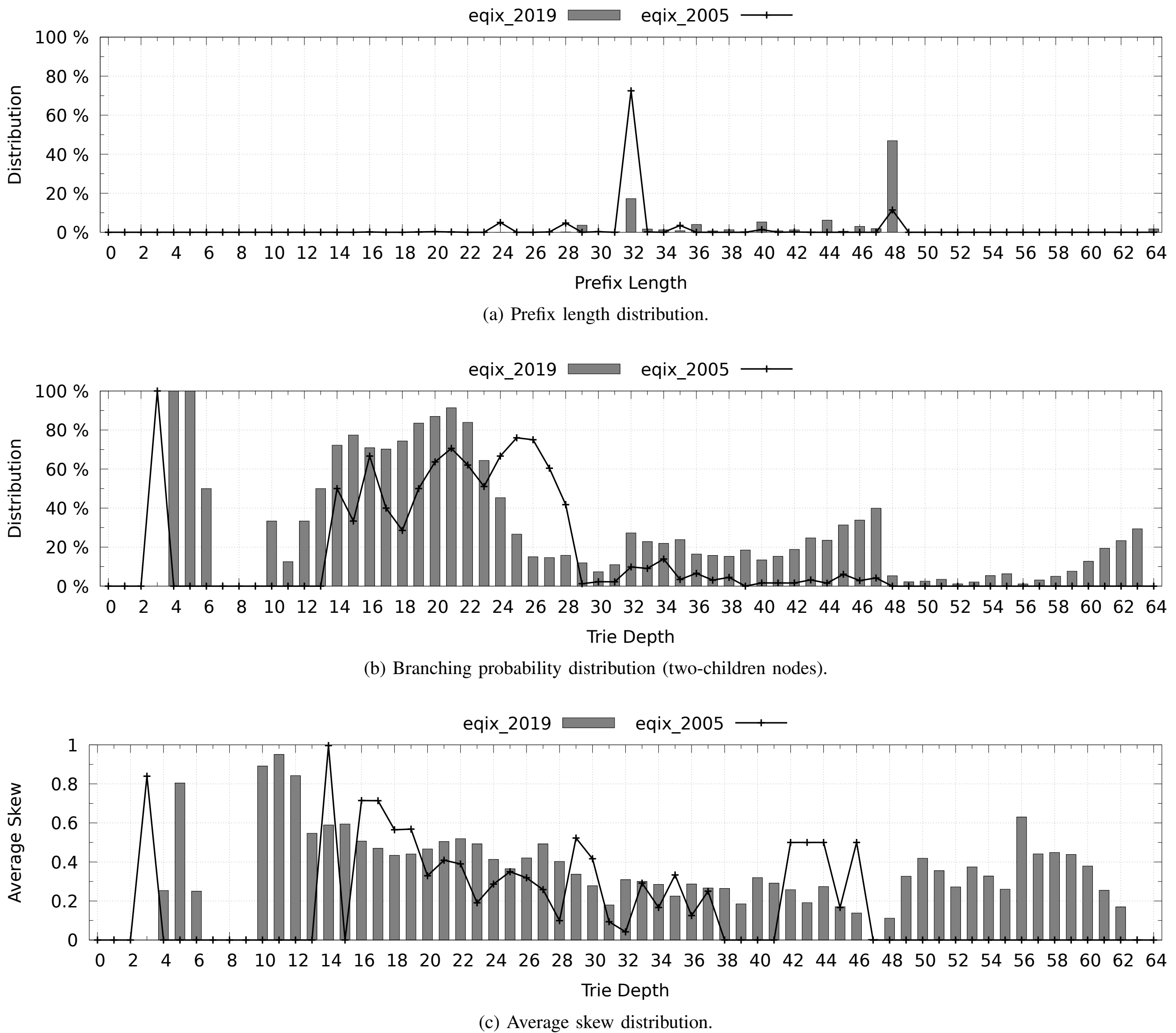
- *WC* — wildcard
- *HI* — user port range [1024 : 65535]
- *LO* — well-known system port range [0 : 1023]
- *AR* — arbitrary range
- *EM* — exact match

While the source port field is always treated with a wildcard, the destination shows an interesting property over the time. In particular, arbitrary range (*AR*) values and wildcard (*WC*) entries increase at the expenses of exact match (*EM*) ones.

TABLE III: Distribution of rules over port classes.

Data Set	Port Classes				
	WC	HI	LO	AR	EM
Source Port					
uni_2010	100.0 %	0.0 %	0.0 %	0.0 %	0.0 %
uni_2015	100.0 %	0.0 %	0.0 %	0.0 %	0.0 %
Destination Port					
uni_2010	26.0 %	0.0 %	0.0 %	5.2 %	68.8 %
uni_2015	38.5 %	0.0 %	0.0 %	8.2 %	53.3 %

Finally, we analyzed the distribution of rules over combined source-destination port pair classes (PPCs). Figures 6 and 7 are based on the uni_2015 data set and refer to TCP protocol and UDP protocol-based rules, respectively. The most common class pair being adopted in the TCP case is *WC-EM*, which represents rules specifying a wildcard for the source



(a) Prefix length distribution.

(b) Branching probability distribution (two-children nodes).

(c) Average skew distribution.

Fig. 5: Comparison between `eqix` IPv6 prefix sets in 2005 and 2019.

port and an exact value for the destination. The exact match values refer mostly to the SMTP protocol, widely used for e-mail transmission. On the other hand, the UDP case shows a big utilization of the *WC-AR* class pair. The rising of new applications and the massive usage of the RTP protocol-based solutions have led to specifically designed classification rules.

The analysis reported in this section shows that wildcard and TCP matching are commonly used in the protocol declaration. There is also an increasing usage of arbitrary ranges in the destination port field selection. As new applications arise, the need for arbitrary ranges become mandatory, thus justifying the obtained result.

C. OpenFlow

This section provides an analysis of real OpenFlow rule sets taken from a cloud data center in operation. We focused our study on understanding the statistical properties of OpenFlow-based rule sets as well as their temporal behavior. This is a once-in-a-lifetime opportunity to observe technological changes on such a grand scale, which is both practically and scientifically important. We first focus on a header fields distribution (Section III-C1). Then we moved our attention to fields dependency (Section III-C2) and rule set dynamics (Section III-C3).

1) Header Fields: OpenFlow 1.0.0 extends the standard 5-tuple, i.e., *ip_src*, *ip_dst*, *l4_src*, *l4_dst*, and *ip_proto*, with seven more header fields [29]. Figure 8 shows the header field distribution in rule sets `of1` and `of2` introduced in Table I. Fields from the standard 5-tuple present a non-wildcard value in at least 20 % of rules, while, except for *mac_dst* and *eth_type*, the others show a big predominance of wildcard entries. Moreover, header fields *vlan_id*, *vlan_prio*, and *ip_tos* are never specified. It is clear that in this case the network configuration plays a key role, i.e., virtual LANs are not enabled.

TABLE IV: Per-field count of unique values and associated *uniqueness factor* expressed in percentage (in parenthesis).

Rule Set	in_port	mac_src	mac_dst	eth_type	ip_proto	ip_src	ip_dst	l4_src	l4_dst
of1	123 (86.6)	27 (3.2)	593 (4.7)	1 (<0.1)	3 (0.3)	478 (4.6)	109 (0.9)	4 (2.9)	48 (2.2)
of2	140 (86.4)	19 (8.1)	791 (5.0)	1 (<0.1)	3 (0.1)	390 (2.8)	97 (0.7)	4 (<0.1)	8227 (92.7)

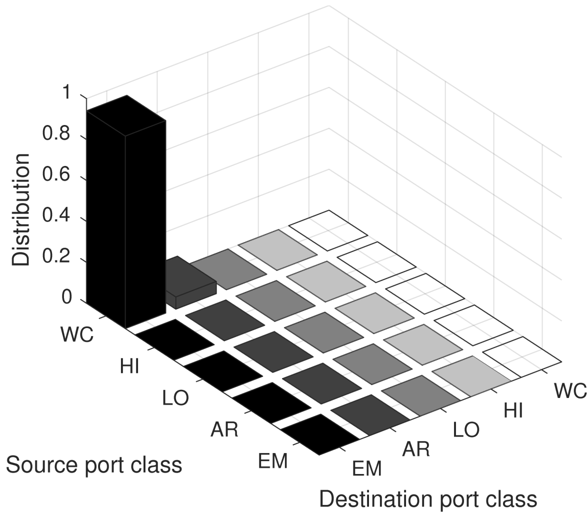


Fig. 6: PPC matrix for protocol TCP (rule set uni_2015).

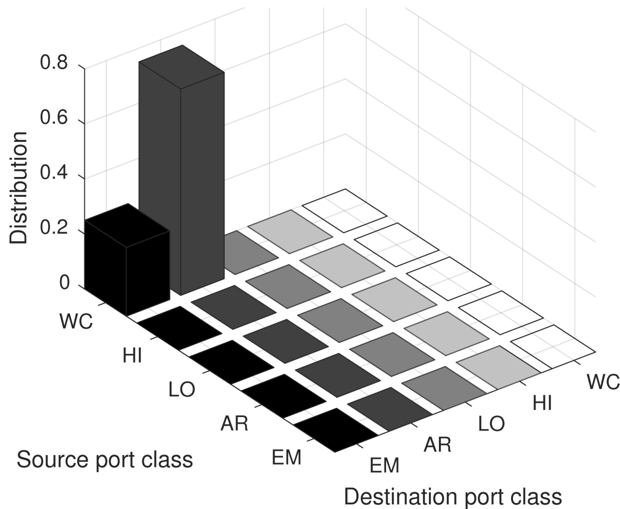


Fig. 7: PPC matrix for protocol UDP (rule set uni_2015).

Table IV shows a per-field count of unique values[2] being used in rule sets of1 and of2, alongside their *uniqueness factor* expressed in percentage. The factor estimates the per-field variance between rules. For instance, a value close to zero suggests little variance, i.e., rules specifying that field tend to use every time the same value, while a value close to one suggests the exact opposite. The *uniqueness factor* shows an interesting property of the of1 data set. While the *mac_dst* field has the highest number of unique values, its *uniqueness factor* is close to zero. In contrast, the *in_port* field has the highest *uniqueness factor*. Therefore, we can state that rules specifying a value for *in_port* are physical-port-oriented, i.e., the value of *in_port* represents the most important part of the rule. Things changes in the of2 data set. In this case, we can assert that rules specifying a value for the *l4_dst* field are application-oriented.

[2]*eth_type* presents just one value referred to the IPv4 type – 0x0800

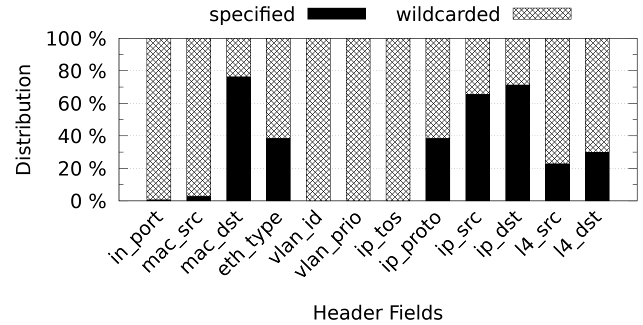


Fig. 8: Per-field distribution of rules from the combined of1+of2 rule set over specified and wildcarded classes.

Figure 9 shows the prefix length distribution for the *ip_src* field in data set of1. The most common prefix lengths are 0 (a wildcard rule), 10, and 32 (an exact match rule). Similar trends can also be seen for the *ip_dst* field of the of1 data set and both IP fields belonging to the of2 data set. The differences between the presented prefix length distribution and the one from Figure 4a are big. We justify this considering the nature of OpenFlow rules: they are not dictated by any routing protocol unless a given daemon is running on the top of the controller. In addition, the different environment (a core router for the previous study and a cloud data center for this one) plays an important role.

A further analysis of data sets of1 and of2 shows that the TCP protocol is specified only in 14.03 % of rules while 10.59 % of rules specify the ICMP protocol. Trends similar to what was shown in Section III-B can also be shown for the distribution of source/destination port values over five port classes and their combination into source-destination port pair

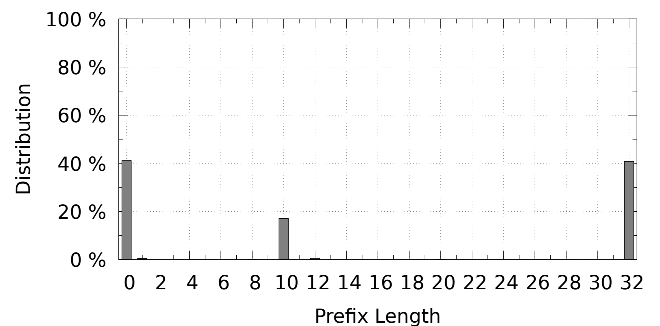


Fig. 9: Prefix length distribution of source prefixes from the of1 rule set.

classes.

2) Rule Types: In this section we provide an analysis of fields dependency. In particular, we characterize the relationship between header fields to study which fields are more likely to be specified together. Figure 10 shows the results of our analysis on the combined `of1+of2` rule set. We define *rule type* as a template that indicate which header fields are specified, i.e., have a non-wildcard value in a rule. To easier the graph representation, each *rule type* has been associated to a 12-bit number (*rule type number*) where each bit is referred to a given header field. The bit set to `1` stands for a specified field, while `0` for a wildcard. While it is clear that *rule type number* 0 refers to the combination of all header fields with a wildcard and 4 095 the exact opposite, it is important to define the bit-field correlation to correctly read the proposed graph. Starting from the most significant bit we used the following order: *in_port*, *mac_src*, *mac_dst*, *eth_type*, *vlan_id*, *vlan_prio*, *ip_tos*, *ip_proto*, *ip_src*, *ip_dst*, *l4_src*, and *l4_dst*. Given the proposed encoding scheme, *rule type number* 796 refers to rules where *mac_dst*, *eth_type*, *ip_proto*, *ip_src*, and *ip_dst* present specified values, while other fields a wildcard. Despite there are 4 096 possible *rule types*, the amount of *rule types* being used is much lower. In practice, our OpenFlow data sets `of1` and `of2` contain rules of 18 types only. Six of them are the most common and appear in more than 5 % of the cases.

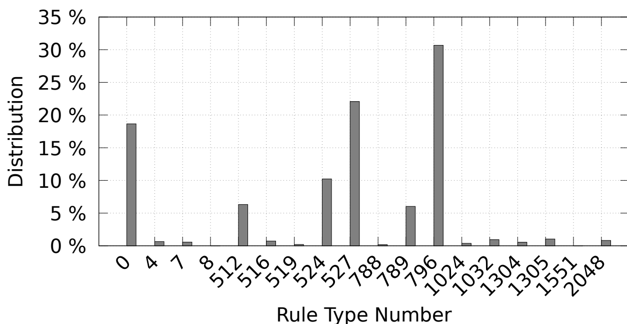


Fig. 10: Distribution of rules from the combined `of1+of2` rule set over *rule types*.

Figure 8 shows that *eth_type* and *ip_proto* are specified by the same number of rules. Moreover, *eth_type* is always defined as IPv4 (value `0x0800`) and it appears only in rules that define also *ip_proto* (note *rule types* 788, 789, 796, 1304, and 1305 in Figure 10). For the sake of analysis they can be considered redundant. Thus, *mac_dst* is the only OpenFlow header field that is specified in all the most common rule types.

3) Dynamics: Figure 11 shows the dynamics of rule set `of3` over a two-week period. We define the rate of changes as the size (cardinality) of symmetric difference divided by the size of union of `of3` in two subsequent days.

The studied data center environment has 220 physical hypervisors. The analysis has been performed exporting a flow table snapshot from the same hypervisor every day at the same time. Users creating/deleting virtual machines (VMs) or updating security profiles on any VM trigger a flow change. While the rate remains stable in June (not shown) and for the first week of July, it presents a spike on 7th July 2015.

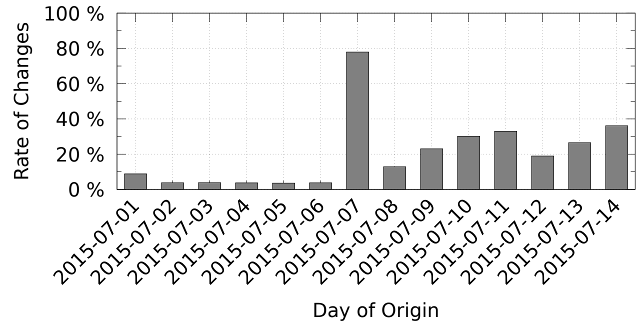


Fig. 11: Rate of changes (compared to the previous day) of rule set `of3` between 1st and 14th July 2015.

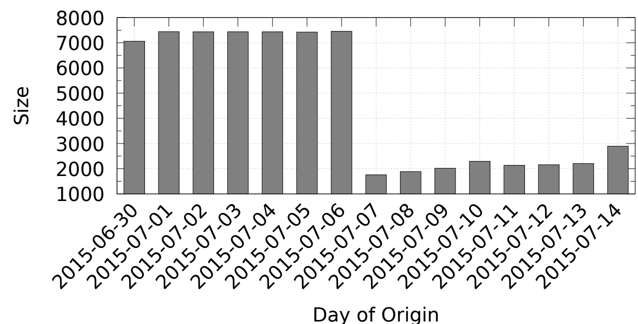


Fig. 12: Size of rule set `of3` between 30th June and 14th July 2015.

The behavior can be justified with Figure 12. On that day, in fact, the number of rules decreased drastically, thus creating the big spike in the rate of changes.

IV. CLASSBENCH-NG: NEXT GENERATION CLASSBENCH

This section discusses the design of ClassBench-ng. Figure 13 shows its high-level architecture composed by three main building blocks. The *Analyser* takes as input either IPv4/IPv6 5-tuples or OpenFlow rules to produce seeds that can be then used to feed the *Rule Generator* module in charge of producing synthetic rule sets. The *Trace Generator* instead produces a sequence of packet headers to exercise packet classification algorithms with respect to a given filter set.

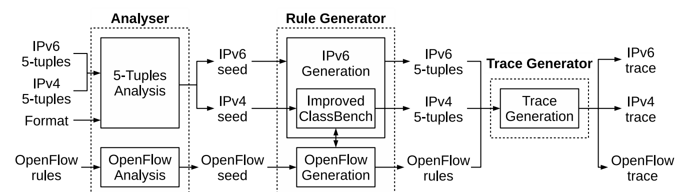


Fig. 13: High-level architecture of ClassBench-ng.

A. Analyser

ClassBench-ng already provides seeds for rule generation. However, we believe a seed generator is necessary to adapt the tool to a number of different scenarios, especially for

OpenFlow-enabled networks where the dynamics are bounded by applications running on top of a controller.

Both the 5-Tuples and OpenFlow Analysis blocks have been built from scratch. Analyser of 5-tuples parses either an IPv4 or IPv6 rule set according to a provided format description and produces a corresponding seed with the same structure as defined by original ClassBench. This analyser is therefore able to parse rule sets in common formats (e.g., those defined by `ipfw` and `iptables` tools) as well as any unusual and non-standard formats, as long as they can be described in the format file. On the other hand, OpenFlow analyser is able to correctly parse a rule set in the format used by the `ovs-ofctl` tool [30] and generate the appropriate OpenFlow 1.0 seed.

An OpenFlow seed is composed of three main elements: (1) a *rule type* distribution, (2) a 5-tuple seed, and (3) an OpenFlow-specific fields seed. The first provides an overview of fields dependency (as shown in Section III-C2) and the second supplies 5-tuple-related distributions. Finally, an OpenFlow-specific fields representation is based on the following types:

- *values* — a distribution over a set of original values;
- *parts* — a distribution over a set of the selected part of original values;
- *size* — a total number of unique original values;
- *null* — no representation.

The pairing between a type and a particular header field reflects different requirements. As an example, the *values* representation contains specific information from the original rule set. Therefore, it is appropriate only for fields that do not carry confidential data, i.e., *in_port* and *eth_type*. On the other hand, *null* and *size* representations do not include values from the original rule set, thus they are suitable for header fields carrying confidential content. The former (*null*) is used for header fields with a relatively small number of possible values, i.e., *vlan_prio* and *ip_tos*, while the latter (*size*) is used for header fields with a potentially big subset, i.e., *vlan_id*. Finally, *parts* represents a trade-off between *values* and *null*. ClassBench-ng uses this representation for the *mac_src* and *mac_dst* header fields, as it stores their vendor part in a seed.

B. Rule Generator

This module can successfully generate rule sets of various length given an input seed reflecting either IPv4, IPv6 or OpenFlow semantic. The first two capabilities have been built on top of the original ClassBench while improving its original fidelity. The OpenFlow rule generator has been created from scratch, instead.

1) IPv4 and IPv6: Our first insight is that it is possible to restructure the original ClassBench to generate both IPv4 and IPv6 rule sets. The main reason is that the construction mechanic adopted by ClassBench does not depend on specific IPv4 features and thus can be potentially extended to support larger fields, i.e., IPv6. To improve its generation fidelity (Section II-A) we designed a solution that iteratively build an output rule set with characteristics as close as possible to the input seed. The pseudocode in Figure 14 shows the process of rule set construction in our *Improved ClassBench* block. The tool first creates a big rule set using the original ClassBench application (line 3). Then it prunes the tries representing source and destination IP prefix sets to converge on a solution which is accurate and contain the target number of IP prefixes (lines 4, 5). The algorithm performing the trie pruning is described in Figure 15.

```
1: function ImprovedClassBench(seed, size)
2:     output_rules ← ∅
3:     rules ← ClassBench(seed, size · 100)
4:     src_trie ← TriePruning(rules.src_trie, seed, size, 4)
5:     dst_trie ← TriePruning(rules.dst_trie, seed, size, 4)
6:     max_match ← MaxBiMatch(src_trie, dst_trie, rules)
7:     for each rule ∈ max_match do
8:         output_rules ← output_rules ∪ {rule}
9:         rules ← rules \ {rule}
10:        RemovePrefix(src_trie, rule.src_prefix)
11:        RemovePrefix(dst_trie, rule.dst_prefix)
12:    for each dst_prefix ∈ dst_trie do
13:        if not TrieIsEmpty(src_trie) then
14:            rule ← SelectRule(rules, dst_prefix)
15:            rules ← rules \ {rule}
16:            src_prefix ← GetAnyPrefix(src_trie)
17:            RemovePrefix(src_trie, src_prefix)
18:            ReplaceSrcPrefix(rule, src_prefix)
19:            output_rules ← output_rules ∪ {rule}
20:    return output_rules
21: end function
```

Fig. 14: Pseudocode of rule set construction in *Improved ClassBench*.

The main idea behind *Improved ClassBench* is to select rules from the initial set, i.e., $rules$, that contain source/destination IP prefixes available also in the pruned tries, i.e., src_trie and dst_trie. To find these rules, the tool employs the maximum matching in a bipartite graph algorithm (line 6). The selected rules are added to the final set, i.e., $output_rules$, as shown in line 8. Every time a new rule is added, it is also removed from the initial set (line 9) and its source and destination prefixes are removed from the pruned tries as well (lines 10, 11). In case the maximum matching does not return the target number of rules, the last loop (line 12) creates the remaining rules by replacing a source prefix with an arbitrary prefix from src_trie (lines 14 to 18).

Figure 15 shows the pseudocode of the previously introduced trie pruning process. In addition to its parameters $trie$, $seed$ (target values of trie parameters are extracted from line 3 to 6), and $target_size$, parameter n is used to fix the number of iterations over the last two pruning steps. These iterations try to minimize the negative effect of the convergence over the target amount of prefixes on average skew. While each iteration decreases the number of prefixes in the trie by $\frac{1}{n} \cdot orig_size$ (line 13), the last iteration adjusts the number of prefixes to the target value ($target_size$ parameter), as shown in line 11.

Branching Probability Adjustment: This step (line 7) adjusts branching probability at each trie level (starting from the root of the trie) by removing a subtree of two-children nodes and then a subtree of one-child nodes. Sub-trees to be removed are selected increasingly according to the number of

```
1: function TriePruning(trie, seed, target_size, n)
2:     orig_size ← GetSize(trie)
3:     prefixes ← GetParam(seed, "prefix_length_distr")
4:     one_child ← GetParam(seed, "one_child_prob")
5:     two_children ← GetParam(seed, "two_children_prob")
6:     skew ← GetParam(seed, "skew_distr")
7:     AdjustBranching(trie, one_child, two_children)
8:     for each i ∈ [1, n] do
9:         AdjustSkew(trie, skew)
10:        if i = n then
11:            AdjustPrefixes(trie, prefixes, target_size)
12:        else
13:            AdjustPrefixes(trie, prefixes, (n−i)/n · orig_size)
14:    return trie
15: end function
```

Fig. 15: Pseudocode of trie pruning.

```
1: function OpenFlowGeneration(seed, size)
2:     of_rules ← ∅
3:     ipv4_5tuples ← ImprovedClassBench(seed, size)
4:     for each rule ∈ ipv4_5tuples do
5:         rule_type ← Generate(seed, "rule_type")
6:         for each field ∈ IPv4 5-tuple fields do
7:             if field ∉ rule_type then
8:                 Remove(rule, field)
9:         for each field ∈ OpenFlow-specific fields do
10:            if field ∈ rule_type then
11:                field_value ← Generate(seed, field)
12:                Add(rule, field_value)
13:        of_rules ← of_rules ∪ {rule}
14:    return of_rules
15: end function
```

Fig. 16: Pseudocode of OpenFlow rules generator.

prefixes they carry. Moreover, this step never removes the last branch with the maximum prefix nesting to not alter the prefix nesting threshold (already met by original ClassBench).

Average Skew Distribution Adjustment: This step (line 9) increases or decreases average skew at each trie level (starting from the leaves of the trie). In particular, it removes prefixes from the lighter or the heavier subtree of two-children nodes. As in the previous case, nodes are selected increasingly according to the total number of prefixes in their subtrees. This step does not remove the last prefix from the leaf nodes and it tries to not alter average skew when removing prefixes at already adjusted levels, i.e., levels below the current level.

Prefix Length Distribution and the Total Number of Prefixes Adjustment: This step (lines 11 and 13) removes prefixes at each trie level (starting from the root of the trie) to get their total number matching the target value. When removing the prefixes, the algorithm also tries to not alter the skew of two-children nodes; this is obtained by tracking the number of prefixes that should be removed from each subtree. Similarly to the average skew distribution adjustment, this step does not remove the last prefix from leaf nodes: doing so would imply the deletion of the whole branch, thus altering the branching probability.

2) OpenFlow Generation: The *OpenFlow Generation* block generates a set of OpenFlow rules from an input seed. Figure 16 shows the pseudocode of the generation process.

```
1: function TraceGen(rules, size, overlap_foc, par_a, par_b)
2:     gen_num ← NoReuseGen(rules, size)
3:     overlap ← ∅
4:     while gen_num < size do
5:         prim ← SelPrim(overlap_foc, overlap, rules)
6:         sec ← SelSec(prim, rules)
7:         hdr ← GenHdr(prim, sec)
8:         gen_num += PrintHdrs(hdr, par_a, par_b)
9:         if Overlap(prim, sec) then
10:            overlap ← overlap ∪ {prim, sec}
11:            mrg ← MergeRules(prim, sec)
12:            nsec ← SelNSec(mrg, prim, sec, rules)
13:            if Overlap(mrg, nsec) then
14:                overlap ← overlap ∪ {nsec}
15:                hdr ← GenHdr(mrg, nsec)
16:                gen_num += PrintHdrs(hdr, par_a, par_b)
17: end function
```

Fig. 17: Pseudocode of header trace generator.

IPv4 5-tuples are generated according to the OpenFlow seed using the modules present in the *Improved ClassBench* block (line 3). Each generated 5-tuple is then transformed to an OpenFlow rule that complies with the generated $ruletype$ (line 5). In particular, some of the created fields might be removed (function Remove in line 8) and some OpenFlow-specific fields might be added (function Add in line 12).

To generate consistent OpenFlow rules, some dependency among fields has to be ensured. As an example, the value of *eth_type* depends on the presence of several others header fields, e.g., the presence of a VLAN tag. Per-field constraints are also taken into account: the value of *ip_tos* is randomly selected from a pool of values defined by IANA [31], while the values of `0x000` and `0xFFF` for *vlan_id* are not allowed (the VLAN standard [32] reserves these values for a special purpose). A similar approach is applied when generating the value of *mac_src* and *mac_dst*, which use the *parts* representation. Their vendor part is generated according to the distribution from the seed, but the device part is randomly generated.

C. Trace Generator

This module generates a sequence of packet headers that match an input rule set, i.e., IPv4, IPv6 or OpenFlow, and cover a high percentage of non-empty regions at the same time (Section II-B). The pseudocode of the trace generator is shown in Figure 17. First, the function NoReuseGen randomly iterates over the rules provided as input and generates a matching header for each of them (line 2). The function exits when either the required trace size is reached or all the input rules have been scanned. If the latter, a new loop starts to reach the requested size (line 4). The main idea behind this loop is to augment the trace with headers matching more than one rule at a time. To do so, it is important to find *overlapping* rules. Depending on the amount of present overlaps and the overall size of a rule set, finding all overlaps may be impossible due to memory requirements and time complexity. Instead of performing such analysis, the generator iteratively selects a random primary rule *prim* (line 5) with a bias towards known overlapping rules. The function SelSec tries to pair *prim* with an overlapping secondary rule (line 6). If a match is not found

after a number of tries, the last unsuccessful pairing is used. A header is generated (line 7) by choosing values that match both rules where possible, otherwise *prim*'s conditions are favored. The header is then inserted into the trace multiple times, where the number of repetitions is sampled from a Pareto distribution (line 8). If an overlapping pair is found, a new temporary rule is created by merging the two rules (line 11). It is then used as a new primary rule and the same process of searching for an overlap repeats. This time a header is generated only if an overlapping pair is found.

V. CLASSBENCH-NG EVALUATION

This section first evaluates ClassBench-ng's Rule Generator, focusing on the generation of IPv4 prefixes (Section V-A), IPv6 prefixes (Section V-B), and OpenFlow rules (Section V-C). In the case of IPv4 prefixes we compare ClassBench-ng against ClassBench [18] and FRuG [19], while IPv6 prefixes generation fidelity is compared against Non-random Generator [33]. Finally, the *OpenFlow Generation* block is evaluated against FRuG [19]. We do not asses layer four ports and protocol generation, as ClassBench-ng relies directly on ClassBench for them.

The evaluations of Rule Generator use the *root-mean-square error* (RMSE), defined in Equation 2, to fairly compare the different tools. In the equation, n represents the number of generated rule sets, $\bar{y}$ is the target value, and y_i stands for the generated ones. The experiments are carried on by generating 10 rule sets, i.e., $n = 10$, using tool-specific seeds extracted from an *original rule set*. In this case, the characteristics of the *original rule set* represent the target values, i.e., $\bar{y}$, against which we compare the same characteristics extracted from rule sets generated by various tools, i.e., y_i.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{y} - y_i)^2} \tag{2}$$

The last part of this section is devoted to the evaluation of ClassBench-ng's Trace Generator (Section V-D). We compare ClassBench-ng against original ClassBench using three parameters: the coverage of a rule set by the generated trace, generator's run time and its memory consumption. Similarly to the evaluation of Rule Generator, presented results are based on 10 independently generated traces for each experimental setting.

A. IPv4 Prefixes Generation

This section compares the RMSE of ClassBench-ng, ClassBench, and FRuG on IP prefix set parameters. We first generated an *original rule set* with ClassBench using the `acl4` seed provided with this tool. Then, capitalizing on FRuG/ClassBench-ng capabilities of producing input seeds from an input rule set, we created the appropriate seeds for FRuG, ClassBench-ng, and ClassBench. We then used these seeds to generate back rule sets whose characteristics are assessed using their RMSE.

The comparison of ClassBench-ng, ClassBench, and FRuG on IP prefix sets generation is shown in Figures 18. In terms of a branching probability distribution (Figure 18b), ClassBench-ng outperforms ClassBench and results to be worse than FRuG at only one trie level. The situation is more balanced with respect to an average skew distribution (Figure 18c). In this case, ClassBench-ng is more precise in approximately 50 % of trie levels when compared against ClassBench and in more than 80 % of levels when compared against FRuG. On the other hand, Figure 18a shows poor performance of ClassBench-ng with respect to prefix length distribution fidelity. Although it is not possible to improve ClassBench-ng generation fidelity for this parameter without impacting negatively on the other ones, it is worth noting that in this case the RMSE is ten times lower than for the other parameters, making ClassBench-ng overall a more accurate solution. In fact, Figure 19 shows the average RMSE per trie level when all the evaluated parameters are considered at once. In this case, ClassBench-ng outperforms

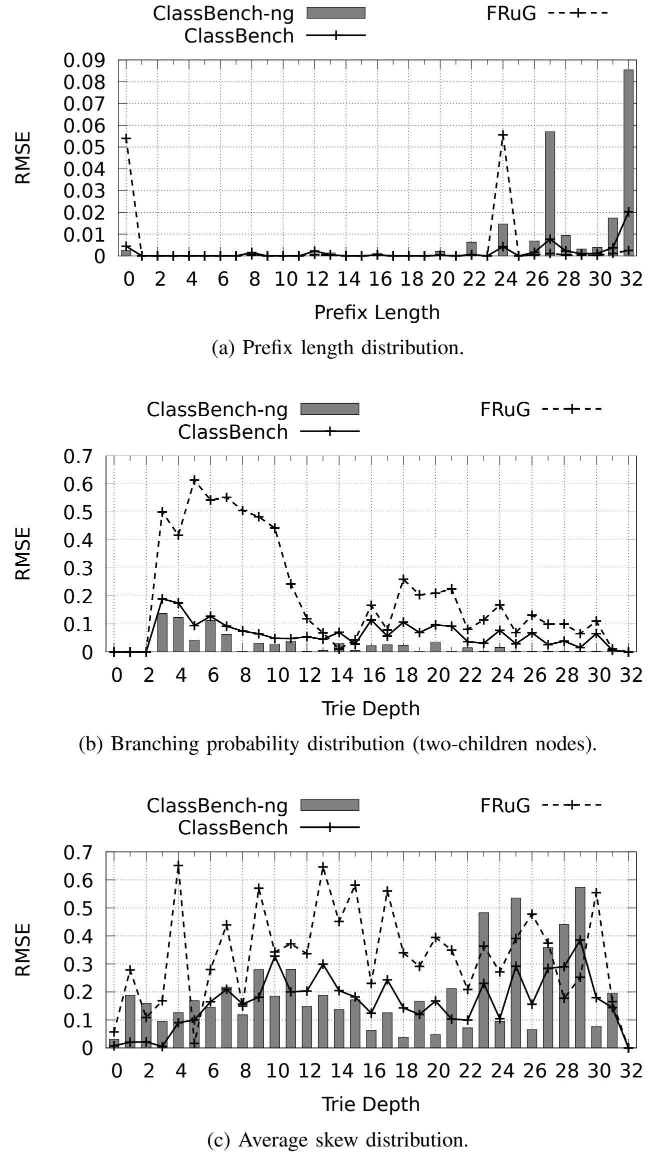


Fig. 18: Comparison of *root-mean-squared error* of ClassBench-ng, ClassBench, and FRuG in IPv4 prefix sets generation.

the other solutions in most of the trie levels, and in particular the $24th$, which is the most commonly used in operation (Section III-A).

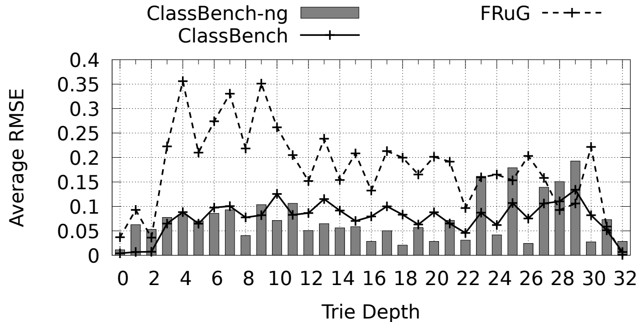


Fig. 19: Average *root-mean-squared error* of ClassBench-ng, ClassBench, and FRuG in IPv4 prefix sets generation.

B. IPv6 Prefixes Generation

To evaluate the quality of IPv6 prefix set generation of ClassBench-ng against Non-random Generator, we used two prefix sets that come from the same core router. An input seed for ClassBench-ng was extracted from IPv6 prefix set `rrc00_2019`, while Non-random Generator's input consisted directly of IPv4 prefix set `rrc00_2019`. Although such a setup leads to a not entirely fair comparison of the tools, we note that Non-random Generator requires an IPv4 prefix set to generate an IPv6 prefix set.

Results of the comparison are shown in Figures 20. Both ClassBench-ng and Non-random Generator achieve comparable quality of generation in terms of a prefix length distribution (Figure 20a). However, ClassBench-ng is more precise with respect to a branching probability distribution (Figure 20b) and Non-random Generator wins the comparison on an average skew distribution (Figure 20c).

C. OpenFlow Rules Generation

OpenFlow rules generation capability of ClassBench-ng is compared against FRuG on two different aspects: (1) field dependencies represented by the *rule type* parameter introduced in Section III-C2 and (2) generation of selected OpenFlow-specific fields. As a common *original rule set*, which is required to fairly asses the two tools using an RMSE, we chose `of1`.

Figure 21a compares the ClassBench-ng *rule type* RMSE against the one obtained with FRuG. With respect to this experiment, our tool clearly outperforms FRuG as it achieves higher RMSE only for *rule types* 1304 and 2048. Therefore, ClassBench-ng is more accurate in characterizing the relationship between header fields, i.e., which fields are more likely to be specified together in a rule. ClassBench-ng also proves to be more accurate in the generation of selected OpenFlow-specific header fields (Figure 21b). As *vlan_id*, *vlan_prio*, and *ip_tos* are always wildcarded in available rule sets, we focus the assessment of OpenFlow field generation on the *in_port*, *mac_src*, *mac_dst*, and *eth_type* header fields. While the average RMSE of ClassBench-ng and FRuG is almost the same (and very low) for *in_port*, in the case of other fields our tool is clearly better. Finally, Figure 21c shows the RMSE for the values of vendor part of the *mac_dst* field. ClassBench-ng outperforms FRuG for all generated values.

D. Header Trace Generation

The evaluation of ClassBench-ng's Header Trace Generator was performed using at least one input rule set for each supported type (see Table V). These rule sets were generated by ClassBench-ng's Rule Generator using seeds extracted from corresponding real rule sets (see Table I) or a seed taken from original ClassBench (i.e., `acl4_gen`). With the exception of an OpenFlow rule set, the size of the generated rule sets is of the same order of a magnitude as in case of corresponding real rule sets.

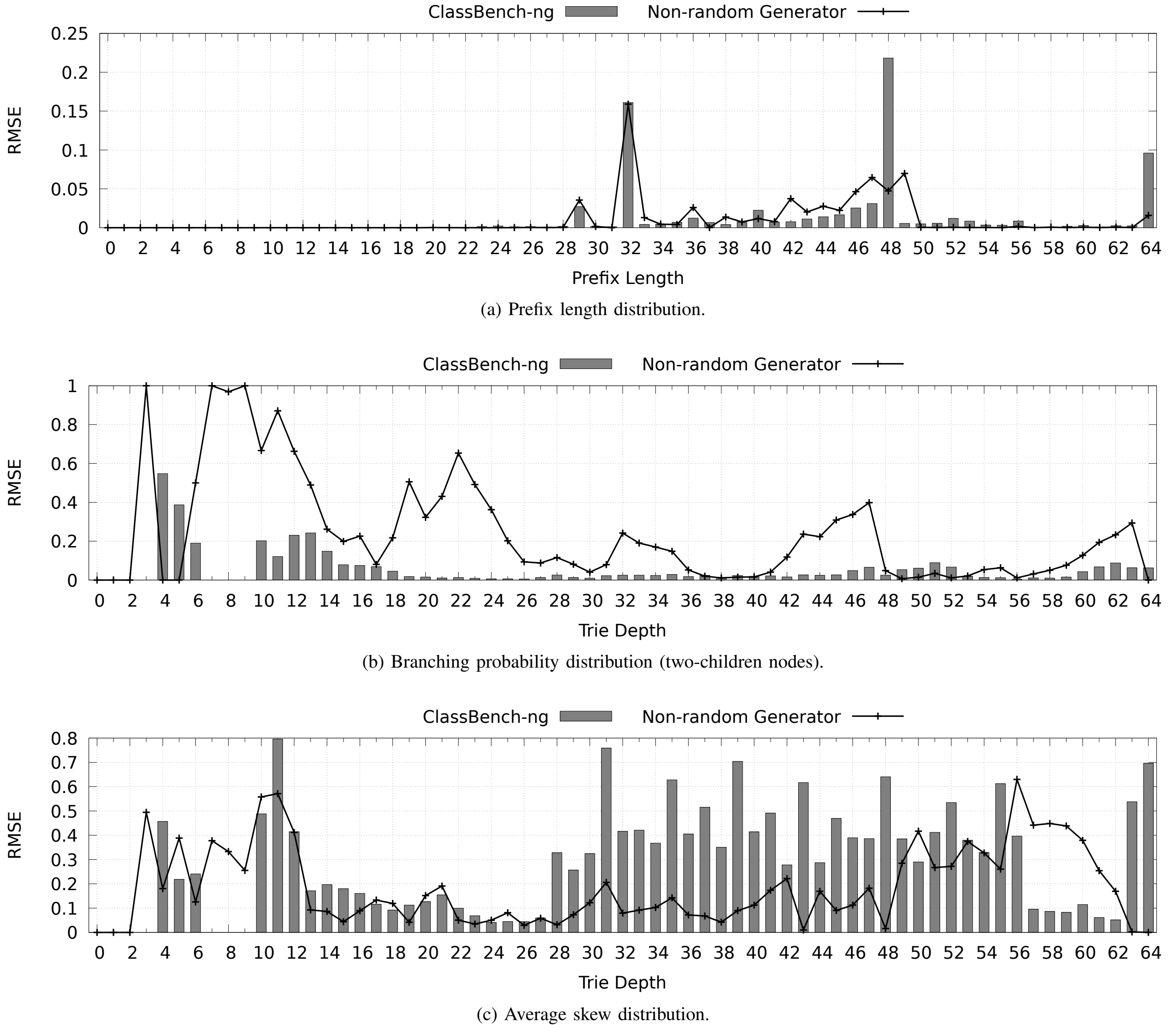
TABLE V: Gereated rule sets used as trace generator's input.

Name	Rules	Overlaps	Regions
`ipv4_rrc00_2019_gen`	100 000	12 095	100 000
`ipv6_rrc00_2019_gen`	10 000	1	10 000
`acl4_gen`	1 000	332	1 287
`of1_gen_1k`	1 000	40 800	41 800

Table V shows the number of rules, rule overlaps, and distinct regions (as defined in Section II) for each rule set. While overlaps do not introduce new regions in case of IPv4 and IPv6 prefixes (a longer prefix is always fully contained in a shorter prefix), this is not the case for more complex rules like ACL and OpenFlow. Therefore, to allow a full coverage of rule set's regions, the size of a header trace generated by the trace generator has to be proportional to the number of regions in the input rule set, not its size in terms of rules.

Firstly, we evaluated the coverage of rule sets' regions by header traces of a various size generated using Header Trace Generators of ClassBench and ClassBench-ng. Since the trace generator of ClassBench is able to produce traces that cover up to 100 % of regions in IP prefix sets (see Figure 3), Figure 22 presents the results for more complex rule sets (i.e., ACL and OpenFlow) only. The figure clearly shows that in case of reasonably large ACL traces (the same or larger number of headers compared to the number of regions), ClassBench-ng's trace generator is able to produce traces that cover approximately 20 % more regions than traces generated by ClassBench's trace generator. Moreover, the results are even better for the OpenFlow rule set. Its coverage keeps increasing with larger traces generated by ClassBench-ng's trace generator, although it is constant (and very small) in case of traces generated by the trace generator of ClassBench.

Finally, we evaluated memory consumption and execution time of the trace generators. Although memory consumption of ClassBench-ng's trace generator is always little higher compared to the trace generator of ClassBench (see Table VI), both generators have almost constant memory requirements, regardless the size of a generated trace. ClassBench-ng's trace generator thus achieves a better coverage at the cost of higher excution time, which rapidly increases with the complexity of an input rule set and the size of a generated trace. For



(a) Prefix length distribution.

(b) Branching probability distribution (two-children nodes).

(c) Average skew distribution.

Fig. 20: Comparison of *root-mean-squared error* of ClassBench-ng and Non-random Generator in IPv6 prefix sets generation.

instance, to generate a trace containing 5-times more headers compared to the number of regions, ClassBench-ng needs 3 orders of a magnitude longer time compared to original ClassBench in case of IPv4, IPv6, and ACL rule sets and 2 orders of a magnitude longer time in case of an OpenFlow rule set. Nevertheless, absolute execution time of ClassBench-ng's trace generator (between 29.4 s for ACL and 1497.8 s for IPv4) can be seen as affordable, considering that a header trace corresponding to a rule set is generated only once and then used multiple times.

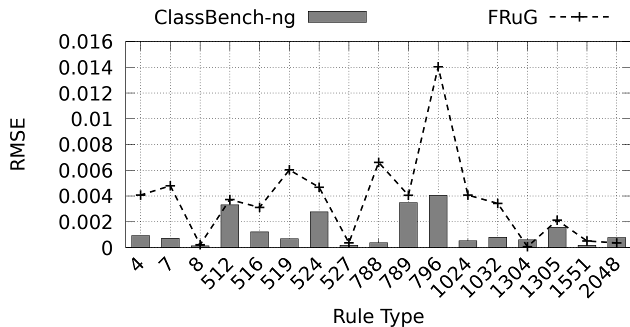
VI. Related Work

In the absence of publicly available classification rule sets, past researchers faced the problem of how to realistically assess the performance of new packet classification algorithms. While a limited number of research groups obtained access to real rule sets through confidentiality agreements, others dealt with frameworks for synthetic rule sets generation. In this scenario, ClassBench [18] is the well known and commonly used framework for IPv4 classification rules generation. So far, it has been a very useful tool but it does not reflect anymore current research community needs, as it focuses only on IPv4.

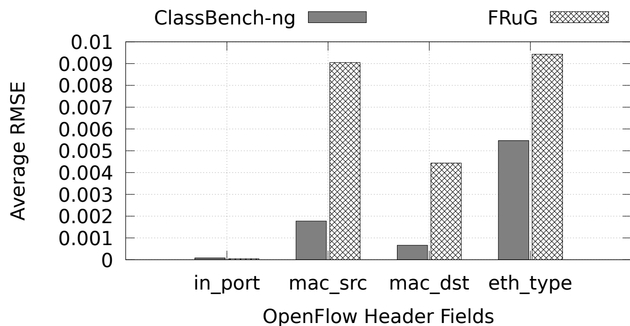
Sun et al. [20] responded to the increasing interest towards IPv6 protocol proposing ClassBenchv6, a reshaped version of the ClassBench framework for the IPv6 world. With a focus on IPv6 lookup tables only, Wang et al. [33] developed new

TABLE VI: Peak real memory usage of ClassBench's and ClassBench-ng's trace generators when generating header traces with size between 1 % and 500 % of regions in a corresponding input rule set (average of 10 runs).

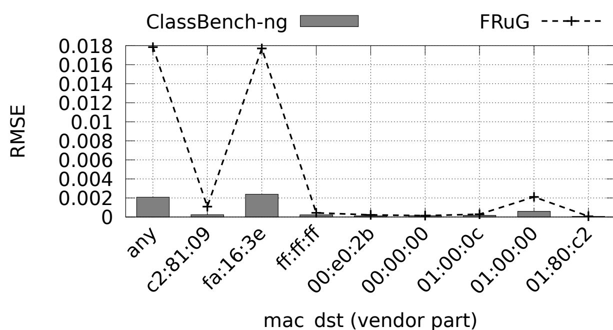
Header Trace Generator		**IPv4**	**IPv6**	**ACL**	**OpenFlow**
ClassBench	min [MB]	306.0	186.9	175.1	175.0
	max[MB]	339.0	191.0	175.4	175.2
ClassBench-ng	min [MB]	317.7	188.1	175.3	175.3
	max[MB]	351.0	192.2	175.6	175.3



(a) OpenFlow *rule types*.



(b) Average RMSE for selected OpenFlow-specific fields.



(c) Vendor part of *mac_dst*.

Fig. 21: Comparison of *root-mean-squared error* of ClassBench-ng and FRuG in OpenFlow rule sets generation.

algorithms for the synthetic generation of IPv6 forwarding tables. Following this effort, Zheng et al. [34] developed a scalable IPv6 prefix generator, called V6Gene, for IPv6-based route lookup algorithms benchmarking.

With an eye towards new future protocols, Ganegedara et al. [19] proposed FRuG, a generic synthetic rule generator. It allows the user to select the protocol fields and the characteristics of each field, which can either be defined by the user or configured to follow a distribution from an input seed file. The user has complete control over the structure and the size of the rule table which makes it a powerful benchmark to assess various packet forwarding algorithms and for different types of routers. However, only MAC and IP addresses fields can be set to follow an input distribution. The other OpenFlow-related fields need to be manually configured by the user, making this solution less attractive if a realistic set of synthetic rules needs to be generated.

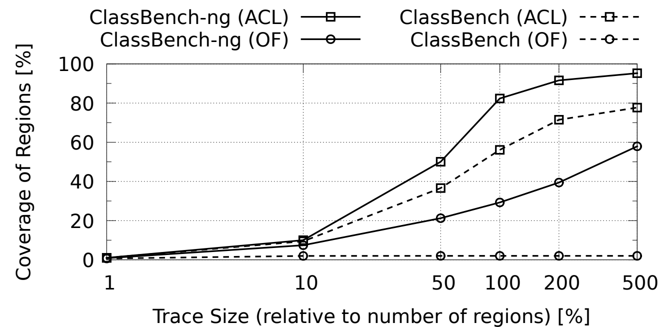


Fig. 22: Comparison of complex rule sets' regions coverage by header traces of various size generated using trace generators of original ClassBench and ClassBench-ng (average of 10 traces).

ClassBench-ng has been designed to provide the flexibility of generating IPv4, IPv6, and OpenFlow rule sets. It accepts an input seed file which can specify a distribution for all the OpenFlow 1.0.0 matching fields, making this solution very attractive when a realistic rule set generation is needed. The detailed analysis performed on real sets allows to include in the tool input seeds that reflect the real world properties. In addition, the ability to self-generate seeds from real sets allows to create a repository for a number of seeds that reflect different scenarios, e.g., data center, Internet Service Provider, or Internet eXchange Point.

VII. CONCLUSION

This paper presents ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow classification rules alongside associated header trace matching them. We analyzed real sets taken from backbone routers, edge firewalls and cloud data centers to gain a better understanding of the statistical properties of nowadays' classification rules. We used our insight to design specific input paramter files that feed our generators. Furthermore, to make this solution attractive in the long term and for a wide number of different use cases, we upgraded our tool with the possibility of creating input parameter files from real rule sets.

We aim to use the tool's repository as a place where researchers and operators can continuously upload new parameter files that match a number of different environments or use cases, e.g., data center, Internet Service Provider, Internet eXchange Point. We believe that this aspect will further increase the impact of ClassBench-ng on the research community.

REFERENCES

[1] J. Matoušek *et al.*, "ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution," in *ANCS*. IEEE, 2017, pp. 204—216.
[2] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[3] J. Czyz *et al.*, "Measuring IPv6 Adoption," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 87–98, Aug. 2014.
[4] "IPv6 – Google," https://www.google.com/intl/en/ipv6/statistics.html.

[5] M. Bruyere *et al.*, “Rethinking IXPs’ Architecture in the Age of SDN,” *IEEE J. Select. Areas Commun.*, vol. 36, no. 12, pp. 2667–2674, Dec. 2018.
[6] C.-Y. Hong *et al.*, “Achieving High Utilization with Software-Driven WAN,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, Oct. 2013.
[7] ——, “B4 and after: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google’s Software-Defined WAN,” in *SIGCOMM*. ACM, 2018, pp. 74–87.
[8] B. Schlinker *et al.*, “Engineering Egress with Edge Fabric: Steering Oceans of Content to the World,” in *SIGCOMM*. ACM, 2017, pp. 418–431.
[9] A. Singh *et al.*, “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Oct. 2015.
[10] D. E. Taylor, “Survey and Taxonomy of Packet Classification Techniques,” *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.
[11] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, “Optimizing Many-field Packet Classification on FPGA, Multi-core General Purpose Processor, and GPU,” in *ANCS*. IEEE, 2015, pp. 87–98.
[12] T. Yang *et al.*, “Fast OpenFlow Table Lookup with Fast Update,” in *INFOCOM*. IEEE, 2018, pp. 2636–2644.
[13] C.-L. Hsieh, N. Weng, and W. Wei, “Scalable Many-Field Packet Classification for Traffic Steering in SDN Switches,” *IEEE Trans. Netw. Serv. Manage.*, vol. 16, no. 1, pp. 348–361, Mar. 2019.
[14] H. Alimohammadi and M. Ahmadi, “Clustering-based many-field packet classification in Software-Defined Networking,” *Journal of Network and Computer Applications*, vol. 147, p. 102428, Dec. 2019.
[15] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 147–160, Oct. 1999.
[16] H. Song and J. S. Turner, “ABC: Adaptive Binary Cuttings for Multidimensional Packet Classification,” *IEEE/ACM Trans. Networking*, vol. 21, no. 1, pp. 98–109, Feb. 2013.
[17] H. Lim *et al.*, “Boundary Cutting for Packet Classification,” *IEEE/ACM Trans. Networking*, vol. 22, no. 2, pp. 443–456, Apr. 2014.
[18] D. E. Taylor and J. S. Turner, “ClassBench: A Packet Classification Benchmark,” *IEEE/ACM Trans. Networking*, vol. 15, no. 3, pp. 499–511, Jun. 2007.
[19] T. Ganegedara, W. Jiang, and V. K. Prasanna, “FRuG: A Benchmark for Packet Forwarding in Future Networks,” in *IPCCC*. IEEE, 2010, pp. 231–238.
[20] Q. Sun *et al.*, “ClassBenchv6: An IPv6 Packet Classification Benchmark,” in *GLOBECOM*. IEEE, 2009, pp. 1–6.
[21] F. Baboescu, S. Singh, and G. Varghese, “Packet Classification for Core Routers: Is there an alternative to CAMs?” in *INFOCOM*. IEEE, 2003, pp. 53–63 vol.1.
[22] M. E. Kounavis *et al.*, “Chapter 13 – Directions in Packet Classification for Network Processors,” in *Network Processor Design*. Morgan Kaufmann, 2004, vol. 2, pp. 273–298.
[23] Y. Qi *et al.*, “Packet Classification Algorithms: From Theory to Practice,” in *INFOCOM*. IEEE, 2009, pp. 648–656.
[24] “OpenFlow Switch Specification – Version 1.5.1,” https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.
[25] “University of Oregon Route Views Project,” http://www.routeviews.org/routeviews.
[26] “RIS Raw Data,” https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.
[27] “IPv4 address report,” http://www.potaroo.net/tools/ipv4.
[28] “IPv6 Deployment Status,” https://www.vyncke.org/ipv6status.
[29] “OpenFlow Switch Specification – Version 1.0.0,” https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.
[30] “Open vSwitch Manual: ovs-ofctl(8),” https://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.html.
[31] “Differentiated Services Field Codepoints (DSCP),” http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.
[32] “IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks,” *IEEE Std 802.1Q-2018*, pp. 1–1993, 2018.
[33] M. Wang, S. Deering, T. Hain, and L. Dunn, “Non-random Generator for IPv6 Tables,” in *HOTI*. IEEE, 2004, pp. 35–40.
[34] K. Zheng and B. Liu, “V6Gene: A Scalable IPv6 Prefix Generator for Route Lookup Algorithm Benchmark,” in *AINA*. IEEE, 2006, pp. 6 pp.–152.