ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution

Jiří Matoušek[1]
imatousek@fit.vutbr.cz

Gianni Antichi[2]
gianni.antichi@cl.cam.ac.uk

Adam Lučanský[3]
xlucan01@stud.fit.vutbr.cz

Andrew W. Moore[2]
andrew.moore@cl.cam.ac.uk

Jan Kořenek[1]
korenek@fit.vutbr.cz

[1]Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations
Brno, CZ

[2]University of Cambridge
Cambridge, UK

[3]CESNET
Praha, CZ

ABSTRACT

Internet evolution is driven by a continuous stream of new applications and users driving the demand for services. To keep up with this, a never-stopping research has been transforming the Internet ecosystem over the time. Technological changes on both protocols (the uptake of IPv6) and network architectures (the adoption of Software Defined Networking) introduced new challenges for ASIC designers. In particular, IPv6 and OpenFlow increased the complexity of the rule matching problem, pushing researchers to build new packet classification algorithms capable to keep pace with a steady growth of link speed.

A lot of research effort identifies better lookup techniques capitalizing on the characteristics of rule sets. So far, the availability of small numbers of real rule sets and synthetic ones, generated with tools such as ClassBench, has boosted research in the IPv4 world. Starting from an analysis of rule sets taken from operational environments, we present ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow 1.0 rule sets exposing the same properties of real ones. We feel this tool can meet the requirements of nowadays researchers, boosting the rule matching research as ClassBench has done since ten years ago.

CCS Concepts

•**Networks** → **Packet classification; Network experimentation; Network performance analysis;** *Network performance modeling; Network measurement; Packet-switching networks;* Network dynamics; •**Hardware** → **Networking hardware;** *Hardware accelerators; Reconfigurable logic applications;*

Keywords

ClassBench, OpenFlow, packet classification

1. INTRODUCTION

Internet enables users to access applications and services through the network at a global scale. It is a very dynamic ecosystem, where the evolution is driven by a continuous stream of new applications, as well as users driving the demand for a steady growing number of services. Today's Internet is no more the one of a decade ago. The continuous innovation, a desirable property of Internet, shapes network devices operation. Technological changes/improvements on both protocols and network architecture transformed device operations over the time.

From a protocol perspective, the last few years have shown a steady rise of IPv6 deployments. In particular, Czyz et al. [6] show that, while raw IPv6 Internet traffic is still a small fraction, the nature of its use and the trajectory of growth have shifted dramatically. Consequently, IPv6 should no longer be dismissed by researchers as an uninteresting rarity. On the other side, from a network architecture point of view, the Software Defined Networking (SDN) paradigm, with its predominant realization, the OpenFlow protocol [12], is getting more and more interest in production networks. New OpenFlow-enabled switches are hitting the market from a number of different companies, e.g., Pica8, Arista, Corsa, facilitating real SDN implementations as for the CARDIGAN project [15], the Google WAN [8], or other deployments around the world [10].

However, the basic operation of each networking device is still the same: packet classification at physical link speed. As a packet arrives, the system must compare one or more header fields against a set of rules to assign a flow identifier. This is used for the basic forwarding operation, to apply security policies, application-specific processing, or quality-of-service guarantees. The packet classification problem is not new and has been driving a lot of research in the last years [17, 19, 13, 9, 21]. As the matching complexity and link speed increase, we argue the rule matching problem still remains a hot topic. In particular, the IPv6 protocol quadrupled the size of IP addresses, making the lookup process more complicated compared to the IPv4 case. Moreover, OpenFlow extended the matching criteria to multiple fields, resulting in new challenges for ASIC designers.

A lot of research effort in the past identified better packet classification techniques leveraging the characteristics of real rule sets for faster searches [14, 11]. It has also been demonstrated that the capacity and efficiency of the most prominent packet classification solution, Ternary Content Addressable Memory (TCAM), is also subject to the characteristics of rule sets [17]. The lack of publicly available rule sets has been mitigated by a number of synthetic rule set generators [18, 7, 16]. However, none of them provide a flexible so-

lution that allows for an accurate generation of IPv4, IPv6, and OpenFlow rule sets.

Into this lacuna, we launch ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow 1.0 rule sets. It accepts an input parameter file that can specify the statistical properties for all the matching fields that need to be generated. Therefore, to make the ClassBench-ng output rule set as close as possible to a realistic one, it is important that such properties reflect precisely the current trends. We thus present also an analysis of real IPv4 and IPv6 prefix sets. In addition, we inspected OpenFlow 1.0 flow tables taken from an operational environment. Finally, to make this solution attractive in the long term and for a wide number of different use cases, ClassBench-ng offers a mechanic to create input parameter files from real rule sets. We aim to use the tool 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., datacenter, Internet Service Provider, Internet eXchange Point. This will further increase the (potential) impact of ClassBench-ng on the research community.

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

- In-depth analysis of real classification rule sets based on IPv4, IPv6, and OpenFlow 1.0.
- A new tool which is able to generate and analyze IPv4, IPv6, and OpenFlow 1.0 rule sets.
- The tool is open and available to anyone at: `http://github.com/classbench-ng/classbench-ng`.

The rest of the paper is organized as follows: we first concentrate on the challenges in synthetic rule set generation (Section 2). We then present an analysis of real IPv4, IPv6, and OpenFlow data sets (Section 3), alongside the ClassBench-ng architecture (Section 4) and the experimental evaluation (Section 5). Finally, we cover related works (Section 6) and conclude the paper (Section 7).

2. CHALLENGES IN RULE GENERATION

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 [7]. While the former solution results to be more accurate when an output as close as possible to a real set is required, the latter is (potentially) more flexible in the long term. Indeed, the continuous innovation, a desirable property of Internet, might change the statistic properties of rule sets, thus likely making a tool obsolete, if the input parameters are not updated accordingly. Providing an analysis mechanic, able to generate input seeds from real sets, is thus the most sensible way to ensure the longevity of the tool. Therefore, combining a generation module, which rely on statistic distributions of real sets, with an analysis toolkit is the best combination for a synthetic rule set generator.

ClassBench-ng proposes a tight integration of generation and analysis. While, the generation module creates IPv4, IPv6, or OpenFlow rules from an input seed (Equation 1a), the analysis toolkit can recreate input seed files from real rule sets (Equation 1b). Because of space limitations, this paper describes only the algorithms that enable the first transformation (Section 4), as the other is the inverse operation: it derives rules statistical distributions from real data.

$$rule\_set = f(seed) \tag{1a}$$

$$seed = f^{-1}(rule\_set) \tag{1b}$$

The generation quality, i.e., how similar is the output set to a real one, directly depends on the parameters being used as an input. To this end, we identified the main properties a seed needs to assure:

- *anonymity* — retain all important characteristics of a real set without revealing any confidential information.
- *completeness* — be sufficient for the generation of a new synthetic set.
- *scalability* — allow the generation of synthetic sets of different sizes.

The first property easier the redistribution of seeds that match a number of different environments or use cases, while keeping the output set as close as possible to a real one. The other properties are needed to enable the generation process.

ClassBench-ng meets all three properties allowing for the generation of the desired amount of output rules that match input distributions. A statistical approach in the definition of input parameters enables the first property: the *anonymity* is guaranteed as no real rules are strictly needed (thus avoiding the use of sensitive information). The *completeness* of the rule set representation is proven by the tool itself and the past experience of ClassBench [18]. As for the *scalability*, we feel that the statistical approach being used for the rule set representation impacts favorably when the size of the generated data starts to scale. Indeed, the more entries need to be generated, the easier they will have the required statistical properties.

Understanding the statistical properties of real rule sets is thus the first step towards the realization of ClassBench-ng. Given past research [18] had shown a complete analysis of real IPv4-based classification rules, along with a keen desire to determine if such a study is nowadays still valid, we propose a comparative analysis of IPv4 and IPv6-based rule sets after a decade. We then investigate the properties of OpenFlow 1.0 flow tables taken from an operational environment. The statistical insights from this study will serve as a basis for the creation of appropriate seeds that will be used as an input in the generation process.

3. ANALYSIS OF REAL CLASSIFICATION RULES

This section provides an analysis of IPv4, IPv6, and OpenFlow 1.0 classification rule sets taken from operational environments. IPv4 and IPv6 prefixes have been taken from core routers. Classification rule sets come from access control lists (ACLs) applied at a university network's perimeter, while the analysis of OpenFlow data is based on a set of Open vSwitches running in a cloud datacenter environment. Table 1 summarizes the data sets being used in the analysis.

[1]We call them seeds.

Table 1: 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_2015`	550 511	Route Views	2015-07-02
`eqix_2005`	164 455		2005-07-02
`rrc00_2015`	571 351	RIPE RIS	2015-07-02
`rrc00_2005`	168 525		2005-07-02
IPv6 Prefix Sets			
`eqix_2015`	23 866	Route Views	2015-07-02
`eqix_2013`	13 444		2013-07-02
`eqix_2005`	658		2005-07-02
`rrc00_2015`	24 162	RIPE RIS	2015-07-02
`rrc00_2013`	14 374		2013-07-02
`rrc00_2005`	499		2005-07-02
ACL Rule Sets			
`uni_2010`	96	ACLs from university network	2010-08-30
`uni_2015`	122		2015-01-14
OpenFlow Rule Sets			
`of1`	16 889	OpenFlow Switch in a datacenter	2015-05-29
`of2`	20 250		2015-05-29
`of3`	1 757 to 7 456		2015-06-18 to 2015-07-14

In the following sections, we first analyze properties of IP prefix sets from core routers (Section 3.1) and classification rules from university ACLs (Section 3.2). Then we conduct an analysis of real rule sets from Open vSwitches deployed in a cloud datacenter environment (Section 3.3).

3.1 IP Prefixes

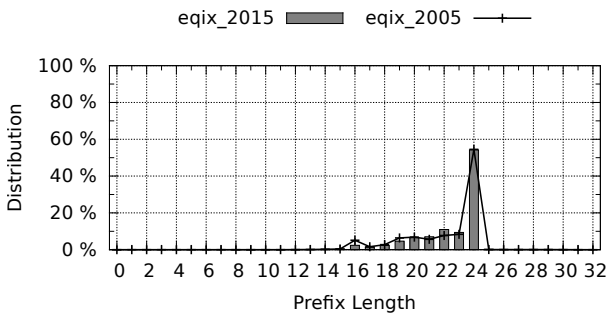
3.1.1 IPv4

A binary prefix tree, i.e., a trie, is the most common data structure being used to represent a set of IP prefixes. The main statistical parameters that influence its shape are four: a prefix length distribution, a branching probability distribution, an average skew distribution, and a prefix nesting threshold [18]. A prefix length distribution characterizes prefixes span. A branching probability distribution represents the probability, at each trie level, of having one-child or two-children nodes. Skew is defined in Equation 2.

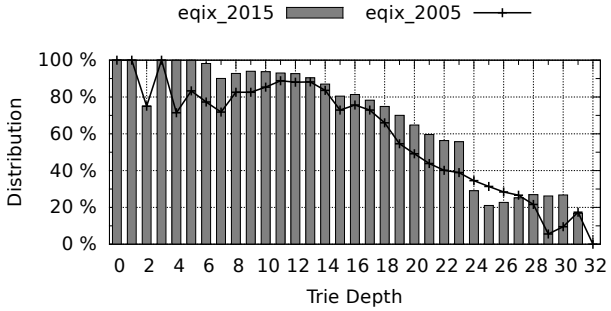
$$skew = 1 - \frac{weight(lighter)}{weight(heavier)} \quad (2)$$

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, a prefix nesting threshold specifies the maximum number of prefix nodes that appear on an arbitrary path from the root to the leaves.

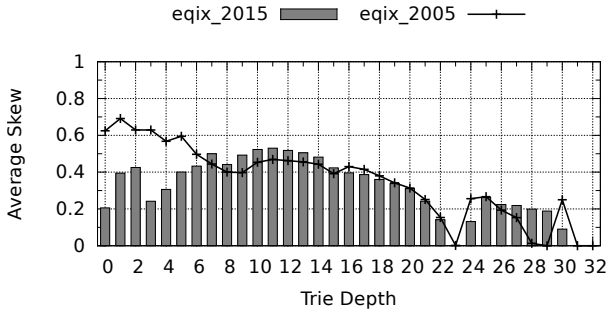
Figures 1 compare the same prefix set (`eqix`) in ten years time. While the prefix length distribution is almost the same between years 2005 and 2015 (Figure 1a), nowadays we are facing an increase of two-children nodes in the trie (Figure 1b) and the average skew is lower (Figure 1c). The prefix nesting threshold remained unchanged between 2005 and 2015. The same results are also confirmed in prefix sets `rrc00`. Growing number of two-children nodes and their smaller skew correlates with more than three times higher number of prefixes after 10 years, as shown in Table 1. Branching probability and average skew distributions



(a) Prefix length distribution.



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



(c) Average skew distribution.

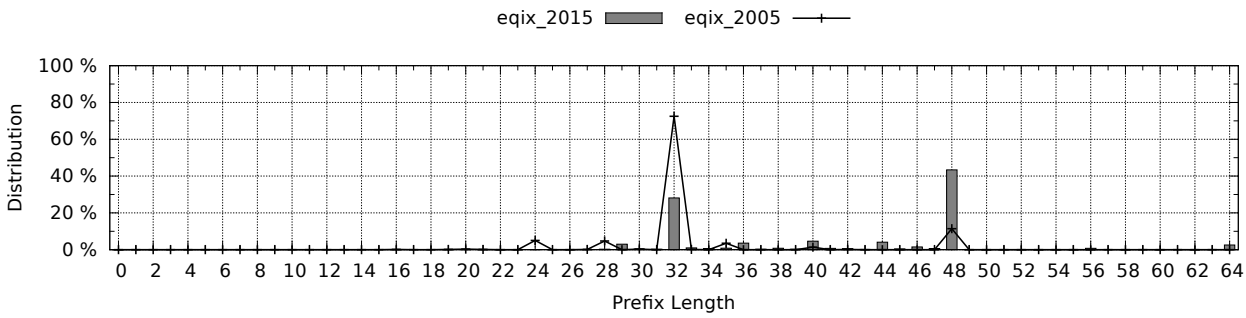
Figure 1: Comparison between `eqix` IPv4 prefix sets in 2005 and 2015.

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 [2].

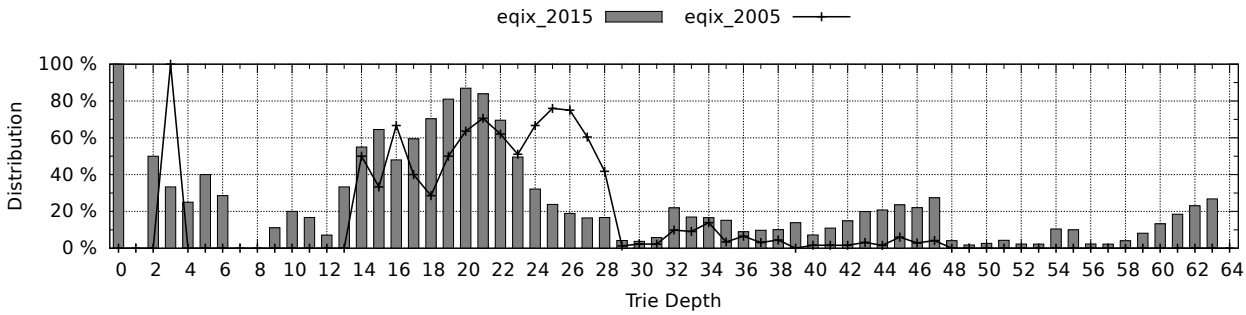
3.1.2 IPv6

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 ten years.

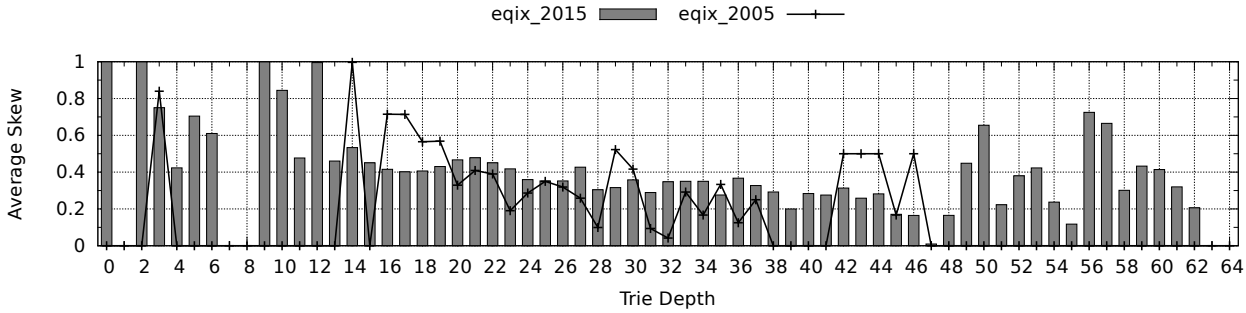
Figures 2 compare the selected parameters between the `eqix` prefix sets from years 2005 and 2015. Only the first 64 trie levels are shown as there were no IPv6 prefixes longer than 48 bits in 2005. Figure 2a shows that the prefix length distribution has changed significantly in the last 10 years. While 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 2b) and the average skew distribution (Figure 2c). We



(a) Prefix length distribution.



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



(c) Average skew distribution.

Figure 2: Comparison between `eqix` IPv6 prefix sets in 2005 and 2015.

believe that such a big difference in the prefix length distribution is related to the steady growth of IPv6 deployments over the last years. In 2005, most of the allocated prefixes belonged to ISPs/RIRs, while nowadays most of the prefixes belong to end users (organizations) [3]. Changes of branching probability and average skew between 2005 and 2015 have also been caused by the emergence of prefixes longer than 64 bits. Prefix set `rrc00` shows similar behavior, except for the prefix nesting threshold parameter, which remains unchanged.

In 2005, both the `eqix` and `rrc00` prefix sets contained only a few hundreds of IPv6 prefixes, while there are currently more than 23 thousands of prefixes in both sets (Table 1). 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 2013 and 2015), 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 2013 and 2015.

Table 2: 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 %

3.2 Ports and Protocol

The following analysis is performed using rule sets taken from ACLs in a university campus network, which were presented in Table 1. The data spans over a period of five years to enable a comparative analysis over the time. We first concentrated on the distribution of rules over protocol values (Table 2). 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 3 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 3: 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 3 and 4 are based on the `uni_2015` dataset 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 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.

3.3 OpenFlow

This section provides an analysis of real OpenFlow rule sets taken from a cloud datacenter 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 3.3.1). Then we move our attention to fields dependency (Section 3.3.2) and rule set dynamics (Section 3.3.3).

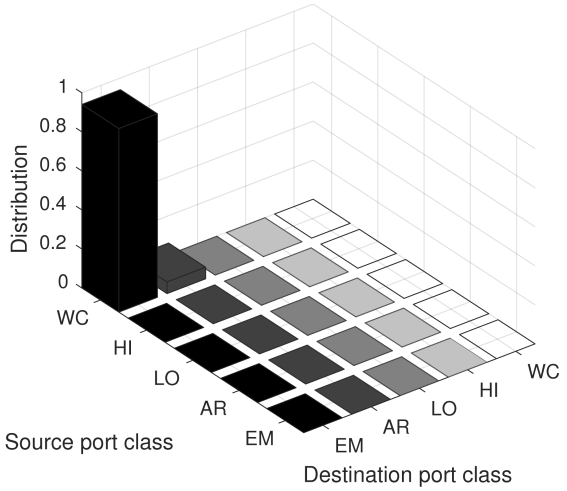


Figure 3: PPC matrix for TCP protocol (rule set `uni_2015`).

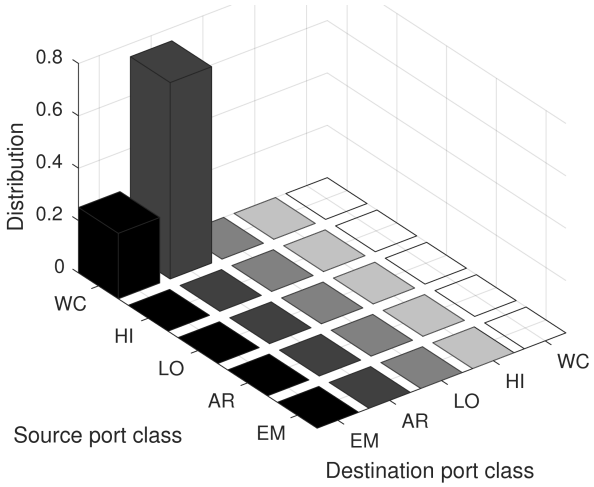


Figure 4: PPC matrix for UDP protocol (rule set `uni_2015`).

3.3.1 Header Fields

OpenFlow 1.0 extends the standard 5-tuple, i.e., *ip_src*, *ip_dst*, *l4_src*, *l4_dst*, and *ip_proto*, with seven more header fields [4]. Figure 5 shows the header field distribution in rule sets `of1` and `of2` introduced in Table 1. 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 4 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

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

Table 4: 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)
of1+of2	182 (59.9)	45 (4.2)	1176 (4.1)	1 (<0.1)	3 (<0.1)	498 (2.0)	119 (0.4)	6 (0.1)	8237 (74.2)

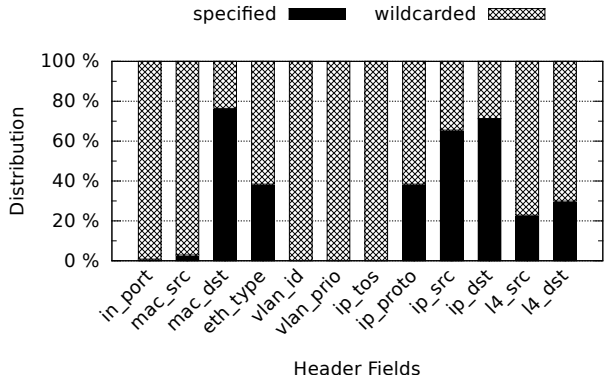


Figure 5: Per-field distribution of rules from combined `of1+of2` rule set over specified and wildcarded classes.

mac_dst field has the highest number of unique values, its *uniqueness factor* is close to zero. In particular, 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.

Figure 6 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 1a 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 datacenter for this one) plays an important role.

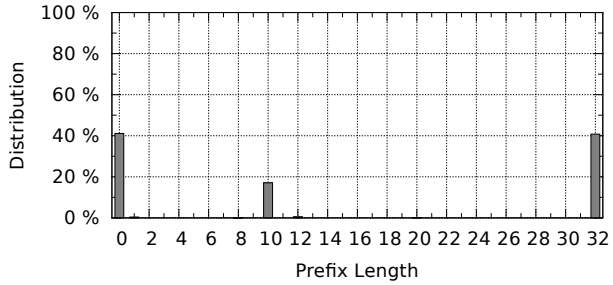


Figure 6: Prefix length distribution of source prefixes from `of1` rule set.

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 3.2 can also be shown for the distribution of source/destination port values over five port classes and their combination into source-destination port pair classes.

3.3.2 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 7 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.

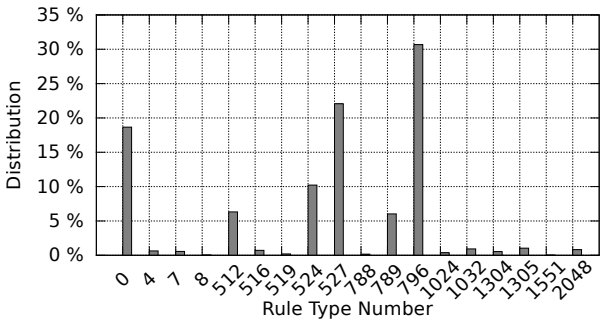


Figure 7: Distribution of rules from combined `of1+of2` rule set over *rule types*.

Figure 5 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 7). 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.

3.3.3 Dynamics

Figure 8 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.

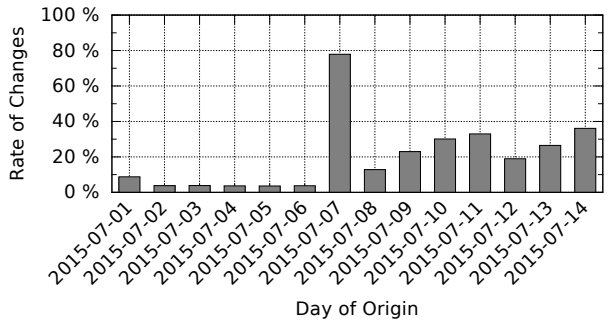


Figure 8: Rate of changes (compared to the previous day) of rule set `of3` between 1st and 14th July 2015.

The studied datacenter 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.

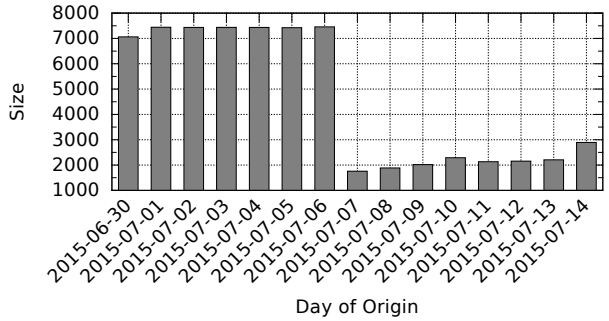


Figure 9: Size of rule set `of3` between 30th June and 14th July 2015.

The behavior can be justified with Figure 9. On that day, in fact, the number of rules decreased drastically, thus creating the big spike in the rate of changes.

4. CLASSBENCH-NG: NEXT GENERATION CLASSBENCH

This section discusses the design of ClassBench-ng. The tool builds upon the original ClassBench software. Figure 10 shows its high-level architecture composed of four main building blocks, which are presented in the following subsections. The *Improved ClassBench* block patches the original tool to improve the IPv4 prefix generation fidelity (Section 4.1), while *IPv6 Generation* block provides IPv6 prefix generation capabilities (Section 4.2). The *OpenFlow Analysis* block analyses OpenFlow rules to produce an output seed, while *OpenFlow Generation* block is in charge of generating synthetic OpenFlow rules (Section 4.4). Despite ClassBench-ng already provides seeds for rule generation, 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.

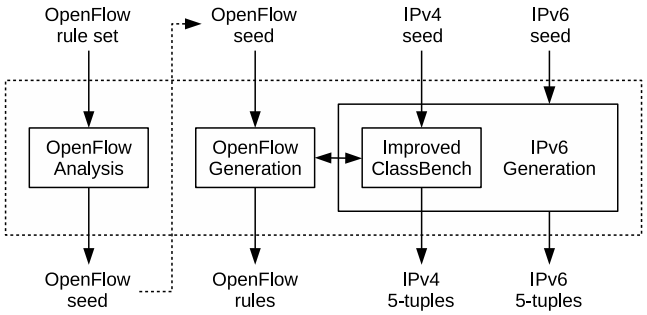


Figure 10: High-level architecture of ClassBench-ng.

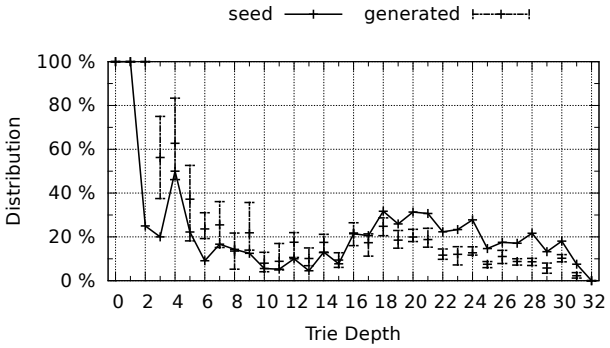
4.1 Improved ClassBench

This section motivates the need for improving the original ClassBench software and describes the algorithm adopted in ClassBench-ng for IPv4 classification rules generation. We performed a test campaign to evaluate the fidelity of ClassBench in order to understand its internals. While the layer four ports and protocol accurately follow the input seed, the IPv4 prefixes show a lower accuracy.

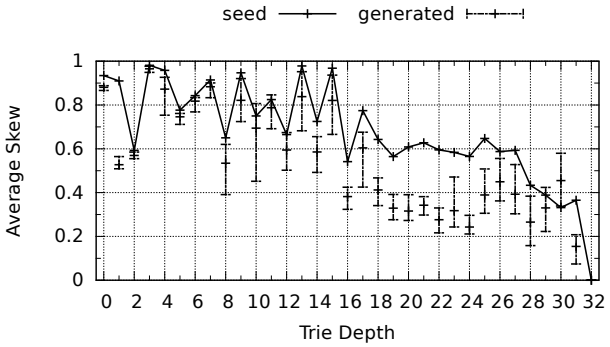
Figures 11 compare prefix set parameters extracted from the input IPv4 seed and from rules generated by original ClassBench, along with their error bars. While the generation process proves to be accurate with respect to a prefix nesting threshold, the other parameters do not precisely follow the required distribution. Indeed, the generated branching probability meets the requirements only for 13 trie levels, while the average skew only for 5. We believe that such errors are caused by parameters interdependence: once the parameter with the highest priority has been fixed, the tool tries to meet the other requirements. A prefix nesting threshold has the highest priority, thus justifying its accuracy.

ClassBench-ng tries to improve the ClassBench generation process by iteratively building an output rule set with characteristics as close as possible to the input seed. The pseudocode in Figure 12 shows the process of rule set construction in the *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 Section 4.1.1.

Improved ClassBench selects rules from the initial set, i.e., *rules*, that contain source/destination IP prefixes available



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



(b) Average skew distribution.

Figure 11: Comparison of destination prefix set parameters from `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.

```
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
```

Figure 12: Pseudocode of rule set construction in *Improved ClassBench*.

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).

4.1.1 Trie Pruning

Figure 13 shows the pseudocode of the 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. Subtrees to be removed are selected increasingly according to the number of prefixes they carry. Moreover, this step never removes the last branch with the maximum prefix nesting to do 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 do 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 do 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.

4.2 IPv6 Generation

The *IPv6 Generation* block is based upon our improved version of original ClassBench. As the trie construction mechanic does not depend on specific IPv4 features, ClassBench-ng takes advantage of the IPv4 toolchain with different input parameters, i.e, IPv6 seeds.

4.3 OpenFlow Analysis

The *OpenFlow Analysis* block takes as an input OpenFlow rules and generates the corresponding seed. ClassBench-ng already comes with seeds for OpenFlow rules generation. However, given the programmability of OpenFlow-enabled networks, we believe that providing a seed generator is im-

```
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
```

Figure 13: 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
```

Figure 14: Pseudocode of OpenFlow rules generator.

portant to adapt the tool to a number of different scenarios. At the current state, ClassBench-ng is able to correctly parse rule sets represented in the format used by `ovs-ofctl` command line tool 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 3.3.2) 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.

4.4 OpenFlow Generation

The *OpenFlow Generation* block generates a set of OpenFlow rules from an input seed. Figure 14 shows the pseudocode of the generation process. 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 others OpenFlow-specific 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 [1], while the values of `0x000` and `0xFFF` for *vlan_id* are not allowed (the VLAN standard [5] 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.

5. CLASSBENCH-NG EVALUATION

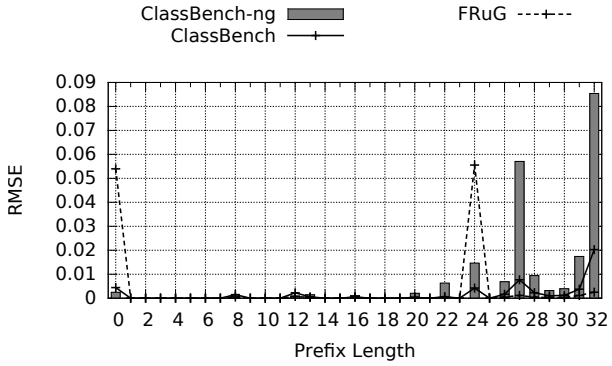
This section evaluates ClassBench-ng, focusing on the generation of IPv4 prefixes (Section 5.1), IPv6 prefixes (Section 5.2), and OpenFlow rules (Section 5.3). In the case of IPv4 prefixes we compare ClassBench-ng against ClassBench [18] and FRuG [7], while IPv6 prefixes generation fidelity is compared against Non-random Generator [20]. Finally, the *OpenFlow Generation* block is evaluated against FRuG [7]. We do not asses layer four ports and protocol generation, as ClassBench-ng relies directly on ClassBench for them.

The evaluation uses the *root-mean-square error* (RMSE), defined in Equation 3, 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.

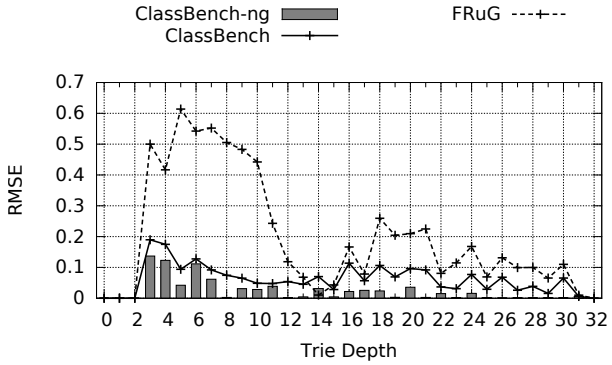
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{y} - y_i)^2} \tag{3}$$

5.1 IPv4 Prefixes Generation

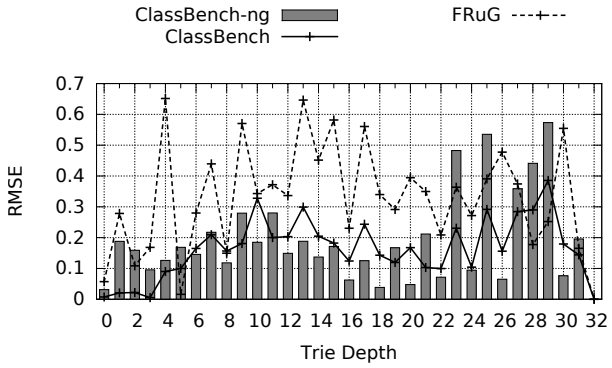
This section compares the RMSE of ClassBench-ng, ClassBench, and FRuG on IP prefix set parameters. We first generated an *original rule set* with ClassBench using the `acl4` seed provided with this tool. Then, capitalizing on



(a) Prefix length distribution.



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



(c) Average skew distribution.

Figure 15: Comparison of *root-mean-squared error* of ClassBench-ng, ClassBench, and FRuG in IPv4 prefix sets generation.

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 15. In terms of a branching probability distribution (Figure 15b), 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 15c). 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 15a 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 16 shows the average RMSE per trie level when all the evaluated parameters are considered at once. In this case, ClassBench-ng outperforms the other solutions in most of the trie levels, and in particular the $24th$, which is the most commonly used in operation (Section 3.1).

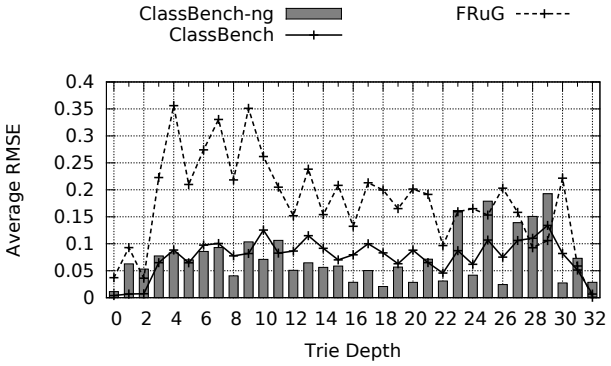


Figure 16: Average *root-mean-squared error* of ClassBench-ng, ClassBench, and FRuG in IPv4 prefix sets generation.

5.2 IPv6 Prefixes Generation

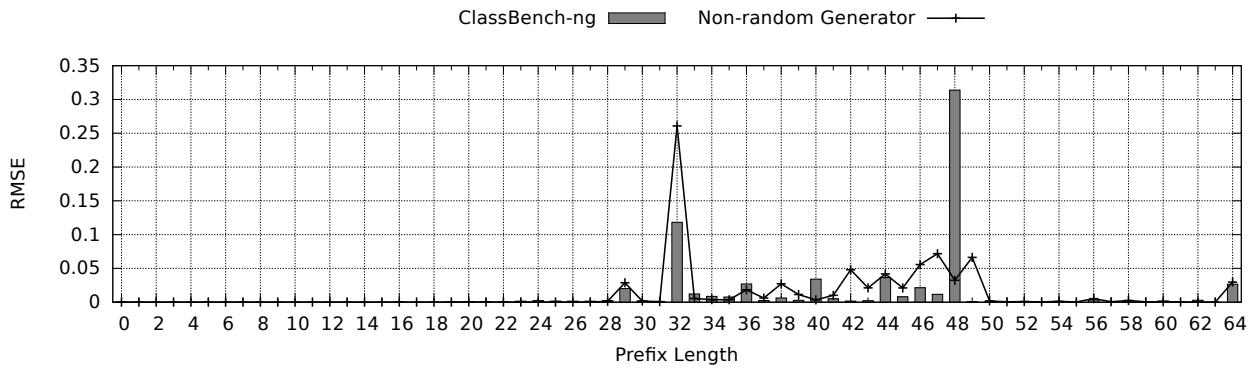
To evaluate the quality of IPv6 prefix set generation of ClassBench-ng against Non-random Generator, we used two prefix sets that come from the same core router. An input seed for ClassBench-ng was extracted from IPv6 prefix set `rrc00_2015`, while Non-random Generator's input consisted directly of IPv4 prefix set `rrc00_2015`. Although such a setup leads to a not entirely fair comparison of the tools, we notice that Non-random Generator requires an IPv4 prefix set to generate an IPv6 prefix set.

Results of the comparison are shown in Figures 17. Both ClassBench-ng and Non-random Generator achieve comparable quality of generation in terms of a prefix length distribution (Figure 17a). However, ClassBench-ng is more precise with respect to a branching probability distribution (Figure 17b) and Non-random Generator wins the comparison on an average skew distribution (Figure 17c).

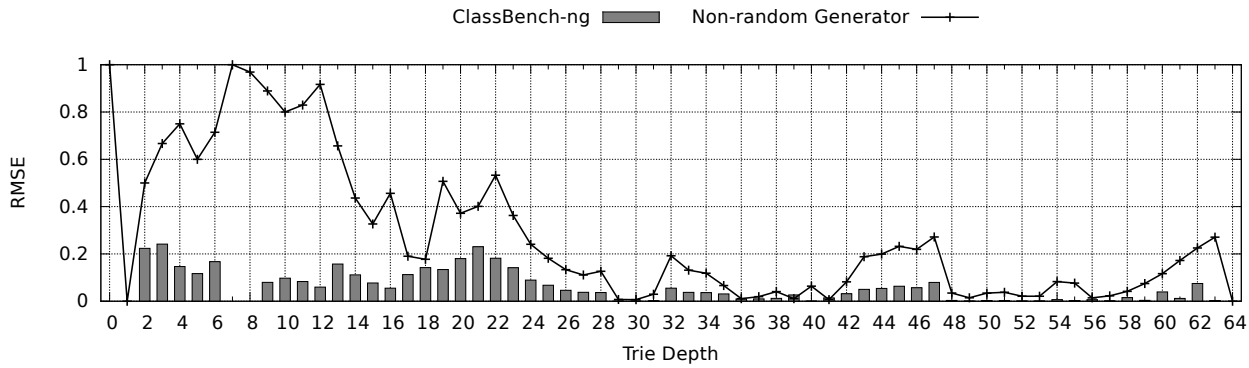
5.3 OpenFlow Rules Generation

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 3.3.2 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`.

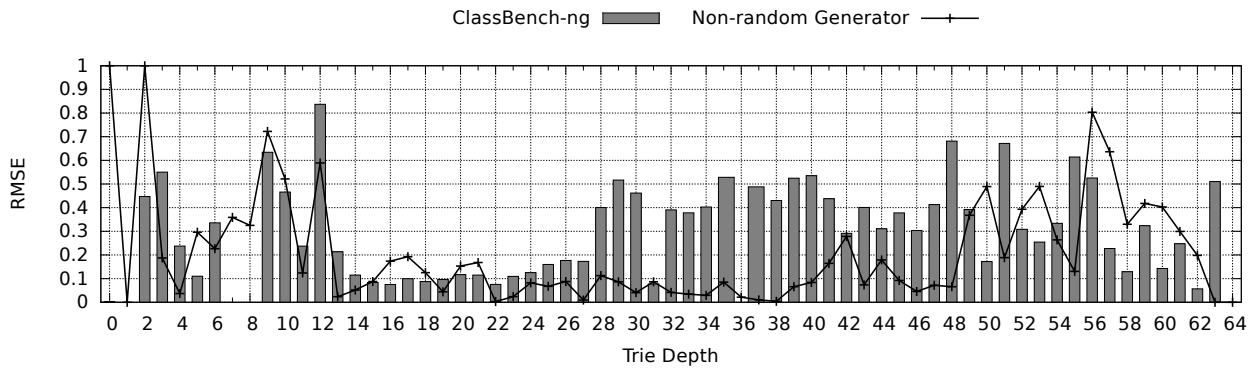
Figure 18a compares the ClassBench-ng *rule type* RMSE against the one obtained with FRuG. With respect to this



(a) Prefix length distribution.



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



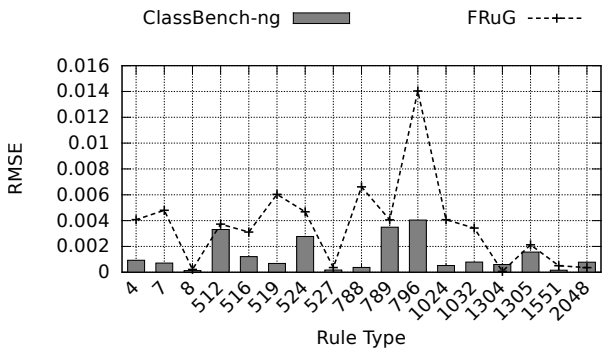
(c) Average skew distribution.

Figure 17: Comparison of *root-mean-squared error* of ClassBench-ng and Non-random Generator in IPv6 prefix sets generation.

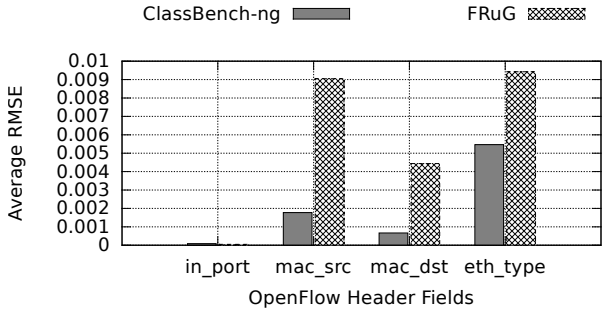
experiment, our tool clearly outperforms FRuG as it achieves higher RMSE only for *rule types* 1304 and 2048. Therefore, ClassBench-ng is more accurate in characterizing the relationship between header fields, i.e., which fields are more likely to be specified together in a rule. ClassBench-ng also proves to be be more accurate in the generation of selected OpenFlow-specific header fields (Figure 18b). 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 18c shows the RMSE for the values of vendor part of the *mac_dst* field. ClassBench-ng outperforms FRuG for all generated values.

6. RELATED WORK

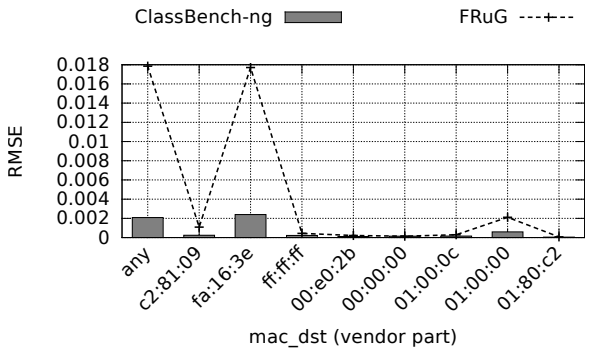
In the absence of publicly available classification rule sets, past researchers faced the problem of how to realistically assess the performance of new packet classification algorithms. While a limited number of research groups obtained access to real rule sets through confidentiality agreements, others 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



(a) OpenFlow *rule types*.



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



(c) Vendor part of *mac_dst*.

Figure 18: Comparison of *root-mean-squared error* of ClassBench-ng and FRuG in OpenFlow rule sets generation.

reflect anymore current research community needs, as it focuses only on IPv4.

Sun et al. [16] 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. [20] developed new algorithms for the synthetic generation of IPv6 forwarding tables. Following this effort, Zheng et al. [22] 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. [7] 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.

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 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., datacenter, Internet Service Provider, or Internet eXchange Point.

7. CONCLUSION

This paper presents ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow 1.0 flow rules. It accepts an input parameter file that can specify the statistical behavior for all the matching fields that need to be generated. Therefore, we analyzed real sets to understand their properties. To the best of our knowledge, this is the first attempt to characterize OpenFlow rules in a real environment. As a result, we provide input seeds that accurately reflect characteristics of different operational scenarios. In addition, to make this solution attractive in the long term and for a wide number of different use cases, ClassBench-ng offers a mechanic to create input parameter files from real rule sets. We aim to use a tool 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., datacenter, Internet Service Provider, Internet eXchange Point. This aspect will further increase the (potential) impact of ClassBench-ng on the research community. Finally, by providing an open source solution we invite everyone from the community to audit (and improve) our implementation as well as adapt it to their needs, e.g., newer version of OpenFlow or introducing a multi-table support.

Given the increasing complexity of the rule matching problem, i.e., from IPv4 to OpenFlow, but also the dependency between characteristics of rule sets and packet classification algorithms optimization, we feel this tool can meet the requirements of nowadays researchers. We hope ClassBench-ng will boost the rule matching research as ClassBench has done since ten years ago.

8. ACKNOWLEDGMENTS

We thank Viktor Puš for his feedback and comments.

This research was supported by the European Union's Horizon 2020 research and innovation programme 2014-2018 under the SSICLOPS (grant agreement No. 644866) and ENDEAVOUR (grant agreement No. 644960). It was also supported by The Ministry of Education, Youth and Sports of the Czech Republic from the "CESNET E-infrastructure" project No. LM2015042 and the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602.

9. REFERENCES

[1] Differentiated Services Field Codepoints (DSCP). http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

[2] IPv4 address report. http://www.potaroo.net/tools/ipv4.

[3] IPv6 Deployment Status. https://www.vyncke.org/ipv6status.

[4] OpenFlow 1.0 specification. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[5] Virtual Bridged Local Area Networks. *IEEE Standard 802.1Q*, 2005.

[6] J. Czyz, M. Allman, J. Zhang, S. Iekel-Johnson, E. Osterweil, and M. Bailey. Measuring IPv6 Adoption. In *SIGCOMM*. ACM, 2014.

[7] T. Ganegedara, W. Jiang, and V. K. Prasanna. FRuG: A benchmark for packet forwarding in future networks. In *IPCCC*. IEEE, 2010.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*. ACM, 2013.

[9] W. Jiang and V. K. Prasanna. Scalable Packet Classification on FPGA. *Transactions on Very Large Scale Integration Systems*, 20(9), 2012.

[10] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. Mckeown. Maturing of OpenFlow and Software-defined Networking Through Deployments. *Computer Network*, 61, 2014.

[11] H. Lim, N. Lee, G. Jin, J. Lee, Y. Choi, and C. Yim. Boundary Cutting for Packet Classification. *Transactions on Networking*, 22(2), 2014.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *CCR*, 38(2), 2008.

[13] H. Song and J. S. Turner. Toward Advocacy-Free Evaluation of Packet Classification Algorithms. *Transactions on Computers*, 6(5), 2011.

[14] H. Song and J. S. Turner. ABC: Adaptive Binary Cuttings for Multidimensional Packet Classification. *Transactions on Networking*, 21(1), 2013.

[15] J. P. Stringer, Q. Fu, C. Lorier, R. Nelson, and C. E. Rothenberg. Cardigan: Deploying a Distributed Routing Fabric. In *HotSDN*. ACM, 2013.

[16] Q. Sun, X. Huang, W. Yang, X. Zhou, Y. Ma, and C. Wang. ClassBenchv6: An IPv6 Packet Classification Benchmark. In *GLOBECOM*. IEEE, 2009.

[17] D. E. Taylor. Survey and Taxonomy of Packet Classification Techniques. *Computing Surveys*, 37(3), 2005.

[18] D. E. Taylor and J. S. Turner. ClassBench: A Packet Classification Benchmark. *Transactions on Networking*, 15(3), 2007.

[19] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*. ACM, 2010.

[20] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random Generator for IPv6 Tables. In *HOTI*. IEEE, 2004.

[21] Y. Xu, Z. Liu, Z. Zhang, and H. J. Chao. High-throughput and Memory-efficient Multimatch Packet Classification Based on Distributed and Pipelined Hash Tables. *Transactions on Networking*, 22(3), 2014.

[22] K. Zheng and B. Liu. V6Gene: A Scalable IPv6 Prefix Generator for Route Lookup Algorithm Benchmark. In *AINA*. IEEE, 2006.