BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

ADDRESSING ISSUES IN RESEARCH ON PACKET CLASSIFICATION IN CORE NETWORKS

ŘEŠENÍ PROBLÉMŮ VE VÝZKUMU KLASIFIKACE PAKETŮ V PÁTEŘNÍCH SÍTÍCH

PHD THESIS
DISERTAČNÍ PRÁCE

AUTOR **Ing. JIŘÍ MATOUŠEK**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN KOŘENEK, Ph.D.**
ŠKOLITEL

BRNO 2018

Abstract

Although the Internet has changed significantly since the beginning of the 21st century, packet classification is still one of the most common operations implemented in networking devices. Nevertheless, the requirements on its performance are continuously increasing, especially in core networks. Currently, packet classification algorithms have to support 100 Gbps throughput. In addition, classification rule sets are becoming larger and the number of bits involved in the classification decision is growing due to 128-bit IPv6 addresses and classification according to more than 5 header fields in the OpenFlow protocol. Therefore, the majority of contemporary research on packet classification in core networks address the performace of packet classification algorithms, which has to keep pace with continuously increasing requirements. However, the researchers also focus on benchmarking newly developed algorithms because they have to be benchmarked using real rule sets, but such data are not available for most of the packet classification use cases. This thesis deals with both of these issues because it is important not only to design packet classification algorithms having high performance but also to assess their parameters by benchmarking based on proper data sets.

Regarding the performace of packet classification algorithms, this thesis focuses on improving prefix matching, which is used in the majority of 1-dimensional and also multi-dimensional algorithms. Since a software implementation of prefix matching cannot fulfill the requirements imposed on packet classification in core networks, the thesis proposes a novel pipelined prefix matching architecture that targets Xilinx FPGA chips and utilizes their distributed on-chip memory. To fit the whole prefix matching data structure into FPGA's on-chip memory, this thesis also proposes a memory-efficient trie-based representation of a prefix set. The proposed representation is more memory efficient than well-known multibit tries Tree Bitmap and Shape Shifting Trie and for IPv4 prefix sets it also significantly overcomes the Prefix Partitioning lookup algorithm. The architecture then comprises two independent processing pipelines (to utilize both ports of on-chip memory blocks) that are together able to perform almost 255 million lookups per second, which translates into throughput of 170 Gbps for the shortest Ethernet frames.

To allow realistic packet classification algorithms benchmarking, the thesis introduces a new open source synthetic rule set generator called ClassBench-ng, which integrates the generation of IPv4, IPv6, and OpenFlow 1.0.0 classification rule sets following the statistical properties specified in an input seed. Apart from the rule set generation, ClassBench-ng also supports an analysis of a real rule set in the `ovs-ofctl` format producing a corresponding seed that may be used for the generation of a similar synthetic rule set later on. Therefore, researchers having access to real classification rule sets can share their benchmarking data with other members of the community via statistical-based (thus anonymous) seeds produced by ClassBench-ng. With respect to the precision of the rule set generation process, ClassBench-ng is better than original ClassBench and FRuG in case of IPv4 prefixes and than Non-random Generator in case of IPv6 prefixes, when considering an average score for all IP prefix-related parameters. Moreover, it also clearly outperforms FRuG in the precision of OpenFlow rule sets generation.

Abstrakt

Ačkoliv se Internet od počátku 21. století znatelně proměnil, klasifikace paketů je stále jednou z nejběžnějších operací implementovaných v síťových zařízeních. Požadavky na její výkonnost však neustále narůstají, zvláště pak v oblasti páteřních sítí, kde musejí současné klasifikační algoritmy podporovat propustnost 100 Gb/s. Navíc dochází i k nárůstu počtu používaných klasifikačních pravidel a v souvislosti se zavedením 128bitových adres v protokolu IPv6 a klasifikace paketů založené na více než 5 položkách v protokolu OpenFlow se také zvyšuje počet bitů majících vliv na výsledek procesu klasifikace. Většina současného výzkumu v oblasti klasifikace paketů v páteřních sítích se proto zaměřuje na zvyšování výkonnosti klasifikačních algoritmů, které musejí držet krok s neustále narůstajícími požadavky. Výzkumníci se však také věnují měření výkonnostních parametrů nově vyvinutých klasifikačních algoritmů, respektive generování vhodných syntetických sad klasifikačních pravidel pro tato měření, protože reálné sady pravidel nejsou pro většinu oblastí, v nichž se klasifikace paketů používá, dostupné. Tato práce se zaměřuje na obě uvedené oblasti, protože je nutné nejen vyvíjet klasifikační algoritmy s vysokou výkonností, ale také ověřovat jejich výkonnostní parametry s využitím vhodných datových sad.

V oblasti výkonnosti klasifikačních algoritmů se tato práce věnuje především úloze vyhledávání shodných prefixů, která je součástí většiny jednodimenzionálních i vícedimenzionálních klasifikačních algoritmů. Jelikož softwarová implementace vyhledávání shodných prefixů nemůže dostát požadavkům na klasifikaci paketů v páteřních sítích, tato práce navrhuje implementaci dané úlohy pomocí zřetězené architektury určené pro FPGA čipy firmy Xilinx a využívající distribuovaných paměťových bloků dostupných na těchto čipech. Kromě toho je v této práci navržena i paměťově efektivní reprezentace sady prefixů založená na binárním prefixovém stromu (tzv. trie), která umožňuje uložit celou datovou strukturu potřebnou pro vyhledávání shodných prefixů do paměti na FPGA čipu. Navržená reprezentace prefixové sady je s ohledem na paměťové nároky efektivnější než reprezentace používané ve známých vícebitových trie Tree Bitmap a Shape Shifting Trie a v případě IPv4 prefixů také výrazně překonává reprezentaci používanou algoritmem Prefix Partitioning. Samotná architektura pro vyhledávání shodných prefixů pak sestává ze dvou zřetězených linek využívajících oba dva porty paměťových bloků dostupných na FPGA čipu, které jsou dohromady schopné provést téměř 255 milionů vyhledání za sekundu, což pro nejkratší rámce protokolu Ethernet znamená propustnost 170 Gb/s.

Pro podporu realistického měření výkonnostních parametrů klasifikačních algoritmů představuje tato práce nový volně dostupný generátor syntetických sad klasifikačních pravidel nazývaný ClassBench-ng, který umožňuje generování IPv4, IPv6 a OpenFlow 1.0.0 pravidel, jejichž statistické vlastnosti odpovídají rozložením popsaným ve vstupním souboru parametrů. Kromě generování syntetických klasifikačních pravidel podporuje tento nástroj také analýzu reálných pravidlových sad ve formátu `ovs-ofctl`, jejímž výstupem je odpovídající soubor parametrů, který může být později použit pro generování podobné syntetické sady pravidel. Výzkumníci mající přístup k reálným sadám klasifikačních pravidel tudíž mohou sdílet svá testovací data s ostatními členy výzkumné komunity prostřednictvím souborů parametrů vytvořených nástrojem ClassBench-ng, které popisují pouze statistické vlastnosti původních pravidlových sad, a tudíž zachovávají jejich anonymitu. S ohledem na přesnost generování syntetických sad klasifikačních pravidel dosahuje nástroj ClassBench-ng lepších průměrných výsledků než nástroje ClassBench a FRuG v případě IPv4 prefixů a než nástroj Non-random Generator v případě IPv6 prefixů. Generování syntetických sad OpenFlow pravidel je pak v tomto nástroji přesnější než v nástroji FRuG.

Keywords

Packet classification, core networks, prefix matching, FPGA, OpenFlow, ClassBench-ng.

Klíčová slova

Klasifikace paketů, páteřní sítě, vyhledávání shodných prefixů, FPGA, OpenFlow, ClassBench-ng.

Reference

MATOUŠEK, Jiří. *Addressing Issues in Research on Packet Classification in Core Networks*. Brno, 2018. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Kořenek, Ph.D.

Addressing Issues in Research on Packet Classification in Core Networks

Declaration

I hereby declare that this Ph.D. thesis is my original work written under the supervision of Ing. Jan Kořenek, Ph.D. All relevant information sources, which were used during the preparation of this thesis, are properly cited and included in the list of references.

. . . .
Jiří Matoušek
November 30, 2018

Acknowledgements

First of all, I thank to my supervisor Ing. Jan Kořenek, Ph.D., for his numerous valuable comments and suggestions as well as great patience and support during the whole period of my Ph.D. studies. My thanks also go to my former supervisor doc. Ing. Zdeněk Kotásek, CSc., who took me under his wing in the first two years of this period. Further, I am grateful to Dr. Andrew W. Moore for hosting me at the Computer Laboratory, University of Cambrige, between October and December 2014 and to Dr. Gianni Antichi, who was my mentor during that visit and has virtually been my co-supervisor since that time. I thank to my co-authors for inspiring discussions and enjoyable cooperation; colleagues from the Department of Computer Systems, Faculty of Information Technology, Brno University of Technology, for their helpful comments on various parts of this thesis; and colleagues from the Security and Administration Tools department at CESNET for creating a friendly working environment. Last but not least, I am grateful to my family, friends, and my girlfriend Karolína for their endless support and patience.

The work presented in this thesis was supported by the Ministry of Education, Youth and Sports of the Czech Republic (under the CESNET E-infrastructure project LM2015042 and the IT4Innovations excellence in science project LQ1602), the Technology Agency of the Czech Republic (under the project TA03010561), the IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT FIT projects FIT-S-11-1, FIT-S-14-2297, and FIT-S-17-3994. The visit to the Computer Laboratory, University of Cambridge, was supported by the Ministry of Education, Youth and Sports of the Czech Republic from the Programme of Student Mobility Support.

Contents

1 Introduction **3**
1.1 Thesis Goals . . . 4
1.2 Thesis Organization . . . 5

2 Packet Classification in Core Networks **6**
2.1 Definition . . . 6
2.2 Use Cases . . . 7
2.3 Approaches to Packet Classification . . . 9
2.3.1 Naive Approaches . . . 10
2.3.2 TCAM . . . 11
2.3.3 Representation Using Tuples . . . 12
2.3.4 Geometric Representation . . . 14
2.3.5 Combinatorial Representation . . . 15
2.3.6 Range Matching . . . 16
2.3.7 Prefix Matching . . . 17
2.4 Research Issues . . . 17
2.4.1 Performance of Algorithms . . . 17
2.4.2 Algorithms Benchmarking . . . 19

3 Related Work **21**
3.1 1-Dimensional Packet Classification Algorithms . . . 21
3.1.1 Trie . . . 22
3.1.2 Tree Bitmap . . . 23
3.1.3 Shape Shifting Trie . . . 24
3.1.4 Multiple SRAM-based Lookup Algorithm . . . 25
3.1.5 Prefix Partitioning Lookup Algorithm . . . 26
3.2 Multi-Dimensional Packet Classification Algorithms . . . 27
3.2.1 Exhaustive Search . . . 27
3.2.2 Decision Tree . . . 28
3.2.3 Decomposition . . . 31
3.2.4 Tuple Space . . . 37
3.3 1-Dimensional Rule Set Generators . . . 39
3.3.1 Non-random Generator . . . 39
3.3.2 V6Gene . . . 40
3.4 Multi-Dimensional Rule Set Generators . . . 41
3.4.1 ClassBench . . . 42
3.4.2 ClassBenchv6 . . . 44
3.4.3 FRuG . . . 44

4 Addressing Performance of Algorithms **47**
4.1 Analysis of Real Prefix Sets 48
4.2 Proposed Prefix Set Representation 51
4.3 Proposed Hardware Architecture 54
4.4 Experimental Evaluation 57
4.5 Summary 60

5 Addressing Algorithms Benchmarking **62**
5.1 Challenges in Rule Set Generation 63
5.2 Analysis of Real Rule Sets 64
5.2.1 IP Prefixes 64
5.2.2 Ports and Protocol 68
5.2.3 OpenFlow 70
5.3 ClassBench-ng: Next Generation ClassBench 73
5.3.1 Improved ClassBench 75
5.3.2 IPv6 Generation 78
5.3.3 OpenFlow Analysis 78
5.3.4 OpenFlow Generation 80
5.4 Experimental Evaluation 81
5.4.1 IPv4 Prefixes Generation 81
5.4.2 IPv6 Prefixes Generation 83
5.4.3 OpenFlow Rules Generation 83
5.5 Summary 86

6 Conclusions **88**
6.1 Summary of Contributions 90

Bibliography **91**

Chapter 1

Introduction

Computer networks are an indispensable part of our everyday life. We use them as a source of information and a means for communication at work as well as in our free time. The most prominent computer network today is a global network called the Internet. It evolved from research networks during 1970s and 1980s and established as a publicly available global network in 1990s. However, since the beginning of 21st century we have experienced rapid technology development that has significantly changed the Internet. The current Internet is no more the one we knew at the beginning of this century.

There are two main technological drivers of Internet changes: (1) growing transfer rates and (2) increasing number of mobile subscriptions. Significant growth of transfer rates can be illustrated on Ethernet, which is the most utilized LAN (*Local Area Network*) technology for fixed subscriptions. While the highest standardized transfer rate for Ethernet was 1 Gbps in 2000 [6], the current operational maximum is 100 Gbps [9]. In addition, since the standard for 400 Gbps Ethernet [13] has been approved in December 2017, the upgrade of the maximum transfer rate in core networks can be expected in the near future. On the other hand, accessing the Internet from mobile devices has been enabled mainly by the 802.11 family of standards [12] and the IMT-2000 standard [34], which together started the advent of mobile-broadband subscriptions. According to ITU (*International Telecommunication Union*) data [33], in 2016 there were 52.2 active mobile-broadband subscriptions per 100 inhabitants worldwide.

Technology development made access to the Internet more affordable, even in developing countries. The number of individuals using the Internet grew from 495 million in 2001 to 3385 million in 2016 [33]. Moreover, users often own several devices that are able to access the Internet (e.g., personal computer, tablet, smartphone, smart TV, or intelligent sensor). This resulted in 17.1 billion of connected devices (i.e., 2.3 devices per capita) in 2016 and forecasted growth to 27.1 billion (3.5 per capita) in 2021, as reported by Cisco [14]. Because the maximum number of distinct IPv4 (*Internet Protocol version 4*) addresses is less than 4.3 billion, all RIRs (*Regional Internet Registries*) except AFRINIC (*AFRIcan Network Information Center*) exhausted their IPv4 allotments between 2011 and 2015 [14]. Therefore, since 2011 we have experienced accelerated adoption of IPv6 [30], the successor of IPv4. However, not only the principal protocol of the Internet is changing. Architecture of computer networks is being redefined as well, especially due to the concept of network virtualization. Although there are numerous, often competing, technologies enabling this concept at various levels of network architecture [35], one of the most promising approaches is SDN (*Software-Defined Networking*), which regained interest after the introduction of OpenFlow [44], currently the most common protocol for communication between control

and data planes of a switch. Since that time, OpenFlow-based SDN has proven to be a viable approach through a number of successful deployments in networks ranging from LANs to WANs (*Wide Area Networks*) [37].

Despite all the changes of the Internet were brought to life by upgrading its infrastructure, packet classification at physical link speed is still one of the most common operations implemented in networking devices. Upon arrival, a networking device classifies every packet according to one or more of its header fields and uses the result of classification for further processing of the packet. Depending on the function of a device, the result of classification may be used for basic forwarding operation, to apply security policies, application-specific processing, or QoS (*Quality of Service*) guarantees.

Even though packet classification has not been replaced by another functionality, requirements on its performance are continuously increased. For instance, because of growth of transfer rates to 100 Gbps and extremely high utilization of the IPv4 address space, a core router has to be able to make a forwarding decision according to a forwarding table containing more than 680 thousand IPv4 prefixes [3] every 6.72 ns. With respect to the IPv6 protocol, due to 4-times longer IP address involved in the forwarding process, the situation is even worse. Currently, the number of IPv6 prefixes in a forwarding table of core routers is almost 43 thousand [3]; however, this number is expected to grow together with IPv6 penetration. Another example of growing demands on packet classification is extending the set of packet header fields involved in the classification process. While the most common set of header fields involved in packet classification consists of 5 header fields, the OpenFlow protocol initially extended this set to 12 fields [7] and the latest version of the OpenFlow protocol defines packet classification based on 45 fields [10]. Both these examples demonstrate that packet classification requires continuous attention of researchers.

From a high level perspective, there can be identified two issues that are addressed in the research on packet classification in core networks.

The first issue is related to the performance of packet classification algorithms, which has to keep pace with growing transfer rates. Parameters that have the greatest influence on the performance are the number of bits involved in packet classification (i.e., the number of utilized header fields and their length) together with the number of utilized classification rules. Therefore, new algorithms have to deal with growing popularity of SDN-based network virtualization utilizing the OpenFlow protocol (more fields) [35] and/or increasing IPv6 penetration (longer fields) [30]. Moreover, since current CPUs (*Central Processing Units*) do not provide enough performance for even 1-dimensional packet classification according to destination IP address (i.e., IP lookup) on a 100 Gbps link, packet classification algorithms targeting core networks have to be accelerated in hardware.

The second issue is related to benchmarking packet classification algorithms, which are continuously improved to meet ever-increasing requirements on their performance. Because real classification rule sets are not usually available for benchmarking, researchers designed and implemented several tools capable of generating synthetic rule sets [66, 67, 63, 59, 29]. Nevertheless, even together these tools are not able to generate all data sets necessary for benchmarking current packet classification algorithms. In addition, it can be shown that the process of rule set generation in the currently available tools is not always accurate.

1.1 Thesis Goals

This thesis aims to address identified issues in the research on packet classification in core networks via achieving the following two goals.

The **first goal** is to address the issue related to the performance of packet classification algorithms by designing a hardware-accelerated prefix matching algorithm that will be able to achieve 100 Gbps throughput for both IPv4 and IPv6 protocols.

The **second goal** is to address the issue related to benchmarking new packet classification algorithms using a tool capable of generating synthetic IPv4, IPv6, and OpenFlow 1.0.0 rule sets with parameters similar to real rule sets.

1.2 Thesis Organization

A brief introduction of the research area and the goals of the thesis in Chapter 1 is followed by their detailed description in Chapter 2. Firstly, this description focuses on the definition of packet classification and its typical use cases. It also presents various 1-dimensional and multi-dimensional approaches to packet classification, which form the basis of current packet classification algorithms. The last section of Chapter 2 then contains a detailed discussion of two main issues in the research on packet classification that are addressed by the goals of this thesis.

Before addressing the performance of packet classification algorithms (the first issue) and their benchmarking (the second issue), the thesis summarizes previous work related to these issues in Chapter 3. With respect to the first issue, related work is represented by selected packet classification algorithms. On the other hand, the second part of this chapter presents several generators, which are able to produce synthetic classification rule sets that are utilized when addressing the second issue. In the whole chapter, 1-dimensional and multi-dimensional approaches are described separately.

The first issue in research on packet classification is addressed in Chapter 4. The core of this chapter describes a memory-efficient IP prefix set representation and a hardware architecture implementing IP lookup using this representation. The chapter also contains an analysis of properties of real IP prefix sets and an experimental evaluation of both the IP prefix set representation and the hardware architecture, including their comparison to related work.

Chapter 5 addresses the second issue in packet classification research. The first part of the chapter is devoted to a detailed analysis of real classification rule sets. After that, ClassBench-ng, a new tool capable of generating synthetic rule sets, is introduced and its properties are compared with other similar tools.

The thesis is concluded with Chapter 6 that evaluates the proposed goals, summarizes contributions of the thesis and outlines possible future work in the given research area.

Chapter 2

Packet Classification in Core Networks

The main aim of this chapter is to introduce the area of packet classification and discuss in details related research issues. First of all, Section 2.1 defines the term packet classification and introduces various ways of specifying conditions that form a classification rule. To support the claim that packet classification is utilized in virtually every networking device, Section 2.2 describes typical use cases of this operation along with the most common formats of classification rules. Next, Section 2.3 is devoted to 1-dimensional as well as multi-dimensional approaches to packet classification, which represent the key ideas behind current packet classification algorithms. The chapter is concluded in Section 2.4 containing the discussion of issues addressed in the current research on packet classification, especially in core networks. Because of continuously increasing requirements on the performance of packet classification algorithms, research in this area is still ongoing and it is centered around the performance of packet classification algorithms and their benchmarking.

2.1 Definition

Packet classification is a process determining a class (often referred to as a network flow) that a packet belongs to. The input of packet classification consists of selected header fields extracted from the packet and a set of classification rules with defined priorities, in which each rule represents one class. A classification rule defines a condition for every header field extracted from input packets. The condition is usually specified in one of the following four ways.

- **Value** — exactly one allowed value (typical for transport layer protocol).
- **Prefix** — a range of allowed values having a common binary prefix (typical for source and destination IP addresses).
- **Range** — an arbitrary range of allowed values (typical for source and destination transport layer ports).
- **Wildcard** — any value is allowed (typical for header fields not important for the corresponding class).

The most general of the presented specifications is the range because every other specification can be represented as a single range. However, a condition using the range speci-

fication is the most difficult to verify. Therefore, the prefix specification is often used as a uniform way of specifying a condition. An exact value and a wildcard can be replaced by a single prefix of maximum and zero length, respectively, but the number of prefixes required for replacing a range is $2l - 2$ [46], l being the length of the corresponding header field, in the worst case. Such a scenario is shown in Figure 2.1.

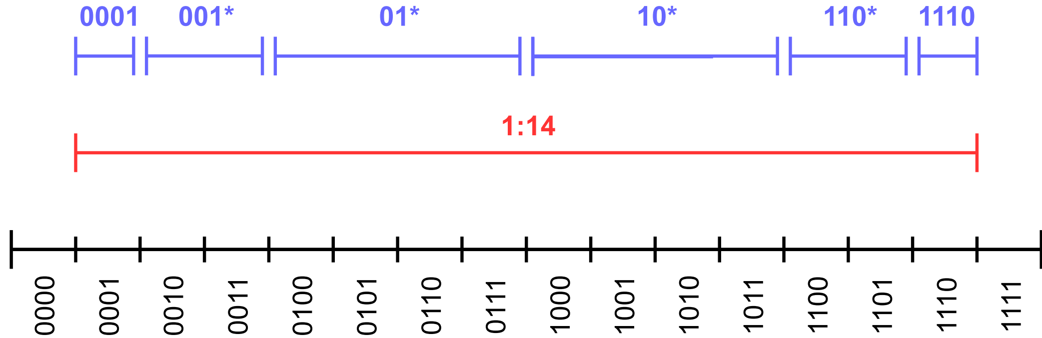


Figure 2.1: Example of replacing the range 1:14 (red interval) on a 4-bit header field by a set of $2 \cdot 4 - 2 = 6$ prefixes (blue intervals).

For the sake of completeness, it should be noted that any of the specifications can also be replaced by a set of exact values containing one item for each value from the specified range. On the other hand, the wildcard specification can only replace special cases of a range (full range) and a prefix (empty prefix).

Regardless the utilized specification, a condition is satisfied when the corresponding header field of a packet contains one of the allowed values. If all conditions of a classification rule are satisfied, then the packet belongs to the corresponding class. Note that classes may overlap, thus multiple classification rules can match the packet. In such a case the matching rule with the highest priority is selected as the output of packet classification. Since packet classes usually define specific processing for their packets, the output of packet classification can also be an action that is going to be applied on the classified packet.

2.2 Use Cases

Packet classification is implemented in a vast majority of networking devices deployed in operating environments ranging from small edge networks to core networks forming the backbone of the Internet. Although each implementation complies with the definition presented in the previous section, its details may vary according to a specific scenario it is used in. The most common use cases of packet classification are briefly introduced in the following list. For the sake of clarity, the list does not present each use case individually, but it focuses on five categories that group similar use cases together.

Routing. In these use cases, the result of packet classification determines a route to a packet's destination. The simplest form of routing—IP routing—is based on matching a destination IP address only. However, routing may also be based on more sophisticated policies employing packet classification according to multiple header fields (e.g., routing packets with the same destination IP address but different transport layer protocol via different links).

Packet filtering. This group of packet classification use cases belongs to the area of network security. Each classification rule defines a specific action (e.g., accept, drop, reject, intercept) applied to a corresponding network flow (i.e., to all packets that are matched by the rule). The most common example of packet filtering is a firewall, which is able to filter network traffic flowing through a given point of a network. For instance, a firewall deployed at the border of a network may filter malicious traffic destined for the network. Another use case of packet filtering is represented by lawful interception systems implementing the interception of communication specified by law enforcement agency in the course of a criminal investigation.

Application-specific processing. Similarly to packet filtering, packet classification allows to process the traffic of a specific application—as long as it can be characterized by header fields utilized in a classification rule—differently from other traffic. Nevertheless, a set of actions that can be used for processing the traffic is richer than in the packet filtering case. Even though it is not the only example of application-specific processing, a typical use case in this group is the application of QoS guarantees via traffic shaping (i.e., delaying packets of some or all network flows in order to achieve a desired traffic profile).

Traffic management. Packet classification may also be used for intelligent traffic management that allows to treat network flows differently from each other rather than in the same way. For instance, load balancing and rate limiting enriched by the notion of network flows can perform fine-grained management decisions that do not affect flows carrying sensitive network traffic. Using packet classification, service providers can also do traffic accounting and billing on a per-user basis, even when all users access the service via a shared link.

SDN. Although this group does not really introduce any new use cases, it highlights the utilization of packet classification in a data plane of SDN-enabled devices. With the help of a programmable control plane an SDN architecture can therefore implement any of the already introduced uses cases. Clearly, an SDN Controller can either install necessary rules into flow tables of an SDN Datapath (e.g., for IP routing, policy-based routing, and firewall) or instruct the SDN Datapath to send packets not matching any rule to the SDN Controller for further processing (e.g., for lawful interception, accounting, and billing).

In different use cases, the number of header fields involved in the packet classification process ranges from a single field (destination IPv4/IPv6 address in case of IP routing) up to 45 fields (classification based on OpenFlow 1.5.1 [10]). Table 2.1 specifies the structure of the most often utilized types of packet classification rules that are also considered further in the thesis. Mapping of labels used in the table to particular header fields is as follows.

- *in_port* — input port number
- *mac_src* — source MAC address
- *mac_dst* — destination MAC address
- *eth_type* — EtherType
- *vlan_id* — VLAN ID

- *vlan_prio* — VLAN priority
- *ip_tos* — DSCP (originally ToS)
- *ip_proto* — IP protocol
- *ip_src* — source IP address (IPv4 or IPv6)
- *ip_dst* — destination IP address (IPv4 or IPv6)
- *l4_src* — source TCP/UDP port
- *l4_dst* — destination TCP/UDP port

Header Field	*in_port*	*mac_src*	*mac_dst*	*eth_type*	*vlan_id*	*vlan_prio*	*ip_tos*	*ip_proto*	*ip_src* (IPv4)	*ip_src* (IPv6)	*ip_dst* (IPv4)	*ip_dst* (IPv6)	*l4_src*	*l4_dst*
Data Width [b]	x	48	48	16	12	3	6	8	32	128	32	128	16	16
IPv4 prefix											✓			
IPv6 prefix												✓		
IPv4 5-tuple								✓	✓		✓		✓	✓
IPv6 5-tuple								✓		✓		✓	✓	✓
OpenFlow 1.0.0	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓

Table 2.1: Packet header fields utilized in various types of classification rules considered in this thesis. The table also shows standard data width of each field (data width of *in_port* depends on a specific implementation).

While some types of classification rules specified in Table 2.1 differ from each other by the version of IP protocol only (i.e., IPv4 vs. IPv6), others utilize a different number of header fields and are used in different use cases. A typical use case of a single destination IPv4/IPv6 prefix is IP routing. Next, 5 header fields of an IPv4/IPv6 5-tuple are used in many use cases, for instance in a firewall to name just one of them. Last but not least, OpenFlow 1.0.0 rules allowing specification of a condition for up to 12 header fields are used for various purposes in SDN-enabled networks, for example in datacenters.

2.3 Approaches to Packet Classification

The problem of packet classification has been approached in many different ways that are described in this section. To facilitate an understanding of these approaches, they are illustrated on classification of the packet P with *address* = 110 and *port* = 110 using a rule set from Table 2.2, which is used as a running example throughout this section. The rule set utilized in the running example consists of rules specifying an address prefix and a port range, both on corresponding 3-bit header fields. The priority of the rules is descending with their position in the rule set (i.e., the first rule has the highest priority).

Apart from publications cited in particular subsections, the whole section is based on information gathered from [46, 60, 8].

Rule	Address	Port
R1	0*	5:7
R2	*	6:6
R3	010	0:7
R4	11*	5:7
R5	11*	1:4

Table 2.2: An example rule set for 2-dimensional packet classification according to 3-bit header fields *address* and *port*. The rules are sorted from the highest to the lowest priority.

2.3.1 Naive Approaches

The simplest approach to packet classification is a linear search of a rule set with rules sorted from the highest to the lowest priority. An input packet is sequentially matched against classification rules and the first matching rule is selected as the output of packet classification. Such an early termination is correct because if there are other matching rules, they will have lower priority. In the running example introduced at the beginning of Section 2.3, the linear search stops after matching the packet P against the rule R2, although the rule R4 also matches the packet.

Both search time and memory requirements of this approach are linear with respect to the number of rules. While linear search time represents the worst performance among the presented approaches, linear memory requirements are close to the optimal solution (note that different rules may redundantly use the same condition for a particular header field). Because of its search performance, this approach is feasible only for small rule sets, for example the one from Table 2.2, in which a search takes 5 steps in the worst case.

An orthogonal approach trades-off memory requirements for better search time by pre-computing the best matching rule for every possible packet and storing this information into a table. Classification of an incoming packet then consists of addressing the table by concatenated header field values and reading the best matching rule information. This is illustrated using Table 2.3, which shows an excerpt of a precomputed best-matching-rule table for the running example. The excerpt is centered around the entry indicating that the rule R2 is the best matching rule for the packet P with $address = 110$ and $port = 110$.

Address	Port	Rule
⋮	⋮	⋮
110	100	R5
110	101	R4
110	110	R2
110	111	R4
111	000	N/A
⋮	⋮	⋮

Table 2.3: An excerpt of a precomputed best-matching-rule table for the example rule set (Table 2.2). The best matching rule for the last entry is not available (N/A) because no rule matches the corresponding packet.

Search time of this second naive approach is constant (classification is done in a single step) but its memory requirements are exponentially dependent on the number of bits involved in the classification process. The example rule set from Table 2.2 classifies packets according to two 3-bit header fields, thus the corresponding table of best matching rules consists of $2^{3+3} = 64$ items. However, in a real-world example of packet classification according to the IPv4 5-tuple, the required table size is $2^{32+32+8+16+16} = 2^{104}$ items, even for a rule set containing only a few rules. Therefore, despite its excellent search time, this approach to packet classification is important mainly from a theoretical point of view.

2.3.2 TCAM

Another straightforward approach to packet classification is to use TCAM (*Ternary Content-Addressable Memory*). This special kind of memory is based on CAM (*Content-Addressable Memory*), which consists of rows addressed by their content. The output of CAM is the address of a row, whose content matches the input value. While CAM supports only exact matching of the input value, TCAM extends matching functionality to so-called ternary matching. Apart from the bit values 0 and 1, a row can also contain the bit value X (don't care), which allows the input to contain an arbitrary value at the corresponding position.

Since TCAM supports addressing by content and ternary matching, it can be viewed as the second naive approach with the ability to use the prefix specification. Indeed, each TCAM record (i.e., a rule after the conversion of all conditions to prefixes augmented by the X value for each unspecified bit) represents all entries of the second naive approach's table that correspond to the packets matching the record. The number of TCAM records is thus usually significantly smaller than the number of entries in the best-matching-rule table of the second naive approach. Nevertheless, the use of the prefix specification causes that more than one record may match a given packet. Therefore, packet classification based on TCAM has to consist of ternary matching an input packet in TCAM and the following selection of the highest priority rule among those associated with the matching TCAM records. Table 2.4 shows TCAM records representing the running example's rule set (see Table 2.2). The example packet P is matched by records 3 and 6, which have associated rules R2 and R4, respectively. Because of its higher priority, the rule R2 is selected as the best matching one.

#	Address	Port	Rule
1	0XX	101	R1
2	0XX	11X	R1
3	XXX	110	R2
4	010	XXX	R3
5	11X	101	R4
6	11X	11X	R4
7	11X	001	R5
8	11X	01X	R5
9	11X	100	R5

Table 2.4: TCAM records created by the range-to-prefix conversion of rules from the rule set in Table 2.2. The table also shows an original rule associated to each TCAM record. Note that records with the same associated rule do not overlap (i.e., they cannot match the same packet).

Parallel matching of the input against all rows allows constant search time of TCAM, but only at a price of increased utilization of hardware resources. Unlike the second naive approach, TCAM has to explicitly store each rule using the number of bits equal to the width of concatenated header fields. On the other hand, requirements on TCAM's capacity are linear with respect to the size of a represented rule set. However, it is important to note that because of the range-to-prefix conversion, the number or utilized records may be higher than the number of represented rules. For instance, 5 rules of the example rule set utilize 9 TCAM records, as shown in Table 2.4.

Constant search time and reasonable memory requirements are arguments behind the extensive use of TCAM in commercial devices. Nevertheless, this approach suffers from several non-negligible issues. Parallel matching, which allows constant search time, leads to high power consumption of this kind of memory. Because of supporting ternary matching, its cost per bit is also higher than of other memories. Moreover, if TCAM is used for packet classification, its capacity is not utilized efficiently due to rules replication during the range-to-prefix conversion. Last but not least, the need for storing records of width equal to the number of bits involved in packet classification limits scalability of this approach to classification according to more and/or longer header fields. All these disadvantages motivate research in algorithmic solutions to packet classification.

2.3.3 Representation Using Tuples

The representation of the packet classification problem using tuples has been introduced by Srinivasan, Suri, and Varghese in [56]. It is the first out of three seminal approaches to multi-dimensional packet classification described in the thesis. In this approach, each classification rule is represented by a tuple, whose elements define the number of bits used for specification of the corresponding rule's conditions. Such a representation is motivated by the observation that real rule sets contain only a few combinations of specification lengths. Therefore, the number of distinct tuples representing a rule set is expected to be much lower than the actual number of rules.

While the number of bits used in the value, wildcard, and prefix specifications is clear, the value of tuple elements corresponding to the range specification is not so straightforward to obtain. To overcome limitations of the range-to-prefix conversion, the authors of the approach proposed an alternative representation of utilized ranges that is based on a hierarchy of non-overlapping ranges. In the hierarchy, ranges are organized into several levels with the highest level containing the most general (i.e., the longest) ranges and each successive layer containing more specific (i.e., shorter) ranges than the previous level. Each range is then represented by a pair (*nesting level*, *range ID*), which characterizes its position within the hierarchy. The hierarchy of ranges used in the example rule set (Table 2.2) as well as *nesting level* and *range ID* of the ranges are shown in Figure 2.2.

Even though *nesting level* does not precisely characterize the number of bits used in the range specification, its meaning is similar. Clearly, *nesting level* of the full range (equal to the wildcard specification) will always be 0, while its value for the shortest ranges (equal to the value specification) will always be maximal. Therefore, the authors of [56] proposed to represent a range in a tuple by its *nesting level* and to replace the range itself by its *range ID*. Following this encoding scheme, Table 2.5 shows the example rule set (Table 2.2) together with tuples representing its rules. Note that rules R4 and R5 are represented by the same tuple, although they specify different ranges.

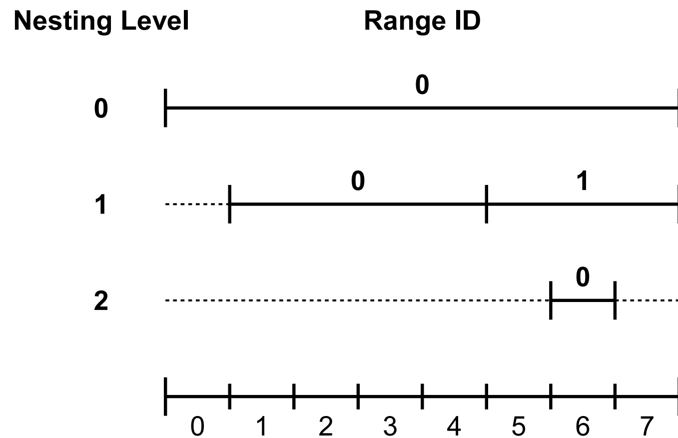


Figure 2.2: The hierarchy of port ranges from the example rule set (Table 2.2). The figure also shows *nesting level* and *range ID* values assigned to individual ranges.

Rule	**Address**	**Port**	**Tuple**
R1	0*	1	[1, 1]
R2	*	0	[0, 2]
R3	010	0	[3, 0]
R4	11*	1	[2, 1]
R5	11*	0	[2, 1]

Table 2.5: The example rule set (Table 2.2) and tuples representing its rules. Port ranges are replaced by their *range ID*.

Classification of an incoming packet using a rule set represented by tuples is done by a (possibly parallel) linear search of a tuple set. Each tuple represents a subset of the original rules that is searched for a matching rule using only a limited amount of information from packet's header fields. In case of conditions specified by a value, a wildcard, or a prefix, the number of utilized bits is directly given in the tuple. However, in case of the range specification, the search utilizes *range ID* of a range that matches the corresponding header field and appears in a hierarchy of ranges at *nesting level* specified in the tuple. The search for a matching rule within the tuple is thus the exact matching problem, which can be for example solved using a hashing table. Unfortunately, the representation of a rule set using tuples does not respect rules' priorities, thus the linear search cannot be terminated prematurely (like in the first naive approach described in Section 2.3.1) and all tuples have to be searched. If the searches of multiple tuples return a matching rule, the highest priority rule is selected as the best matching one.

Classification of the packet P from the running example, introduced at the beginning of Section 2.3, using the example rule set represented by tuples is illustrated in Table 2.6. The table shows the information from packet's header fields that is utilized when particular ranges are searched for a matching rule. The utilized information match rule's conditions in case of rules R2 and R4 (c.f. Table 2.5), which belong to tuples [0, 2] and [2, 1]. Since the rule R2 has higher priority than the rule R4, it is selected as the output of packet classification.

Tuple	Address Bits	Range ID	Rule
[1, 1]	1	1	N/A
[0, 2]	-	0	R2
[3, 0]	110	0	N/A
[2, 1]	11	1	R4

Table 2.6: Address bits and *range ID* used while searching particular tuples from Table 2.5 for rules matching the running example's packet P. The table also shows found matching rules (or N/A if a tuple does not contain a matching rule).

2.3.4 Geometric Representation

Another representation of the general packet classification problem is based on multi-dimensional space where each dimension corresponds to one header field utilized in packet classification. In this space, each condition of a classification rule can be represented as an interval in the corresponding dimension, thus the rule is equal to a multi-dimensional rectangle defined by intervals corresponding to its conditions. Since a packet contains exact values in its header fields, it is represented as a point in the multi-dimensional space. The geometric representation of the running example introduced at the beginning of Section 2.3 is shown in Figure 2.3. The figure shows 2-dmensional rectangles representing the example rule set (Table 2.2) as well as a point representing the example packet P.

In case of a geometrically represented rule set, a rule matching an input packet is represented by a rectangle that contains a point of the multi-dimensional space that corresponds

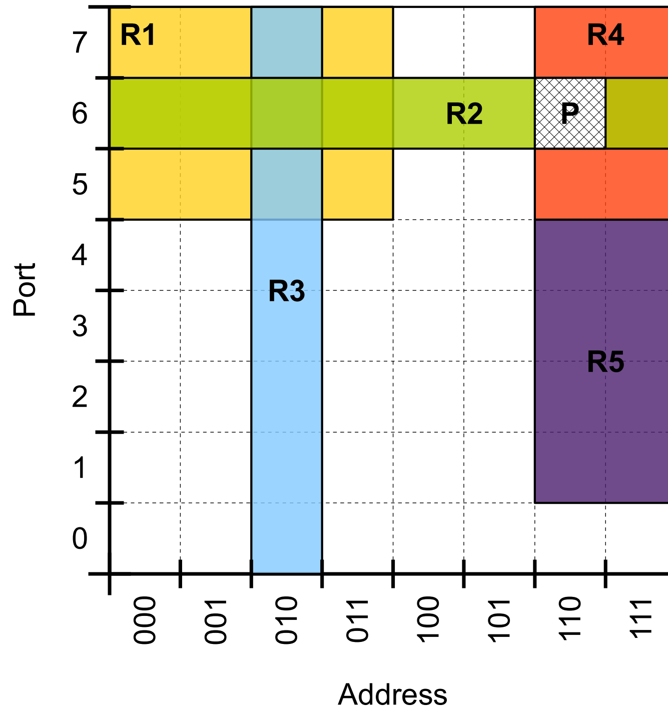


Figure 2.3: The geometric representation of the example rule set from Table 2.2 and the example packet P.

to the packet. As can be seen in Figure 2.3, rectangles representing rules can overlap, which illustrates a situation when more than one rule match an input packet. For instance, in Figure 2.3, both rules R2 and R4 match the packet P. Therefore, because of its higher priority, rule R2 is selected as the output of packet classification.

2.3.5 Combinatorial Representation

The last seminal approach to multi-dimensional packet classification described in this thesis is based on viewing a classification rule as a combination of the given number of conditions. In order to speed up a classification process, this approach builds a Cartesian product of sets of conditions utilized in particular dimensions. The entries of the Cartesian product correspond to all possible combinations of matching results for individual dimensions and each entry has the best matching rule associated to it. This is similar to the second naive approach to packet classification. However, the condition sets of particular dimensions of a real rule set are usually significantly smaller than the number of rules in the rule set [31]. Therefore, the number of Cartesian product entries is much smaller than the number of all possible packets.

Table 2.7 shows the Cartesian product entries along with assigned best matching rules that together represent the example rule set from Table 2.2. Five out of the sixteen entries (1, 6, 11, 13, and 16) directly correspond to the original rules. Another five entries (2, 9, 10, 12, and 14) have associated the best matching rule, i.e., the highest priority rule that matches all input packets, which are matched by the conditions specified for particular dimensions in the entry. However, because the example rule set does not contain a default rule (i.e., a rule with the wildcard specification in all dimensions, which matches every packet), six entries of the Cartesian product (3, 4, 5, 7, 8, and 15) have no associated best matching rule.

#	Address	Port	Rule	#	Address	Port	Rule
1	0*	5:7	R1	9	010	5:7	R1
2	0*	6:6	R1	10	010	6:6	R1
3	0*	0:7	N/A	11	010	0:7	R3
4	0*	1:4	N/A	12	010	1:4	R3
5	*	5:7	N/A	13	11*	5:7	R4
6	*	6:6	R2	14	11*	6:6	R2
7	*	0:7	N/A	15	11*	0:7	N/A
8	*	1:4	N/A	16	11*	1:4	R5

Table 2.7: The entries of a Cartesian product created from the sets of unique conditions utilized in *address* and *port* dimensions of the example rule set (Table 2.2). If it exists, the best matching rule from the example rule set is associated with a Cartesian product's entry. Otherwise, the best matching rule is set to N/A.

Packet classification based on combinatorial representation of the rule set fully utilizes properties of the constructed Cartesian product. First of all, matching conditions for individual dimensions are determined, possibly using different 1-dimensional packet classification approaches in each dimension. As mentioned in Section 2.3.6, some approaches to 1-dimensional packet classification may return more than one matching result. Therefore, combining the matching results from individual dimensions together may lead to multiple corresponding Cartesian product's entries, each of which may have associated the best

matching rule. The packet classification process is finalized by selecting the highest priority rule among these rules.

Considering the running example's packet P, prefix matching on the address field will return a single matching prefix 11* while range matching on the port field will return a set of matching ranges ([5:7], [6:6], and [0:7]). Combinations of these matching results are represented by entries 13, 14, and 15 of the Cartesian product, the first two of which has associated the best matching rules R4 and R2, respectively (see Table 2.7). Since rule R2 has higher priority, it is the output of classification of the packet P.

2.3.6 Range Matching

Apart from approaches to multi-dimensional packet classification, there are also specific approaches to 1-dimensional packet classification. They reflect different condition specifications utilized in classification rules. Since matching the value specification (exactly one value matches) and the wildcard specification (every value matches) is trivial, the next two sections will focus on matching range and prefix specifications, both of which define a set of matching values.

As already mentioned, the range specification is the most general among all the specifications, but also the most difficult for matching. Some approaches try to avoid range matching using the range-to-prefix conversion. However, the worst-case number of prefixes required for replacing a single range is $2l-2$ [46], where l is the length of the corresponding header field. Therefore, it is beneficial to have the option of direct range matching when classification rules contain the range specification.

The seminal range matching approach to 1-dimensional packet classification has been introduced by Lakshman and Stiliadis in [38]. It is based on non-overlapping intervals (often referred to as elementary intervals), which are created by dividing the full range by start and end points of each range utilized in a rule set. For example, Figure 2.4 shows elementary intervals created by port ranges utilized in the example rule set (Table 2.2). In this example, rules R1 and R4 utilize the same range and several start/end points have the same value, thus the full range is divided to only 5 elementary intervals. Nevertheless, each of N classification rules may utilize a unique range and a start point as well as an end point of each range (including the full range) may be a unique value. If this worst case happens, the number of elementary intervals is $2N + 1$.

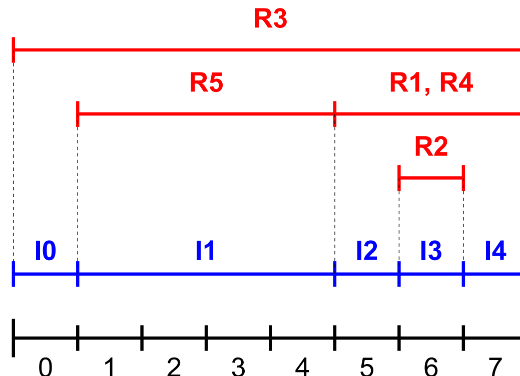


Figure 2.4: Elementary intervals (blue) created by port ranges (red) utilized in the example rule set (Table 2.2). The figure also shows rules that use particular ranges.

In order to allow packet classification, each elementary interval has to store a list of classification rules, whose range overlaps with the interval. In Figure 2.4, the list is illustrated by the name of rules utilizing port ranges that are above the elementary interval. Packet classification then consists of the search for an elementary interval covering the header field value extracted from a packet, followed by returning the list of overlapping (i.e., matching) rules. For example, the packet P from the running example introduced at the beginning of Section 2.3 has the port field set to 110. Because this value is covered by interval I3, the packet's port value is matched by rules R1, R2, R3, and R4 (see Figure 2.4). The covering elementary interval can be find by an arbitrary range location algorithm, for instance binary search.

2.3.7 Prefix Matching

Similarly to range matching, prefix matching is beneficial especially when classification rules contain the prefix specification. However, the prefix specification is also often used as a uniform way of specifying a condition in a classification rule. Even though the range specification is more general and more than one prefix may be necessary for replacing a single range, prefix matching is easier to implement.

The most utilized prefix matching approach to 1-dimensional packet classification is called LPM (*Longest Prefix Matching*). This approach looks for all prefixes matching the input value and returns the longest one (i.e., the most specific one) as the best matching prefix. For example, consider the running example's packet P with $address = 110$, which was introduced at the beginning of Section 2.3. Its address is matched by address prefixes utilized in rules R2 and R4/R5 (note that rules R4 and R5 use the same prefix) of the example rule set (Table 2.2). However, since the address prefix of rules R4/R5 is the longest one among them, it is the only result of LPM.

Some packet classification algorithms based on combinatorial representation might require prefix matching that returns all matching prefixes instead of just the longest one. In such a case, LPM can be easily modified to satisfy this requirement by omitting its last step.

2.4 Research Issues

Current research on packet classification techniques for core networks faces two issues. The first one can be characterized as *performance of packet classification algorithms*. Clearly, the performance of classification algorithms has to be increased to meet ever-increasing requirements on packet classification in core networks. The second issue rises from the first one and it revolves around *packet classification algorithms benchmarking*. As new classification algorithms are designed, it is necessary to benchmark their performance and compare their properties to each other.

While addressing performance of packet classification algorithms and their benchmarking represent the core of this thesis (see Chapters 4 and 5, respectively), the following sections introduce the addressed issues.

2.4.1 Performance of Algorithms

Even though the Internet is still growing and accelerating, packet classification stays one of the most common operations implemented in networking devices. Nevertheless, the changes

of the Internet increase requirements on the performance of packet classification algorithms, which has to keep pace with growing transfer rates. The standard for 400 Gigabit Ethernet [13] has been approved in December 2017, but current core networks widely support 100 Gigabit Ethernet defined in [9]. Therefore, packet classification algorithms targeted at core networks are required to support throughput of 100 Gbps.

Considering a 64-byte Ethernet frame (i.e., the shortest frame) together with 7-byte Preamble, 1-byte SFD (*Start Frame Delimiter*), and 12-byte interpacket gap (i.e., the shortest gap), the maximum packet rate of 100 Gigabit Ethernet is approximately 148.81 MPPS (*Million Packets Per Second*). Thus, packet classification algorithms have to be able to provide a classification result every 6.72 ns. To achieve such matching performance, their designers have to deal with growing requirements on selected parameters, which include the number of bits involved in packet classification (i.e., the number of utilized header fields and their length) and the number of specified classification rules.

The number of bits involved in packet classification depends on a specific use case. However, in general it is increasing because nowadays more as well as longer header fields are being utilized in packet classification. Extensions of the set of utilized header fields are closely related to growing interest towards network virtualization, which may be realized, for instance, by OpenFlow-based SDN [35]. Each version of the OpenFlow protocol incrementally extended the set and its latest version has defined packet classification according to 45 header fields [10]. On the other hand, longer header fields are mainly due to increasing IPv6 penetration [30]. Compared to IPv4, the length of both source and destination addresses has been quadrupled in IPv6.

While the exact number of utilized header fields may vary according to the used version of the OpenFlow protocol (e.g., OpenFlow 1.0.0 utilizes "only" 12 header fields), IPv6 always requires supporting 128-bit address prefixes. Therefore, with respect to the number of bits involved in packet classification, supporting IPv6 may be more expensive than supporting additional header fields. This can be illustrated using types of classification rules from Table 2.1. Comparing packet classification based on an IPv6 prefix and an IPv4 5-tuple, the former involves 128 bits while the latter involves 104 bits. Moreover, the situation is the same when comparing an IPv6 5-tuple and OpenFlow 1.0.0 (296 bits vs. $x + 237$ bits).

The number of specified classification rules varies even for different instances of the same use case. In case of IP routing, forwarding tables of core routers currently contain 680 thousand IPv4 and 43 thousand IPv6 prefixes [3] but these numbers continuously grow as the allocation of prefixes from IPv4 and IPv6 address spaces progresses. The situation is different with firewalls, in which the actual number of installed classification rules depends on security policies of a particular network. Since firewall rule sets are not usually publicly available (because of security reasons), the researchers evaluating 5-tuple-based classification algorithms often use synthetic data sets consisting of thousands or tens of thousands of classification rules [64, 55, 40]. Focusing on SDN-enabled switches in a datacenter, each OpenFlow rule may correspond to an active virtual machine, thus their number may be in the order of tens of thousands [41].

In summary, packet classification algorithms targeting core networks have to be able to classify an incoming packet according to tens or hundreds of bits into tens or hundreds of thousands of classes and provide a new classification result every 6.72 ns. Such requirements on performance prohibit software implementation of packet classification algorithms [19]. Although hardware implementation can be realized using both an ASIC (*Application-Specific Integrated Circuit*) or an FPGA (*Field-Programmable Gate Array*), because of its availability, flexibility, and configurability, FPGA-based implementation will

be considered further in this thesis. The viability of this approach in an often changing networking environment has also been proved by successful FPGA-based networking platforms: NetCOPE [28] and NetFPGA [68].

2.4.2 Algorithms Benchmarking

The requirements on the performance of packet classification algorithms are continuously increasing, as demonstrated in the previous section. Therefore, the development of classification algorithms that meet these requirements is a never-ending story. To benchmark a new classification algorithm, benchmarking tools (e.g., Netbench [48]) usually assess the following parameters of the algorithm:

- **classification speed** defined as the number of memory accesses required for classification of a single packet;
- **memory requirements** of a data structure representing a set of classification rules.

In addition, the speed of updates of a classification rule set representation might also be assessed in the course of algorithm's benchmarking.

The practical implementation of a packet classification algorithm has to consider the worst case performance of the algorithm in terms of the above mentioned parameters, but the worst case performance solely depends on a utilized classification rule set. This means that in the ideal case a real rule set is utilized while benchmarking the classification algorithm. Nevertheless, real rule sets are not publicly available for the majority of packet classification use cases (often because of security reasons). One of the few exceptions to this are IPv4/IPv6 prefix sets from forwarding tables of core routers [18, 50], which can be used for benchmarking IP lookup (i.e., the key part of IP routing). However, since IPv6 penetration is expected to grow exponentially in the future and the size of IPv6 forwarding tables depends on the number of allocated IPv6 prefixes, current IPv6 prefix sets cannot be directly used for benchmarking IP lookup algorithms in the future environment.

The researchers addressed the issue of missing real benchmarking data in various ways. While a limited number of research groups obtained access to real rule sets via NDAs (*Non-Disclosure Agreements*), others developed several tools capable of generating synthetic data sets for common packet classification use cases. Because the size of IPv4 forwarding tables is not expected to grow significantly in the future, the tools for generating IP prefix sets focus on IPv6 prefix sets only. To get this kind of benchmarking data, it is possible to use for instance Non-random Generator [66] or V6Gene [67]. On the other hand, in the area of 5-tuple rules it makes sense to generate both IPv4 and IPv6 5-tuples. The former is almost exclusively generated using ClassBench [63], while the latter can be generated using ClassBenchv6 [59]. The most problematic is the situation with OpenFlow rules generators. Although FRuG (*Flexible Rule Generator*) [29] is able to generate a set of rules that specify a condition for an arbitrary number of header fields, it does not explicitly consider any specifics of OpenFlow rule sets.

Even though the existing generators are capable of producing IPv4/IPv6 prefixes and 5-tuples as well as rules specifying a condition for more than 5 header fields, none of them explicitly provides support for OpenFlow rules generation. In addition, none of the tools is able to generate all types of benchmarking data, which complicates the situation in case of benchmarking packet classification algorithms targeted at various use cases. Moreover,

it can be shown that the process of rule set generation in available generators is not always accurate. These issues show that packet classification benchmarking is still an open problem.

Chapter 3

Related Work

While the previous chapter introduced the packet classification operation along with various approaches to its implementation, this chapter contains the selection of already implemented packet classification algorithms and synthetic rule set generators. These algorithms and generators represent current solutions to the performance of packet classification algorithms issue and the packet classification algorithms benchmarking issue, respectively, which were also discussed in the previous chapter. Since the goal of the thesis is to address these issues as well, the algorithms and generators presented in this chapter serve as a starting point for work described in this thesis and they will be used as a baseline for the evaluation of proposed solutions to the identified issues in research on packet classification in core networks.

The first part of this chapter focuses on existing packet classification algorithms, separately for the 1-dimensional and multi-dimensional case. Section 3.1 describes 1-dimensional algorithms, all of which implements the prefix matching approach to packet classification. The reasons for including prefix matching algorithms only are that the prefix specification is utilized in IP lookup (i.e., probably the most common type of packet classification in core networks) and it is also commonly used as a uniform way of specifying all conditions of a classification rule. On the other hand, multi-dimensional algorithms presented in Section 3.2 implement various approaches to packet classification. Actually, a single multi-dimensional algorithm often combines multiple approaches together.

Similarly to packet classification algorithms, synthetic rule set generators, which were already briefly introduced in Section 2.4.2, are also described separately for the 1-dimensional and multi-dimensional case. All 1-dimensional generators presented in Section 3.3 implement the generation of IP prefixes (i.e., classification rules for benchmarking IP lookup). More precisely, they are specialized on generating IPv6 prefixes because currently available real IPv6 prefix sets are not suitable for benchmarking as they are expected to significantly grow in size in the future. The situation is different in case of multi-dimensional generators, which are described in Section 3.4. They primarily support the generation of IPv4 5-tuples, but some of them also allow to generate IPv6 5-tuples or classification rules specifying a condition for more than 5 header fields.

3.1 1-Dimensional Packet Classification Algorithms

Probably the most common type of 1-dimensional packet classification in core networks is prefix matching on destination IP address, which is known as IP lookup. Because of its

constant matching performance and a direct support of the prefix specification, many commercial devices implement IP lookup using TCAM. However, such implementation suffers from high power consumption, high cost per bit, storage inefficiency, and limited scalability, which makes also algorithmic solutions to prefix matching a viable option for IP lookup implementation [51]. Although prefix matching algorithms employ various approaches (e.g., hashing), the majority of them encode a prefix set using a binary prefix tree that is usually called trie. Therefore, this section presents selected trie-based prefix matching algorithms.

Individual prefix matching algorithms are illustrated on the set of *address* prefixes from the example rule set introduced in Table 2.2. Table 3.1 shows this prefix set sorted from the shortest to the longest prefix.

Prefix	**Value**
P1	*
P2	0*
P3	11*
P4	010

Table 3.1: The set of *address* prefixes from the example rule set (Table 2.2) sorted from the shortest to the longest prefix.

3.1.1 Trie

The trie data structure, which has been proposed by Fredkin [27], represents the basis of prefix set encoding in the majority of prefix matching algorithms. It is a binary prefix tree in which each node represents a prefix. The root node represents the empty prefix and left and right child nodes of any trie node represent prefixes created from their parent's prefix by appending 0 and 1, respectively. Trie nodes representing prefixes from a prefix set are called prefix nodes, while other trie nodes are referred to as place holder nodes. The representation of the prefix set from Table 3.1 using a trie is shown in Figure 3.1.

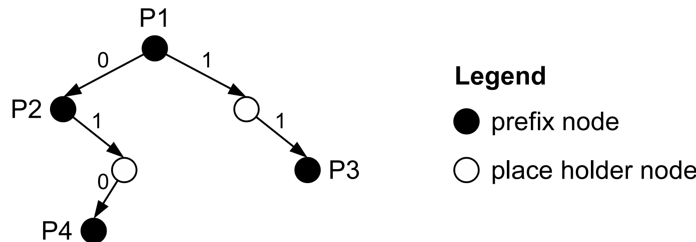


Figure 3.1: The representation of the prefix set from Table 3.1 using a trie.

Matching prefixes of a prefix set represented by a trie is done by traversing the trie from the root to the leaves according to the bits of an input value (e.g., a destination address in case of IP lookup) taken from the MSB (*Most Significant Bit*) to LSB (*Least Significant Bit*). All prefix nodes visited during such a traversal represent prefixes matching the input value and the last visited prefix node represents the longest matching prefix, which is the output of the LPM operation.

Since a trie is a binary prefix tree, the implementation of prefix matching based on a trie is straightforward because operations on a trie (adding, removing, or matching prefixes)

can be realized using standard operations on a binary tree. However, a trie allows to match only one bit of the input value in each step, which means linear time complexity with respect to the bit width of an input value. Therefore, in the worst case it is necessary to perform 32 and 128 steps for matching a full IPv4 and IPv6 prefix, respectively. Due to the high number of pointers in a trie, this data structure also suffers from high memory requirements.

In order to diminish the main disadvantage of a trie (i.e., matching only one input bit in each step), modern trie-based prefix matching algorithms employ the concept of so called multibit trie. These algorithms propose new types of node that represent subtrees of a trie and allow to match multiple bits per step. Thus, although a trie is not directly applicable for high-performance prefix matching, it is a seminal data structure for modern prefix matching algorithms.

3.1.2 Tree Bitmap

Tree Bitmap (TBM), which has been developed by Eatherton, Varghese, and Dittia [26], is one of the best known multibit trie algorithms. It represents a set of prefixes using a 2^{SL}-tree, where parameter SL (stride length) determines the number of input bits matched in each step. This tree is built on the top of a trie by mapping TBM nodes, each of which corresponds to trie's subtree of the maximum depth equal to SL, onto the trie in a non-overlapping fashion such that each trie node is covered by a TBM node. An example of the mapping of TBM nodes with $SL = 3$ onto the trie from Figure 3.1 is shown in Figure 3.2.

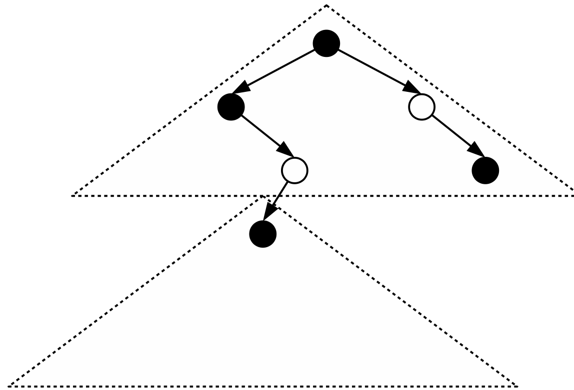


Figure 3.2: The mapping of TBM nodes with $SL = 3$ onto the trie representing the prefix set from Table 3.1.

The details of TBM node's encoding are illustrated in Figure 3.3 that shows the root TBM node from Figure 3.2 and its encoding using two bitmaps and two pointers. The external bitmap contains 2^{SL} bits determining whether a corresponding child node is present (value 1) or missing (value 0). Note that MSB of the external bitmap corresponds to the leftmost child, while its LSB corresponds to the rightmost child. On the other hand, $2^{SL}-1$ bits of the internal bitmap correspond to the nodes of the underlying trie in breadth-first order and each bit encodes information whether a corresponding internal node is a prefix node (value 1) or non-prefix node (value 0).

The child and prefix pointers refer to information about child nodes and prefix-related data, respectively. In order to minimize the size of the TBM node's data structure and speed up its retrieving from a memory, these information are stored outside the node itself.

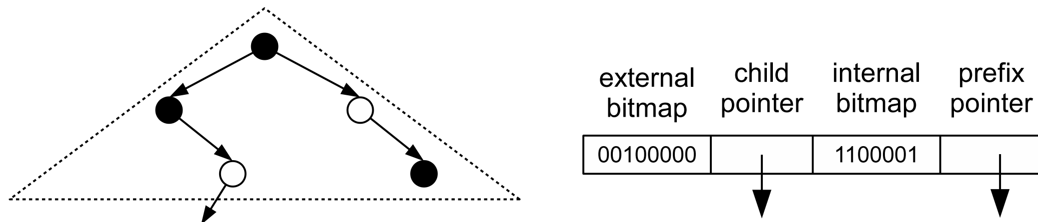


Figure 3.3: The encoding of the root TBM node from Figure 3.2.

Nevertheless, both types of externally stored information are in a continuous block of the memory and they are stored in the same order in which they appear in the corresponding bitmaps. Such organization allows to use pointer arithmetic to directly retrieve information about any child node or prefix-related data using just the corresponding pointer and bitmap (the child pointer and external bitmap in case of information about child nodes, the prefix pointer and internal bitmap in case of prefix-related data).

The compact representation of a TBM node makes possible to retrieve it from a memory in just one clock cycle. Additionally, due to the use of bitmaps for node's encoding, this prefix matching algorithm is easily implementable in hardware. The TBM algorithm is thus able to achieve high matching performance. In addition, the fixed structure of a node allows easy updates of the represented prefix set. However, it may also cause high memory overhead when the underlying trie is sparse. Trade-off between matching performance and memory requirements of the TBM algorithm for a given prefix set can be tuned via parameter SL. Clearly, a higher value of parameter SL results in a lower number of matching steps but higher memory overhead and vice versa.

3.1.3 Shape Shifting Trie

Another important multibit trie algorithm called *Shape Shifting Trie* (SST) has been introduced by Song, Turner, and Lockwood [54]. This algorithm is based on TBM, but it tries to overcome its main drawback by introducing an adaptive shape of a node, which reduces memory overhead when the underlying trie is sparse. Instead of parameter SL, the shape of SST nodes is influenced by parameter K, which determines the maximum number of underlying trie nodes that can be represented by an SST node. Therefore, this parameter no longer determines the number of input bits matched in each step of the SST algorithm. Although the number of matched bits can theoretically be equal to K (when an SST node corresponds to a non-branching subtree of a trie consisting of K nodes), it is typically lower and varying throughout an SST tree in practice.

The encoding of SST nodes, the example of which is shown in Figure 3.4, is based on TBM node's encoding. The function of $K + 1$ bits of the external bitmap, K bits of the internal bitmap, and both the child and prefix pointers is exactly the same as of the corresponding elements in the TBM node's data structure. Apart from these bitmaps and pointers, the data structure of an SST node also contains the shape bitmap consisting of $2K$ bits, which encodes the shape of the node. Similarly to the internal bitmap, pairs of shape bitmap's bits correspond to the nodes of the underlying trie in breadth-first order. The first bit of each pair determines whether the SST node contains the left child of the corresponding node (value 1) or its left child is missing (value 0), while the second bit encodes the same information for the right child of the corresponding node.

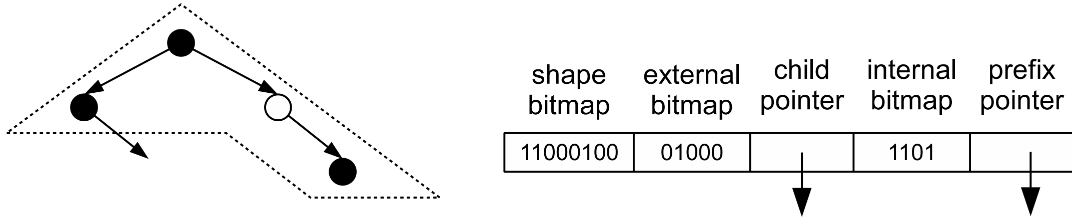


Figure 3.4: The encoding of an example SST node for $K = 4$.

The adaptive shape of SST nodes allows to represent a prefix set using a very small amount of memory. However, the variability of nodes makes the construction of the prefix set representation a computationally complex task and it also significantly limits the possibility of prefix set updates. Moreover, the need to decode the shape of a node from its shape bitmap negatively influences matching performance. Because of all these limitations, SST is not a viable option for prefix matching in core networks, which is also illustrated by the fact that there is no hardware architecture implementing this algorithm. Nevertheless, it may be useful as a reference algorithm for an assessment of memory requirements.

3.1.4 Multiple SRAM-based Lookup Algorithm

The first of two current prefix matching algorithms described in this thesis is *Multiple SRAM-based Lookup Algorithm* (MSLA) proposed by Jiang and Prasanna [36]. This algorithm primarily aims to achieve high matching performance. To this end, a hardware architecture implementing the algorithm employs several parallel processing pipelines that utilize independent SRAM (*Static Random Access Memory*) blocks connected to their stages. Apart from focusing on matching performance, the MSLA algorithm also tries to keep the size of the utilized SRAM blocks reasonable. Therefore, it represents a prefix set using a leaf-pushed trie introduced in [57], in which all prefixes are stored in the leaf nodes only. This means that for each input the leaf-pushed trie contains only one (i.e., the longest) matching prefix.

As shown in Figure 3.5, the architecture implementing MSLA employs P processing pipelines, each of which is responsible for prefix matching in a subset of at most 2^I subtries starting at level I. To guarantee that these subtries contain all prefix nodes of the original leaf-pushed trie (i.e., that the length of each prefix is at least I), controlled prefix expansion—another prefix set transformation from [57]—is applied on the trie before its mapping onto pipelines. This transformation makes possible to directly access the root node of each subtrie using the first I bits of an input value without skipping over any prefix node.

Because in general the number of processing pipelines P may be lower than the number of subtries starting at level I and these subtries may contain various number of nodes, the mapping of the subtries onto the pipelines is determined using a polynomial-time approximation algorithm, which tries to balance memory requirements over different pipelines. Similarly, the nodes of all subtries mapped to a particular pipeline are equally distributed among the stages of the pipeline. Apart from this two-stage memory balancing, MSLA also implements techniques called flow pre-caching and payload exchange that address traffic balancing and intra-flow packet reordering issues, respectively.

The proposed hardware architecture with 8 parallel pipelines consisting of 25 stages can perform over 10 GLPS, which translates into throughput of 3.2 Tbps for 40-bytes packets.

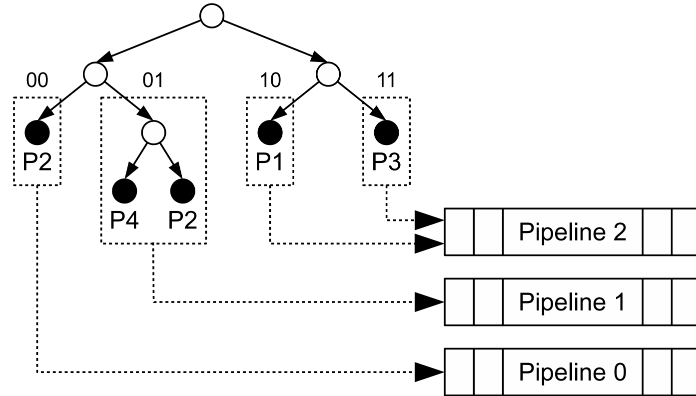


Figure 3.5: An example mapping of the subtries starting at level $I = 2$ of the leaf-pushed trie from Figure 3.1 onto $P = 3$ processing pipelines.

Such matching performance should be sufficient also in the near future. At the same time, memory requirements of the architecture are reasonable. For storing the representation of a prefix set comprising 248 856 unique prefixes, the architecture requires slightly over 28 Mb of memory organized in $8 \cdot 25 = 200$ independent SRAM blocks with a capacity of 144 kb. However, both matching performance and memory requirements depend on a time-consuming mapping of the prefix set representation onto available memory resources in a memory-balanced fashion. Thus, it can be expected that these parameters will degrade in an environment with frequent incremental updates of the prefix set.

3.1.5 Prefix Partitioning Lookup Algorithm

Prefix Partitioning Lookup Algorithm (PPLA) developed by Le and Prasanna [39] is the second current prefix matching algorithm described in this thesis. Differently from MSLA, this algorithm focuses on minimizing the memory footprint of a prefix set representation, while keeping sufficient throughput for 100 Gbps networks. As all other prefix matching algorithms described so far in this section, PPLA represents a prefix set by a trie. However, it utilizes the trie representation just for partitioning the set of prefixes into k disjoint subsets that are used for prefix matching in k separate processing pipelines.

Each of k prefix subsets consists of prefixes expanded using controlled prefix expansion [57] to the closest one of k trie levels. In order to achieve balanced distribution of prefixes among processing pipelines, the PPLA algorithm determines these levels using dynamic programming. Once the prefix set is partitioned, each subset is represented by a separate binary search tree (for minimum memory requirements) or 2-3 tree (for easy prefix set updates) and mapped onto stages of a corresponding pipeline. An illustration of partitioning the example prefix set from Table 3.1 to two subsets, each of which is represented by a binary search tree mapped to a separate processing pipeline, is shown in Figure 3.6.

PPLA searches for prefixes matching a given input value in all pipelines in parallel. Since a prefix subset in each pipeline consists of prefixes expanded to the same level of the original trie, at most one of the prefixes can match an arbitrary input value. Nevertheless, multiple (possibly all) pipelines can return a valid matching prefix for the given input. In such a case, the prefix returned by the pipeline with the highest corresponding level is selected as the longest matching prefix.

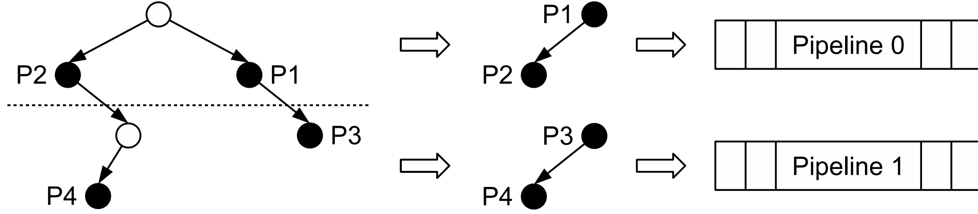


Figure 3.6: The partitioning of the trie from Figure 3.1 to $k = 2$ levels and the representation of prefixes expanded to these levels using binary search trees, each of which is mapped to a separate processing pipeline.

Although an FPGA implementation of PPLA is able to perform 410/390 MLPS (*Million Lookups Per Second*) for an IPv4/IPv6 prefix set consisting of over 330 000 unique prefixes, the main advantage of the PPLA algorithm is its high memory efficiency. When prefix subsets are represented by a binary search tree, an average memory efficiency ratio (i.e., the number of bytes of memory needed for storing one byte of a prefix) achieved by the algorithm is 1.00 for IPv4 prefix sets and 0.91 for IPv6 prefix sets. The main drawback of this algorithm is connected with the initial partitioning of a prefix set, which introduces very high preprocessing overhead. Therefore, similarly to the MSLA algorithm, PPLA may experience the degradation of matching performance and memory efficiency when the prefix set is frequently updated.

3.2 Multi-Dimensional Packet Classification Algorithms

Approaches to packet classification, which were described in Section 2.3, represent the main ideas utilized in multi-dimensional packet classification algorithms. However, the majority of these ideas are not implemented in their pure form. They are rather implemented in various ways, which may result into different time and space complexity of packet classification algorithms, even when the algorithms utilize the same approach. It is also common that a single algorithm utilizes more than one approach to packet classification. Therefore, this section describes seminal multi-dimensional packet classification algorithms.

Inspired by a survey conducted by Taylor [60], the description is divided into four subsections, each of which includes algorithms based on the same high-level concept. In addition, the representation of a rule set in the packet classification algorithms described in this section is illustrated on the example rule set from Table 2.2.

3.2.1 Exhaustive Search

The first and simplest high-level concept is exhaustive search through all classification rules. Its serial implementation is very efficient with respect to utilized computational resources but its low performance is prohibitive. Although the performance can be improved by parallel implementation, the amount of required computational resources grows linearly with the degree of parallelism, which is unfavorable.

Contrary to almost all other algorithms described in further sections, both representatives of the exhaustive search group directly implement selected approaches from Section 2.3. A serial version of exhaustive search—linear search—corresponds to the first naive approach to packet classification. Linear search time of this approach limits usability of the linear search algorithm to the final step of some advanced packet classification al-

gorithms only. On the other hand, massively parallel implementation of exhaustive search in TCAM results in constant search time that makes this solution extensively utilized in commercial devices. However, because of its non-negligible limitations (high power consumption, high cost per bit, storage inefficiency, and limited scalability), TCAM is not the ultimate solution to packet classification.

3.2.2 Decision Tree

A common feature of algorithms described in this section is that they represent the packet classification problem using a decision tree, whose leafs contain either a single classification rule or a small rule set. Classifying a packet is then realized by traversing the tree from the root to the leaves according to bits read from packet's header fields. After reaching a leaf node, a matching rule is either directly read from the leaf (when the leaf contains a single rule only) or it is found using linear search (when the leaf contains a set of rules).

Packet classification algorithms utilizing a decision tree can be divided into two branches. Algorithms from the first branch construct a decision tree according to cuts of multi-dimensional space that geometrically represents a rule set. Therefore, these algorithms are often referred to as *cutting algorithms*. On the other hand, algorithms from the second branch utilize a binary decision tree encoding a prefix set, which is a basic data structure for prefix matching. Because this data structure is often called trie, algorithms from this branch will be further referred to as *trie-based algorithms*.

Cutting Algorithms

The seminal cutting algorithm called HiCuts (*Hierarchical intelligent Cuttings*) has been introduced by Gupta and McKeown [32]. This algorithm recursively cuts the multi-dimensional space until the number of rules in each sub-space decreases below a given threshold (referred to as *binth*). Each cutting step can be done along an arbitrary dimension and it divides the partitioned (sub-)space into equal-sized intervals. In [32], the authors also propose heuristics exploiting the inherent structure of the rule set that guide the construction of the decision tree (i.e., which dimension to select, how many cuts to do, etc.).

To illustrate basic principles of the HiCuts algorithm, Figure 3.7 shows cuts made by the algorithm in geometric representation of the example rule set (Table 2.2) and a decision tree constructed according to these cuts when the threshold for the number of rules per sub-space (*binth*) is set to 2. However, this example also reveals three limitations of the HiCuts algorithm. The first one is ability to make a cut along a single dimension only, which is illustrated by separately made cuts 1 and 2 in Figure 3.7. If these cuts were made in a single step, the decision tree would be one level lower. The next limitation is successive cutting the same dimension because of the requirement for equal-sized cuts. After relaxing this requirement, it would be possible to omit cut 2 together with the whole corresponding level of the decision tree. The last limitation is caused by the value of *binth*, which is lower than the maximum number of overlapping rules. While $binth = 2$, the rule set contains three rules (R1, R2, and R3) that overlap at point defined by $address = 010$ and $port = 6$. This results in the decision tree of maximum depth and violation of *binth* in its node representing the previously mentioned point.

Further cutting algorithms often address the presented limitations of the HiCuts algorithm. Singh et al. have introduced the HyperCuts algorithm [52], which allows cutting rule (sub-)space along multiple dimensions at the same time, while Vamanan, Voskuilen, and Vijaykumar have allowed cuts of non-equal size in their EffiCuts algorithm [64]. However,

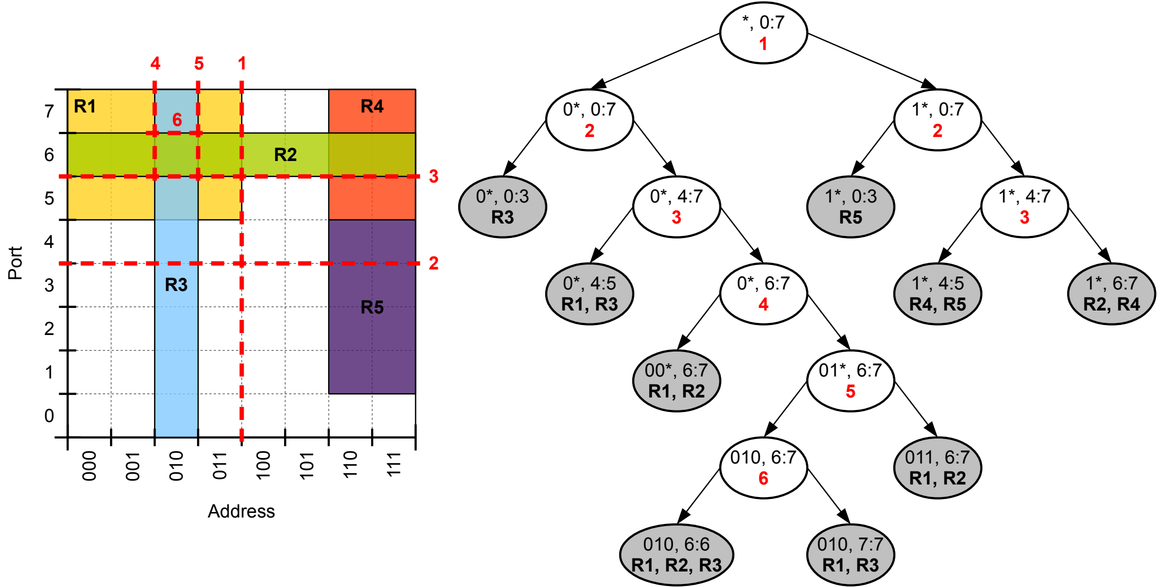


Figure 3.7: Cuts of 2-dimensional space that geometrically represents the example rule set from Table 2.2 (left) and a decision tree constructed according to these cuts (right). Nodes of the tree contain an address prefix and port range of the represented (sub-)space and either the number of a cut defining its children (white internal nodes) or the represented subset of the example rule set (gray leaf nodes). The threshold for the number of rules per sub-space (*binth*) is set to 2.

there are also cutting algorithms that improve upon HiCuts in a different way. For instance, the HyperSplit algorithm [49] introduced by Qi et al. makes no more than one cut in a single cutting step but it allows cutting the (sub-)space into non-equal-sized parts. Because of these constraints, HyperSplit represents a rule set by a binary decision tree employing range matching.

Search time of the cutting algorithms depend on the depth of the decision tree and the number of rules represented by a leaf node, which is bounded by parameter *binth*. The simplest way of lowering the tree's depth is increasing the number of children a node can have. This also leads to decreasing the number of nodes in the tree, but the more children a node can have, the more data for a decision making process it has to store. Therefore, lowering the tree's depth may or may not have a positive impact also on memory requirements of the cutting algorithms. Moreover, the actual structure of the decision tree is heavily dependent on the represented rule set, especially on rules' overlapping, as illustrated in Figure 3.7.

To get the worst-case bounds on search time and memory requirements, it is possible to reduce the packet classification problem in the cutting algorithms to the point location problem in computational geometry. Then, considering the case of n non-overlapping d-dimensional classification rules, it can be shown that the packet classification problem may require $O(\log n)$ time and $O(n^d)$ memory (when optimized for time) or $O(\log^{d-1} n)$ time and $O(n)$ memory (when optimized for memory) [45]. Note that an arbitrary rule set can be converted to a rule set with non-overlapping rules by representing each condition as a range and constructing the elementary intervals in each dimension.

Trie-Based Algorithms

Packet classification algorithms from this branch combine two separate tries (i.e., binary decision trees encoding prefix sets, each of which can be used for 1-dimensional prefix matching) into a single data structure called grid-of-tries [58]. This approach was proposed by Srinivasan et al., who designed grid-of-tries for handling 2-dimensional rule sets that use the prefix specification in both dimensions (e.g., destination and source IP addresses). Similarly to set-pruning tries [24] introduced by Decasper et al., grid-of-tries encodes the full prefix set of the first dimension using a single trie in which each node representing a prefix has associated another trie that encodes a pruned prefix set of the second dimension. However, differently from set-pruning tries, pruned tries used in grid-of-tries encode only second-dimension prefixes that appear in rules specifying the prefix the pruned trie is associated to in the first dimension. Thus, grid-of-tries does not replicate rules and it requires only $O(nw)$ memory, where n is the number of classification rules and w is the maximum prefix length allowed in the first or second dimension. To keep also search time reasonable ($O(w)$), grid-of-tries uses switch pointers between pruned tries and performs precomputation of the highest priority matching rule for each prefix node of pruned tries. Switch pointers and precomputation together guarantee that the rule matching process always progresses forward and it never has to apply backtracking.

Since grid-of-tries allows prefix matching only, in order to show the representation of the example rule set (Table 2.2) using this data structure, it is necessary to perform the range-to-prefix conversion first. Table 3.2 shows the example rule set after the range-to-prefix conversion that increased the number of rules from 5 to 9. The representation of this converted rule set using grid-of-tries is shown in Figure 3.8. Grid-of-tries in the figure comprises a single address trie and multiple pruned port tries, each of which is associated to an address prefix node via a dotted arrow. Each port trie represents classification rules that has a common address prefix only. Therefore, to ensure that all possible rules can be matched, grid-of-tries also contains switch pointers (dashed arrows between nodes of the port tries), which allow to continue with matching longer port prefixes in a trie associated to a shorter address prefix. However, switch pointers are allowed only if following them may lead to a higher priority rule. Because of this constraint, grid-of-tries in Figure 3.8 does not contain a switch pointer between nodes representing rules R1b and R2.

Although the authors of the grid-of-tries algorithm propose an extension that supports standard 5-tuple rules (by constructing a separate grid-of-tries for each tuple defined by

Rule	Address	Port
R1a	0*	101
R1b	0*	11*
R2	*	110
R3	010	*
R4a	11*	101
R4b	11*	11*
R5a	11*	001
R5b	11*	01*
R5c	11*	100

Table 3.2: The example rule set (Table 2.2) after the range-to-prefix conversion.

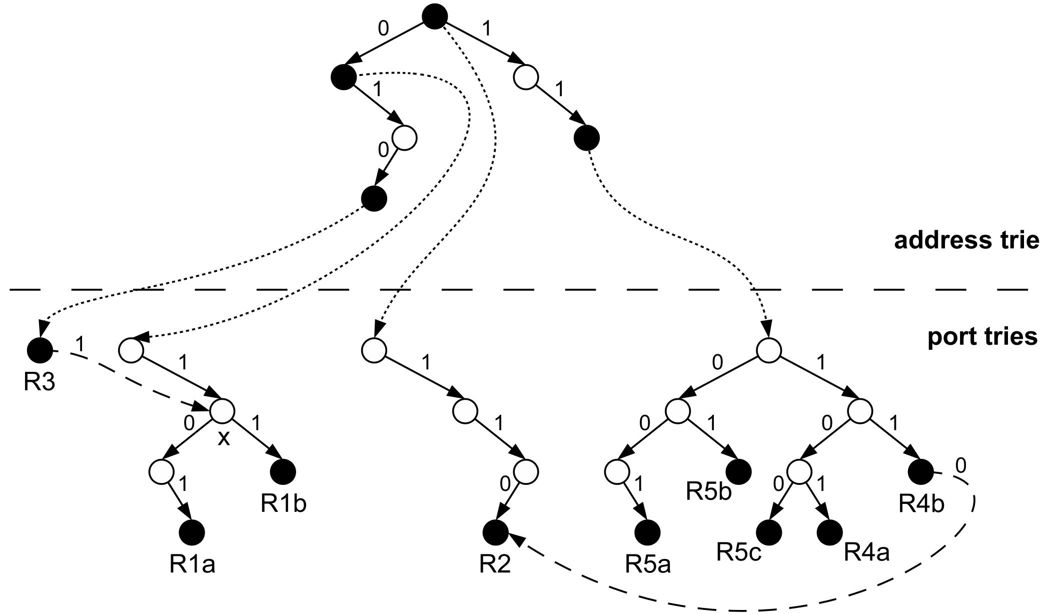


Figure 3.8: Grid-of-tries representing the example rule set after range-to-prefix conversion (see Table 3.2).

source port, destination port, and protocol), more memory efficient extension called EGT (*Extended Grid-of-Tries*) has been proposed by Baboescu, Singh, and Varghese [20]. EGT is based on original grid-of-tries but it implements two extensions. First, instead of directly representing a classification rule, prefix nodes of the second-dimension tries store a pointer to a list of rules sharing the already matched prefixes. To select the best matching rule from such a list, EGT employs linear search. The second extension replaces switch pointers by jump pointers that direct the search process to all possible matching filters, thus ensuring correctness of the classification result.

3.2.3 Decomposition

Packet classification algorithms utilizing the principle of decomposition first perform individual classification in each dimension and subsequently combine the results from particular dimensions together. This allows the algorithms to employ various 1-dimensional classification approaches (e.g., range matching and prefix matching) and perform classification in particular dimensions in parallel. However, combining the results of these independent classification steps is not usually straightforward. The best matching rule does not always consist of the individual best matches, thus the employed 1-dimensional classification approaches are often required to return all entries matching a given packet. A high number of intermediate results increases the number of their possible combinations, which makes determining the final best matching filter the major challenge that has to be addressed in the decomposition-based packet classification algorithms.

There can be identified three branches of packet classification algorithms based on decomposition. Algorithms in the first branch, which will be further referred to as *combinatorial algorithms*, represent variations on the combinatorial representation approach to packet classification. The second branch includes the RFC (*Recursive Flow Classification*) algorithm only, which can be characterized as a recursive implementation of the second

naive approach leveraging the properties of real classification rule sets. A common feature of algorithms from the third branch is the representation of range matching results in individual dimensions using a bit-vector. Therefore, these algorithms will be further referred to as *bit-vector-based algorithms*.

Combinatorial Algorithms

This branch of decomposition-based classification algorithms has been initiated by the cross-producting algorithm introduced by Srinivasan et al. [58], which was designed as a multi-dimensional alternative to the grid-of-tries technique introduced in the same article. The cross-producting algorithm employs the combinatorial representation of packet classification. However, differently from the pure combinatorial representation approach, it only employs prefix matching for packet classification in individual dimensions. Therefore, before constructing the table of Cartesian product entries, all classification rules' conditions specified by a range has to be converted to the prefix specification using the range-to-prefix conversion (note that an exact value and the wildcard are the special cases of a prefix). Although this conversion may lead to increasing the number of unique conditions, it also simplifies the final steps of the algorithm. Since prefix matching usually returns only the longest matching prefix, combining the classification results from individual dimensions can be simply realized by their concatenation. Moreover, concatenation produces only one result, which can be directly used for retrieving the best matching rule from the table of Cartesian product entries.

Table 3.3 shows the Cartesian product entries along with assigned best matching rules that together represent the example rule set after the range-to-prefix conversion (see Table 3.2). Even though the range-to-prefix conversion increased the number of classification rules from 5 to 9 (cf. Tables 2.2 and 3.2), the number of unique *port* conditions increased from 4 to 7 only due to the conversion. Thus, every 4 entries sharing the same *address* prefix in Table 2.7 correspond to 7 entries sharing that prefix in Table 3.3. In total, due to the range-to-prefix conversion, the number of Cartesian product entries created from the sets of unique conditions utilized in *address* and *port* dimensions of the example rule set increased from 16 to 28.

The cross-producting algorithm is excellent with respect to search time. Because packet classification in individual dimensions can be performed in parallel and the final steps of the algorithm (i.e., the concatenation of prefix matching results and the retrieval of the best matching rule) have constant time complexity, the search time of the algorithm depends on the search time of employed prefix matching technique(s) only. However, memory requirements of this algorithm are prohibitive. The number of Cartesian product entries depends on the number of unique conditions in particular dimensions. This number is usually lower than the number of rules in a rule set but such redundancy cannot be guaranteed. Therefore, the space complexity of the cross-producting algorithm is $O(n^d)$ where n is the number of rules in the rule set and d is the number of dimensions of the packet classification problem. The authors of the cross-producting algorithm addressed the memory explosion problem of this algorithm by a caching-like technique called on-demand cross-producting [58]. Despite this way they were able to significantly reduce both average memory requirements and time needed for the construction of the table of Cartesian product entries, the worst case space complexity of on-demand cross-producting is the same as of the original cross-producting algorithm.

#	Address	Port	Rule	#	Address	Port	Rule
1	0*	101	R1a	15	010	101	R1a
2	0*	11*	R1b	16	010	11*	R1b
3	0*	110	R1b	17	010	110	R1b
4	0*	*	N/A	18	010	*	R3
5	0*	001	N/A	19	010	001	R3
6	0*	01*	N/A	20	010	01*	R3
7	0*	100	N/A	21	010	100	R3
8	*	101	N/A	22	11*	101	R4a
9	*	11*	N/A	23	11*	11*	R4b
10	*	110	R2	24	11*	110	R2
11	*	*	N/A	25	11*	*	N/A
12	*	001	N/A	26	11*	001	R5a
13	*	01*	N/A	27	11*	01*	R5b
14	*	100	N/A	28	11*	100	R5c

Table 3.3: The entries of a Cartesian product created from the sets of unique prefixes utilized in *address* and *port* dimensions of the example rule set after the range-to-prefix conversion (Table 3.2). If it exists, the best matching rule from the example rule set after the range-to-prefix conversion is associated with a Cartesian product's entry. Otherwise, the best matching rule is set to N/A.

Since the introduction of the cross-producting algorithm, numerous algorithms employing the combinatorial representation approach and addressing the memory explosion problem have been proposed.

Taylor and Turner have developed the DCFL (*Distributed Crossproducting of Field Labels*) algorithm [62], which performs the aggregation of classification results from individual dimensions in a distributed fasion rather than in a single step using the full table of Cartesian product entries. Distributed aggregation is implemented using pairwise cross-producting followed by a set membership query based on Bloom filters [22] (after each aggregation step) and the highest priority rule selection (after the last aggregation step only). It is also important to note that because the aggregation is performed in a distributed fashion, prefix matching technique(s) employed in DCFL are required to return all matching prefixes instead of just the longest one.

Another algorithm called MSCA (*Multi-Subset Crossproduct Algorithm*) has been proposed by Dharmapurikar et al. [25]. This algorithm uses the cross-producting algorithm as a starting point; however, the Cartesian product entries that have associated the best matching rule but do not directly correspond to the original rules (i.e., entries 3, 15, 16, 17, 19, 20, 21, and 24 in Table 3.3) are replaced by so called pseudo-rules. To reduce the space complexity of MSCA, the authors propose to split the original rule set into multiple subsets, which together contain a significantly smaller number of pseudo-rules. Nevertheless, introducing multiple rule subsets increases the number of memory accesses that have to be performed when retrieving the best matching rule. MSCA addresses this issue by employing Bloom filters, which eliminate the majority of combinations of the best matching prefixes from individual dimensions that do not correspond to a rule or pseudo-rule. Because the set of prefix conditions utilized in particular dimensions may differ among the subsets, it is also necessary to slightly modify employed prefix matching techniques to return the

longest matching prefix for each of the subsets. Further reduction of the number of pseudo-rules can be achieved by removing spoilers (i.e., rules that cause the excessive amount of pseudo-rules) from the rule subsets and treating them separately in a small TCAM.

The concept of pseudo-rules has also been used by Puš and Kořenek in their PHCA (*Perfect Hashing Crossproduct Algorithm*) [47]. However, this algorithm does not try to reduce the number of pseudo-rules. Instead, it constructs a perfect hash function that intentionally maps all pseudo-rules to the same position in a hash table as their corresponding original rule, which allows not to store the pseudo-rules explicitly. Since the perfect hash function employed in PHCA maps every combination of the best matching prefixes from individual dimensions to some rule, regardless it represents a (pseudo-)rule or not, the last step of the algorithm has to check whether the selected rule really matches the input packet.

Recursive Flow Classification

The RFC algorithm by Gupta and McKeown [31] builds on a unique view of the packet classification problem as the reduction of an input bit vector comprising the concatenated header fields to significantly shorter bit vector representing the set of rules matching the packet. Despite its similarities with the cross-producting algorithm, this algorithm represents a separate branch of decomposition-based packet classification methods. Differently from cross-producting, which combines the results of classification in individual dimensions in a single step, RFC performs reduction in several phases, each of which is based on precomputed aggregation tables that map an input index to *eqID* (equivalence class identifier) representing a set of potentially matching rules. Such *eqIDs* are often significantly shorter than input indexes to an aggregation table because real classification rule sets commonly utilize only a limited number of unique conditions in particular dimensions [31]).

In the first phase of reduction, the input bit vector is decomposed into several chunks (which may or may not correspond to header fields), utilized for indexing the first-phase aggregation tables. Further reduction phases index their tables by two or more *eqIDs* from the previous stage(s) that are combined together (e.g., using concatenation). This way the number of *eqIDs* passed to the next stage is successively reduced until the indexing of the last aggregation table returns only one *eqID*, which represents the set of rules matching the input packet. To obtain the best matching rule, the last *eqID* has to be decoded and the highest priority rule has to be selected.

Figure 3.9 shows the structure of 2-phase RFC and the content of its precomputed aggregation tables that allow to reduce a 6-bit input comprising the *address* and *port* header fields to 4-bit *eqID* using the example rule set introduced in Table 2.2. In the first phase, the input is divided into two 3-bit chunks corresponding to header fields. Both chunks are reduced in their respective first-phase aggregation tables to 2-bit *eqIDs*, each of which represent four different sets of potentially matching rules. Indexing the second-phase aggregation table using concatenated *eqIDs* obtained in the previous phase results into 4-bit *eqID* encoding 10 distinct rule sets, including an empty set, that can match an input packet (rule sets represented by particular *eqIDs* are shown in light grey in Figure 3.9). Note that the second-phase aggregation table performs reduction even though its input as well as output are 4-bit values—16 input combinations are reduced to 10 output values. For the sake of completeness it should also be noted that 2-phase version of the RFC algorithm (e.g., the one in Figure 3.9) is equivalent to the the cross-producting algorithm.

The RFC algorithm is similar to the cross-producting technique also in its search time and memory requirements. The implementation of reduction using aggregation tables in-

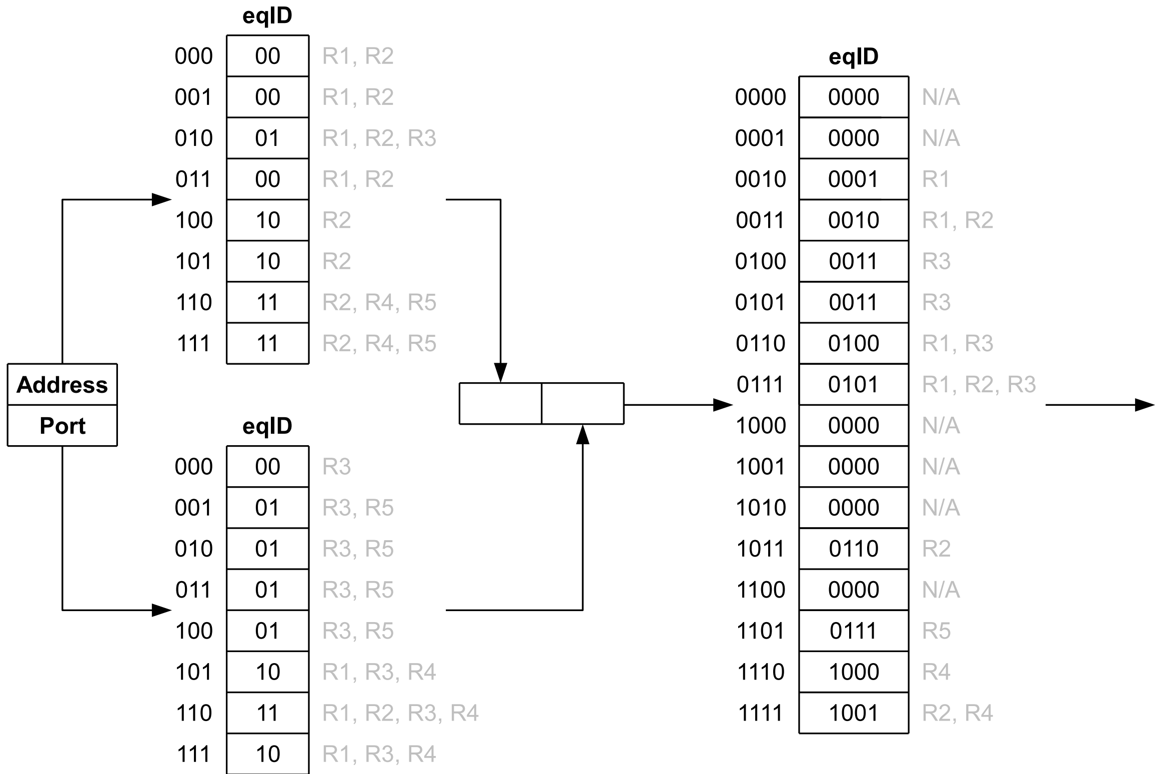


Figure 3.9: The structure of 2-phase RFC and the content of its precomputed aggregation tables for the example rule set (Table 2.2). Tables in the first phase are indexed by 3-bit chunks corresponding to the *address* and *port* header fields, respectively, while the second-phase table is indexed by 4-bit concatenation of *eqIDs* retrieved from the first-phase tables. The sets of (potentially) matching rules represented by particular *eqIDs* are shown in light gray. Note that $eqID = 0000$ in the second-phase table represents the situation when a rule matching the input packet is not available (N/A).

dexed by the chunks of input data or combined results from the previous phases allows the RFC to find the set of matching rules very fast. Moreover, the performance of the algorithm can be tuned via the number of reduction phases and the structure of a reduction tree. However, excellent performance is achieved at the cost of excessively high memory requirements and long preprocessing time, which is necessary for the construction of aggregation tables. Both these drawbacks get worse with the growing size of a classification rule set. The authors of RFC addressed this issue by merging similar rules together into so called adjacency groups. Nevertheless, such improvement of the RFC algorithm allows to just double the size of supported rule sets.

Bit-Vector-Based Algorithms

The seminal bit-vector-based packet classification algorithm, which is commonly referred to as parallel bit-vectors, has been published by Lakshman and Stiliadis [38]. To decompose a multi-dimensional packet classification problem into multiple 1-dimensional instances, this algorithm employs the geometric representation of packet classification. The edges of n rectangles, which represent n classification rules, are projected to the axes of the multi-

dimensional space, forming a set of at most $2n + 1$ elementary intervals in each dimension. The elementary intervals are then utilized in range matching, which is employed as 1-dimensional packet classification technique in this algorithm.

The authors of the parallel bit-vectors algorithm propose to represent the set of rules overlapping an elementary interval using an n-bit bit-vector, in which each bit corresponds to one of n classification rules sorted by their priority. The bits of the bit-vector corresponding to rules that overlap the elementary interval are set to 1, while the other bits are set to 0. Since such bit-vectors are also the result of range matching in individual dimensions, the aggregation step of the algorithm simply combines them together via the bitwise AND operation. The bits set to 1 in the resulting bit-vector correspond to the rules matching an input packet in all dimensions and the first such bit represents the highest priority matching rule.

Using the geometric representation of the example rule set (Table 2.2), Figure 3.10 illustrates the decomposition of packet classification, which is used in the parallel bit-vectors algorithm. Although 5 rules of the example rule set could project into $2 \cdot 5 + 1 = 11$ elementary intervals in *address* and *port* dimensions, there are only 5 elementary intervals in both cases. The reasons for this situation in the *port* dimension, which are also relevant for the *address* dimension, are explained in Section 2.3.6. Each elementary interval has assigned a 5-bit bit-vector, the bits of which correspond to rules R1, R2, R3, R4, R5, respectively. Thus, for instance, the *address* bit-vector assigned to the elementary interval corresponding to prefix 11* is 01011 because this interval is overlapped by rules R2, R4, and R5.

Storing an n-bit bit-vector for each elementary interval in a single dimension may require $n \cdot (2n + 1)$ bits in the worst case. Therefore, the space complexity of the algorithm in case of d-dimensional packet classification is $O(d \cdot n^2)$. On the other hand, its time complexity $O(d \cdot n)$ is determined by the time necessary for bitwise AND operation over d bit-vectors of n bits (note that the matching elementary intervals in individual dimensions can be found in parallel using binary search with logarithmic time complexity). Because the parameter d is usually a small constant, the parallel-bit vectors algorithm can be considered to have quadratic memory requirements and linear search time. Both these characteristics are mainly affected by the linear dependence of bit-vector length on the number of classification rules, which can be considered as the main drawback of this algorithm. Even though its hardware implementation, which the authors targeted on, can take advantage of bit-level parallelism (i.e., processing multiple bits of a bit-vector in parallel), its memory requirements as well as search time make it impractical for packet classification using a large rule set.

In [38], Lakshman and Stiliadis have also proposed an improved version of the parallel bit-vectors algorithm that is based on incremental reads. Replacing all but one bit-vectors in each dimension by pointers to bits that flip between elementary intervals decreases the memory requirements of the algorithm to $O(n \cdot \log n)$ and allows to reconstruct any bit-vector at the price of $O(n \cdot \log n)$ increase of search time. On the other hand, the Aggregated Bit Vector algorithm introduced by Baboescu and Varghese [21] addresses poor performance of the parallel bit-vectors algorithm, especially when a large set of classification rules is used. In order to reduce the number of memory accesses, it arranges the bits of bit-vectors regardless the priority of their corresponding rules and aggregates chunks of a rearranged bit-vector into a so called aggregate bit-vector, which is used in the course of packet classification. The bit-vector-based algorithms have also been used in the BV-TCAM architecture by Song and Lockwood [53], which combines the aggregated bit vector

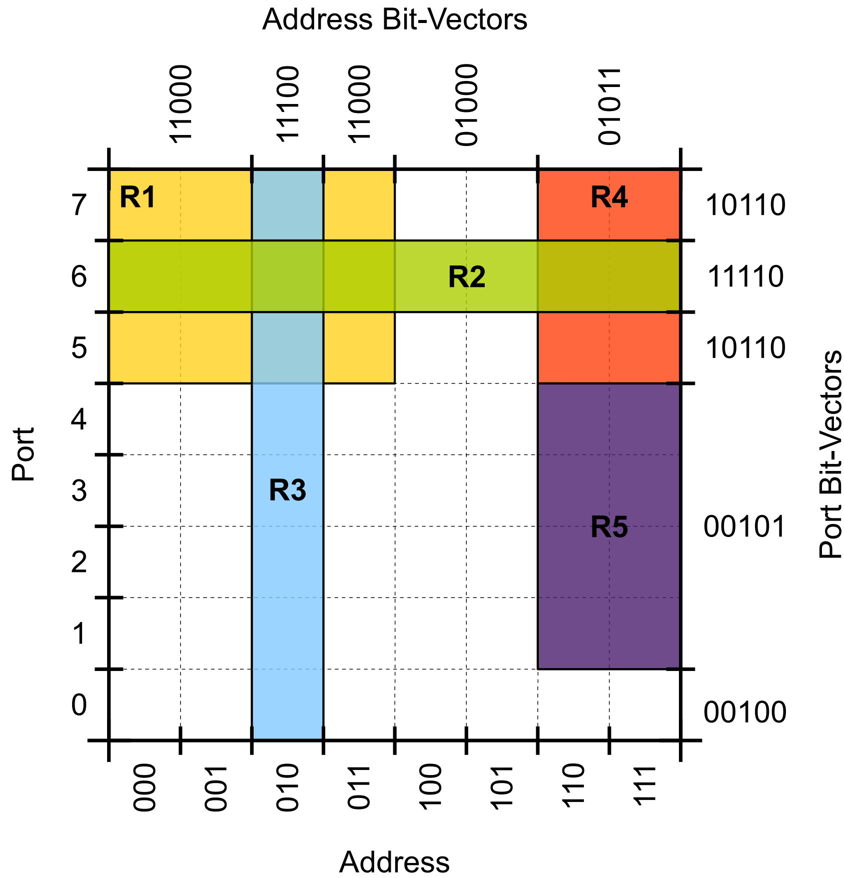


Figure 3.10: The geometric representation of the example rule set from Table 2.2 together with *address* and *port* bit-vectors representing the set of rules overlapping elementary intervals in these dimensions. The bits of the bit-vectors correspond to rules R1, R2, R3, R4, R5, respectively.

algorithm (for source and destination ports) and TCAM (for other header fields) to further reduce both memory requirements and search time.

3.2.4 Tuple Space

This section is devoted to the description of packet classification algorithms that are primarily based on the representation of a classification rule set using tuples. The majority of these algorithms have been introduced by Srinivasan, Suri, and Varghese in their seminal paper on tuple space search [56]. A simple linear search of tuple space, which was already described in Section 2.3.3, represents the basis of all tuple space search algorithms. It benefits from the observation that the number of tuples representing a rule set is usually lower than the number of rules in that set, thus it is usually faster than the linear search of the original rule set. However, memory requirements of both approaches are linear with respect to the number of classification rules.

To further improve the search time of tuple space search, the tuple pruning algorithm leverages another property common in real classification rule sets: the number of prefixes matching a given IP address is inherently limited. For instance, in [56] the authors report

no more than six simultaneously matching prefixes for a prefix set taken from a core router. Therefore, the space of tuples that have to be searched can be significantly limited by performing prefix matching for source and destination IP addresses and pruning out the tuples not compatible with the matching prefixes. In general, tuple space can be pruned according to matches in any subset of dimensions. Nevertheless, the authors of the algorithm suggest pruning according to source and destination address prefixes as a good choice, at least for IPv4 5-tuples.

The implementation of the tuple pruning algorithm utilizes a tuple list (or bitmap) that is assigned to each source and destination address prefix and comprises tuples containing at least one rule with that prefix. Figure 3.11 shows tuple lists assigned to *address* and *port* prefixes of the example rule set after the range-to-prefix conversion (see Table 3.4), which are encoded in address and port tries. The tuple pruning algorithm first performs prefix matching in each dimension individually and applies the union operation on tuple lists assigned to all matching prefixes. Next, the algorithm determines the intersection of the resulting tuple lists from particular dimensions and linearly searches the pruned tuple space represented by this intersection. Considering a packet with $address = 110$ and $port = 110$, prefix matching results in tuple list [0,3], [2,3], [2,2] in the *address* dimension and [3,0], [1,2], [2,2], [0,3] in the *port* dimension. Their intersection [0,3], [2,2] then defines pruned tuple space.

Rule	**Address**	**Port**	**Tuple**
R1a	0*	101	[1,3]
R1b	0*	11*	[1,2]
R2	*	110	[0,3]
R3	010	*	[3,0]
R4a	11*	101	[2,3]
R4b	11*	11*	[2,2]
R5a	11*	001	[2,3]
R5b	11*	01*	[2,2]
R5c	11*	100	[2,3]

Table 3.4: The example rule set (Table 2.2) after the range-to-prefix conversion. For each rule the table also shows a tuple the rule belongs to.

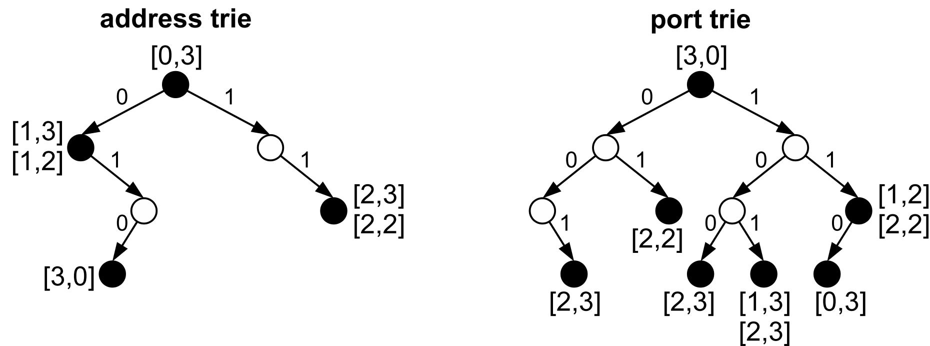


Figure 3.11: Address and port tries encoding the prefixes of the *address* and *port* dimensions of the example rule set after the range-to-prefix conversion (Table 3.4). Each *address* and *port* prefix has associated the list of tuples containing rules with that prefix.

As long as the assumption about the limited number of prefixes matching an IP address holds, tuple pruning outperforms linear tuple space search with respect to search time. This technique can also perform fast updates of the rule set if the implementation of tuple lists is augmented with the reference count of distinct filters using the corresponding prefix in each listed tuple.

In their seminal paper [56], Srinivasan, Suri, and Varghese have also explored several approaches to improving the search time of linear tuple space search via precomputation and markers introduced by Waldvogel et al. in [65]. Among other algorithms, their effort resulted in a general 2-dimensional tuple space search algorithm called rectangle search, whose name refers to the representation of 2-dimensional tuple space by a rectangular grid of tuples. Since the most common type of 2-dimensional packet classification performs prefix matching according to source and destination IP addresses, the rectangle search algorithm primarily targets a square grid of $w \times w$ tuples, where w is the maximum length of an IP address prefix. In such grid, the utilization of precomputation and markers allows the algorithm to probe only tuples on a staircase-like path leading from the bottom left corner of the grid (tuple $[w,1]$) to its upper right corner (tuple $[1,w]$). Therefore, rectangle search requires only $2w - 1$ steps for fully searching the space of $w \times w$ tuples, which has been proved to be optimal search performance for 2-dimensional tuple space search [56].

3.3 1-Dimensional Rule Set Generators

Benchmarking IP lookup algorithms, which implement the most common type of 1-dimensional packet classification in core networks, requires a set of classification rules where each rule consists of a destination IP address prefix. Real IP prefix sets can be obtained from forwarding tables of core routers for both IPv4 and IPv6 case [18, 50]. Nevertheless, differently from sets of IPv4 prefixes, current IPv6 prefix sets cannot be directly used for benchmarking future IP lookup algorithms because their size is expected to grow significantly with progress in the adoption of the IPv6 protocol. Thus, all rule set generators presented in this section implement the generation of synthetic IPv6 prefix sets.

3.3.1 Non-random Generator

Non-random Generator, which has been developed by Wang et al. [66], is the first of two IPv6 prefix set generators described in this thesis. Its authors expected that many of the features observed in IPv4 prefix sets will also emerge in IPv6 prefix sets because of similar allocation policies, retained network topology, and continuing evolution of the Internet. Therefore, they designed the generator such that it converts an input IPv4 prefix set to an output IPv6 prefix set in a way that preserves selected features of the input and generates the output with respect to IPv6 allocation policies. During the generation of the output prefix set, Non-random Generator focuses on three main properties: (1) prefix set size, (2) prefix length distribution, and (3) prefix value distribution.

As each input IPv4 prefix is converted to a single output IPv6 prefix, the size of the output prefix set is directly inherited from the input set. The length of generated prefixes is limited to the maximum of 64 bits, which reflects the internal structure of the global unicast IPv6 address, where upper 64 bits represent a network address (i.e., the information relevant in IPv6 lookup algorithms). To determine the exact length of an output IPv6 prefix, Non-random Generator doubles the length of an input IPv4 prefix. However, the length of every fourth IPv6 prefix is converted to an odd number by either adding or subtracting 1 (to

comply with the distribution of odd to even prefix lengths in real IPv4 prefix sets) and the length of an output prefix is directly set to 23 in case of an 8-bit input prefix (to account for IPv6 allocation policies).

Once the length of the output IPv6 prefix is determined, its value is composed according to Figure 3.12. The first three bits are always set to `001` because Non-random Generator supports prefixes of global unicast IPv6 addresses only. Next 16 bits are filled with the number of AS (*Autonomous System*) corresponding to the input IPv4 prefix, which is expected to be a suitable top-level network address representation. The rest of the output prefix then consists of the input prefix with randomized first three bits (to remove an artifact in prefix value distribution of real IPv4 prefix sets). In case the IPv6 prefix should consist of more bits than specified by the previous three steps, the remaining bits are generated randomly.

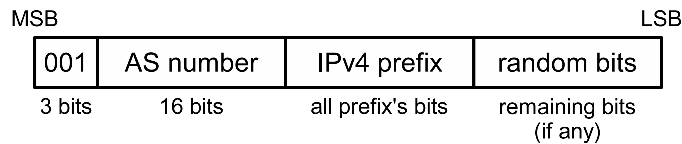


Figure 3.12: The composition of IPv6 prefixes generated by Non-random Generator.

Since Non-random Generator was the first generator of synthetic IPv6 prefix sets, it was extremely useful at the time of its origin. Nevertheless, prefix sets generated by this tool do not correspond to current real IPv6 prefix sets because their generation is based on sets of IPv4 prefixes. This fact limits their usability for the benchmarking of IPv6 lookup algorithms, which has to be done using synthetic prefix sets that are as similar as possible to real prefix sets. Therefore, newer generators are based on currently available real sets of IPv6 prefixes. Such approach is valid because the adoption of the IPv6 protocol is no longer in its initial phase.

3.3.2 V6Gene

A well-known example of a generator based on real sets of IPv6 prefixes is called V6Gene [67]. This tool has been designed by Zheng and Liu, who aimed to overcome the reasons that prevent real IPv6 prefix sets from being used for benchmarking IPv6 lookup algorithms. V6Gene thus starts from a real prefix set of a small size and systematically enlarges this set by new prefixes generated in a way that simulates the allocation of address prefixes in the real IPv6 world. The generation of an output prefix set is performed in three phases (Figure 3.13), the first two of which are briefly described in the following paragraphs.

During the initiation phase, V6Gene reads the input IPv6 prefix set along with several parameters that either define target properties of the output prefix set (e.g., the number of prefixes and distribution of their lengths) or constrain the process of its generation (e.g., required accuracy). It also examines the input prefix set in order to identify and remove invalid or redundant prefixes. The finial step of this phase then comprises the construction of a trie representing the pruned input prefix set in all subsequent phases of the V6Gene's run.

The generation phase is divided into two parallel branches, which implement different algorithms introducing new IPv6 prefixes into the input prefix set. While the first branch simulates the allocation of IPv6 prefixes from LIR (*Local Internet Registry*) prefixes that already exist in the prefix set to their subscribers, the second branch allocates IPv6 prefixes

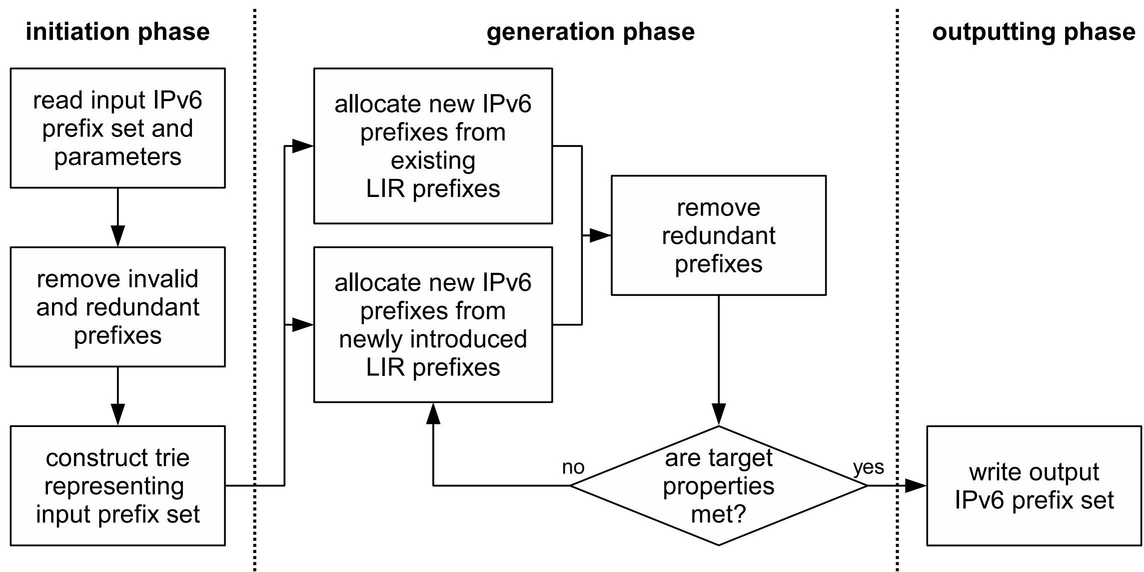


Figure 3.13: Three phases of IPv6 prefix set generation in V6Gene.

from a limited number of newly introduced LIR prefixes. After combining the outputs of both branches together, V6Gene removes redundant prefixes and verifies the properties of the resulting prefix set. If the prefix set does not contain the specified number of IPv6 prefixes or its properties are not close enough to the given target properties, the second branch is repeatedly run again until the output prefix set successfully pass the verification step.

As the generation process is based on real IPv6 prefix sets in V6Gene, this tool can generate synthetic prefix sets that are closer to real IPv6 prefix sets than the output of Non-random Generator. However, the generator does not explicitly account for prefix aggregations and splits known from real prefix sets. It delegates the responsibility for their generation to the user, who can control their presence in the output prefix set by tuning the prefix length distribution only. Moreover, although the authors highlighted the availability of their tool, currently there is no publicly available V6Gene's implementation.

3.4 Multi-Dimensional Rule Set Generators

Since multi-dimensional rules are utilized in more than just one packet classification use case and real rule sets for these use cases are not publicly available, multi-dimensional packet classification algorithms have to be benchmarked using various types of synthetic rule sets. All synthetic rule set generators that are described in this section support the generation of IPv4 5-tuples, which still represent the basic type of a multi-dimensional classification rule. However, with an increasing IPv6 penetration the number of algorithms that perform packet classification according to IPv6 5-tuples is rising. Even more is growing the number of OpenFlow deployments, which translates into rapidly increasing utilization of algorithms that implement packet classification based on more than 5 header fields. Therefore, some of the presented multi-dimensional synthetic rule set generators also allow to generate sets of IPv6 5-tuples or classification rules based on more than 5 header fields.

3.4.1 ClassBench

ClassBench [63], which has been introduced by Taylor and Turner, is probably the most often used multi-dimensional synthetic rule set generator. It is actually a toolset consisting of three independent tools that together allow to produce realistic data sets for benchmarking packet classification algorithms. The most important component of the toolset is the Filter Set Generator that implements the generation of synthetic IPv4 5-tuples according to an input seed (referred to as a parameter file in [63]) and a small set of high-level parameters. The remaining tools then implement the construction of the seed from a real rule set (the Filter Set Analyzer) and the generation of a synthetic header trace, which is able to comprehensively exercise a packet classification algorithm utilizing a given rule set (the Trace Generator).

The Filter Set Generator's input seed contains several statistical distributions that jointly make possible to generate a condition for each header field belonging to an IPv4 5-tuple. The first group of these distributions is related to the transport layer protocol and source/destination port fields (i.e., the specification of communicating applications). While the protocol field is directly represented by the distribution of rules over unique protocol values, representation is not so straightforward in case of source/destination port fields because ClassBench encodes their value using the following five classes.

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

Therefore, the seed contains the distribution of rules over 25 source-destination PPCs (*Port Pair Classes*), separately for each protocol value. In addition, for both source and destination ports the seed also contains distributions of rules specifying AR and EM over unique arbitrary ranges and exact port values, respectively.

The second group of statistical distributions stored in the seed characterizes source/destination address prefixes (i.e., the specification of communicating subnets). The majority of these distributions relates to the properties of a trie representing either source or destination address prefix set. Probably the simplest one of them is the prefix length distribution, which is defined for source prefixes and whole prefix pairs in the seed (note that the destination prefix length is equal to the difference of the prefix pair length and the source prefix length). Another trie-based distribution included in the seed—the branching probability distribution—is defined for non-leaf nodes and it expresses the probability that a node at a given trie level has either one child or two children. Last but not least, the seed contains the average skew distribution at individual trie levels. Skew of a two-children node is computed using Equation 3.1, where the $weight()$ function returns the number of prefixes in a specified subtree and $lighter/heavier$ represent subtrees of the node with a lower/higher number of prefixes, respectively.

$$skew = 1 - \frac{weight(lighter)}{weight(heavier)} \tag{3.1}$$

Separately for the source and destination trie, the seed also defines the prefix nesting threshold, which specifies the maximum number of prefix nodes that may appear on an arbitrary path from the root node to the leaves. The only distribution common for both source and destination address prefixes describes their correlation in the prefix pair, or more precisely probability that they continue to be the same for a given prefix length.

Figure 3.14 illustrates the presented trie-based distributions on a trie that encodes a set of ten prefixes having length between 1 and 5 bits. To ease an understanding of the example, leaf nodes are shown in black, while one-child and two-children nodes are highlighted by red and green, respectively. In addition, next to two-children nodes there is also the value of their skew shown in blue. Apart from the prefix nesting threshold, which is 2 in this example, all the distributions are defined over individual trie levels. For instance, the last level encodes 2/10 of all prefixes and 3/4 of non-leaf nodes at the third level have only one child whereas 1/4 of them has two children (note that the sum of branching probability distributions for one-child and two-children nodes is always 1, if they are defined). In a similar manner, the average skew at the second level is $(1/2 + 0 + 1/2)/3 = 1/3$.

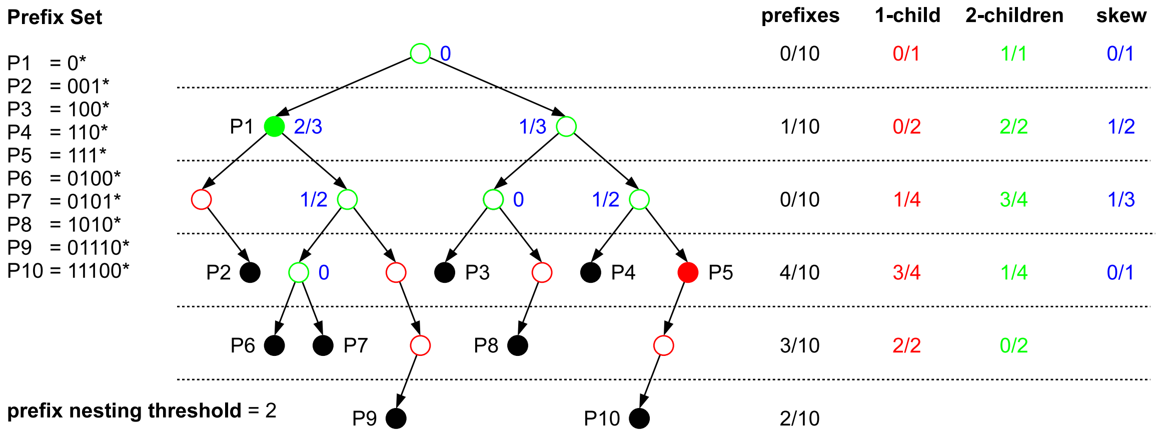


Figure 3.14: An example prefix set encoded using a trie along with the value of the prefix nesting threshold and the values of trie-based distributions for individual levels.

In general, the Filter Set Generator implements the generation of individual classification rule's conditions as sampling corresponding distributions stored in the input seed. However, because of interdependence of some distributions (e.g., the PPC distribution depends on the protocol value and the prefix pair length distribution depends on the selected PPC), sampling has to be done in a specific order. Moreover, generating some conditions of a classification rule is not as straightforward as sampling a distribution, which is true especially for source/destination address prefixes. Their length has to be determined from a combination of the prefix pair length and source prefix length distributions and also to obtain their value it is necessary to first construct tries complying with the prefix nesting threshold and all trie-based distributions. With respect to rule set generation it is also important to note that for the sake of simplicity the Filter Set Generator does not remove redundant rules on the fly, but postpones their removing to the final step of the generation process. Therefore, the output rule set might contain less than the specified number of classification rules.

A great popularity of ClassBench has been caused not only by the Filter Set Generator's ability to produce realistic IPv4 5-tuples for benchmarking packet classification algorithms,

but also by providing a set of twelve input seeds extracted from real classification rule sets of various origin (access control lists, firewalls, IP chains). Although these seeds were extracted from rule sets that are not publicly available, they can be freely distributed with no concerns about security and confidentiality because their representation of an original rule set is based on statistical distributions, thus they are anonymous. Therefore, since the time of ClassBench's publication, the seeds provided with it have become de facto standard inputs for the generation of synthetic data for benchmarking various packet classification algorithms. However, ClassBench is no longer sufficient for current needs of the research community as it focuses on IPv4 5-tuples only and does not support the generation of IPv6 5-tuples or classification rules comprising a condition for more than 5 header fields. These drawbacks have been addressed by more recent multi-dimensional synthetic rule set generators ClassBenchv6 and FRuG, which are introduced in the following sections.

3.4.2 ClassBenchv6

In order to allow the generation of synthetic classification rules with IPv6 address prefixes, Sun et al. have proposed ClassBenchv6 [59], a reshaped version of ClassBench for the IPv6 world. Similarly to original ClassBench, this tool generates IPv6 5-tuples according to statistical distributions stored in an input seed. However, at the time of designing ClassBenchv6, the authors of this tool did not consider IPv6 deployment to had already passed its initiation phase and they expected that IPv6 prefix set-related distributions will undergo non-negligible changes in the future. Therefore, rather than using a seed extracted from a real set of IPv6 5-tuples, ClassBenchv6 builds on similarities between IPv4 and IPv6 environments (similar allocation policies, retained Internet topology and classification use cases, continuing evolution of the Internet) and predicts an IPv6 seed from an IPv4 seed corresponding to a real rule set.

To predict the trie-based distributions characterizing an IPv6 prefix set, ClassBenchv6 utilizes two piecewise functions: (1) M_l for the prefix length distribution and (2) M_{bs} for distributions related to a prefix value (the branching probability and average skew distributions). Both these functions have been designed according to correspondence between IPv4 and IPv6 address allocation policies and they essentially define the transformation of an IPv4 trie level to an equivalent IPv6 trie level. This means that level i of the IPv4 prefix length distribution is transformed to level $M_l(i)$ of the IPv6 prefix length distribution and the same transformation, but using the M_{bs} function, applies also to the branching probability and average skew distributions.

Since ClassBenchv6 generates IPv6 5-tuples in a similar way as the Non-random Generator produces IPv6 prefixes, it also has similar benefits and drawbacks. The prediction of IPv6 trie-based distributions based on corresponding IPv4 distributions was useful during the initiation phase of IPv6 deployment. Nevertheless, it is no longer valid. As shown by Czyz et al. [23], the IPv6 protocol has already shifted into production mode, thus it is desirable to extract IPv6 seeds from real rule sets.

3.4.3 FRuG

Even though both ClassBench and ClassBenchv6 theoretically allow to generate conditions for selected header fields beyond IPv4/IPv6 5-tuple (e.g., IPv4 flags in ClassBench and IPv6 flow label in ClassBenchv6), the full support of rules with a condition for more than 5 header fields was not available until Ganegedara, Jiang, and Prasanna had introduced FRuG [29]. This synthetic rule set generator allows the user to fully control the size and structure of

generated rule sets and also to define specific distribution for each included header field. As shown in Figure 3.15, it consists of three generation engines (IPv4 prefix generator, MAC address generator, and FRuG engine), which are augmented by three analysis/parser modules (IPv4 prefix analyzer, MAC address analyzer, and configuration file parser) that analyze corresponding distributions in given input sets of IPv4 prefixes and MAC addresses or parse user-defined instructions specified in parameter files. Such a modular structure allows FRuG to not only generate rule sets with flexible structure, but also to separately generate IPv4 prefix sets.

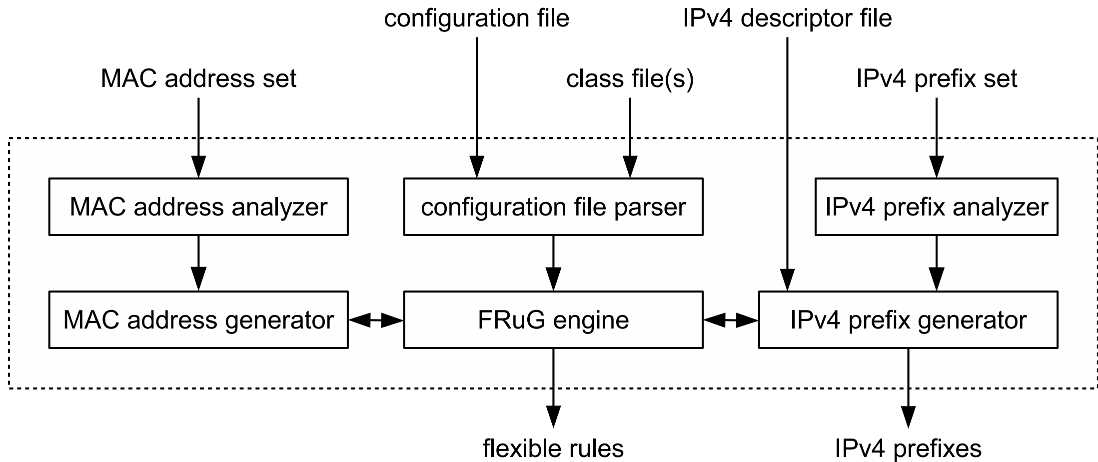


Figure 3.15: The high-level architecture of FRuG.

The user can control the rule set generation process of FRuG via parameter files of three types. The highest level parameters are specified in a configuration file that defines the structure of each generated class of rules (including the type of specification for all supported header fields), its percentage of the whole rule set, and a corresponding class file. Particular class files then specify distributions for header fields present in rules of this class. The only exception is represented by IPv4 prefixes, which are defined in a separate IPv4 descriptor file using the prefix length percentage and the branching probability distribution (i.e., a relative number of prefixes with a given bit set to 1) for each prefix length. Apart from these parameter files, FRuG can also optionally utilize input sets of IPv4 prefixes and/or MAC addresses and extract the distributions for corresponding header fields from them.

Each of the FRuG's generation engines is responsible for the generation of conditions for a subset of supported header fields. IPv4 prefixes are generated in the IPv4 prefix generator, which straightforwardly follows the prefix length percentage and the branching probability distribution obtained either directly from the IPv4 descriptor file or indirectly from the input IPv4 prefix set using the IPv4 prefix analyzer. It also allows to bias the generated IPv4 prefixes by specifying their common prefix. In the course of generating MAC addresses, the MAC address generator also utilizes distribution that is either defined by the user in the class file or extracted from the input MAC address set by the MAC address analyzer. However, this distribution refers to the vendor part of a MAC address only, thus the device part is generated randomly. Conditions for the remaining header fields are generated in the FRuG engine, which incrementally produces rules of particular classes defined in the configuration file until the final rule set contains the specified number of rules for each of the classes. During this step, the FRuG engine generates a condition for

individual header fields sequentially in the order of their appearance in the configuration file and with respect to their descriptors (wildcard, exact value, range, and not-for-print). For the sake of completeness it is worth to note that all the generation engines remove redundant conditions and rules on the fly.

FRuG gives the user a complete control over the size and structure of the generated rule set, which makes it a powerful benchmark to assess various packet classification algorithms. However, only MAC address and IP address fields can be set to follow corresponding distributions from an input set. Distributions for other header fields have to be manually configured by the user, making this generator less attractive if a realistic set of synthetic rules needs to be generated. Moreover, even though FRuG currently supports the generation of condition for 12 header fields utilized in OpenFlow 1.0.0, it neither explicitly considers specifics of OpenFlow rule sets nor allows to easily specify correlation among header fields of generated rules.

Chapter 4

Addressing Performance of Algorithms

As described in details in Section 2.4.1, packet classification algorithms targeting core networks are required to support growing transfer rates and an increasing number of rules as well as bits involved in packet classification. Current classification algorithms have to be able to classify an incoming packet according to tens or hundreds of bits into tens or hundreds of thousands of classes and provide a new classification result every 6.72 ns. Therefore, this chapter deals with improving the performance of classification algorithms in order to meet the above mentioned requirements.

Since matching a condition specified as an exact value or wildcard is trivial, the greatest improvement of classification algorithms' performance can be achieved by optimizing either prefix matching or range matching. Because prefix matching is utilized in IP lookup (i.e., probably the most common type of packet classification in core networks) and also in the majority of multi-dimensional packet classification algorithms (e.g., trie-based algorithms, combinatorial algorithms, or the tuple pruning algorithm), this thesis addresses the performance of packet classification algorithms via improvements of prefix matching. Focus on matching prefixes is also beneficial because a prefix is a typical representation for conditions on IP addresses, which are the widest dimension in current classification rules. Moreover, as noted in Section 2.4.1, supporting IPv6 may be more expensive than supporting additional header fields with respect to the number of bits involved in packet classification.

The performance of current CPUs prohibits software implementation of packet classification algorithms. Even for relatively simple IP lookup, software implementation is not able to achieve 148.81 MLPS, which is the minimum required performance in 100 Gbps networks. Thus, packet classification algorithms have to be implemented in hardware. Because of availability, flexibility, and configurability, this thesis targets implementation in FPGAs. The inherent parallelism of this technology also seamlessly supports pipelined processing, which is a necessity for a high-performance implementation of classification algorithms.

The bottleneck of pipelined implementation lies in accessing a memory that stores a data structure encoding a set of classification rules. Since each of n pipeline stages may access the memory in any clock cycle, the whole pipeline may generate n parallel memory accesses per clock cycle in the worst case. The requirement on supporting multiple parallel memory accesses makes the use of an external memory, which could provide high capacity, practically impossible. However, this requirement can be easily fulfilled by using a distributed on-chip memory that is available on FPGA chips. Each pipeline stage may have allocated

one or more on-chip memory blocks, which can be accessed by that pipeline stage only. Nevertheless, because of a limited number of on-chip memory blocks, the rule set has to be represented using a memory-efficient data structure.

In the previous paragraphs it has been argued that the performance of packet classification algorithms may be addressed via pipelined implementation of prefix matching that targets an FPGA and utilizes its distributed on-chip memory. Although the Trie [27], TBM [26], and MSLA [36] prefix matching algorithms can be implemented in an FPGA, their prefix set representations do not fit into a memory available on FPGA chips. On the other hand, the SST [54] algorithm represents a prefix set in a compact way that is suitable for an on-chip memory with a limited capacity, but there is no hardware architecture implementing this algorithm. To meet both requirements at the same time, it is possible to implement prefix matching using the PPLA [39] algorithm. Nevertheless, its initial prefix set partitioning introduces very high preprocessing overhead.

To overcome the drawbacks of existing prefix matching algorithms, this chapter presents a memory-efficient trie-based representation of a prefix set together with a pipelined hardware architecture for prefix matching based on this represenation in an FPGA. The prefix set representation was designed according to the results of the analysis of available prefix sets (real IPv4 and IPv6, generated IPv6) and it allows to store the whole data structure representing a prefix set into memory blocks available on current FPGAs. Subsets of these memory blocks are allocated to particular pipeline stages of the architecture, which allows the pipeline stages to access their memory blocks independently of each other. Thus, the architecture achieves matching performance required in 100 Gbps networks.

Both the representation and architecture were published in [43, 42] and the rest of this chapter, which is organized as follows, is based on these papers. Section 4.1 describes the performed analysis of real prefix sets, the results of which were used when designing the proposed prefix set representation introduced in Section 4.2. Next, the hardware architecture for prefix matching based on the proposed representation is described in Section 4.3. Section 4.4 presents the results of experimental evaluation of the proposed representation as well as the hardware architecture. The chapter is summarized in Section 4.5.

4.1 Analysis of Real Prefix Sets

The goal of this chapter is to develop a prefix matching algorithm based on a trie, i.e., the most common prefix set representation in algorithmic approaches to prefix matching, which is also suitable for pipelined processing. Specifically, the algorithm should utilize a multibit trie with memory-efficient encoding of its nodes that would allow to store a prefix set representation in on-chip memory blocks of FPGA chips. Since the algorithm is expected to be used in packet classification techniques targeting core networks, its prefix set representation can leverage inherent properties of prefix sets from this environment. Therefore, this section uses prefix sets extracted from forwarding tables of core routers and analyzes their representation by a trie as well as by multibit tries utilized in TBM and SST algorithms.

Basic information about prefix sets used in the course of the analysis are summarized in Table 4.1. In order to obtain results relevant for many different situations, real sets of IPv4 and IPv6 prefixes originate from various sources and they were acquired on different days. Moreover, to model the situation in future core networks, IPv6 prefix sets generated by [66] are also included. The analysis of prefix set representation is performed using the Netbench tool [48].

Prefix Set	Prefixes	Source	Date
		IPv4	
rrc00	332 118	http://data.ris.ripe.net/	2010-06-03
IPv4-space	220 779	http://bgp.potaroo.net/	2011-12-21
route-views	442 748	http://archive.routeviews.org/	2012-09-20
		IPv6	
AS1221	10 518	http://bgp.potaroo.net/	2012-09-21
AS6447	10 814	http://bgp.potaroo.net/	2012-09-21
		Generated IPv6	
rrc00_ipv6	319 998	generated using [66] from rrc00	
IPv4-space_ipv6	150 157	generated using [66] from IPv4-space	
route-views_ipv6	439 880	generated using [66] from route-views	

Table 4.1: Basic characteristics of used prefix sets.

The first part of the analysis was focused on memory requirements of a trie and multibit tries utilized in TBM and SST algorithms when representing used prefix sets. Its results are shown in Table 4.2. In this part of the analysis, the value of parameters SL and K was chosen with respect to the minimum memory requirements. The changing value of TBM's parameter SL reflects varying density of the underlying trie. Indeed, denser tries (IPv4 prefix sets) are more effectively represented by greater TBM nodes, while for sparser tries (IPv6 prefix sets), smaller TBM nodes introduce lower memory overhead. On the other hand, SST's parameter K was set to the same value for all prefix sets, which corresponds with the adaptivity of SST nodes. The chosen value of this parameter ($K = 32$) allows the SST nodes to represent approximately the same number of underlying trie nodes as TBM nodes for parameter $SL = 5$. The missing results of SST's memory requirements for generated IPv6 prefix sets could not be provided because of very high computational complexity of the SST algorithm.

The results of the performed analysis confirm expected properties of the examined prefix set representations. As Table 4.2 shows, a multibit trie of the SST algorithm has the lowest

Prefix Set	Prefixes	Memory Requirements [kb]		
IPv4		**Trie**	**TBM** ($SL = 5$)	**SST** ($K = 32$)
rrc00	332 118	47 639.7	9 689.4	6 930.4
IPv4-space	220 779	24 252.4	5 702.1	4 081.0
route-views	442 748	62 650.5	11 942.1	8 775.0
IPv6		**Trie**	**TBM** ($SL = 3$)	**SST** ($K = 32$)
AS1221	10 518	3 518.3	1 076.9	588.5
AS6447	10 814	3 673.8	1 125.1	617.1
Generated IPv6		**Trie**	**TBM** ($SL = 4$)	**SST**
rrc00_ipv6	319 998	307 641.5	87 257.1	N/A
IPv4-space_ipv6	150 157	153 877.3	43 958.7	N/A
route-views_ipv6	439 880	418 663.7	118 889.4	N/A

Table 4.2: Memory requirements of a trie and multibit tries utilized in TBM and SST algorithms when representing prefix sets from Table 4.1.

memory requirements, while a simple trie is the least effective representation. Such results would propose the SST algorithm to be a candidate for further optimizations. However, SST suffers from high computational complexity and there is no hardware architecture implementing this algorithm. Another candidate for further optimizations would be the TBM algorithm, which can be easily implemented in hardware. Nevertheless, its memory requirements are higher than those of SST. Thus, in order to develop a prefix matching algorithm with a memory-efficient representation of a prefix set that would be amenable for processing in hardware, it will be necessary to combine positive aspects of both TBM and SST. To support this, the further analysis focuses on the structural characteristics of a multibit trie utilized in TBM in order to identify possibilities for its optimizations with respect to required memory.

The analysis of TBM's prefix set representation was performed by classifying TBM nodes according to the number of child nodes and the number of prefixes represented by a node. The results of such classification for selected IPv4, IPv6 and generated IPv6 prefix sets are shown in Tables 4.3, 4.4, and 4.5, respectively. Despite all tables show the classification of TBM nodes with $SL = 3$, similar results were achieved also for other settings of parameter SL.

The results of TBM nodes' classification for all selected prefix sets show two interesting groups of nodes that introduce high memory overhead when encoded in the standard TBM format. The first group comprises leaf nodes (i.e., the leftmost column in Tables 4.3, 4.4, and 4.5), which do not utilize child-related information (the external bitmap and child pointer)

	Child Nodes								
Prefixes	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**26 829**	**11 859**	**6 876**	**5 422**	**3 679**	**3 547**	**4 297**	**14 138**
1	**278 804**	6 220	4 244	2 840	4 463	1 683	2 416	876	2 051
2	**21 005**	3 270	4 198	1 599	2 688	724	792	393	842
3	**5 716**	1 093	2 000	596	806	293	286	160	306
4	**3 786**	447	543	220	322	106	129	102	267
5	**679**	63	55	22	48	20	25	25	78
6	**298**	30	22	9	23	3	9	6	64
7	**70**	6	3	3	8	4	3	7	46

Table 4.3: Classification of nodes comprising the TBM's representation of the `route-views` prefix set (434 552 Nodes, $SL = 3$).

	Child Nodes								
Prefixes	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**11 303**	**1 666**	**812**	**538**	**184**	**145**	**131**	**249**
1	**8 965**	547	142	19	17	3	2	1	1
2	**193**	21	14	4	3	0	1	0	0
3	**50**	3	3	3	1	0	1	0	0
4	**29**	3	1	1	3	1	1	0	0
5	**0**	1	0	1	0	0	0	0	0

Table 4.4: Classification of nodes comprising the TBM's representation of the `AS1221` prefix set (25 063 Nodes, $SL = 3$).

Prefixes	Child Nodes 0	1	2	3	4	5	6	7	8
0	0	**1 597 683**	**143 258**	**39 958**	**21 056**	**9 332**	**5 637**	**3 958**	**4 462**
1	**406 100**	3 503	746	263	108	45	15	10	6
2	**2 623**	171	118	40	39	21	8	6	1
3	**475**	37	24	5	7	3	1	3	1
4	**155**	11	7	3	1	1	0	0	0
5	**44**	1	4	3	2	1	0	0	0
6	**12**	0	1	0	0	0	0	0	0
7	**2**	0	0	0	0	0	0	0	0

Table 4.5: Classification of nodes comprising the TBM's representation of the `route-views_ipv6` prefix set (2 239 971 Nodes, $SL = 3$).

available in the data structure encoding a TBM node. On the other hand, the second group comprises internal nodes without prefixes (i.e., the uppermost row in Tables 4.3, 4.4, and 4.5), which do not utilize prefix-related information (the internal bitmap and prefix pointer) available in the data structure encoding a TBM node. Because these groups contain the majority of TBM nodes encoding a given prefix set, even small optimization of these nodes' encoding may significantly reduce overall memory requirements of the TBM algorithm.

4.2 Proposed Prefix Set Representation

The analysis performed in the previous section has shown that the multibit trie representation of a prefix set in the TBM algorithm comprises, among other things, two groups of nodes, which do not fully utilize the TBM node's data structure. Since the representation of a prefix set in TBM consists mainly of nodes belonging to either of these groups, memory requirements of this algorithm may be substantially reduced by even a bit more efficient encoding of these nodes. To this end, this section proposes a new multibit trie representation of a prefix set that utilizes thirteen different types of node.

The nodes of the proposed prefix set representation can be divided into two groups: (1) nine newly proposed nodes and (2) four variants of a standard TBM node. The newly proposed nodes are illustrated in Figure 4.1, which is organized as a grid of three columns and three rows. The nodes of each column and row share the same property, which is also reflected in their name. Left, middle, and right columns contain nodes that can encode a subtree of an underlying trie consisting of one branch (1B), two branches (2B), and three branches (3B), respectively. In addition, each row contains nodes designed for encoding a specific situation with respect to prefix and leaf nodes of the underlying trie. The nodes in the top row can only encode internal nodes without prefixes. In the middle row, there are nodes that can also encode internal prefix nodes (P), but such prefix nodes are only allowed at the lowest level of the underlying subtree and they may not occur in all branches. Finally, the bottom row contains nodes that can only encode subtrees in which each branch is terminated by a leaf node (-L). Utilized variants of a TBM node ensure completeness and efficiency of the representation in less common situations. They include a standard node for $SL = 3$ (TBM3) and a set of leaf nodes for $SL = 3, 4, 5$ (TBM3-L, TBM4-L, TBM5-L), which do not contain the external bitmap and child pointer.

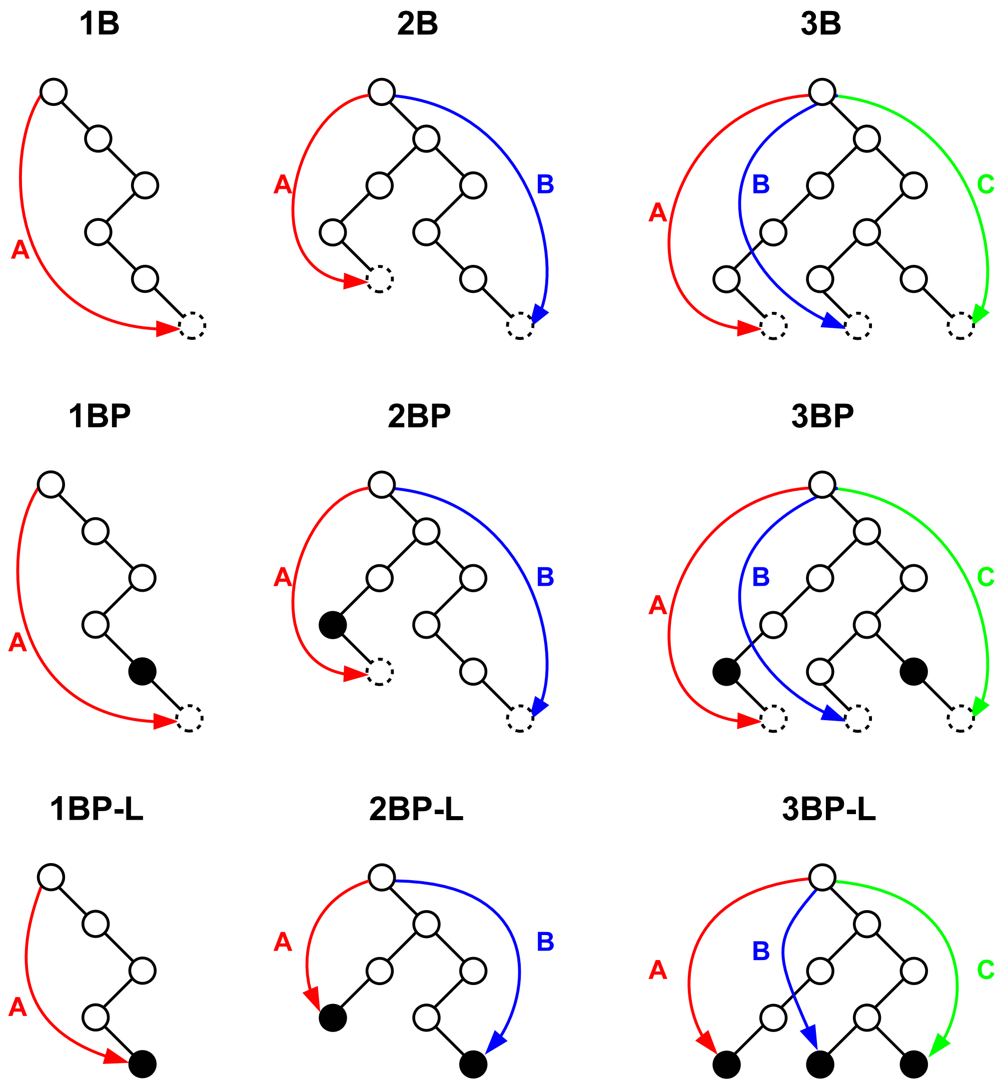


Figure 4.1: Newly proposed nodes.

In order to make a hardware implementation of prefix matching based on the proposed prefix set representation feasible, it is necessary to align the size of each node's representation to some boundary. A smaller boundary implies smaller memory overhead but also a higher number of different node sizes, hence higher utilization of resources required for processing such data structures. Therefore, alignment on both 8-bit and 16-bit boundary is considered further in this chapter. These two options should allow to find a reasonable compromise between memory overhead and resource utilization of the proposed prefix set representation. The properties of both options are evaluated in Section 4.4.

The size of each newly proposed node mainly depends on its maximum allowed branch length, which is the same for all branches of a node. However, as shown in Table 4.6, the value of this parameter differs among nodes as well as between 8-bit and 16-bit alignment of the same node. Apart from the branch length, the size of newly proposed nodes is also influenced by the presence of child and prefix pointers, which is already encoded in the node's name. The child pointer is utilized in every node without the -L suffix in its name and the prefix pointer is present in every node, whose name contains P. When present, the

child pointer is encoded on 23 and 22 bits in case of 8-bit and 16-bit alignment, respectively. In contrast to that, the prefix pointer always uses 19 bits. Such values allow to address all child nodes and prefixes when representing any prefix set from Table 4.1. The size of TBM node's variants is determined by the number of bits in the external and internal bitmaps as well as by the presence of the child pointer, which is utilized in TBM3 only (note that all TBM-based nodes contain the prefix pointer). Both child and prefix pointers used in the variants of a TBM node are of the same size as in newly proposed nodes.

	Size Aligned to 8 bits			**Size Aligned to 16 bits**		
Node Type	**Branch Length**	**Unaligned Size** [bits]	**Aligned Size** [bits]	**Branch Length**	**Unaligned Size** [bits]	**Aligned Size** [bits]
1B	24	56	56	17	48	48
1BP	19	72	72	13	64	64
1BP-L	20	48	48	20	48	48
2B	16	72	72	14	64	64
2BP	10	80	80	11	80	80
2BP-L	12	55	56	15	61	64
3B	11	78	80	12	78	80
3BP	5	80	80	6	80	80
3BP-L	7	53	56	9	62	64
TBM3	3	75	80	3	67	80
TBM3-L	3	30	32	3	30	48
TBM4-L	4	38	40	4	38	48
TBM5-L	5	54	56	5	54	64

Table 4.6: Key characteristics of nodes from the proposed prefix set representation when aligned on the 8-bit and 16-bit boundary.

Along with the maximum allowed branch length for each of thirteen nodes comprising the proposed prefix set representation, Table 4.6 also contains unaligned as well as aligned size of each node for both 8-bit and 16-bit alignment. The difference of these two values represents memory overhead introduced by the alignment. The greatest overhead can be observed for TBM-based nodes, especially when aligned on the 16-bit boundary. On the other hand, newly proposed nodes introduce memory overhead only rarely regardless their alignment. This complies with what was expected because the maximum allowed branch length of newly proposed nodes was selected with respect to minimum memory overhead, while the size of TBM nodes can be tuned by the value of parameter SL only.

The mapping of proposed nodes onto a trie is done according to the algorithm in Figure 4.2, which starts from the root node and continues in breadth-first order towards the leaves of the trie. At each position, the algorithm performs a trial mapping of all proposed types of node and determines the best type for the current position using Equation 4.1, where p is the number of covered prefix nodes, n is the number of all covered trie nodes, and $size$ is the size of a selected node type. Except for standard queue operations ENQUEUE and DEQUEUE, the algorithm also uses the following auxiliary functions.

- MAP_COST — returns the cost of mapping a given type of node at the specified position in the trie according to Equation 4.1.

- MAP — actually performs the mapping of the selected type of node at the specified position in the trie (i.e., it removes a subtree of the trie and replaces it with the selected node).
- CHILDREN — returns a list of child nodes of the specified node.

Input: pointer *root* to the root node of the trie
Output: pointer *root* to the root node of the tree mapped onto the trie

```
1:  Q ← ∅
2:  if root ≠ NULL then
3:      ENQUEUE(Q, root)
4:  end if
5:  while Q ≠ ∅ do
6:      subtrie ← DEQUEUE(Q)
7:      max_cost ← 0
8:      best_type ← NULL
9:      for each type ∈ node_types do
10:         cost ← MAP_COST(subtrie, type)
11:         if cost > max_cost then
12:             max_cost ← cost
13:             best_type ← type
14:         end if
15:     end for
16:     MAP(subtrie, best_type)
17:     for each child ∈ CHILDREN(subtrie) do
18:         ENQUEUE(Q, child)
19:     end for
20: end while
```

Figure 4.2: Pseudocode of an algorithm that maps proposed nodes onto a trie.

$$cost = \begin{cases} \frac{p}{size} & \text{if } p > 0 \\ \\ \frac{n}{size} & \text{otherwise} \end{cases} \tag{4.1}$$

Although the algorithm from Figure 4.2 does not guarantee a globally optimal mapping of proposed nodes onto the trie, it represents a working solution that is locally optimal and has acceptable time complexity.

4.3 Proposed Hardware Architecture

In order to achieve performance required in 100 Gbps networks, prefix matching based on the proposed representation has to be implemented in hardware using a processing pipeline, in which each processing element (PE) performs one step of the matching algorithm. Since the prefix set representation proposed in the previous section can be classified as a multibit trie, the result of prefix matching is available after processing at most n nodes, where n is the height of a tree representing the prefix set; therefore, the pipeline has to consist of

n PEs. Because each of these PEs accesses a memory storing a prefix set representation independently, the memory has to be able to support n parallel memory accesses per clock cycle. To satisfy this requirement, it is convenient to implement the processing pipeline in an FPGA chip, which allows to allocate one or more independent on-chip memory blocks to each PE.

A proposed hardware architecture for prefix matching based on the prefix set representation introduced in the previous section is shown in Figure 4.3. This architecture consists of two processing pipelines with uniform PEs and dual-port memories shared between PEs from corresponding stages. By utilizing the dual-port nature of memory blocks available in an FPGA, the architecture can achieve double the performance of a single-pipeline architecture without the need to compromise on memory accesses. A memory allocated to each pipeline stage comprises two parallel parts, each of which has data width of 80 bits (i.e., the maximum aligned size of a node, as shown in Table 4.6). While the first part of

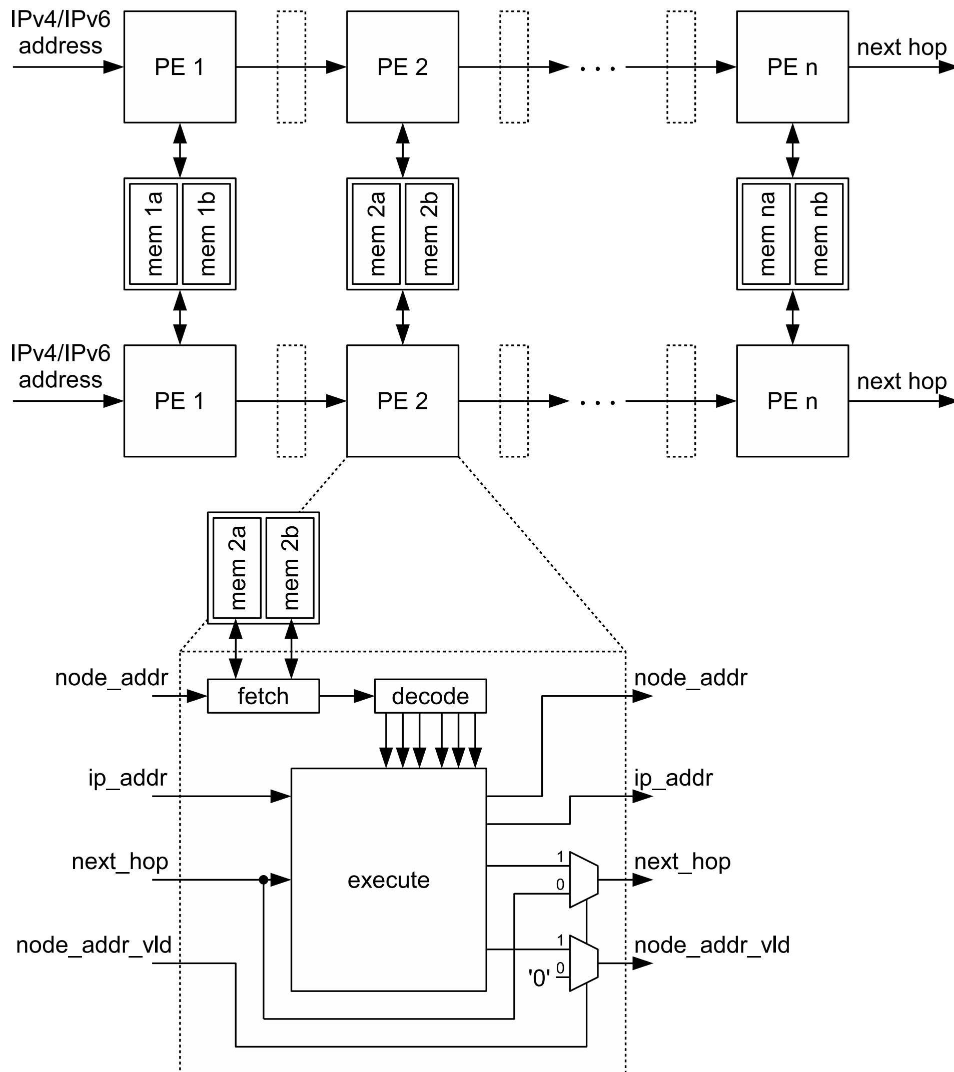


Figure 4.3: The pipelined hardware architecture that implements prefix matching based on the prefix set representation from Section 4.2. The figure also shows a high-level architecture of a PE utilized in employed processing pipelines.

the memory stores data words with an even address, the second part is used for storing data words with an odd address. Such internal organization of the memory makes possible to read the whole node in one clock cycle even if its representation is not aligned on the beginning of a data word, which may cause storing the node in two consecutive data words.

Figure 4.3 also shows a high-level architecture of a PE that implements one step of the prefix matching algorithm. The processing of a prefix set representation's node in the PE follows the *fetch-decode-execute paradigm*, which is known from processing instructions in a CPU. First of all, the PE fetches from the memory the representation of the node, which is subsequently decoded and the obtained values are sent in parallel to the *execute* submodule. This submodule actually executes one step of prefix matching (i.e., it searches for matching prefixes and determines the address of the next node) and sets the PE's outputs accordingly.

Because the *execute* submodule is the most complex part of the PE, Figure 4.4 illustrates its internal structure. The majority of processing within this submodule is performed in *branch A proc*, *branch B proc*, *branch C proc*, and *TBM node proc* blocks. The first three of them are designed for the processing of corresponding branches in newly proposed types of node (recall that branches are marked by letters A, B, and C in Figure 4.1), while the last one is utilized when one of the variants of a standard TBM node is processed. Since the processing of different branches and node types is done in parallel, the *select branch* and *select result* blocks are used to choose correct values for the PE outputs.

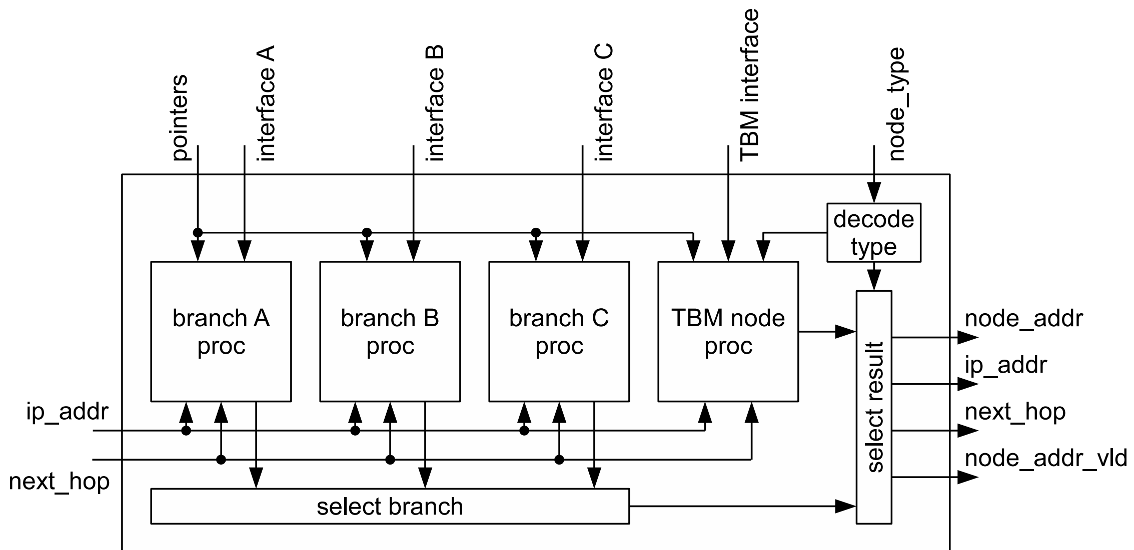


Figure 4.4: An internal structure of the PE's *execute* submodule.

Combinatorial logic of the *fetch* and *execute* submodules of the PE is relatively complex, which limits the maximum frequency they can safely operate on. Therefore, in order to achieve desired matching performance, each of them contains two sets of intra-stage registers. In total, each PE contains four sets of intra-stage registers, which means that the latency of processing a node within the PE is five clock cycles.

In Section 4.2, there are described two variants of node alignment in a memory—on the 8-bit or 16-bit boundary. Both of these variants can be processed using a PE having conceptually the same architecture with only some minor changes in the *fetch*, *decode*, and *execute* submodules. Different node alignment has the greatest influence on data reorder logic in the *fetch* submodule. It also has to be reflected in the *decode* submodule by

different interconnection of decoding logic. Relatively the smallest changes have to be done in the *execute* submodule, where it is sufficient to change data width of some internal interconnections.

4.4 Experimental Evaluation

This section evaluates the prefix set representation proposed in Section 4.2 and the pipelined hardware architecture implementing prefix matching based on the representation, which was designed in Section 4.3. The mapping of the proposed representation's nodes on a trie has been implemented—according to the pseudocode available in Figure 4.2—in the Netbench tool [48]. Thus, it is possible to conduct similar experiments as those performed during the analysis of the prefix set representations utilized in TBM and SST algorithms. On the other hand, the evaluation of the designed architecture is allowed by its implementation on a Xilinx Virtex-6 XC6VSX475T FPGA [11], which has been done using Xilinx ISE 14.3. (Since newer FPGA chip series from Xilinx—such as 7 Series [15], UltraScale, and UltraScale+ [17]—use basically the same set of primitives, but manufactured under more advanced process, the evaluation of the architecture on a newer FPGA chip would probably yield the same results with respect to the number of utilized resources and better results with respect to the maximum frequency.) The prefix set representation as well as the hardware architecture are evaluated for both 8-bit and 16-bit alignment of prefix set representation's nodes.

First of all, memory requirements of the proposed prefix set representation were assessed on IPv4 and IPv6 prefix sets from Table 4.1. The results of the assessment for both variants of nodes' alignment in memory are presented in Table 4.7. As expected, memory requirements are always lower for 8-bit alignment, but a significant difference between 8-bit and 16-bit alignment can be observed only in case of IPv4 prefix sets. This can be explained by a different density of tries representing prefix sets, which is higher for IPv4 than for IPv6 prefix sets. Therefore, IPv4 tries are mainly covered by variants of a TBM node (which introduce the highest memory overhead when aligned on the 16-bit boundary), while tries representing IPv6 prefix sets are mainly covered by newly proposed nodes (which introduce almost the same memory overhead for both 8-bit and 16-bit alignment). Table 4.7 also shows the height of a tree that encodes particular prefix sets using the proposed representation.

The utilization of FPGA resources (both absolute and percentage) and the maximum operating frequency after place & route of the proposed architecture on the target FPGA chip are shown in Table 4.8. As can be seen, the main difference between the architectures supporting 8-bit and 16-bit alignment of nodes lies in the number of utilized LUTs, where lower numbers in case of the 16-bit architecture are mainly due to simpler reorder logic in the *fetch* submodule of the PE. With respect to the number of utilized registers, both variants of the architecture are practically the same. However, the 8-bit architecture slightly overcomes the other variant in the maximum supported operating frequency.

Apart from resource utilization and the maximum operating frequency of a single PE, Table 4.8 also shows these values for a complete processing pipeline and the whole proposed architecture comprising two pipelines. Even though the length of each processing pipeline (23 PEs) makes possible to perform prefix matching using the prefix set represented by the highest tree (real IPv6 set `AS6447`, see Table 4.7), the whole architecture fits into the target FPGA, regardless the alignment of prefix set representation's nodes in memory. Moreover, the resources utilized by both variants of the architecture are significantly lower than the resources available in the target FPGA, which allows to govern the selection of preferred

Prefix Set	Prefixes	Memory [kb] 8-bit Alignment	Memory [kb] 16-bit Alignment	Tree Height
IPv4				
rrc00	332 118	6 330.8	7 287.6	12
IPv4-space	220 779	3 571.4	4 297.4	12
route-views	442 748	7 779.8	9 039.6	12
IPv6				
AS1221	10 518	475.8	489.0	18
AS6447	10 814	493.8	506.6	23
Generated IPv6				
rrc00_ipv6	319 998	21 264.3	21 373.2	21
IPv4-space_ipv6	150 157	10 412.2	10 421.4	18
route-views_ipv6	439 880	29 039.5	29 207.4	20

Table 4.7: Memory requirements and tree height of the proposed representation for prefix sets from Table 4.1.

nodes' alignment mainly by their memory requirements, whose optimization is the main objective in this chapter. Therefore, the representation of a prefix set utilizing nodes aligned on the 8-bit boundary is preferred in further evaluations.

The selected alignment of nodes in memory does not only lead to lower memory requirements, but it can also be processed by a hardware architecture that operates on a little higher frequency, which implies higher matching performance. Each processing pipeline is able to provide one matching result per clock cycle, thus the total matching performance of the whole architecture is almost 255 MLPS, translating into throughput of 170 Gbps for the shortest Ethernet frames. Operating frequency and the number of pipeline stages also together determine the overall latency of the proposed architecture. Because each PE consists of five pipeline stages, the whole pipeline contains $5 \cdot 23 = 115$ stages. Since processing in one stage takes approximately 7.86 ns, the overall latency of processing in the full pipeline is 903.90 ns. Overall latency also dictates the size of a buffer for packets that wait for the result of prefix matching in the proposed architecture. To support throughput of 170 Gbps, the capacity of the packet buffer has to be at least 18.75 kB. Nevertheless, this buffer can be implemented in an external memory in order to save precious memory resources available on an FPGA chip.

The comparison of the proposed prefix set representation with the TBM and SST algorithms in terms of memory requirements is provided in Table 4.9 that shows memory required for the proposed representation and the percentage of memory it saves when compared to TBM and SST. The proposed representation overcomes both TBM and SST, but the amount of saved memory is higher for TBM (between 34.67 % and 76.31 %) than for SST (between 8.65 % and 19.98 %), which was designed with the aim of minimizing memory requirements. Moreover, it is shown that sparse prefix trees of IPv6 prefix sets allow to save more memory because they are well suited for the utilization of memory-efficient newly proposed nodes introduced in Figure 4.1. It is also clear that all prefix sets from Table 4.1 can be stored in an on-chip memory available on the target FPGA, when encoded using the proposed representation.

8-bit Alignment	**LUTs (% of All)**	**Registers (% of All)**	**Frequency [MHz]**
1 PE	3 647 (1.23 %)	1 825 (0.31 %)	127.162
1 pipeline (23 PEs)	83 881 (28.19 %)	41 957 (7.05 %)	127.162
2 pipelines (46 PEs)	167 762 (56.37 %)	83 950 (14.11 %)	127.162
16-bit Alignment	**LUTs (% of All)**	**Registers (% of All)**	**Frequency [MHz]**
1 PE	3 194 (1.07 %)	1 817 (0.31 %)	123.183
1 pipeline (23 PEs)	73 462 (24.69 %)	41 791 (7.02 %)	123.183
2 pipelines (46 PEs)	146 924 (49.37 %)	83 582 (14.04 %)	123.183

Table 4.8: Resource utilization and the maximum frequency of the proposed pipelined hardware architecture after place & route on Xilinx Virtex-6 XC6VSX475T using Xilinx ISE 14.3.

In order to compare the memory efficiency of the proposed prefix set representation also with the PPLA algorithm, Table 4.10 provides a memory efficiency ratio (i.e., the number of bytes required for storing one byte of a prefix) of the proposed representation when encoding prefix sets from Table 4.1. In addition, the table also shows the value of this parameter for TBM and SST in configurations that require the minimum memory for representing particular prefix sets. According to [39], the average memory efficiency ratio of PPLA on generated IPv6 prefix sets is 0.90 when prefixes are encoded using a set of binary search trees. Therefore, the proposed representation is slightly worse than PPLA on generated IPv6 prefix sets. However, it is significantly better on IPv4 prefix sets, for which [39] reports the average memory efficiency ratio of 1.00. Additionally, both TBM and SST, which were not taken into account in [39], achieve a better memory efficiency ratio than PPLA on IPv4 prefix sets. The memory efficiency of the proposed representation on real IPv6 prefix sets cannot be compared with PPLA, because PPLA was not evaluated on this type of data.

Since both the proposed prefix set representation and TBM are based on a trie, in which multiple prefixes share the same path through a prefix tree up to some level, they should exhibit a better memory efficiency ratio on large prefix sets (e.g., generated IPv6) then on small prefix sets (e.g., real IPv6). Nevertheless, according to Table 4.10, this is not the case on real and generated IPv6 prefix sets used in the performed evaluation. The most probable explanation of this unexpected situation is that the utilized Non-random Generator [66] does not model the process of IPv6 address allocation correctly, as discussed in Section 3.3.1. Although there are other IPv6 prefix set generators (e.g., V6Gene [67]), the Non-random Generator was used in order to fairly compare the results of the performed evaluation with results presented in [39].

Prefix Set	Prefixes	Memory [kb]	Saved Memory	
IPv4		**Proposed**	**TBM (*SL=5*)**	**SST (*K=32*)**
`rrc00`	332 118	6 330.8	34.67 %	8.65 %
`IPv4-space`	220 779	3 571.4	37.37 %	12.49 %
`route-views`	442 748	7 779.8	34.85 %	11.34 %
IPv6		**Proposed**	**TBM (*SL=3*)**	**SST (*K=32*)**
`AS1221`	10 518	475.8	55.82 %	19.16 %
`AS6447`	10 814	493.8	56.11 %	19.98 %
Generated IPv6		**Proposed**	**TBM (*SL=4*)**	**SST**
`rrc00_ipv6`	319 998	21 264.3	75.63 %	N/A
`IPv4-space_ipv6`	150 157	10 412.2	76.31 %	N/A
`route-views_ipv6`	439 880	29 039.5	75.57 %	N/A

Table 4.9: Memory requirements of the proposed prefix set representation and the percentage of memory it saves when compared to the TBM and SST algorithms on prefix sets from Table 4.1.

Prefix Set	Prefixes	Memory Efficiency Ratio		
IPv4		**Proposed**	**TBM (*SL=5*)**	**SST (*K=32*)**
`rrc00`	332 118	0.610	0.934	0.668
`IPv4-space`	220 779	0.518	0.826	0.592
`route-views`	442 748	0.562	0.863	0.634
IPv6		**Proposed**	**TBM (*SL=3*)**	**SST (*K=32*)**
`AS1221`	10 518	0.724	1.638	0.895
`AS6447`	10 814	0.731	1.665	0.913
Generated IPv6		**Proposed**	**TBM (*SL=4*)**	**SST (*K=32*)**
`rrc00_ipv6`	319 998	1.063	4.363	N/A
`IPv4-space_ipv6`	150 157	1.109	4.684	N/A
`route-views_ipv6`	439 880	1.056	4.324	N/A

Table 4.10: A memory efficiency ratio (bytes of memory/bytes of prefix) of the proposed prefix set representation, TBM, and SST for prefix sets from Table 4.1.

4.5 Summary

This chapter addresses the performance of packet classification algorithms via improvements of the prefix matching operation, which is utilized in the majority of packet classification algorithms. To this end, it introduces a novel representation of a prefix set along with a pipelined hardware architecture implementing prefix matching based on this representation in an FPGA chip. In order to fit into a limited amount of on-chip memory blocks available on an FPGA, the prefix set representation was designed with the primary aim to minimize its memory requirements.

The introduced multibit trie representation of a prefix set was designed in accordance with the results of analysis focused on the representation of various prefix sets using existing multibit trie approaches. The analysis based on real IPv4/IPv6 prefix sets from forwarding tables of core routers and artificial IPv6 prefix sets generated using the Non-random Generator [66] revealed that regardless the type of a prefix set, the majority of multibit

trie nodes can be classified as either leaf nodes or internal non-prefix nodes. Therefore, the proposed representation comprises thirteen different types of node—nine newly proposed nodes illustrated in Figure 4.1 and four variants of a standard TBM node. While the newly proposed nodes provide means for a highly memory-efficient representation of the most common trie's subtrees, the variants of a TBM node ensure the completeness and memory efficiency of the representation also in less common situations.

According to the performed evaluation of memory efficiency, the proposed prefix set representation is much more efficient than TBM and it also overcomes the SST algorithm. Moreover, when compared with PPLA, the representation achieves significantly better memory efficiency on IPv4 prefix sets and only slightly worse results on generated IPv6 prefix sets. However, most importantly, because of its memory efficiency the proposed representation allows to store all evaluated prefix sets in an on-chip memory available on the target FPGA chip (Xilinx Virtex-6 XC6VSX475T).

The proposed representation of a prefix set is suitable for processing in hardware, which is demonstrated by the pipelined hardware architecture introduced in Figure 4.3 that implements prefix matching based on this representation in the target FPGA. The architecture consists of two processing pipelines, whose corresponding stages share a common memory comprising several dual-port on-chip memory blocks. The processing element of each stage is able to process any of the thirteen proposed types of node, thus it can perform a single step of prefix matching regardless the bit-length of an input value. As shown in the performed evaluation, the whole architecture, which is able to perform prefix matching using any of the considered prefix sets, easily fits into the target FPGA and its matching performance is almost 255 MLPS, translating into throughput of 170 Gbps for the shortest Ethernet frames.

Chapter 5

Addressing Algorithms Benchmarking

Because the requirements on the performance of packet classification algorithms are continuously increasing, the development of classification algorithms that meet these requirements is still an active process. To verify that newly developed algorithms fulfill given requirements, they have to be benchmarked using a set of classification rules. However, since many packet classification algorithms leverage inherent properties of real classification rule sets to improve their performance [60], benchmarking using an arbitrary set of classification rules would not provide valid results. Therefore, to correctly benchmark a packet classification algorithm, a real rule set has to be used. The problem of this requirement is that real rule sets are not publicly available for the majority of packet classification use cases, mostly because of security reasons.

The lack of publicly available rule sets for benchmarking has been mitigated by a number of synthetic rule set generators [66, 67, 63, 59, 29] developed by the researchers. These tools can generate artificial rule sets that have similar inherent properties as real sets and comprise almost all types of classification rules considered in this thesis (see Table 2.1). Nevertheless, FRuG [29]—the only tool that allows to generate rules specifying a condition for more than five header fields—does not explicitly provide support for OpenFlow rules generation. In addition, none of the tools provides the flexibility to generate all types of classification rules on its own and even the generation of a single rule type is not always accurate. These issues show that packet classification benchmarking is still an open problem and there is space for improvements of synthetic rule set generators.

This chapter addresses the issues of packet classification benchmarking by introducing ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow 1.0.0 rule sets (i.e., all types of rules described in Table 2.1). Its generation process is based on an input seed that specify statistical properties of all header fields, for which the matching conditions are to be generated. Therefore, to make the ClassBench-ng output rule set as close as possible to a real classification rule sets, it is important to ensure that such a seed contains properties that precisely reflect the current trends. The chapter thus presents also the analysis of real IPv4 and IPv6 prefix sets as well as OpenFlow 1.0.0 flow tables taken from an operational environment. Finally, to make the proposed tool attractive in the long term and for a wide number of different use cases, ClassBench-ng offers a mechanic to create input seeds from real rule sets. To further increase a potential impact of this tool on the research community, its repository is aimed to be used as a place

where researchers and operators can continuously upload new seeds that match a number of different environments or use cases (e.g., datacenter, Internet Service Provider, and Internet eXchange Point).

ClassBench-ng was published in [41], which represents a basis for the rest of this chapter that is organized as follows. First of all, Section 5.1 introduces challenges that are connected with synthetic rule set generation and discusses how they are addressed in existing generators as well as in ClassBench-ng. Then, in order to understand statistical properties of current classification rule sets, Section 5.2 provides an analysis of real IPv4, IPv6, and OpenFlow data sets. Next, Section 5.3 describes ClassBench-ng, which was designed with the results of the performed analysis in mind, while Section 5.4 is devoted to its experimental evaluation on IPv4/IPv6 prefixes and OpenFlow rules generation. Finally, the summary of this chapter is provided in Section 5.5.

5.1 Challenges in Rule Set Generation

Since packet classification algorithms cannot be benchmarked using arbitrarily generated classification rule sets, synthetic rule set generators have to utilize an input that controls the process of output rule set's generation. For instance in 1-dimensional generators, the generation of an IPv6 prefix set is based on real sets of IPv4 or IPv6 prefixes, which are available and can be easily transformed or extended to the output prefix set. On the other hand, multi-dimensional generators utilize input seeds and parameter files that specify statistical properties the output rule set should meet. While existing 1-dimensional generators are sufficient for benchmarking prefix matching algorithms, multi-dimensional generators suffer from several issues (e.g., none of them explicitly supports OpenFlow rules nor allows to generate all types of classification rules) that limit their usability. Therefore, this chapter addresses the issues of multi-dimensional generators.

The content of seeds utilized in existing multi-dimensional classification rule set generators is of various origin. In case of ClassBench [63] and ClassBenchv6 [59], a seed contains statistical distributions extracted from a real rule set. In contrast to this, rule set generation in FRuG [29] is based on a parameter file with user-defined characteristics of a target rule set. While the former solution results to be more accurate when an output as close as possible to a real rule set is required, the latter is (potentially) more flexible in the long term. Indeed, continuous innovation, a desirable property of the Internet, may change the statistical properties of classification rule sets and make a tool obsolete, if the input seeds are not updated accordingly. Because both accuracy and flexibility are important properties of rule set generators, combining a generator that relies on statistical properties specified in a seed together with an analyzer able to extract such properties from a real rule set is the most sensible way to ensure the longevity of the toolkit.

To be usable in a rule set generator, a seed has to meet the following properties.

- **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 various sizes.

The first property makes the redistribution of seeds that match a number of different environments or use cases easier, while keeping the output rule set as close as possible to a

real one. The remaining two properties ensure that the process of rule set generation may end successfully, if sufficient time and memory are provided.

ClassBench is the only publicly available multi-dimensional rule set generator that utilizes seeds extracted from real rule sets. In addition, these seeds meet all three above mentioned properties, allowing for the generation of a given number of output rules that match input distributions. The definition of output rule set's properties using statistical distributions enables the first property: the anonymity is guaranteed as no real classification rules are stored in a seed, thus avoiding the use of sensitive information. The completeness of the seed is proven by ClassBench itself, which has been actively used by the researchers for over a decade. Finally, similarly to the first property, the scalability is enabled by the definition of desired output rule set's properties using statistical distributions. Clearly, the more entries need to be generated, the easier they will meet the specified statistical properties.

As ClassBench seeds meet all three required properties and their content is extracted from real rule sets, they have the potential to be used also in ClassBench-ng, which proposes a tight integration of rule set generation and analysis. However, the structure of ClassBench seeds was designed ten years ago and it does not provide a space for defining statistical distributions that would cover the specific properties of IPv6 5-tuples and OpenFlow rules. Therefore, in order to allow addressing these issues, the following section presents an analysis of current real classification rule sets. Given Taylor and Turner have shown a complete analysis of real IPv4-based classification rules in [63], the following section provides a comparison of the same type of classification rules after ten years of Internet's evolution. Moreover, it also investigates changes in IPv6-based rule sets during the last decade and the current properties of OpenFlow 1.0.0 flow tables taken from an operational environment. The insights from this analysis will be used for designing the structure of seeds utilized in ClassBench-ng and they will also help to properly design a rule set analyzer able to extract statistical properties from real rule sets.

5.2 Analysis of Real Rule Sets

The analysis of IPv4, IPv6, and OpenFlow 1.0.0 rule sets taken from operational environments, which is presented in this section, represents the first step towards an accurate and flexible toolkit that allows to extract statistical properties from real rule sets into a seed and utilize this seed for the generation of synthetic rule sets. Data sets used in the analysis along with their main properties are summarized in Table 5.1. Both IPv4 and IPv6 prefix sets have been taken from core routers, while the used sets of 5-tuples originate from ACLs (*Access Control Lists*) applied at a university network's perimeter. The analysis of OpenFlow rules is based on data sets from Open vSwitches [4] running in a datacenter environment.

5.2.1 IP Prefixes

The most common data structure for the representation of an IP prefix set is a trie, i.e., a binary prefix tree. As shown in Section 3.4.1, a trie is characterized by the following four parameters in ClassBench seeds:

- prefix length distribution,
- branching probability distribution,

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 a university network	2010-08-30
uni_2015	122		2015-01-14
		OpenFlow Rule Sets	
of1	16 889	Open vSwitches 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

Table 5.1: Real rule sets used in the analysis. OpenFlow set of3 exists in several instances, one for each day in the given interval.

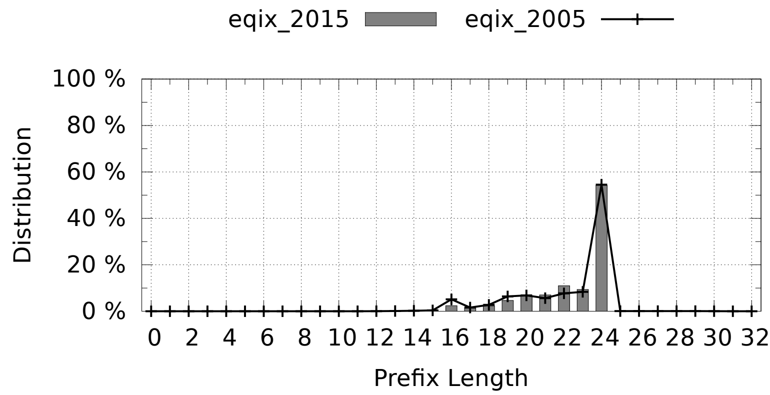
- average skew distribution, and
- prefix nesting threshold.

Since ClassBench seeds can potentially form a basis of seeds utilized in ClassBench-ng, the same set of trie-related parameters was used also in the analysis of available IPv4 and IPv6 prefixes, the results of which are presented in this section.

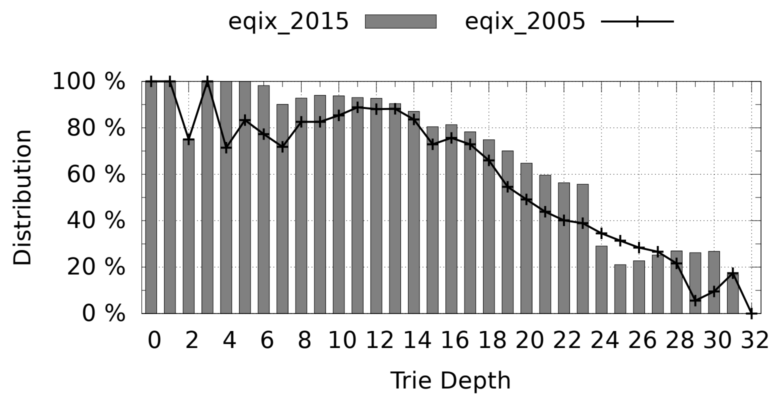
IPv4

Figures 5.1 compare the same IPv4 prefix set (eqix) in ten years time. While the prefix length distribution was almost the same between years 2005 and 2015 (Figure 5.1a), the number of two-children nodes in the trie was increasing (Figure 5.1b) and the average skew was decreasing (Figure 5.1c). The prefix nesting threshold, which is not shown in the figures, remained unchanged between 2005 and 2015. The same results were also confirmed in prefix set rrc00.

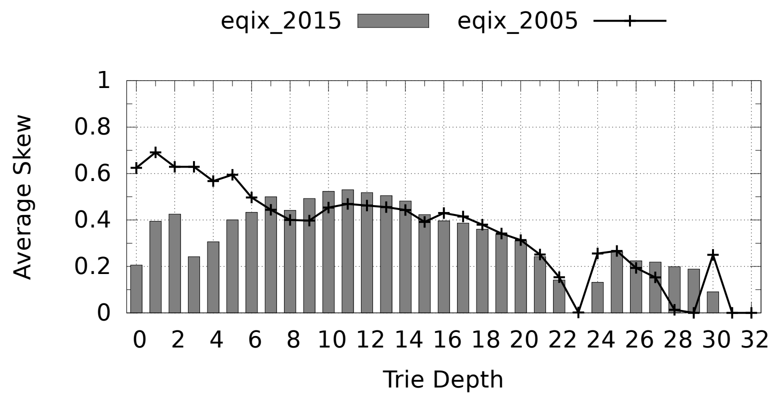
A growing number of two-children nodes and their smaller skew correlates with more than 3-times higher number of prefixes after ten years, as shown in Table 5.1. Nevertheless, the branching probability and average skew distributions of the prefix set from year 2015 follow similar trends as in 2005 and although the prefix set grew in size, the prefix length distribution is the same. These results are aligned with the path towards the saturation of the IPv4 address space [14].



(a) Prefix length distribution.



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

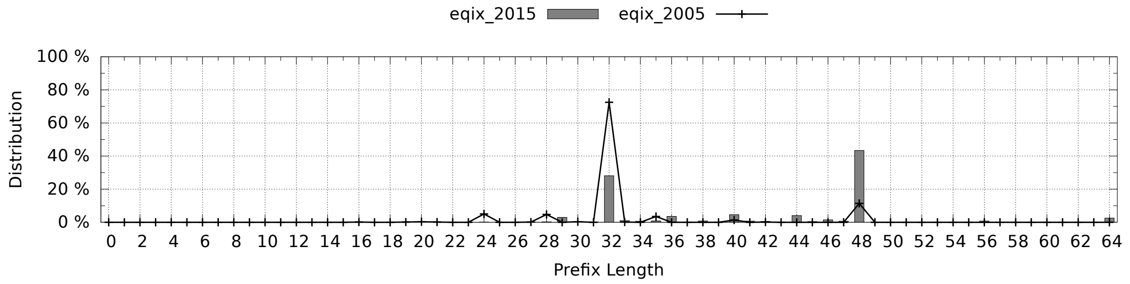


(c) Average skew distribution.

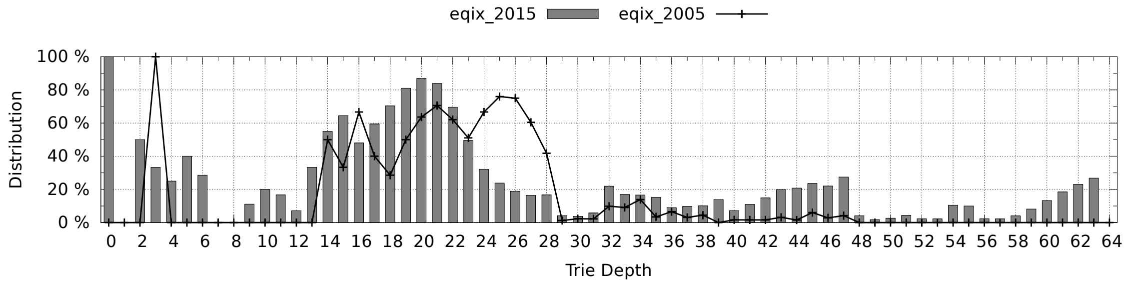
Figure 5.1: A comparison of `eqix` IPv4 prefix sets from years 2005 and 2015.

IPv6

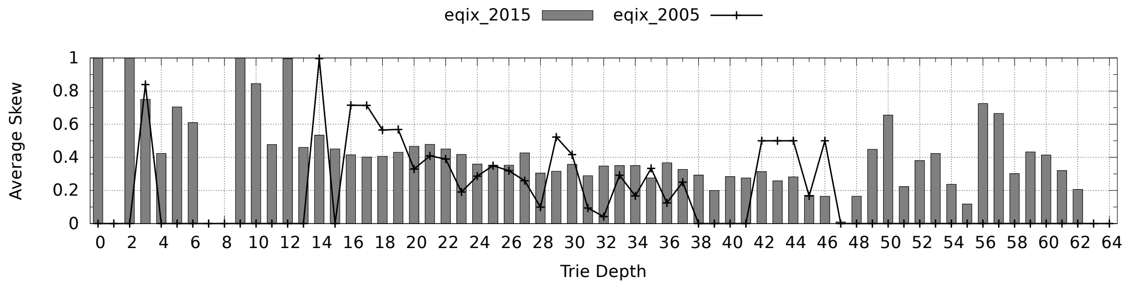
Figures 5.2 compare the selected parameters on the `eqix` IPv6 prefix sets from years 2005 and 2015. These figures display only the first 64 trie levels because there were no IPv6 prefixes longer than 48 bits in 2005. Figure 5.2a shows that the prefix length distribution changed significantly during the monitored interval. While prefix length 32 dominated the distribution in 2005, ten years later the most common prefix length was 48. This affected



(a) Prefix length distribution.



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



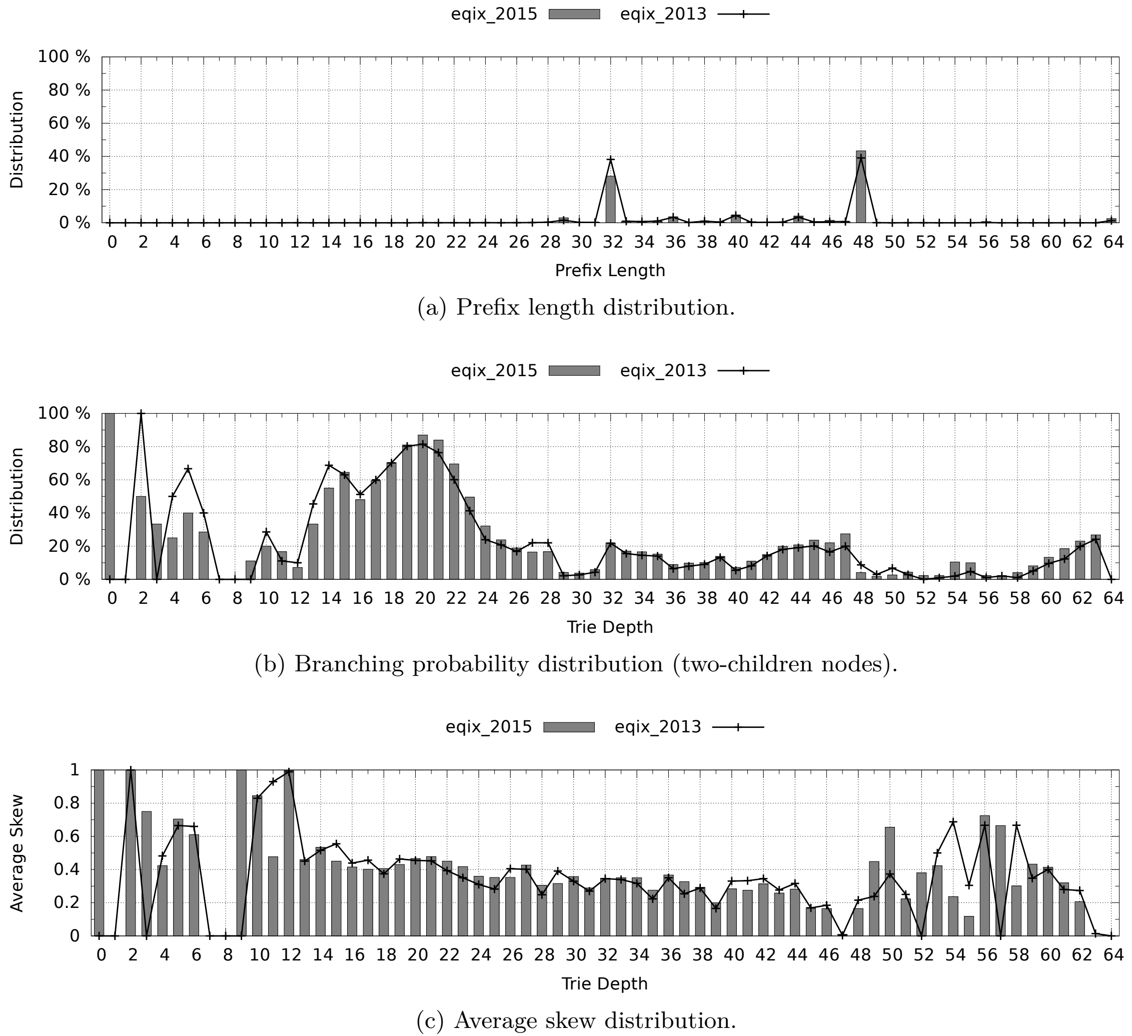
(c) Average skew distribution.

Figure 5.2: A comparison of `eqix` IPv6 prefix sets from years 2005 and 2015.

both the branching probability distribution (Figure 5.2b) and the average skew distribution (Figure 5.2c).

Even though the prefix set contained 36-times more prefixes after ten years of evolution (see Table 5.1), the most probable reason for such a great difference in the prefix length distribution is a steady growth of IPv6 deployments over the monitored decade. In 2005, most of the allocated prefixes belonged to ISPs/RIRs, while after ten years prefixes belonging to end users (organizations) became dominant [2]. In addition, the changes of the branching probability and average skew distributions between 2005 and 2015 were also caused by the emergence of prefixes longer than 64 bits. Prefix set `rrc00` shows similar behavior, except for the prefix nesting threshold, which remained unchanged.

In 2005, both the `eqix` and `rrc00` prefix sets contained only a few hundreds of IPv6 prefixes, while in 2015 there were more than 23 thousands of prefixes in both sets (see Table 5.1). This is a clear demonstration of the steady growth of IPv6 deployments around the world. In this context, great changes of the parameter distributions (i.e., branching probability and average skew) are not surprising. However, when compared over a shorter span (e.g., between 2013 and 2015, where the prefix length distribution is almost stable), the



(a) Prefix length distribution.

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

(c) Average skew distribution.

Figure 5.3: A comparison of `eqix` IPv6 prefix sets from years 2013 and 2015.

values of the branching probability and average skew distributions follow similar trends, as shown in Figures 5.3. Note that the number of IPv6 prefixes in the `eqix` set almost doubled between 2013 and 2015.

5.2.2 Ports and Protocol

The following analysis was performed using rule sets taken from ACLs in a university network presented in Table 5.1. The data span over a period of five years to enable a comparative analysis over the time.

The first part of the analysis focused on the distribution of rules over protocol values and its results are summarized in Table 5.2. The results show an increasing number of rules specifying a wildcard or UDP, while the number of rules specifying TCP is decreasing. The ICMP protocol was not specified at all in the available rule sets.

Next, Table 5.3 presents the distribution of rules over port classes proposed in ClassBench, separately for source and destination port fields. The source port field of all rules contained only the wildcard specification in 2010 as well as in 2015, which is aligned with

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

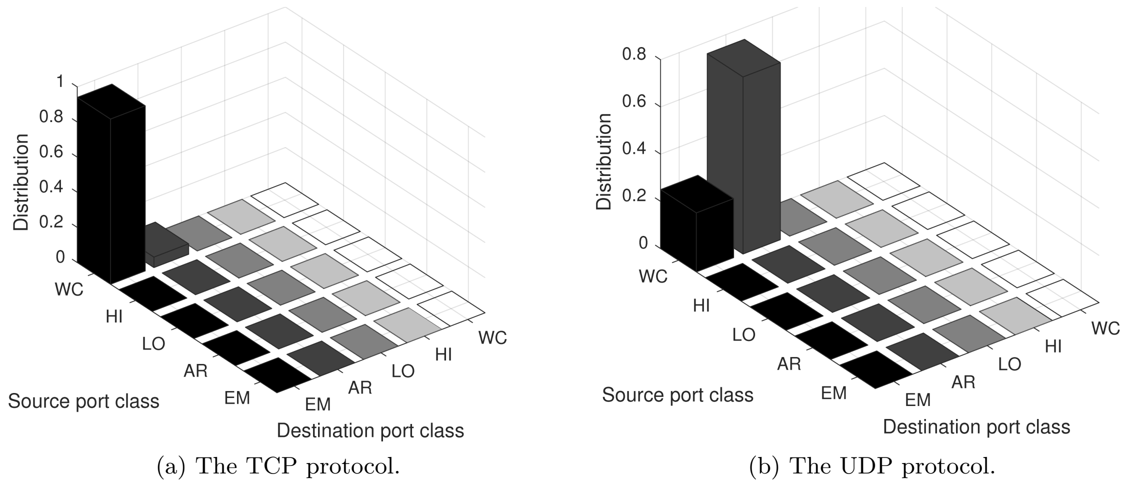
Table 5.2: The distribution of rules over protocol values.

observations regarding ACL rule sets made by Taylor and Turner [61]. In contrast, the distribution of rules over destination port classes was different between these years. Especially the number of arbitrary range and wildcard entries increased at the expenses of exact match ones.

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 %

Table 5.3: The distribution of rules over port classes.

Finally, the distribution of rules over combined source-destination PPCs was analyzed. Figures 5.4 are based on the uni_2015 data set and they show PPC matrices for subsets of rules that specify the TCP and UDP protocols. The most common class pair being adopted in the TCP case (Figure 5.4a) is WC-EM, which represents rules specifying a wildcard for the source port and an exact value for the destination port. On the other hand, the UDP case (Figure 5.4b) shows a great utilization of the WC-AR class pair.



(a) The TCP protocol.

(b) The UDP protocol.

Figure 5.4: PPC matrices for various transport layer protocols (rule set uni_2015).

The analysis reported in this section shows that a condition for the protocol header field is commonly defined using the wildcard specification or an exact value corresponding to the TCP protocol. There is indeed still more interest towards connection-oriented applications, but the utilization of connectionless applications is rising, at least according to a growing number of classification rules matching the UDP protocol. The rising of new applications and a massive usage of the RTP protocol-based solutions, both of which utilize UDP as a transport layer protocol, have led to specifically designed classification rules. While rules specifying TCP define almost exclusively an exact value for the destination port (in `uni_2015` this value refers mostly to the SMTP protocol), in case of UDP the majority of rules specify a condition for the destination port using an arbitrary range, which allows to keep the size of a rule set reasonable even when the number of applications is growing.

5.2.3 OpenFlow

Despite the OpenFlow protocol is getting more and more interest in production networks and the number of deployments around the world is rising, it is difficult to find publicly available real rule sets. This section provides an analysis of real OpenFlow rule sets taken from a datacenter in operation. The analysis focuses on understanding the statistical properties of OpenFlow-based rule sets as well as their temporal behavior. While the statistical properties are studied on header fields distribution and their composition into various rule types, with respect to temporal behavior this section analyzes rule set dynamics.

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 [7]. For each of twelve OpenFlow header fields, Figure 5.5 shows its distribution over the specified and wildcarded classes in rules from sets `of1` and `of2`, which were introduced in Table 5.1. The fields of the standard 5-tuple present a non-wildcard value in at least 20 % of rules, while other fields (except for *mac_dst* and *eth_type*) show a great predominance of wildcard entries. Moreover, header fields *vlan_id*, *vlan_prio*, and *ip_tos* are never specified, although it is clear that in this case a network configuration plays a key role (i.e., virtual LANs are not enabled).

Tables 5.4 contain a per-field count of unique values being used in rule sets `of1` and `of2` (e.g., *eth_type* presents just the value `0x0800`, which refers to the IPv4 protocol). For each header field and rule set, the tables also show the *uniqueness factor*, which estimates per-field variance among the rules of that set. For instance, a value close to zero suggests little variance (i.e., rules specifying that field tend to use the same value every time), while a value close to one suggests the exact opposite. The *uniqueness factor* shows an interesting property of the `of1` rule set. Although the *mac_dst* field has the highest number of unique values, the highest *uniqueness factor* can be observed for the *in_port* field. Therefore, the rules of this set that specify a value for *in_port* can be called physical-port-oriented because the value of *in_port* represents the most important part of these rules. Similarly, the rules of the `of2` rule set that specify a value for the *l4_dst* field can be called application-oriented.

Figure 5.6 shows the prefix length distribution for the *ip_src* field in rule set `of1`. The most common prefix lengths are 0 (a wildcard), 10, and 32 (an exact value). Similar trends can also be observed for the *ip_dst* field of the `of1` rule set and both IP address fields of rules belonging to the `of2` set. Even though there are great differences between the presented prefix length distribution and the one from Figure 5.1a, they can be justified considering the nature of OpenFlow rules, which are not dictated by any routing protocol, unless a

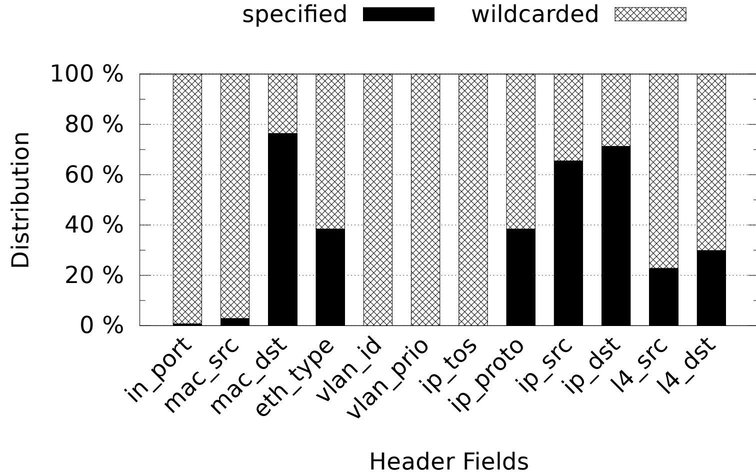


Figure 5.5: A per-field distribution of rules from the combined `of1+of2` rule set over the specified and wildcarded classes.

Rule Set	*ip_proto*	*ip_src*	*ip_dst*	*l4_src*	*l4_dst*
`of1`	3 (0.3 %)	478 (4.6 %)	109 (0.9 %)	4 (2.9 %)	48 (2.2 %)
`of2`	3 (0.1 %)	390 (2.8 %)	97 (0.7 %)	4 (<0.1 %)	8227 (92.7 %)
`of1+of2`	3 (<0.1 %)	498 (2.0 %)	119 (0.4 %)	6 (0.1 %)	8237 (74.2 %)

(a) 5-tuple header fields.

Rule Set	*in_port*	*mac_src*	*mac_dst*	*eth_type*
`of1`	123 (86.6 %)	27 (3.2 %)	593 (4.7 %)	1 (<0.1 %)
`of2`	140 (86.4 %)	19 (8.1 %)	791 (5.0 %)	1 (<0.1 %)
`of1+of2`	182 (59.9 %)	45 (4.2 %)	1176 (4.1 %)	1 (<0.1 %)

(b) OpenFlow-specific header fields (only the specified ones).

Table 5.4: A per-field count of unique values and associated *uniqueness factor* (in parenthesis).

given daemon is running on the top of the controller. In addition, a different environment (a core router for the previous study and a datacenter for this one) also plays an important role.

A further analysis of rule sets `of1` and `of2` revealed that the TCP protocol is specified only in 14.03 % of rules, while 10.59 % of rules specify the ICMP protocol. However, the analysis also showed that the distribution of source and destination port values over five port classes as well as the distribution of source-destination port pair values over twenty-five PPCs are similar to what was shown in case of ACL rule sets from a university network.

Rule Types

Apart from studying individual header fields distribution, the analysis of real OpenFlow rules also focuses on fields dependency. The relationship among header fields was studied using a *rule type* that represents a template indicating which header fields are specified and which are wildcarded. To simplify its representation, each *rule type* has been associated

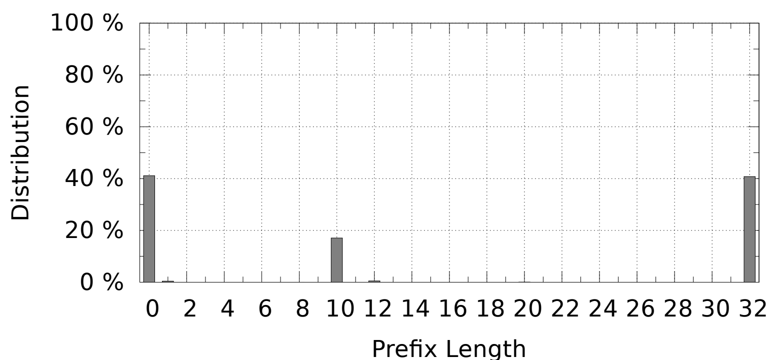


Figure 5.6: The prefix length distribution of source prefixes from the `of1` rule set.

to a 12-bit *rule type number*, in which each bit corresponds to a given header field. A bit set to 1 stands for a specified field and 0 stands for a wildcard. While it is clear that the *rule type number* 0 refers to the combination of all header fields with a wildcard and 4 095 refers to a fully specified rule, it is important to define the bit-field correlation to correctly decode a given *rule type number*. The correlation of header fields to individual bits of the *rule type number* is presented in Table 5.5. This table also displays a binary encoding of *rule type* 796, which clearly shows that rules of this type specify a non-wildcard condition for *mac_dst*, *eth_type*, *ip_proto*, *ip_src*, and *ip_dst* header fields and a wildcard for the remaining fields.

Header Field	*in_port*	*mac_src*	*mac_dst*	*eth_type*	*vlan_id*	*vlan_prio*	*ip_tos*	*ip_proto*	*ip_src*	*ip_dst*	*l4_src*	*l4_dst*
Rule Type 796	0	0	1	1	0	0	0	1	1	1	0	0
	MSB											LSB

Table 5.5: The correlation of header fields to the bits of a *rule type number* illustrated on the *rule type* 796.

Despite there are 4 096 possible *rule types*, the number of *rule types* being used is much lower. In practice, Figure 5.7 shows that the `of1` and `of2` sets contain rules of 18 types only. Moreover, just 6 *rule types* are used by more than 5 % of all rules, while another 12 *rule types* are used by only a few rules.

Using the results of the *rule types* analysis, it can be shown that header field *eth_type* is redundant in the `of1` and `of2` rule sets. Clearly, it is defined only in the rules of types 788, 789, 796, 1304, and 1305 (see Figure 5.7), which also define the *ip_proto* field. Because this header field can only appear in IPv4 packets, matching on the *eth_type* value `0x0800` is redundant (note that according to Table 5.4b, only one unique value is specified for *eth_type* in the combined `of1+of2` rule set). Therefore, *mac_dst* is the only non-redundant OpenFlow-specific header field that is specified in all the most common *rule types*.

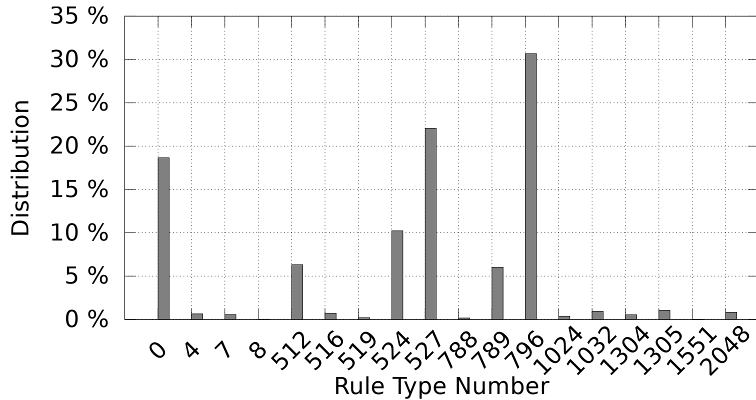


Figure 5.7: The distribution of rules from the combined `of1+of2` rule set over *rule types*.

Dynamics

OpenFlow rule set dynamics was studied on the `of3` set, which originates from a datacenter with 220 physical hypervisors. Each instance of the `of3` rule set represents a flow table snapshot taken from the same hypervisor every day at the same time. In the performed analysis, the dynamics of a rule set was represented by the rate of changes R of rule set S between the previous day $d-1$ and current day d, which is defined in Equation 5.1, where $\triangle$ denotes symmetric difference.

$$R = \frac{|S_{d-1} \triangle S_d|}{|S_{d-1} \cup S_d|} \tag{5.1}$$

Figure 5.8 shows the dynamics of rule set `of3` over a two-week period. While the rate of changes was stable in June (not shown) and for the first week of July, it presented a spike on 7th July 2015. Such behavior can be justified using Figure 5.9, which displays the size of the `of3` set during the monitored interval (plus one day before it because of the rate of changes definition). From this figure it is clear that on 7th July the number of rules decreased drastically, which caused the spike. Since flow changes in the `of3` set may be caused by users creating/deleting VMs (*Virtual Machines*) or updating their security profiles, the observed spike can be attributed to the specific environment the `of3` rule set originates from rather than the OpenFlow protocol itself.

5.3 ClassBench-ng: Next Generation ClassBench

The ClassBench-ng toolkit tightly integrates rule set generation and analysis in order to allow accurate as well as flexible generation of IPv4, IPv6, and OpenFlow rule sets. It also defines the structure of a seed, which stores the results of rule set analysis and serves as an input to the rule set generation process. To meet the required properties specified in Section 5.1 (i.e., anonymity, completeness, and scalability), the ClassBench-ng seed contains several statistical distributions that allow to completely characterize all considered types of generated rule sets in an anonymous and scalable way.

Since original ClassBench already defines seed's structure for the IPv4 case and provides a rule set generator that accepts such seeds, ClassBench-ng utilizes these components and supplements them with a rule set analyzer, which is not publicly available, although it has

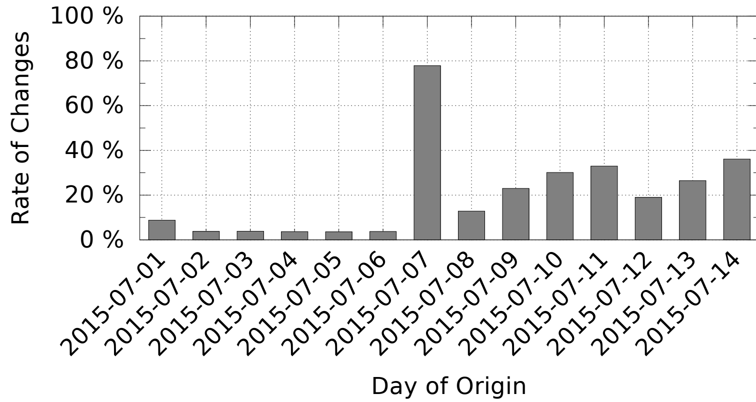


Figure 5.8: The rate of changes of rule set `of3` between 1st and 14th July 2015.

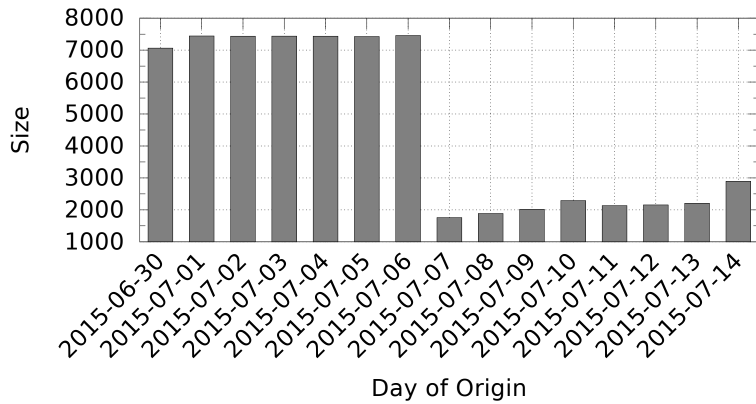


Figure 5.9: The size of rule set `of3` between 30th June and 14th July 2015.

been presented in [63]. As shown in Section 5.2, using parts of ClassBench is valid even though they were designed more than 10 years ago and the Internet has changed significantly since that time. Indeed, the results of the performed analysis demonstrate that the value of IPv4 prefix set parameters has not almost changed and the expected changes were correctly reflected. This is also the case for parameters related to protocol and ports. Moreover, it has been shown in Section 5.1 that seeds of original ClassBench meet all three required properties.

To support the analysis and generation of IPv6 and OpenFlow rules, ClassBench-ng extends seed's structure and also both IPv4 rule set analyzer and generator. Fortunately, as shown in the analysis, the IPv4 prefix set parameters defined in original ClassBench are also able to catch the dynamics of the current IPv6 ecosystem, thus the support of IPv6 can be added by just extending their distributions (129 instead of 33 levels of a trie). On the other hand, OpenFlow support has to be added from scratch using the distribution for OpenFlow *rule types* and separate statistical distributions for OpenFlow-specific header fields, both of which represent important characteristics of OpenFlow rule sets. The analysis and generation tools have to be extended such that they are able to produce and consume, respectively, such a modified seed.

Figure 5.10 shows a high-level architecture of ClassBench-ng comprising four main building blocks, which are presented in details in the following subsections.

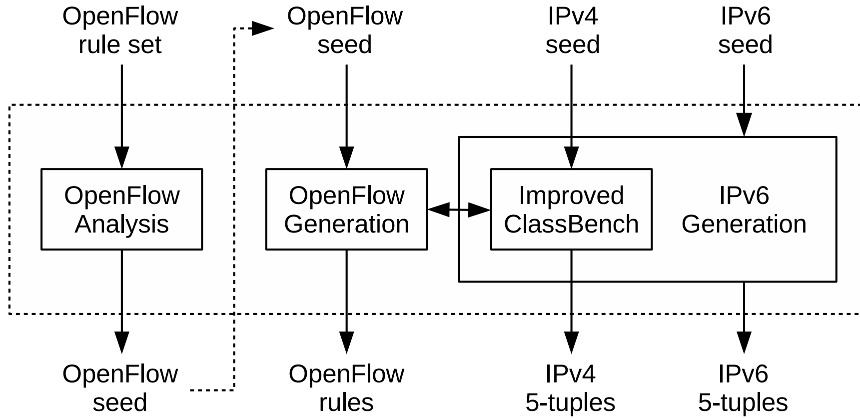


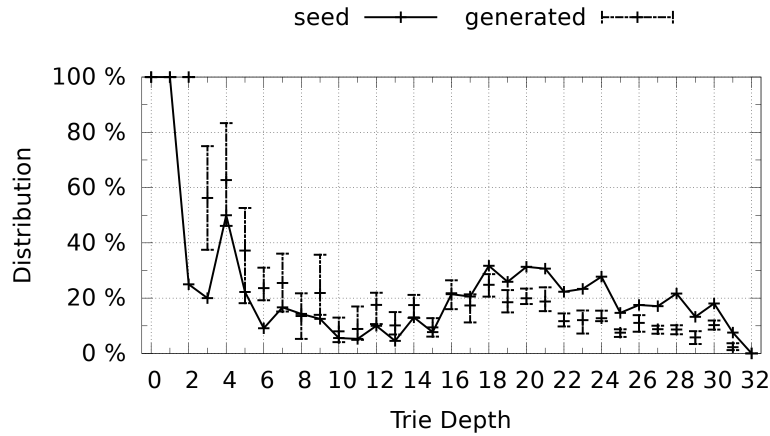
Figure 5.10: The high-level architecture of ClassBench-ng.

5.3.1 Improved ClassBench

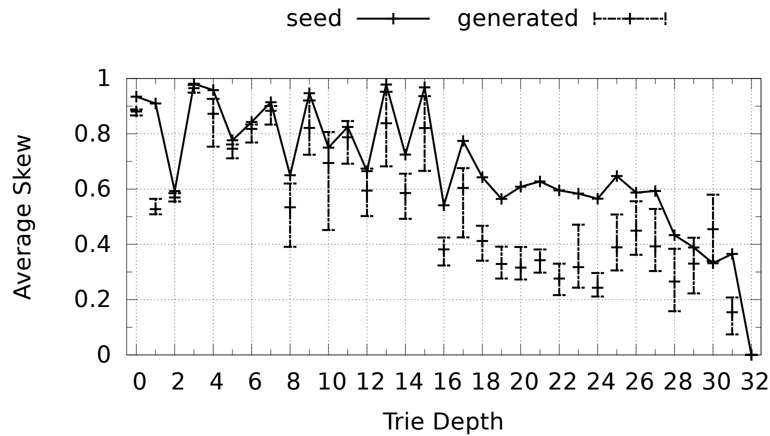
For over a decade, researchers have used original ClassBench in order to generate synthetic classification rule sets for the benchmarking of packet classification algorithms. However, a test campaign evaluating the fidelity of ClassBench, which was performed in the course of this thesis' preparations, revealed that the rule set generation process of ClassBench is not always accurate. While the layer four ports and protocol of generated rules accurately follow the input seed, the IPv4 prefixes show lower accuracy. This is illustrated in Figures 5.11 that compare prefix set parameters extracted from the input IPv4 seed and from rules generated by original ClassBench.

Since the generation process proved to be accurate with respect to the prefix nesting threshold, Figures 5.11 focus on the branching probability and skew, which do not follow the required distributions precisely. Indeed, the generated branching probability meets the requirements only for 13 trie levels, while the average skew only for 5 levels. The most probable explanation of such errors is based on parameters interdependence. To prevent a complex resolving of dependencies among the prefix set parameters, ClassBench assigns each parameter a priority and tries to meet the required distributions in an order given by the priority of corresponding parameters. This hypothesis corresponds with the accuracy of the prefix nesting threshold, which is the highest-priority prefix set parameter in original ClassBench. Note that a comparison of input and generated prefix length distribution could not be shown because this parameter is not directly available in a seed.

ClassBench-ng improves the generation process of ClassBench by iteratively building an output rule set with source and destination prefix set characteristics as close as possible to the distributions from the input seed. The pseudocode in Figure 5.12 shows the process of rule set construction in the *Improved ClassBench* block. First of all, the tool uses original ClassBench and generates a 100-times larger initial rule set *rules* (line 3). Then it prunes the tries representing source and destination IP prefixes of this rule set to converge on a solution which is accurate and contain the target number of IP prefixes (lines 4 and 5). The details of a utilized trie pruning algorithm are described later in this section.



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



(b) Average skew distribution.

Figure 5.11: 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.

Once the pruned source trie (*src_trie*) and destination trie (*dst_trie*) are available, *Improved ClassBench* selects from *rules* those rules that contain source and destination IP prefixes available in the pruned tries. To find these rules, the tool constructs a bipartite graph in which each node represents a prefix from either *src_trie* or *dst_trie* and each edge connecting two nodes represents a rule from *rules* that contains corresponding source and destination IP prefixes. The rules that ClassBench-ng is looking for are represented by maximum matching in the constructed bipartite graph (line 6). Each selected rule is added to final rule set *output_rules* and the rule together with its source and destination IP prefixes are removed from *rules*, *src_trie*, and *dst_trie*, respectively (lines 7 to 11). In case the maximum matching does not represent the target number of rules, the remaining rules are selected from *rules* according to prefixes available in *dst_trie* only and source IP prefixes of these rules are replaced with arbitrarily selected prefixes from *src_trie* (lines 12 to 19).

```
 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 5.12: The pseudocode of rule set construction in *Improved ClassBench*.

Trie Pruning

Figure 5.13 shows a pseudocode of the utilized trie pruning algorithm. 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 minimize a negative effect of the convergence over the target number 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. However, the key steps of the trie pruning algorithm are represented by the functions AdjustBranching, AdjustSkew, and AdjustPrefixes, which are described in the following paragraphs.

Branching Probability Adjustment Starting from the root node of the trie, this step (line 7) adjusts branching probability at each trie level 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, which keeps as much space as possible for adjustments performed in further pruning steps. 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) starts from the leaves of the trie and it increases or decreases average skew at each trie level. In particular, it removes prefixes from the lighter or heavier subtree of two-children nodes. However, this step never removes the last prefix from the leaf nodes and it tries to not alter average skew when

```
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 5.13: The pseudocode of the trie pruning algorithm.

removing prefixes at already adjusted levels (i.e., levels below the current level). Because of the same reason as in the previous pruning step, the nodes of each level are selected increasingly according to the total number of prefixes in their subtrees.

Prefix Length Distribution and the Total Number of Prefixes Adjustment The last trie pruning step (lines 11 and 13) removes prefixes at each trie level (starting from the root node of the trie) to make their total number matching the target value. When removing the prefixes, this step tries to not alter the skew of two-children nodes, which is realized by tracking the number of prefixes that should be removed from each subtree. Similarly to the average skew distribution adjustment, this step also never removes the last prefix from the leaf nodes. Doing so would imply the deletion of a whole branch, thus altering the branching probability.

5.3.2 IPv6 Generation

The *IPv6 Generation* block extends the improved version of the original ClassBench's rule set generator with support for the generation of IPv6 rules. Since both IPv4 and IPv6 prefix sets can be represented using a trie and the trie-related parameters utilized in original ClassBench are able to catch current IPv6 dynamics, an IPv6 seed straightforwardly extends the trie-related parameters to allow the specification of corresponding distributions for up to 129 trie levels. In the same way ClassBench-ng also extends the improved process of IPv4 rule sets generation, i.e., it adds support for the generation of IP prefixes according to trie-related distributions specified in the IPv6 seed.

5.3.3 OpenFlow Analysis

The *OpenFlow Analysis* block takes as an input OpenFlow rules and generates the corresponding OpenFlow seed. Although ClassBench-ng already provides a couple of seeds for OpenFlow rules generation, a rule set analyzer ensures the flexibility of the toolkit. For instance, ClassBench-ng can be easily adapted to the generation of various OpenFlow rule sets

corresponding to different use cases that can appear in programmable OpenFlow-enabled networks.

Currently, ClassBench-ng is able to correctly parse rule sets represented in the format utilized by the `ovs-ofctl` command line tool [5] and generate the appropriate OpenFlow 1.0.0 seed. This format is primarily aimed at representing flow table entries of OpenFlow switches (note that each flow table entry corresponds to an OpenFlow rule). However, since the IPv4 prefix and IPv4 5-tuple are also valid OpenFlow rules, the *OpenFlow Analysis* block is able to parse these types of classification rules as well. Although the `ovs-ofctl` tool supports both IPv4 and IPv6 prefixes, ClassBench-ng currently supports parsing of IPv4 prefixes only.

The analysis described in Section 5.2.3 revealed that apart from distributions related to 5-tuple fields, the most important characteristics of an OpenFlow rule set are the distribution of rules over *rule types* and separate statistical distributions for OpenFlow-specific header fields. Therefore, an OpenFlow seed is a backward-compatible extension of a 5-tuple seed (i.e., the seed utilized in original ClassBench) consisting of three main sections: (1) a rule type distribution, (2) a 5-tuple seed, and (3) an OpenFlow-specific fields seed. The first section represents header fields dependency observed in the analyzed rule set. The structure of the second section is exactly the same as of a standalone 5-tuple seed, thus it is possible to use an OpenFlow seed for the generation of 5-tuple rules using original ClassBench (or its improved version that is employed in ClassBench-ng). Finally, the last section contains separate statistical distributions for OpenFlow-specific header fields. Each of these distributions utilizes one of the following representations:

- *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 representation and a particular header field reflects various requirements. For 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 use any values from the original rule set, thus they are suitable for header fields carrying confidential content. The former (*null*) is used for fields with a relatively small number of possible values, i.e., *vlan_prio* and *ip_tos*, while the latter (*size*) is used for fields with a potentially large set of unique values, i.e., *vlan_id*. Last but not least, *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.

The proposed OpenFlow seed meets the required properties defined in Section 5.1 even after supplementing a 5-tuple seed with a rule type distribution and seed for OpenFlow-specific header fields. Including these sections into an OpenFlow seed directly ensures the completeness of a rule set's representation. As described in the previous paragraph, the requirement of anonymity is reflected in the already introduced pairing between the four representations of statistical distributions and individual header fields. The use of statistical distributions also makes sure that an OpenFlow seed allows a scalable generation of synthetic rule sets.

5.3.4 OpenFlow Generation

The *OpenFlow Generation* block generates a set of OpenFlow rules from an input OpenFlow seed. Figure 5.14 shows the pseudocode of the generation process. At line 3 it starts with the generation of a set of IPv4 5-tuples, which follow the parameters specified in the seed, using the *Improved ClassBench* block of ClassBench-ng. Each generated 5-tuple is then transformed to an OpenFlow rule that complies with a *rule_type* generated according to the rule type distributions stored in the seed (line 5). In particular, some of the 5-tuple fields might be removed (lines 6 to 8) and other OpenFlow-specific fields might be added (lines 9 to 12).

```
 1: function OpenFlowGeneration(seed, size)
 2:     of_rules ← ∅
 3:     ipv4_5tuples ← Improved_ClassBench(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, field_value)
13:         of_rules ← of_rules ∪ {rule}
14:     return of_rules
15: end function
```

Figure 5.14: Pseudocode of OpenFlow rules generator.

The generation of values for OpenFlow-specific header fields in line 11 is driven by representation utilized for particular header fields. However, in order to generate consistent OpenFlow rules, further constraints on the generated values have to be sometimes applied. For instance, values for *in_port* and *eth_type* fields are selected from corresponding distributions recorded in a seed, but the value of *eth_type* must be set to:

- `0x8100` when *vlan_id* or *vlan_prio* is going to be specified,
- `0x0800` or `0x0806` when *ip_src*, *ip_dst*, or *ip_proto* is going to be specified, and
- `0x0800` when *ip_tos*, *l4_src*, or *l4_dst* is going to be specified.

The selection of values from distributions is used also for *mac_src* and *mac_dst* header fields, which utilize the *parts* representation. Their vendor part is generated according to the distribution from the seed, but the device part is generated randomly. The random generation of header field values is mainly used for fields utilizing *size* and *null* representations. Nevertheless, only the value of the *vlan_prio* field is generated completely randomly without any further constraints. The value of *ip_tos* field is randomly selected from a pool of values defined by IANA [1], while the value of *vlan_id* must not be neither `0x000` nor `0xFFF` (the VLAN standard [16] reserves these values for a special purpose) and a total number of unique *vlan_id* values must not exceed the corresponding parameter recorded in the seed.

5.4 Experimental Evaluation

This section evaluates the fidelity of ClassBench-ng's rule set generation for IPv4 prefixes, IPv6 prefixes, and OpenFlow rules. In case of IPv4 prefixes generation, ClassBench-ng is compared against original ClassBench [63] and FRuG [29], while the evaluation of IPv6 prefixes generation is done against Non-random Generator [66]. Finally, the fidelity of OpenFlow rules generation in ClassBench-ng is compared against FRuG [29]. The evaluation does not focus on layer four ports and protocol because ClassBench-ng directly uses the values of these header fields generated by original ClassBench, which provides accurate results in this case.

In order to fairly compare ClassBench-ng with other synthetic rule set generators, the evaluation presented in this section is based on the value of *root-mean-square error* (RMSE) that is computed using Equation 5.2. In this equation, n represents the number of generated rule sets, $\bar{y}$ is the target value of an evaluated parameter, and y_i stands for the parameter's value extracted from the generated sets. The performed experiments were carried on by generating $n = 10$ rule sets using tool-specific seeds extracted from an *original rule set*. The characteristics of the *original rule set* thus represent the target values (i.e., $\bar{y}$) against which were compared the same characteristics of the generated sets (i.e., y_i) obtained from various rule set generators.

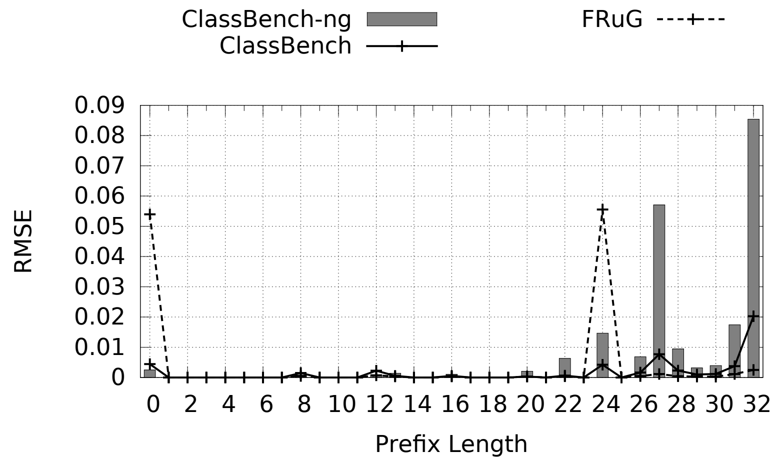
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{y} - y_i)^2} \tag{5.2}$$

5.4.1 IPv4 Prefixes Generation

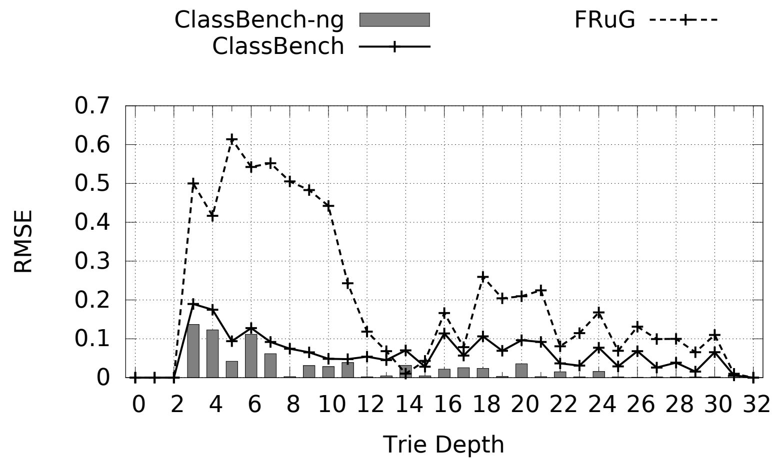
The comparison of ClassBench-ng, ClassBench, and FRuG on the generation of IPv4 prefixes utilized an *original rule set* that had been generated by ClassBench using the `acl4` seed provided with this tool. Because both ClassBench-ng and FRuG support the transformation of an input rule set into the corresponding seed, they were used to generate input seeds for the compared rule set generators from the *original rule set* (note that a seed for ClassBench-ng can also be used in original ClassBench). Finally, these seeds were utilized in the compared tools to generate rule sets, whose trie-related characteristics were assessed using RMSE.

Figures 5.15 show the comparison of RMSE obtained for ClassBench-ng, ClassBench, and FRuG on each trie level. According to these results, ClassBench-ng fully outperforms original ClassBench and except for only one trie level also FRuG in terms of the branching probability distribution (Figure 5.15b). The situation is more balanced with respect to the average skew distribution (Figure 5.15c). In this case, ClassBench-ng is more precise on approximately 50 % of trie levels when compared against ClassBench and on more than 80 % of levels when compared against FRuG. However, Figure 5.15a shows a poor fidelity of ClassBench-ng with respect to the prefix length distribution.

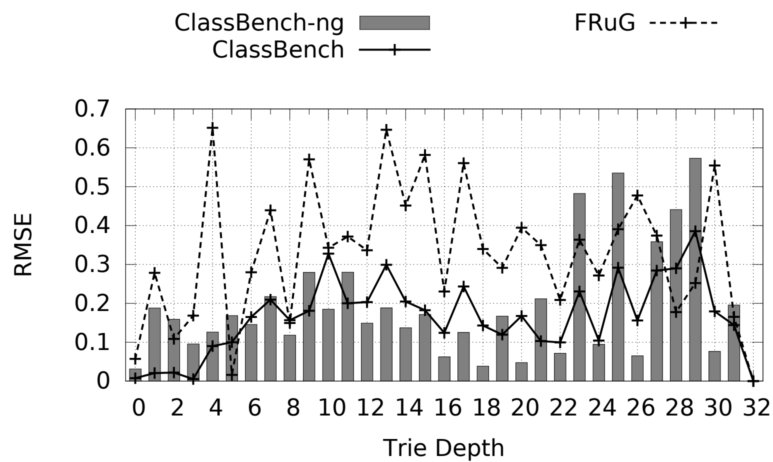
It is not possible to improve ClassBench-ng's generation fidelity for the prefix length distribution without impacting negatively on the other parameters. Nevertheless, in case of the prefix length distribution RMSE is 10-times lower compared to other trie-related parameters. Therefore, ClassBench-ng is the most accurate rule set generator among the compared tools on average. This is confirmed in Figure 5.16, which shows the average RMSE per trie level that is computed from RMSE of all trie-related parameters defined for each level of a trie. Considering such a metric, ClassBench-ng outperforms both ClassBench



(a) Prefix length distribution.



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



(c) Average skew distribution.

Figure 5.15: The comparison of trie-related parameters' *root-mean-square error* on each trie level when generating IPv4 prefix sets using ClassBench-ng, ClassBench, and FRuG.

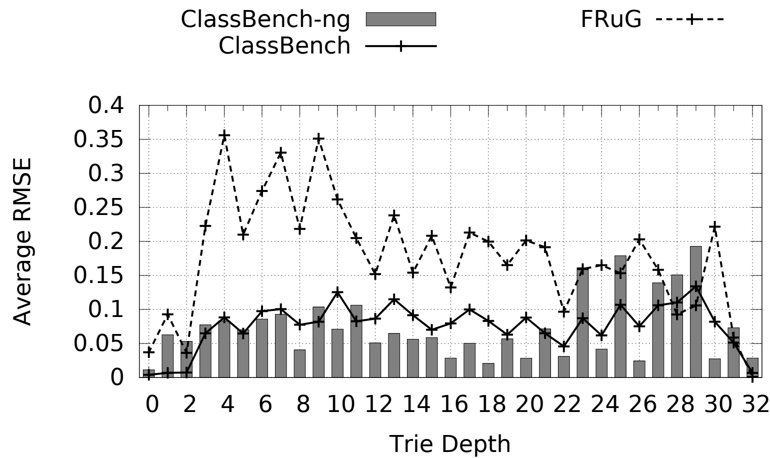


Figure 5.16: An average *root-mean-square error* of ClassBench-ng, ClassBench, and FRuG for each trie level. The average was computed from RMSE of trie-related parameters displayed in Figures 5.15.

and FRuG on the majority of trie levels, including *level 24*, which stores the greatest portion of prefixes in the analyzed IPv4 prefix sets.

Last but not least, the RMSE value for the prefix nesting threshold is 0.63, 0.00, and 0.89 in case of ClassBench-ng, ClassBench, and FRuG, respectively. Even though ClassBench is flawless in achieving the target value of this parameter, since its value can only be an integer, none of the compared generators show a great error.

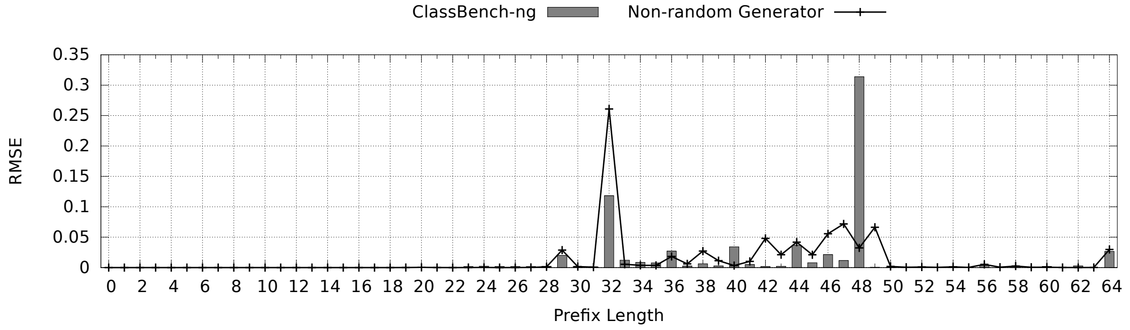
5.4.2 IPv6 Prefixes Generation

When comparing the quality of ClassBench-ng's IPv6 prefix set generation against Non-random Generator, two *original rule sets* were used. 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 does not lead to an entirely fair comparison of the tools, it is enforced by their different requirements on input data. ClassBench-ng requires a seed extracted from a rule set of a target type (an IPv6 prefix set in this case) and Non-random Generator expects an IPv4 prefix set on its input. Thus, using IPv4 and IPv6 prefix sets originating from the same core router leads to the fairest comparison of the tools.

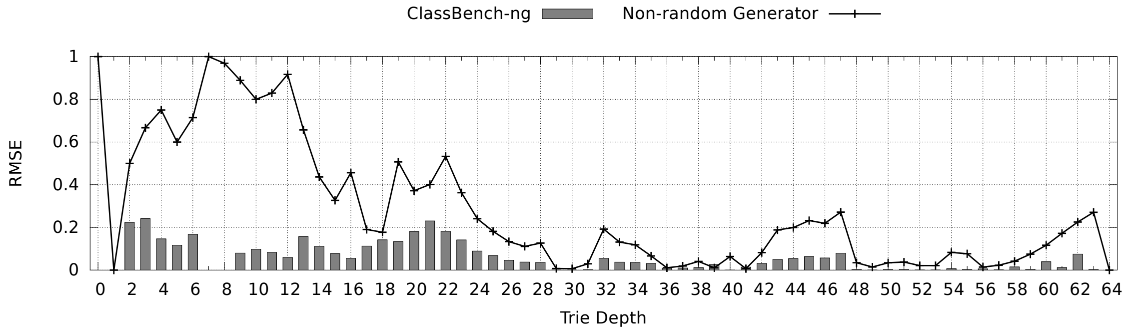
The results of the comparison are shown in Figures 5.17. Both ClassBench-ng and Non-random Generator achieve a comparable quality of IPv6 prefixes generation in terms of the prefix length distribution (Figure 5.17a). However, ClassBench-ng is more precise with respect to the branching probability distribution (Figure 5.17b), while Non-random Generator wins the comparison on the average skew distribution (Figure 5.17c).

5.4.3 OpenFlow Rules Generation

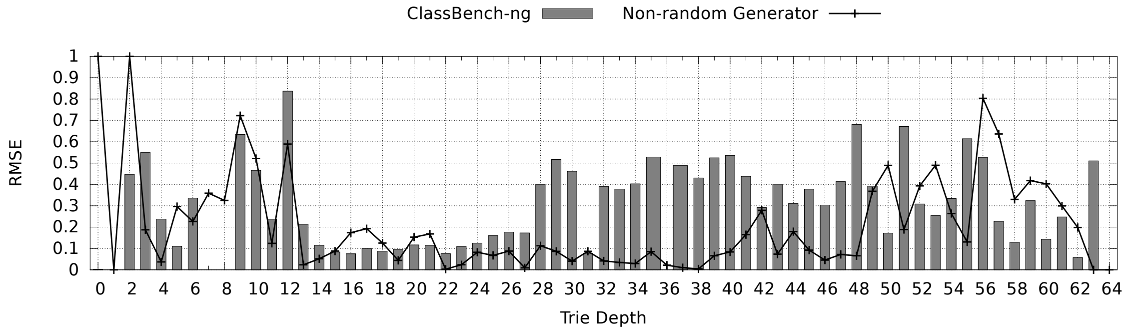
The fidelity of ClassBench-ng's OpenFlow rules generation was compared against FRuG on two important characteristics of an OpenFlow rule set: (1) header fields dependency represented by the *rule type* parameter and (2) separate statistical distributions for OpenFlow-



(a) Prefix length distribution.



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



(c) Average skew distribution.

Figure 5.17: The comparison of trie-related parameters' *root-mean-square error* on each trie level when generating IPv6 prefix sets using ClassBench-ng and Non-random Generator.

specific header fields. To fairly compare ClassBench-ng with FRuG, input seeds for both generators were extracted from rule set `of1`, which was chosen as an *original rule set*.

Figure 5.18 compares ClassBench-ng's RMSE on particular *rule types* utilized in the *original rule set* against RMSE obtained on these *rule types* with FRuG. In this experiment ClassBench-ng clearly outperforms FRuG as it achieves higher RMSE for *rule types* 1304 and 2048 only. 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 is also more accurate than FRuG with respect to the generation of OpenFlow-specific header fields, as shown in Figure 5.19. Since header fields *vlan_id*, *vlan_prio*, and *ip_tos* are always wildcarded in the *original rule set*, the figure compares average RMSE of the generators on the *in_port*, *mac_src*, *mac_dst*, and *eth_type* header

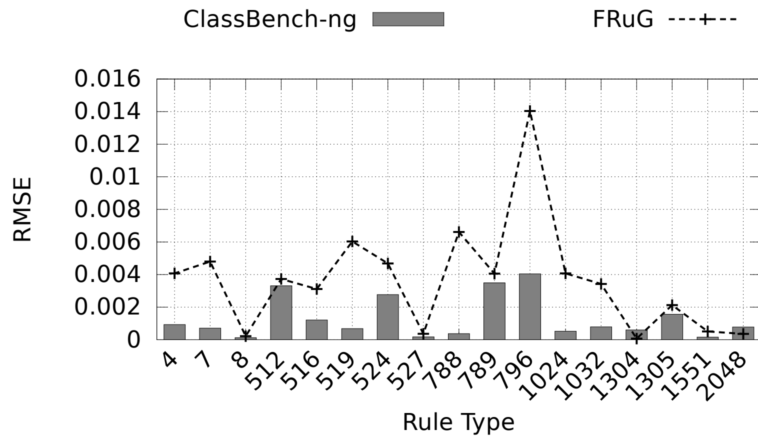


Figure 5.18: The comparison of *root-mean-square error* for *rule types* utilized in OpenFlow rule sets generated by ClassBench-ng and FRuG.

fields only. While the average RMSE of both generators is almost the same (and very low) for *in_port*, ClassBench-ng is clearly more accurate than FRuG for all other fields. Moreover, Figure 5.20 proves that at least in case of the *mac_dst* field (more precisely its vendor part) ClassBench-ng outperforms FRuG not only on average but also on all generated values.

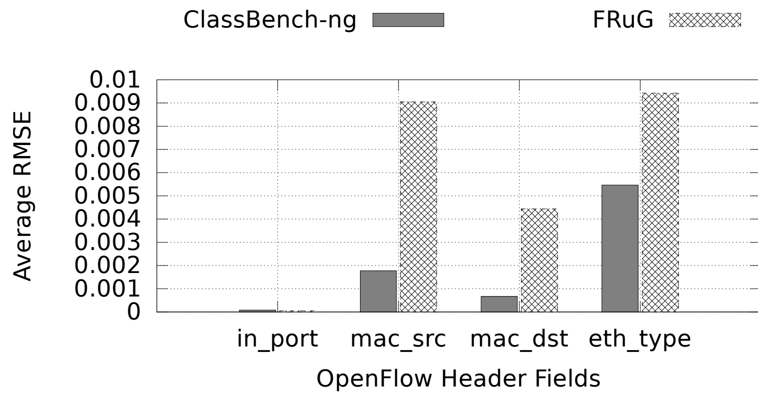


Figure 5.19: An average *root-mean-square error* of ClassBench-ng and FRuG for all evaluated OpenFlow-specific header fields. The average for each header field was computed from all RMSE values of that particular field.

The whole comparison of ClassBench-ng and FRuG is in favor of the sooner. Additionally, while seeds utilized by ClassBench-ng comprise statistical distributions extracted from real rule sets, FRuG requires manually configured distributions for all header fields but IP and MAC addresses. Thus, a low value of RMSE does not have to mean that OpenFlow rules generated by FRuG accurately represent a real rule set because the input distributions could be set incorrectly. It is also important to note that ClassBench-ng produces consistent OpenFlow rules, which satisfy all constraints introduced in Section 5.3.4.

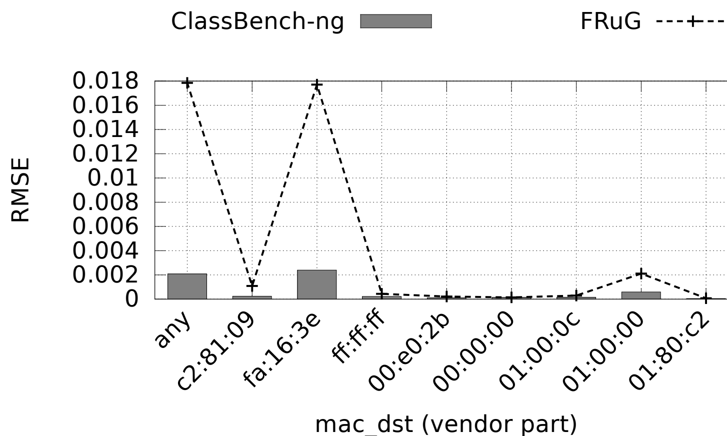


Figure 5.20: The comparison of *root-mean-square error* of ClassBench-ng and FRuG on *mac_dst* field's vendor part.

5.5 Summary

To address the lack of real classification rule sets for benchmarking newly developed packet classification algorithms, this chapter introduces ClassBench-ng, a new open source tool for the generation of synthetic IPv4, IPv6, and OpenFlow 1.0.0 rules. Differently from other similar tools, ClassBench-ng comprises not only a rule set generator but also a rule set analyzer able to extract key characteristics of a real rule set into a seed, which completely characterize the analyzed set in an anonymous and scalable way. Providing the rule set analyzer together with the rule set generator that utilizes analyzer's output makes ClassBench-ng an accurate (with respect to real classification rule sets) as well as flexible tool, because its generation process follows the characteristics of real rule set stored in the seed, which can be regenerated on demand using the rule set analysis feature.

The architecture of ClassBench-ng and the structure of the seed were designed after a thorough analysis of various types of real classification rule sets: IPv4/IPv6 prefix sets from core routers, IPv4 5-tuples from ACLs deployed in a university network, and OpenFlow 1.0.0 flow tables from Open vSwitches running in a datacenter. A comparative analysis of IPv4 prefix sets and 5-tuples after 10 years of Internet's evolution, which focused on the rule set parameters proposed in original ClassBench, demonstrated that these parameters are valid even after more than a decade from their first use. Moreover, as was shown by the analysis of IPv6 prefix sets, conceptually the same set of trie-related parameters is also able to reflect dynamic behavior connected with an accelerating adoption of the IPv6 protocol. On the other hand, the analysis of OpenFlow rules taken from a real deployment in a datacenter is among the very first studies of its kind. Its results mainly suggest to characterize OpenFlow rule sets not only by separate distributions for the OpenFlow-specific header fields (in addition to the parameters capturing the key characteristics of the 5-tuple fields) but also using the distribution of rules over a set of utilized *rule types*.

Although the results of the performed analysis suggest the parameters utilized in original ClassBench as suitable even after more than a decade, a further campaign evaluating the fidelity of the ClassBench's rule set generation process revealed its non-negligible limitations with respect to the generation of IPv4 prefixes. Therefore, ClassBench-ng generates IPv4 5-tuples using an improved version of original ClassBench that employs an iterative trie

pruning algorithm to generate IP prefixes which follow trie-related distributions from the input seed more precisely. In addition, ClassBench-ng extends the structure of the seed as well as the rule set generator of original ClassBench in order to allow the representation and generation of IPv6 prefix sets (i.e., tries of up to 129 levels). Apart from improvements and extensions of original ClassBench related to IPv4 and IPv6 rules, ClassBench-ng also introduces a complete toolchain for OpenFlow rule sets. An OpenFlow rule set analyzer is currently able to correctly parse rule sets in the `ovs-ofctl` format and to store their characteristics into OpenFlow seeds, each of which contains the rule type distribution and separate distributions for the OpenFlow-specific header fields, along with the parameters characterizing the 5-tuple fields. Such seed is then used as an input to an OpenFlow rule set generator that produces a set of consistent OpenFlow rules following the distributions specified in the input seed. Finally, it is worth mentioning that the OpenFlow analyzer is also able to analyze IPv4 rule sets, if they are represented in the `ovs-ofctl` format.

The proposed ClassBench-ng was experimentally evaluated on the generation of classification rules for various use cases and its RMSE was compared against RMSE of existing rule set generators relevant for particular use cases. With respect to the generation of IPv4 prefixes, ClassBench-ng proved to be better than both original ClassBench and FRuG on average, but it did not outperform the other tools on all trie-related parameters. Similar results were obtained from the evaluation of IPv6 prefix sets generation in ClassBench-ng and its comparison against the fidelity of IPv6 prefixes generation in Non-random Generator. ClassBench-ng achieved the best results in case of OpenFlow rules generation. For this use case, it clearly outperformed FRuG with respect to the rule type distribution and it also generated the value of the individual OpenFlow-specific header fields more precisely than FRuG.

Chapter 6

Conclusions

This thesis deals with packet classification that is one of the most common operations implemented in networking devices. Although the basic principle of packet classification has always been the same, a rapid development of the Internet, which we have been experiencing since the beginning of this century, has significantly increased the requirements that have to be met by current packet classification algorithms deployed in core networks. Namely, growing transfer rate has led to a shorter time available for the classification of a single packet, while the number of bits involved in the classification decision has increased due to accelerated adoption of the IPv6 and OpenFlow protocols. Moreover, the complexity of packet classification has also been increased by a growing number of classification rules. Therefore, the majority of current packet classification research address the performance of packet classification algorithms. However, as the requirements on the algorithms are continuously increasing, improving their performance is still an active process producing novel classification algorithms that have to be benchmarked, ideally using real sets of classification rules. Unfortunately, such rule sets are not publicly available for most of the packet classification use cases. Current research thus further focuses on the generation of synthetic rule sets applicable to benchmarking packet classification algorithms. These two issues in current research on packet classification are also addressed in this thesis, which has been directed by the goals set in Section 1.1.

The performance of packet classification algorithms is addressed by an FPGA-based implementation of prefix matching that is able to perform almost 255 MLPS for both IPv4 and IPv6 prefixes, which translates into throughput of 170 Gbps when considering the shortest Ethernet frames. Such lookup performance is enabled by a newly proposed pipelined hardware architecture utilizing on-chip memory blocks available in current FPGA chips. Although the whole architecture consists of two processing pipelines (to use both ports of on-chip memory blocks), each of which comprises 23 stages (to support matching IPv6 prefixes), it easily fits into the target FPGA chip (Xilinx Virtex-6 XC6VSX475T). In addition, because the amount of the on-chip memory is limited, prefix sets are encoded using a novel memory-efficient representation that allows to completely store any of the available prefix sets in the on-chip memory of the target FPGA. The proposed prefix set representation is more memory efficient than representations utilized in both TBM and SST algorithms, especially for IPv6 prefix sets corresponding to sparse prefix trees. Furthermore, even though the proposed representation is slightly worse than the representation utilized in the PPLA algorithm in case of generated IPv6 prefix sets, it is significantly better on real IPv4 prefix sets. Since the proposed prefix set representation and pipelined hardware architecture together allow to perform prefix matching with throughput required in 100 Gbps networks

(regardless the version of the IP protocol), it can be concluded that the first goal of this thesis has been successfully achieved.

To enable a realistic assessment of classification algorithms' performance parameters, this thesis introduces a synthetic rule set generator called ClassBench-ng, which combines features of existing 1-dimensional and multi-dimensional generators in a single tool and explicitly supports also the generation of OpenFlow 1.0.0. rule sets. Though original ClassBench provides all necessary features for the generation of IPv4 5-tuples, generated IPv4 prefix sets do not precisely follow corresponding distributions specified in an input seed. ClassBench-ng thus builds on the original ClassBench tool, but it improves the ClassBench's process of IPv4 prefixes generation and extends this process with the support for the generation of IPv6 prefixes. Nevertheless, the main contribution of ClassBench-ng is a newly added OpenFlow toolchain comprising not only a rule set generator but also a rule set analyzer that is capable of analyzing IPv4 and OpenFlow rule sets specified in the `ovs-ofctl` format. Since ClassBench-ng contains the rule set analyzer producing a seed together with the rule set generator utilizing such seed on its input, it is able to generate synthetic rule sets with properties similar to the analyzed real rule set and also to adjust the seed when the properties of the real set changes. Although the generation of IPv4 prefixes in ClassBench-ng is not more accurate than in original ClassBench or FRuG with respect to all trie-related parameters, on average ClassBench-ng outperforms both of these tools. The situation is similar for IPv6 prefixes generation, in which case the precision of ClassBench-ng is comparable with Non-random Generator, i.e., a specialized 1-dimensional generator of IPv6 prefixes. ClassBench-ng achieves the best results in case of OpenFlow rules generation, where it is clearly more accurate than FRuG with respect to both *rule type* and individual OpenFlow-specific header fields. In summary, the proposed ClassBench-ng toolchain is capable of generating synthetic IPv4, IPv6, and OpenFlow 1.0.0 rule sets, which follow statistical distributions extracted from real rule sets, and the precision of its rule set generation process is comparable or better than the precision of similar rule set generators. Therefore, the second goal of this thesis has also been achieved.

Despite the goals set for this thesis have been successfully achieved, there are numerous options for future work in the addressed areas. Currently, there is an ongoing effort at the development of a trace generator for ClassBench-ng that will allow to generate a packet header trace for a given rule set. Using such trace it will be possible to comprehensively assess not only the expected worst case performance of a packet classification algorithm utilizing the given rule set, but also its actual average performance under a traffic load. Once the trace generator will be finished, the next steps may focus on further extensions of the presented solutions. It would be interesting either to allow incremental updates of a prefix set while keeping a high memory efficiency of its representation or to make the ClassBench-ng's rule set analyzer able to extract a seed from different types of real rule sets specified in various formats. Nevertheless, even more appealing would be to address challenges brought by a continuous and accelerating evolution of the Internet. This category currently includes improving the performance of the prefix matching architecture to support 400 Gbps Ethernet and extending ClassBench-ng to support further versions of the OpenFlow standard.

6.1 Summary of Contributions

The following list provides a quick overview of main contributions of this thesis to research in the area of packet classification, which were made in the course of achieving the goals set in Section 1.1.

- The analysis of various types of classification rules:
  - real IPv4 and IPv6 prefix sets from forwarding tables of core routers;
  - synthetic IPv6 prefix sets generated using Non-random Generator;
  - real IPv4 5-tuple sets from ACLs applied in a university network;
  - real OpenFlow 1.0.0 rule sets from Open vSwitches deployed in a datacenter.
- The novel memory-efficient representation of prefix sets, whose memory efficiency ratio is comparable with the PPLA algorithm on generated IPv6 prefix sets and better than the TBM, SST, and PPLA algorithms on real IPv4 prefix sets.
- The pipelined hardware architecture implementing prefix matching based on the proposed prefix set representation that is able to perform almost 255 MLPS, which translates into throughput of 170 Gbps for the shortest Ethernet frames.
- The open source tool ClassBench-ng, which is publicly available at https://github.com/classbench-ng/classbench-ng and consists of:
  - the rule set analyzer able to produce seeds corresponding to real IPv4 and OpenFlow 1.0.0 rule sets in the `ovs-ofctl` format;
  - the rule set generator capable of generating synthetic IPv4, IPv6, and OpenFlow 1.0.0 rule sets that follow properties specified in a seed.

Bibliography

[1] Differentiated Services Field Codepoints (DSCP). Available online [August 2018] http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

[2] IPv6 Deployment Aggregated Status. Available online [June 2018] https://www.vyncke.org/ipv6status.

[3] IPv6: IPv6 / IPv4 Comparative Statistics. Available online [September 2017]: http://bgp.potaroo.net/v6/v6rpt.html.

[4] Open vSwitch. Available online [June 2018] https://www.openvswitch.org/.

[5] `ovs-ofctl(8)` (Open vSwitch 2.8.90 Manpages). Available online [August 2018] https://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.html.

[6] Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. IEEE Std 802.3, 2000 Edition. IEEE. 3 Park Avenue, New York, NY 10016-5997, USA. October 2000. ISBN 0-7381-2574-8.

[7] OpenFlow Switch Specification. Version 1.0.0 (Wire Protocol 0x01). Open Networking Foundation. December 2009.

[8] Hardwarová akcelerace klasifikace paketů. Technická zpráva. Výzkumná skupina ANT at FIT. Květen 2010.

[9] IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment 4: Media Access Control Parameters, Physical Layers, and Management Parameters for 40 Gb/s and 100 Gb/s Operation. IEEE Std 802.3ba-2010. IEEE. 3 Park Avenue, New York, NY 10016-5997, USA. June 2010. ISBN 978-0-7381-6322-2.

[10] OpenFlow Switch Specification. Version 1.5.1 (Protocol version 0x06). Open Networking Foundation. March 2015.

[11] Virtex-6 Family Overview. DS150 (v2.5). August 2015.

[12] IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2016. IEEE. 3 Park Avenue, New York, NY 10016-5997, USA. December 2016. ISBN 978-1-5044-3645-8.

[13] IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation. IEEE Std 802.3bs-2017. IEEE. 3 Park Avenue, New York, NY 10016-5997, USA. December 2017. ISBN 978-1-5044-4450-7.

[14] The Zettabyte Era: Trends and Analysis. White paper. Cisco. June 2017.

[15] 7 Series FPGAs Data Sheet: Overview. DS180 (v2.6). February 2018.

[16] IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. IEEE Std 802.1Q-2018. IEEE. 3 Park Avenue, New York, NY 10016-5997, USA. July 2018. ISBN978-1-5044-4929-8.

[17] UltraScale Architecture and Product Data Sheet: Overview. DS890 (v3.6). November 2018.

[18] Advanced Network Technology Center, University of Oregon: Route Views Project Page. Available online [March 2018]: http://www.routeviews.org/.

[19] Ahmed, O.; Areibi, S.; Grewal, G.: Hardware Accelerators Targeting a Novel Group Based Packet Classification Algorithm. *International Journal of Reconfigurable Computing*. vol. 2013. January 2013. ISSN 1687-7195.

[20] Baboescu, F.; Singh, S.; Varghese, G.: Packet Classification for Core Routers: Is there an alternative to CAMs? In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE. March 2003. pp. 53–63. ISBN 0-7803-7752-4.

[21] Baboescu, F.; Varghese, G.: Scalable Packet Classification. *ACM SIGCOMM Computer Communication Review*. vol. 31, no. 4. October 2001: pp. 199–210. ISSN 0146-4833.

[22] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*. vol. 13, no. 7. July 1970: pp. 422–426. ISSN 0001-0782.

[23] Czyz, J.; Allman, M.; Zhang, J.; et al.: Measuring IPv6 Adoption. *ACM SIGCOMM Computer Communication Review*. vol. 44, no. 4. October 2014: pp. 87–98. ISSN 0146-4833.

[24] Decasper, D.; Dittia, Z.; Parulkar, G.; et al.: Router Plugins: A Software Architecture for Next Generation Routers. *ACM SIGCOMM Computer Communication Review*. vol. 28, no. 4. October 1998: pp. 229–240. ISSN 0146-4833.

[25] Dharmapurikar, S.; Song, H.; Turner, J.; et al.: Fast Packet Classification Using Bloom Filters. In *2006 Symposium on Architecture For Networking And Communications Systems*. IEEE. December 2006. pp. 61–70. ISBN 978-1-59593-580-9.

[26] Eatherton, W.; Varghese, G.; Dittia, Z.: Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*. vol. 34, no. 2. April 2004: pp. 97–122. ISSN 0146-4833.

[27] Fredkin, E.: Trie Memory. *Communications of the ACM*. vol. 3, no. 9. September 1960: pp. 490–499. ISSN 0001-0782.

[28] Štěpán Friedl; Puš, V.; Matoušek, J.; et al.: Designing a Card for 100 Gb/s Network Monitoring. Technical Report 7/2013. CESNET. Zikova 1903/4, 160 00 Prague 6, Czech Republic. 2013.

[29] Ganegedara, T.; Jiang, W.; Prasanna, V.: FRuG: A Benchmark for Packet Forwarding in Future Networks. In *Proceeedings of the IEEE 29th International Performance Computing and Communications Conference (IPCCC)*. IEEE. December 2010. pp. 231–238. ISBN 978-1-4244-9330-2.

[30] Google: IPv6 - Statistics. Available online [September 2017]: https://www.google.com/intl/en/ipv6/statistics.html.

[31] Gupta, P.; McKeown, N.: Packet Classification on Multiple Fields. *ACM SIGCOMM Computer Communication Review*. vol. 29, no. 4. August 1999: pp. 147–160. ISSN 0146-4833.

[32] Gupta, P.; McKeown, N.: Classifying Packets with Hierarchical Intelligent Cuttings. *IEEE Micro*. vol. 20, no. 1. January 2000: pp. 34–41. ISSN 0272-1732.

[33] International Telecommunications Union (ITU): Active mobile-broadband subscriptions per 100 inhabitants, 2007-2017. Available online [September 2017]: https://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2017/Stat_page_all_charts_2017.xls.

[34] International Telecommunications Union (ITU): List of ITU-R Recommendations on IMT. Available online [September 2017]: https://www.itu.int/net/ITU-R/index.asp?category=information&rlink=imt-advanced-rec.

[35] Jain, R.; Paul, S.: Network Virtualization and Software Defined Networking for Cloud Computing: A Survey. *IEEE Communications Magazine*. vol. 51, no. 11. November 2013: pp. 24–31. ISSN 0163-6804.

[36] Jiang, W.; Prasanna, V. K.: Parallel IP Lookup using Multiple SRAM-based Pipelines. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IPDPS 2008. IEEE. April 2008. pp. 1–14. ISBN 978-1-4244-1693-6.

[37] Kobayashi, M.; Seetharaman, S.; Parulkar, G.; et al.: Maturing of OpenFlow and Software-defined Networking Through Deployments. *Computer Networks*. vol. 61. March 2014: pp. 151–175. ISSN 1389-1286.

[38] Lakshman, T. V.; Stiliadis, D.: High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *ACM SIGCOMM Computer Communication Review*. vol. 28, no. 4. October 1998: pp. 203–214. ISSN 0146-4833.

[39] Le, H.; Prasanna, V. K.: Scalable Tree-Based Architectures for IPv4/v6 Lookup Using Prefix Partitioning. *IEEE Transactions on Computers*. vol. 61, no. 7. July 2012: pp. 1026–1039. ISSN 0018-9340.

[40] Lim, H.; Lee, N.; Jin, G.; et al.: Boundary Cutting for Packet Classification. *IEEE/ACM Transactions on Networking*. vol. 22, no. 2. April 2014: pp. 443–456. ISSN 1063-6692.

[41] Matoušek, J.; Antichi, G.; Lučanský, A.; et al.: ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. May 2017. pp. 204–216. ISBN 978-1-5090-6386-4.

[42] Matoušek, J.; Skačan, M.; Kořenek, J.: Memory Efficient IP Lookup in 100 Gbps Networks. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE. September 2013. pp. 1–8. ISBN 978-1-4799-0004-6.

[43] Matoušek, J.; Skačan, M.; Kořenek, J.: Towards Hardware Architecture for Memory Efficient IPv4/IPv6 Lookup in 100 Gbps Networks. In *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. IEEE. April 2013. pp. 108–111. ISBN 978-1-4673-6136-1.

[44] McKeown, N.; Anderson, T.; Balakrishnan, H.; et al.: OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*. vol. 38, no. 2. April 2008: pp. 69–74. ISSN 0146-4833.

[45] Overmars, M. H.; van der Stappen, F. A.: Range Searching and Point Location among Fat Objects. *Journal of Algorithms*. vol. 21, no. 3. November 1996: pp. 629–656. ISSN 0196-6774.

[46] Puš, V.: *Packet Classification Algorithms*. PhD. Thesis. Brno University of Technology, Faculty of Information Technology. Brno. 2012.

[47] Puš, V.; Kořenek, J.: Fast and Scalable Packet Classification Using Perfect Hash Functions. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. ACM. December 2009. pp. 229–236. ISBN 978-1-60558-410-2.

[48] Puš, V.; Tobola, J.; Košař, V.; et al.: Netbench: Framework for Evaluation of Packet Processing Algorithms. In *2011 Seventh ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. IEEE Computer Society. October 2011. pp. 95–96. ISBN 978-0-7695-4521-9.

[49] Qi, Y.; Xu, L.; Yang, B.; et al.: Packet Classification Algorithms: From Theory to Practice. In *IEEE INFOCOM 2009*. IEEE. April 2009. pp. 648–656. ISBN 978-1-4244-3513-5.

[50] RIPE Network Coordination Centre: RIS Raw Data. Available online [March 2018]: https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[51] Ruiz-Sánchez, M. A.; Biersack, E. W.; Dabbous, W.: Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*. vol. 15, no. 2. March 2001: pp. 8–23. ISSN 0890-8044.

[52] Singh, S.; Baboescu, F.; Varghese, G.; et al.: Packet Classification Using Multidimensional Cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*. ACM. August 2003. pp. 213–224. ISBN 1-58113-735-4.

[53] Song, H.; Lockwood, J. W.: Efficient Packet Classification for Network Intrusion Detection Using FPGA. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*. FPGA '05. ACM. February 2005. pp. 238–245. ISBN 1-59593-029-9.

[54] Song, H.; Turner, J.; Lockwood, J.: Shape Shifting Tries for Faster IP Route Lookup. In *13th IEEE International Conference on Network Protocols (ICNP'05)*. IEEE. November 2005. pp. 358–367. ISBN 0-7695-2437-0.

[55] Song, H.; Turner, J. S.: ABC: Adaptive Binary Cuttings for Multidimensional Packet Classification. *IEEE/ACM Transactions on Networking*. vol. 21, no. 1. February 2013: pp. 98–109. ISSN 1063-6692.

[56] Srinivasan, V.; Suri, S.; Varghese, G.: Packet Classification Using Tuple Space Search. *ACM SIGCOMM Computer Communication Review*. vol. 29, no. 4. August 1999: pp. 135–146. ISSN 0146-4833.

[57] Srinivasan, V.; Varghese, G.: Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*. vol. 17, no. 1. February 1999: pp. 1–40. ISSN 0734-2071.

[58] Srinivasan, V.; Varghese, G.; Suri, S.; et al.: Fast and Scalable Layer Four Switching. *ACM SIGCOMM Computer Communication Review*. vol. 28, no. 4. October 1998: pp. 191–202. ISSN 0146-4833.

[59] Sun, Q.; Huang, X.; Yang, W.; et al.: ClassBenchv6: An IPv6 Packet Classification Benchmark. In *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. IEEE. November 2009. pp. 1–6. ISBN 978-1-4244-4148-8.

[60] Taylor, D. E.: Survey and Taxonomy of Packet Classification Techniques. *ACM Computing Surveys*. vol. 37, no. 3. September 2005: pp. 238–275. ISSN 0360-0300.

[61] Taylor, D. E.; Turner, J. S.: ClassBench: A Packet Classification Benchmark. Technical Report WUCSE-2004-28. Washington University in Saint Louis. Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130, USA. May 2004.

[62] Taylor, D. E.; Turner, J. S.: Scalable Packet Classification using Distributed Crossproducing of Field Labels. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE. March 2005. pp. 269–280. ISBN 0-7803-8968-9.

[63] Taylor, D. E.; Turner, J. S.: ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*. vol. 15, no. 3. June 2007: pp. 499–511. ISSN 1063-6692.

[64] Vamanan, B.; Voskuilen, G.; Vijaykumar, T. N.: EffiCuts: Optimizing Packet Classification for Memory and Throughput. *ACM SIGCOMM Computer Communication Review*. vol. 40, no. 4. August 2010: pp. 207–218. ISSN 0146-4833.

[65] Waldvogel, M.; Varghese, G.; Turner, J.; et al.: Scalable High Speed IP Routing Lookups. *ACM SIGCOMM Computer Communication Review*. vol. 27, no. 4. October 1997: pp. 25–36. ISSN 0146-4833.

[66] Wang, M.; Deering, S.; Hain, T.; et al.: Non-random Generator for IPv6 Tables. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*. IEEE. August 2004. pp. 35–40. ISBN 0-7803-8686-8.

[67] Zheng, K.; Liu, B.: V6Gene: A Scalable IPv6 Prefix Generator for Route Lookup Algorithm Benchmark. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA'06)*, vol. 1. IEEE. April 2006. ISBN 0-7695-2466-4.

[68] Zilberman, N.; Audzevich, Y.; Covington, G. A.; et al.: NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*. vol. 34, no. 5. September 2014: pp. 32–41. ISSN 0272-1732.