BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

NETWORK TRAFFIC GENERATOR FOR TESTING OF PACKET CLASSIFICATION ALGORITHMS

GENERÁTOR SÍŤOVÉHO PROVOZU PRO TESTOVÁNÍ KLASIFIKAČNÍCH ALGORITMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. DAVID JANEČEK**
AUTOR PRÁCE

SUPERVISOR — **Ing. JIŘÍ MATOUŠEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Department of Computer Systems (DCSY) Academic year 2019/2020

Master's Thesis Specification

20499

Student:	**Janeček David, Bc.**
Programme:	Information Technology Field of study: Computer Networks and Communication
Title:	**Network Traffic Generator for Testing of Packet Classification Algorithms**
Category:	Networking

Assignment:

1. Become acquainted with the TCP/IP architecture, network layer protocols IPv4 and IPv6, and the OpenFlow protocol.
2. Study various approaches to benchmarking of packet classification algorithms and analyze existing tools for generation of test data utilized in this area.
3. Design a simple tool for generation of a packet trace, which could be used for dynamic analysis of the packet classification algorithm utilizing a rule set generated by the ClassBench-ng tool.
4. Implement the tool and evaluate its run time and memory consumption for various input rule sets. For each rule set also determine its coverage by a generated packet trace.
5. Based on the performed evaluation, design and implement several optimizations of the simple tool.
6. Evaluate optimized versions of the tool and compare them with the initial version.
7. Select the best optimization according to the evaluation results and compare it with Trace Generator, which is part of the ClassBench tool suite.
8. Discuss the obtained results.

Recommended literature:

- D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," in IEEE/ACM Transactions on Networking, vol. 15, no. 3, pp. 499-511, June 2007.
- J. Matoušek, G. Antichi, A. Lučanský, A. W. Moore, J. Kořenek, "ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution," in 2017 ACM/IEEE Symposium on Architectures for Networking and Communications, IEEE CS, 2017, pp. 204-216.

Requirements for the semestral defence:

- Items 1 to 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Matoušek Jiří, Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2019
Submission deadline:	May 20, 2020
Approval date:	October 25, 2019

Abstract

Efforts to improve classification algorithms are being slowed down by lack of data required for testing. For confidentiality and security reasons it is difficult to obtain real data. Good rule set generation tools, such as ClassBench-ng, exist. However, in order to evaluate proper functioning, throughput, power consumption, and other properties of packet classification algorithms, it is necessary to also use network traffic. Subject of this thesis is creating a network traffic generator that would allow for testing of such properties using IPv4, IPv6, and OpenFlow1.0 rules created by ClassBench-ng. The work explores different ways to achieve this, which resulted in several versions of the generator. Those were experimented with and evaluated. Implementation was done using Python. The primary result is a generator combining multiple approaches to achieve the best properties of created header traces. Another contribution of this thesis is a tool that was necessary to create for analyzing rule sets and evaluating generated header traces.

Abstrakt

Pokrok při zdokonalování klasifikačních algoritmů je zpomalován nedostatkem dat potřebných pro testování. Reálná data je obtížné získat z důvodu bezpečnosti a ochrany citlivých informací. Existují však generátory syntetických sad pravidel, jako například ClassBench-ng. K vyhodnocení správného fungování, propustnosti, spotřeby energie a dalších vlastností klasifikačních algoritmů je zapotřebí také vhodný síťový provoz. Tématem této práce je tvorba takového generátoru síťového provozu, který by umožnil testování těchto vlastností v kombinaci s IPv4, IPv6 a OpenFlow1.0 pravidly vygenerovanými ClassBench-ng. Práce se zabývá různými způsoby, jak toho dosáhnout, které vedly k vytvoření několika verzí generátoru. Vlastnosti jednotlivých verzí byly zkoumány řadou experimentů. Implementace byla provedena pomocí jazyku Python. Nejvýznamnějším výsledkem je generátor, který využívá principů několika zkoumaných přístupů k dosažení co nejlepších vlastností. Dalším přínosem je nástroj, který bylo nutné vytvořit pro analýzu užitých sad klasifikačních pravidel a vyhodnocení vlastností vygenerovaného síťového provozu.

Keywords

Network traffic generation, packet classification, computer networks, OpenFlow, ClassBench-ng.

Klíčová slova

Generování síťového provozu, klasifikace paketů, počítačové sítě, OpenFlow, ClassBench-ng.

Reference

JANEČEK, David. *Network Traffic Generator for Testing of Packet Classification Algorithms.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Matoušek, Ph.D.

Rozšířený abstrakt

Internet se ve velké části světa stal nezbytnou službou, která významně ovlivňuje každodenní životy lidí. Ačkoliv se Internet od svého zrodu významně změnil ve způsobu využití i v jeho dostupnosti, mnohé základní principy zůstávají i po letech stejné. Jedním takovým příkladem je klasifikace paketů, která je klíčovou úlohou na síťových zařízeních. Klasifikace paketů se využívá například při směrování, nebo třídění paketů, což z ní dělá jednu z nejběžnějších síťových operací.

Internetové protokoly, kterými se celý Internet řídí, se neustále vyvíjejí a stejně jako struktura jednotlivých sítí se stávají složitějšími. Tyto faktory a ustavičně se zvyšující propustnost sítí kladou zvýšené nároky i na algoritmy sloužící ke klasifikaci paketů. Nové, potenciálně lepší, algoritmy je třeba řádně otestovat před nasazením do ostrého provozu. Takové testování musí být provedeno na množství a typu dat, které odpovídá reálnému provozu. Taková data je ovšem obtížné získat. Z bezpečnostních důvodů a kvůli ochraně osobních údajů nechtějí organizace zveřejňovat detailní informace o síťovém provozu a použitých klasifikačních pravidlech.

Skutečná, nebo alespoň realistická, klasifikační pravidla jsou k testování potřeba, protože výkon a efektivita klasifikačních algoritmů je na nich závislá. Existuje několik nástrojů, které generují syntetické sady klasifikačních pravidel. Nejnovější z nich, ClassBench-ng, dokáže analyzovat skutečné sady pravidel a na jejich základě vytvořit pravidla syntetická. To umožňuje zachovat vlastnosti původních pravidel bez jejich zveřejnění.

K vyhodnocení správného fungování, propustnosti, spotřeby energie, efektivity využívání mezipaměti a dalších technik klasifikačních algoritmů na síťových zařízeních je zapotřebí také síťový provoz. Konkrétně hodnoty relevantních hlaviček paketů. Vytvoření generátoru takového síťového provozu je hlavním cílem této práce.

Existují tři hlavní nástroje sloužící k testování klasifikačních algoritmů. Nejstarší z nich je ClassBench, který umožňuje analyzovat skutečné sady pravidel, vytvářet syntetické sady pravidel a umožňuje i generovat síťový provoz. Jeho největší nevýhodou je, že umí pracovat pouze s péticemi IPv4 (zdrojová a cílová IP adresa, zdrojový a cílový port a protokol vyšší vrstvy). Druhým nástrojem je FRuG, který je do veliké míry přizpůsobitelný uživatelem. Jeho hlavní výhodou je, že sady pravidel, které generuje, nejsou nijak omezeny počtem použitých hlavičkových polí. Je ovšem stále omezen pouze na IPv4, a navíc neumožňuje generovat síťový provoz. Nejnovějším z nástrojů je ClassBench-ng. Jak už název napovídá, je to nástroj, který do jisté míry navazuje na původní ClassBench. Přichází s několika vylepšeními analýzy a generování klasifikačních pravidel. Dále rozšiřuje funkcionalitu o práci s IPv6 a hlavičkovými poli OpenFlow1.0.0. Neobsahuje však generátor síťového provozu. Jelikož zbývající vlastnosti ClassBench-ng předčí ostatní nástroje, je generátor tvořený v rámci této práce navržen tak, aby byl kompatibilní právě s ClassBench-ng.

Požadovaný generátor má na vstupu sadu klasifikačních pravidel a požadovaný počet hlaviček. Výstupem jsou hodnoty jednotlivých polí hlaviček. Před vytvořením generátoru síťového provozu, který by adekvátně testoval klasifikační algoritmy bylo zapotřebí vytvořit jeho jednoduchou verzi. Jedná se o generátor, který splňuje veškerou základní funkcionalitu, ale hodnoty hlaviček generuje bez jakékoliv optimalizace vzhledem k žádoucím vlastnostem generovaného provozu. Je ovšem nutné, aby vytvořené hlavičky spadaly alespoň do nějakého z pravidel vstupní sady. To je zařízeno tak, že pro každou hlavičku je náhodně vybráno jedno z pravidel. Hodnoty hlavičkových polí jsou poté vybrány náhodně z hodnot povolených tímto pravidlem. Tento i všechny další verze generátoru jsem implementoval v jazyce Python.

V ideálním případě by generátor měl vygenerovat alespoň jednu hlavičku pro každou odlišnou oblast ve vstupní sadě pravidel. Odlišnou oblastí se myslí každá část sady pravidel, která je tvořena unikátní kombinací pravidel. To může být buď jedno individuální pravidlo, nebo překryv více pravidel. Je možné provést kompletní analýzu sady pravidel, která identifikuje všechny tyto oblasti, a vygenerovat právě jednu hlavičku pro každou oblast. V závislosti na počtu pravidel a překryvů mezi nimi se tento přístup může stát neuskutečnitelným kvůli času potřebnému pro všechny výpočty a vyžadovanému množství paměti. Je tedy třeba najít rozumný kompromis mezi pokrytím oblastí a výpočetní náročností.

Autoři původního nástroje ClassBench použili metodu, která se jen lehce liší od zmíněného jednoduchého generátoru. Pro každou hlavičku je náhodně vybráno jedno z pravidel, ale hodnoty jednotlivých hlavičkových polí nejsou vybrány ze všech povolených hodnot tímto pravidlem. Místo toho je vždy vybrána buď maximální nebo minimální hodnota. Výsledkem jsou tedy hlavičky odpovídající nějakému z “rohů” daného pravidla. Tento přístup jsem zreplikoval a rozšířil o fungování s IPv6 a OpenFlow hlavičkami. Ukázalo se, že pokrytí oblastí touto metodou není lepší než jednoduchý generátor.

Po vytvoření, implementaci a vyhodnocení řady různých přístupů jsem dospěl ke generátoru, který funguje následovným způsobem. Nejprve v náhodném pořadí projde všechna pravidla a pro každé z nich vygeneruje jednu hlavičku stejným způsobem jako jednoduchý generátor. To slouží především k pokrytí oblastí tvořených jedním pravidlem. Zbývající hlavičky jsou generovány způsobem zaměřeným na pokrytí překryvů. Hlavní myšlenkou algoritmu je tvoření takových hlaviček, které splňují podmínky více pravidel najednou.

K testování jednotlivých algoritmů byly využity sady pravidel vygenerované nástrojem ClassBench-ng. Ty svými vlastnostmi odpovídají reálným sadám pravidel, jelikož byly vytvořeny na základě jejich analýz. K vyhodnocení výsledků jsem v jazyce Python vytvořil nástroj, který analyzuje danou sadu klasifikačních pravidel a poté spočítá její pokrytí vygenerovanými hlavičkami.

Výsledný generátor síťového provozu poskytuje lepší pokrytí oblastí než všechny ostatní testované metody. V tomto ohledu je výrazně lepší než metoda z nástroje ClassBench, což bylo hlavním cílem. Z hlediska výkonu je generátor podstatně pomalejší. Jeho časová složitost je však lineární, což je uspokojující.

Network Traffic Generator for Testing of Packet Classification Algorithms

Declaration

Hereby I declare that this master's thesis was prepared as my original work under the supervision of Ing. Jiří Matoušek, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
David Janeček
June 2, 2020

Acknowledgements

I would like to thank my supervisor Ing. Jiří Matoušek, Ph.D., for all the useful advice and suggestions he provided me with. As well as granting me access to computational resources to run numerous experiments needed for the thesis.

Contents

Chapter 1

Introduction

In many places in the world, the Internet has become one of vital services that people can hardly imagine living without. And even though it has changed significantly since its inception, a lot of its core functionality remains working on the same principles. One such example is packet classification. It is used for routing, packet filtering and other applications, which makes it one of the most common operations in computer networks.

Internet protocols, that govern the Internet, and the structure of individual computer networks are constantly becoming more complex, which raises requirements on packet classification algorithms. That is further amplified by continuously increasing throughput of computer networks. New, potentially better, algorithms cannot be simply deployed to live traffic. That could have disastrous consequences. They need to be properly tested first. Testing them on realistic amounts and types of data is complicated. The absence of environment that would allow this contributes to so called ossification of Internet infrastructure.

The first problem with testing packet classification algorithms is that it is not easy to get rule sets that are being applied to real traffic [17]. Organizations do not want to publish this data for confidentiality and security reasons. Real, or at least realistic, rule sets are needed because capacity and efficiency of packet classification algorithms are subject to the structure of the rule sets [9]. There already are some existing tools that set out to combat this by generating synthetic rule sets. The other problem is the lack of the actual traffic. As a matter of fact, it is not the traffic that is needed. It is only some of the header fields of the packets. These are needed for evaluation of proper functioning, throughput, power consumption, and effectiveness of caching and other techniques of devices employing the classification algorithm [17]. Again, there have been attempts to solve this, but these solutions are not sufficient considering current demands.

The first goal of this thesis is to create a header trace generator, which could be used for dynamic analysis of the packet classification algorithm utilizing a rule set generated by ClassBench-ng. ClassBench-ng is currently the best tool for creating synthetic rule sets, but it has no network traffic generation functionality. The header generator is aimed to be compatible with ClassBench-ng, so that it could eventually be incorporated in it.

The second goal is to optimize the generator to achieve the best possible coverage for various rule sets while maintaining reasonable computational demands. A rule is considered to be covered if at least one generated header satisfies the rule's conditions. For proper testing of classification algorithms it is not enough to cover individual rules. Instead, it is desirable to also generate at least one header for all distinct regions in the rule set that are formed by overlaps between the rules.

The following chapter, Chapter 2, introduces information regarding computer networks with focus on topics that are most relevant to the thesis. All protocols that contain header fields that ClassBench-ng works with, and therefore that this generator works with, are explained there. Chapter 3 is devoted to packet classification. It defines the term and presents a brief overview of different approaches. Last section of the chapter, section 3.2, examines existing tools that can be used for testing classification algorithms. The process of making the first version of the network traffic generator is described in Chapter 4. This chapter further discusses ways to analyze and evaluate the generated header trace in section 4.2, which is necessary in order to optimize the generator. All of the major improvements that lead to better properties of the generated trace are described in Chapter 5. Final section of the chapter, section 5.7, outlines possible future work that could be done to further advance the generator. The thesis is concluded in Chapter 6.

Chapter 2

Computer Networks

The main aim of this chapter is to introduce the area of computer networking and discuss its parts that are most relevant to the thesis in detail. First, the chapter defines computer network and presents basic terminology. That is followed by origins and brief history of computer networks along with the driving forces behind their evolution. The rest of the chapter contains more detailed information about current state of computer networking. It focuses on the TCP/IP architecture which is the most widely used. It discusses its structure, components, and some important protocols. Both versions of Internet Protocol, and the OpenFlow protocol are discussed in greater detail, as they are the most significant for this thesis.

Computer network is a communications network that interconnects a variety of devices and provides a means for information exchange among them [15]. The connected devices are called hosts or end systems. They are identified by an IP address. Interconnection is done through communication links and packet switches. There are many types of communication links, consisting of different types of physical media and radio spectrum. A packet switch takes a packet arriving on one of its incoming interfaces (links) and forwards it to an outgoing interface toward its destination. Packets are essentially packages containing information that is being sent and information necessary for successful communication (e.g. sender's and receiver's IP addresses). End systems access the Internet through Internet service providers (ISPs). Each ISP is in itself an independently managed network. It interconnects all of its customers' networks, and also connects to other ISPs. These connections ultimately form the Internet [7].

The most common networks are local area networks (LANs). They are present in virtually all office buildings and homes. A LAN consists of a shared transmission medium and a set of hardware and software for interfacing devices to the medium [15]. Whole LAN is often owned by a single organization or person. Its scope is small, usually a single building. Wide area networks (WANs), on the other hand, extend over larger geographical areas. They mostly serve as a connection of LANs and other types of networks.

A special case of LAN is virtual local area network (VLAN). It is a logical network that can group devices even from different physical locations. Hosts within a VLAN communicate with each other as if they were connected to the same switch. Conversely, VLANs can be used to create multiple networks on just one switch, which allows, for example, to create a network for each department in a company without having to buy more networking hardware and changing topology of the physical network. This can also be useful for security and performance of the network [7]. Communication between different VLANs is still possible using system of VLAN tagging. As the name suggests, a tag, that identifies

VLAN to which the frame belongs, is added to the frame. Structure of the tag is discussed further in subsection 2.1.1. Interconnection of two VLANs, over which tagged frames are sent, is called a trunk.

The origins of computer networking can be traced to early 1960s [7]. At the time, a telephone network was the world's dominant communication network. It used circuit switching to transmit information, which is suitable for voice communication since it transmits at a constant rate between a sender and a receiver. Information exchange in a computer network of those times was likely to come in bursts followed by periods of inactivity. Packet switching was invented as an efficient and robust alternative to circuit switching. The first packet-switched computer network was designed in 1969 at the Advanced Research Projects Agency (ARPA) in the United States [7]. It was the first direct ancestor of today's Internet.

It is clear that in order for two computers to communicate through a complex telecommunication network, there must be a high degree of cooperation between them, and also between each of them and the network. Generally, it is required that even devices from different vendors must be capable of communicating with each other. These and many other factors lead to creation of Internet standards. Standards assure that there will be a large open market for equipment and software with a wide variety of vendors, whose products can interface and communicate with each other. There are also a few downsides to standards. Most notably, they slow down the development of new technologies. It takes a lot of time to create, specify and review a standard. By the time it is finished, a better alternative might already exist. Many different groups take interest in Internet standards and participate in their creation. Their goals are often not fully aligned. Negotiating between these groups results in more delays. If things go well, a compromise is reached. In the opposite case, multiple conflicting standards might be created for the same thing, which can lead to further confusion and complications [15]. Each distinct version of an Internet-related standard is published as part of the Request for Comments (RFC) document series. The RFC series of documents on networking began in 1969 as part of the original ARPA wide-area networking project [1]. Nowadays, it is managed by the Internet Engineering Task Force (EITF). In addition to Internet standards, RFCs also cover other topics related to the Internet. Other organizations also specify standards for network components. For example, the IEEE 802 LAN/MAN Standards Committee specifies the Ethernet and wireless WiFi standards [7].

All activity taking place in the Internet that involves multiple communicating parties is governed by protocols. They can be implemented as software, hardware or combination of both. A protocol is a set of rules defining the communication and the structure of its messages. Because the Internet is an incredibly complex system, network designers organize protocols, and their implementations, into layers [7]. In a protocol architecture, the layers are arranged in a vertical stack. Each layer performs certain tasks independently on the other layers. Lower layers provide services to higher layers, but the implementation is hidden to them. That allows for making changes in a layer without interfering with the rest of them. For a successful communication, the same set of layered functions must exist in all participating systems [15].

2.1 TCP/IP Architecture

The TCP/IP architecture is named after its two most important protocols - Transmission Control Protocol and Internet Protocol [12]. The main goal during its creation was to build an interconnection of networks that would provide universal communication services over heterogeneous physical networks. This should allow communication between different networks spread across the entire world. This worldwide set of interconnected networks is called the Internet.

Like most networking software, the TCP/IP architecture consists of layers. TCP/IP's layers are Application, Transport, Internet and Network Interface layer. The term protocol stack refers to the stack of layers. It can be used for positional comparison with other models, such as the Open Systems Interconnection (OSI) model, which consists of seven layers [12]. The comparison of these two models can be seen in Figure 2.1.

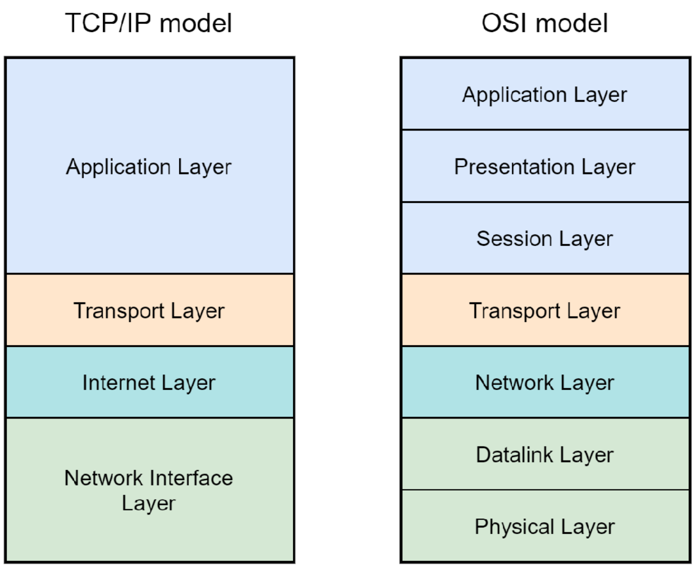

Figure 2.1: Comparison of TCP/IP and OSI models

Separation of functionality into layers allows for easier implementation, testing, and provides space for possible extensions or alternative implementations. Work done by layers is independent, but lower level layers provide functionality for those above them. They communicate through concise interfaces.

When an application wants to send data over a network, first, an application protocol creates a message containing the data. For example, the message can be a HTTP response and its data is HTML code of a website. Next, a transport layer protocol header is added in front of the message to form a packet. Most common are TCP and UDP packets, more on that in subsection 2.1.3. An IP header is placed in front of the packet and the newly formed structure is called IP datagram. Finally, Ethernet frame is created by placing Ethernet header in front of the datagram and frame check sequence, sometimes also called frame footer, behind the end of the datagram. This process is called encapsulation. An example of encapsulation can be seen in Figure 2.2. The entire operation is done in reverse when the frame reaches its destination, where a header is removed at each layer and its information used to get the message to intended application.

2.1.1 Network Interface Layer

The network interface layer is at the lowest position in the stack. It interfaces with the network hardware and allows the traffic to flow over various kinds of physical networks. The most widely used protocol at this layer is Ethernet. There are many reasons for its success. It was the first widely deployed wired LAN technology. Most network adminis-

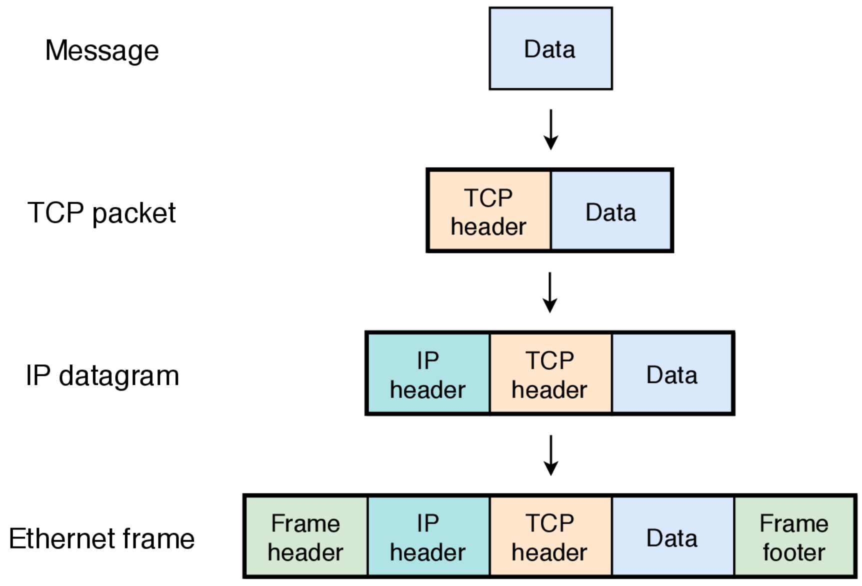

Figure 2.2: Example of data encapsulation

trators became familiar with it and were reluctant to switch to other LAN technologies when they came about. Most other technologies, such as token ring, FDDI, or ATM, were more complex and expensive than Ethernet, which further discouraged from transitioning to them [7]. Ethernet also did not remain stagnant. When new technologies were providing better features and properties, Ethernet adapted through new versions and never stayed behind for long.

Ethernet uses media access control (MAC) addresses for identifying devices participating in communication. MAC address has 48 bits. It serves as a unique identifier of a network interface card (NIC). Apart from some special addresses, the first 24 bits are organizationally unique identifier (OUI), which is used to identify manufacturer of the card. The other 24 bits are assigned by the manufacturer to the specific NIC.

Ethernet frame consists of a preamble, destination and source addresses, type field, data field (payload), and frame check sequence, in that order. Structure of the whole frame is shown in Figure 2.3. Description of its fields can be seen in the list bellow.

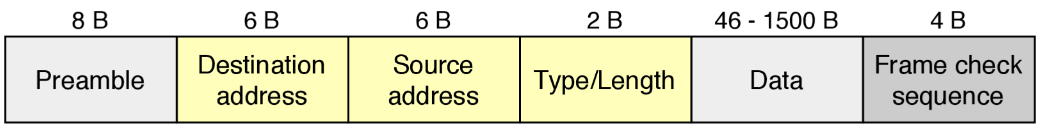

Figure 2.3: Ethernet frame structure

- **Preamble (8 bytes)** This field is always set to the same value: first seven bytes are set to a decimal value of 170, and the last byte to 171. Preamble is used for synchronization and as a "wake up" call for receiving adapters [7].
- **Destination address (6 bytes)** MAC address of network interface card of a receiving device.

- **Source address (6 bytes)** MAC address of network interface card of a sending device.
- **Type/length (2 bytes)** The type field is used to indicate which protocol is encapsulated in the payload of the frame. It is also used to specify the size of certain Ethernet frames.
- **Data (46 - 1500 bytes)** The data field, often referred to as payload, can be anywhere between 46 and 1 500 bytes long. It consists of the information being sent and headers of higher layer protocols included in the frame. If the payload's size is below the minimum required size, then it has to be padded to reach the 46 bytes.
- **Frame check sequence (4 bytes)** The frame check sequence field is a type of cyclic redundancy check (CRC). It allows receiver of the message to detect bit errors in the frame.

Ethernet technology provides an unreliable connectionless service to the internet layer. Data that needs to be sent is encapsulated by the sender into an Ethernet frame, which is then sent without prior handshaking, or connecting in any other way, with the receiver. A receiving device runs a CRC check, but does not inform sending device about its result. If CRC check fails, the receiver simply discards the frame [7].

Networking standard IEEE 802.1Q defines an extended Ethernet frame format for frames crossing VLAN trunks. The IEEE 802.1Q frame is created by inserting 802.1Q header between the source MAC address field and the type field of a normal Ethernet frame. Frame check sequence also has to be recalculated. As shown in figure Figure 2.4, the 802.1Q header consists of tag protocol identifier (TPID) and tag control information (TCI). The TCI is further split into three fields - priority, drop eligible indicator, and VLAN identifier [6, 7].

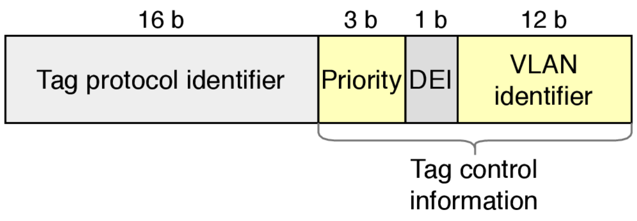

Figure 2.4: 802.1Q header structure

- **Tag protocol identifier (16 bits)** The TPID field is used to identify the frame as a tagged frame. It is set to a fixed hexadecimal value of 81-00.
- **Priority (3 bits)** Priority is a quality of service parameter. Frames with high priority may be given precedence.
- **Drop eligible indicator (1 bit)** The DEI field is used to indicate if a frame is eligible to be dropped during traffic congestion.
- **VLAN identifier (12 bits)** This field contains a number identifying VLAN from which the frame has been sent.

2.1.2 Internet Layer

The internet layer, sometimes also called inter-network or network layer, shields the higher layers from the physical form of networks [12]. Instead, it provides their virtual view. It implements host-to-host communication service and contains functions for forwarding and routing. The internet layer is one of the most complex in the protocol stack.

When a packet arrives at a router's input interface, it determines on which output interface the packet should be sent. This is called forwarding. Routing is a process of determining route from a sender to a receiver through an entire network [7]. Algorithms used for calculating these routes are referred to as routing algorithms. Routers make decisions based on information in their routing and forwarding tables. Those can be set up by administrators or by already mentioned routing algorithms. The main advantage of using algorithms is that routers can exchange routing information with each other and because of that react to changes in network topology.

An internet layer packet is referred to as a datagram. Its structure differs based on the protocol that is being used. The most common is Internet Protocol (IP), which comes in two versions, IPv4 and IPv6. IPv6 was created to replace IPv4, but the transition is taking some time. These two protocols are, due to their complexity, discussed further in section 2.2.

Internet layer provides connectionless best-effort services. Reliability, flow control, and error recovery, if needed, must be provided by higher level layers [12].

2.1.3 Transport Layer

The transport layer builds on functionality provided by the internet layer and plays a central role in network architecture. It delivers data directly to applications on different hosts using ports and sockets. It can provide additional functionality such as congestion control, reliable data delivery, duplicate data suppression, and flow control [12].

Ports and sockets are used to determine which process, at a given host, is communicating. Each process that wants to communicate with another identifies itself by one or more ports. Port is a 16-bit number which provides us with a range from 0 to 65 535. The first 1 024 numbers are reserved for privileged services and designated as so-called well-known ports [12]. Well-known ports belong to standard services, such as Telnet or FTP (File Transfer Protocol). They are controlled and assigned by the Internet Assigned Number Authority (IANA) and allow clients to find services without further configuration information. Processes access network services through sockets. A socket is a special type of file handle, which is requested by processes from an operating system.

The two most important protocols operating on the transport layer are User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). UDP is essentially an application interface to IP. It provides an unreliable, connectionless service to the invoking application. No functionalities such as reliability, flow control, or error correction are present. UDP simply maps incoming traffic based on port numbers to correct processes. Similarly, it assigns port numbers to outgoing traffic based on the processes. That allows one application to communicate with another. Thanks to its simplicity, UDP is very efficient, but it requires the application to take responsibility for any other needed functionality, such as error recovery [12]. Each UDP segment (packet) is sent with a single IP datagram. UDP packets have an 8-byte header that is shown in Figure 2.5 It contains source and destination port numbers, length of the packet including the header, and checksum.

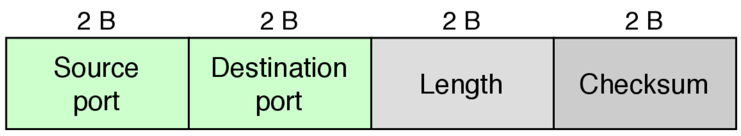

Figure 2.5: UDP header structure

Transmission Control Protocol, on the other hand, provides reliable, connection-oriented service to the invoking application. It also provides facilities, such as error recovery and flow control. Before applications start communicating with each other, they have to establish a connection. They do that by exchanging preliminary segments to establish parameters of the ensuing data transfer. This process is called a "handshake" [7]. A TCP connection is always point-to-point and provides a full-duplex service, which means that data can flow in both directions simultaneously.

Since TCP provides more functionality than UDP, its header needs to carry more information. It can be seen in Figure 2.6. Individual header fields are explained in the following list.

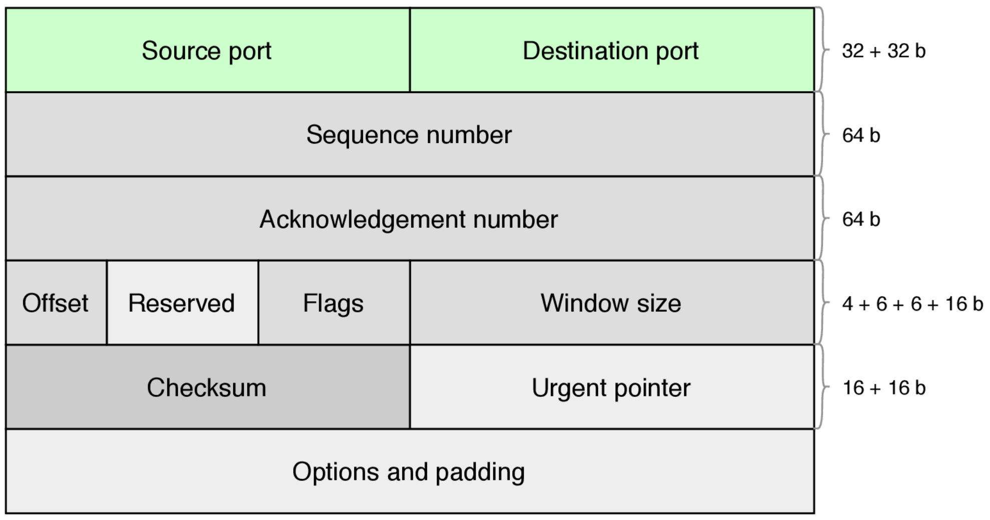

Figure 2.6: TCP header structure

- **Source port (32 bits)** Source port number.
- **Destination port (32 bits)** Destination port number.
- **Sequence number (64 bits)** If the SYN control bit (one of flags discussed under the Flags field) is set, the sequence number is the initial sequence number. Otherwise it is the sequence number of the first data byte in this segment.
- **Acknowledgement number (64 bits)** The acknowledgement number field is only relevant if the ACK control bit (one of flags also discussed under the Flags field) is set. In that case this field contains the value of the next expected sequence number.
- **Offset (4 bits)** The offset field indicates the position of the beginning of data.

- **Reserved (6 bits)** These six bits are reserved for possible future use and should always be set to zero.
- **Flags (6 bits)** The flags field consists of six control bits, also commonly called flags. Purpose of two of them (SYN and ACK) was already mentioned. The other four are URG, PSH, RST, and FIN. If URG is set, then the urgent pointer field is significant in this segment. The PSH flag is an option that allows the sending application to start sending data even if its size is smaller than the minimum transmission unit. RST is used to reset the connection. Finally, if FIN is set, then the sender has no more data to send.
- **Window size (16 bits)** The window size contains a number representing the amount of bytes that the sender is willing to accept.
- **Checksum (16 bits)** In TCP checksum is calculated using both header and data while the checksum itself is considered to be zero.
- **Urgent pointer (16 bits)** The urgent pointer field is only significant if the URG flag is set, and it points to data that is urgently required.
- **Options (multiple of 8 bits) and padding (variable)** Options may occupy space at the end of the TCP header and must be a multiple of 8 bits in length. They are used for specific features enhancing the TCP protocol. Padding is used to ensure that the TCP header ends, and data begins, on a 32-bit boundary. The padding is composed of zeros.

2.1.4 Application Layer

The application layer lies at the very top of the TCP/IP model. It contains logic needed to support various network applications and builds on everything provided by the lower level layers. An application is a user process, usually cooperating with another process on a different host. There are also use cases for communicating applications on the same host. The interface between the application and the transport layers is defined by port numbers and sockets [12]. Those are described in subsection 2.1.3.

2.2 Internet Layer Protocols

Internet Protocol was designed for use in interconnected systems of packet-switched computer communication networks [14]. It provides means for transmitting blocks of data called datagrams. Source and destination hosts of such communication are identified by fixed-length addresses. Currently, two versions of Internet Protocol are being used - version 4 and 6.

The purpose of Internet Protocol is to move datagrams through an interconnected set of networks. That is done by passing the datagrams from one point to another until the destination is reached. The route of the datagrams is determined based on their addresses. It can happen that a network might have its maximum packet size smaller than the size of the datagram. To overcome this, the datagram has to be fragmented (split) into smaller parts. Those two functions, addressing and fragmentation, are the two most fundamental functions of Internet Protocol [14].

2.2.1 IPv4

IPv4 utilizes fixed-length addresses of 32 bits. An address consists of two parts. The first part is a number representing a network. The other part is a number representing a host in the given network. The number of bits that belong to each part can differ. It is specified by a network mask in case of classless addressing or by a class of the address in the original classful addressing scheme which is not being used anymore. There are also address ranges reserved for special purposes.

The protocol's header contains all the information required for its functionality. Structure of an IPv4 header is shown in Figure 2.7. Individual header fields are explained in the following list.

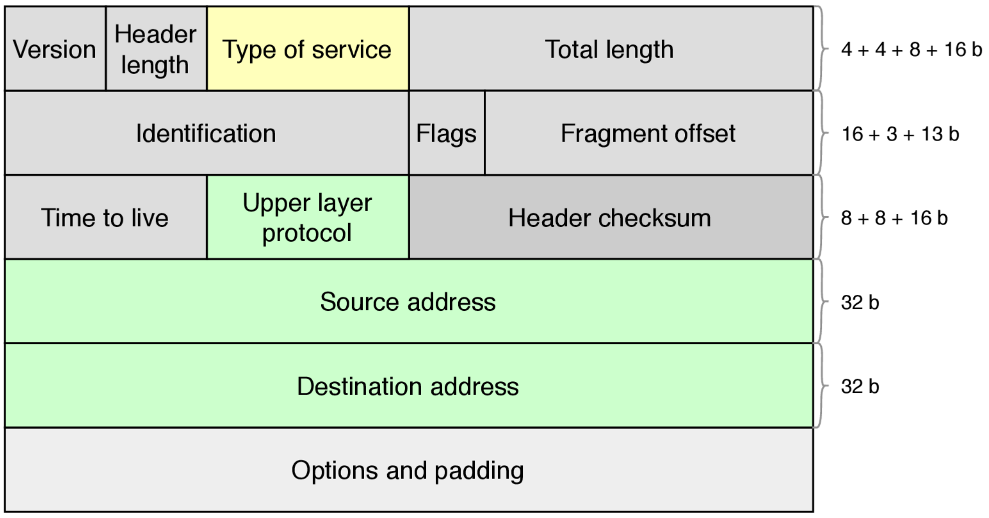

Figure 2.7: IPv4 header structure

- **Version (4 bits)** This field specifies the version of a utilized protocol of the datagram. It is used by routers and other devices to determine how to interpret the remainder of the header.
- **Header length (4 bits)** Because the length of the IPv4 header is not fixed, it is necessary to specify where the header ends and data begins. That is determined, in part, by this value specifying the length of the header in bytes.
- **Type of service (8 bits)** The type of service header field is an 8-bit value introduced to allow and identify different types of IP datagrams.
- **Total length (16 bits)** The total length field indicates a total length of the IP datagram in bytes. That is the length of the header plus the length of data. Since it is 16 bits long, the theoretical maximum size is 65 535, but datagrams are usually not larger than 1 500 bytes.
- **Identification (16 bits)** The identification field is used to uniquely identify a group of fragments belonging to a single IP datagram.
- **Flags (3 bits)** The first of the three flag bits is reserved and must always be set to zero. The second bit is set if a sender wants to prevent fragmentation of the datagram.

The last bit, if set, signals that this is not the last part of an incoming fragmented datagram.

- **Fragment offset (13 bits)** The fragment offset field identifies the fragment location, relative to the beginning of the original unfragmented datagram.
- **Time to live (8 bits)** The time to live (TTL) field is a number which ensures that there will be no datagrams circulating in a network forever, regardless of network's topology. TTL is decremented by one every time the datagram is processed by a router. If it reaches zero, then the datagram must be discarded.
- **Upper layer protocol (8 bits)** The upper layer protocol field is relevant only in packet's final destination. Its value identifies the specific transport layer protocol.
- **Header checksum (16 bits)** The header checksum field is used to detect bit errors. Routers check it and usually discard faulty datagrams.
- **Source address (32 bits)** IP address of the original source of the datagram.
- **Destination address (32 bits)** IP address of the final destination of the datagram.
- **Options and padding (variable)** The options field is optional. There might be zero or more options in an IPv4 datagram. They can be used to convey some extra information about the traffic. If the options do not end on a 32-bit boundary, then padding (octets of zeros), is appended to the end of header to make it so.

2.2.2 IPv6

Internet Protocol version 6 was designed as the successor to IPv4. A prime motivation for this effort was the realization that the 32-bit IP address space was beginning to be depleted. That makes IPv6's expanded addressing capabilities one of the most important improvements. IP address size is increased from 32 to 128 bits, which allows support of more levels of addressing hierarchy and a much greater number of unique addresses. Several header fields have been removed or made optional. Some other fields have also been added, but overall the header format was simplified and allows for faster processing and handling of packets. IPv6 encodes header options in a different way which allows for more efficient forwarding. There are also less strict limits on the length of options and greater flexibility for introducing new options in the future. Extensions to support authentication, data integrity, and data confidentiality have also been added to IPv6 [3].

Addressing in IPv6 is much less complicated than the one in IPv4. IPv6 addresses are assigned to interfaces, not nodes. There are three types of addresses: unicast, anycast, and multicast. A unicast address identifies a single network interface. A packet sent to such address is delivered to the interface specified by that address. Every interface is required to have at least one unicast address. An anycast address identifies a set of interfaces. A packet sent to such address is delivered to the closest one of the interfaces identified by that address. The definition of distance depends on a used routing protocol. A multicast address identifies a set of interfaces. A packet sent to such address is delivered to all of the interfaces identified by that address. Typically, unicast and anycast addresses are composed of two logical parts. The first 64 bits define a network prefix used for routing. The other 64 bits identify specific interface in the given network. Multicast addresses are formed in a different way dependent on the application. Several of them are predefined for special purposes [5].

Optional internet layer information is encoded in separate headers. Those are called extension headers and they are placed between the IPv6 header and the upper layer header in a datagram. The type of an extension header is identified by a distinct value of the next header field. Values identifying extension headers and upper layer protocols do not overlap, which means that the next header field can also be used to indicate if the next item in the datagram is an extension header or not. A special "no next header" value is used if there is no upper layer header [3].

Structure of an IPv6 header is shown in Figure 2.8. A list of individual header fields with explanations can be found below.

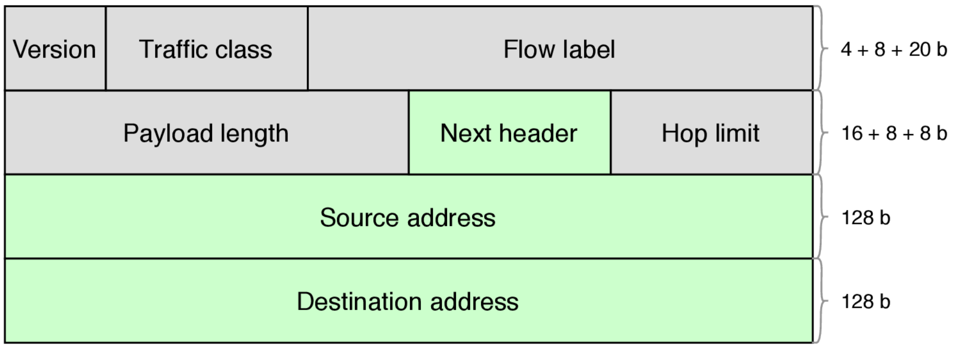

Figure 2.8: IPv6 header structure

- **Version (4 bits)** This field specifies the version of a utilized IP protocol. In case of IPv6 that number is, unsurprisingly, 6. It is used by routers and other devices to determine how to interpret the remainder of the header.

- **Traffic class (8 bits)** The traffic class field is used by the network devices for traffic management.

- **Flow label (20 bits)** The flow label field is used by a source of traffic to label sequences of datagrams that should be treated in the network as a single flow.

- **Payload length (16 bits)** The payload length field is a number specifying the length of datagram's payload in octets. The length of the payload is specified as everything in the datagram following this IPv6 header, which includes any present extension headers.

- **Next header (8 bits)** The next header field serves as an identifier of the type of header immediately following the IPv6 header. It is essentially the same as the upper layer protocol field in IPv4.

- **Hop limit (8 bits)** Hop limit works like TTL in IPv4. It is a positive number that is decremented by one on each node that forwards the datagram. A datagram is discarded if its hop limit reaches zero. A node that is the destination of a datagram should not discard a datagram with hop limit equal to zero, it should process it normally instead.

- **Source address (128 bits)** An IPv6 address of the originator of the datagram.

- **Destination address (128 bits)** An IPv6 address of the intended recipient of the datagram.

2.2.3 ICMP

The Internet Control Message Protocol (ICMP) is an auxiliary protocol to IP. Its purpose is to provide feedback about problems in the communication environment [13]. It does not guarantee that a datagram will be delivered or a control message will be returned, higher level protocols must be used in order to achieve that. The ICMP messages usually report errors in processing of a datagram. For example when a datagram cannot reach its destination, or when the router does not have enough buffering capacity to forward a datagram. In order to avoid creating infinite loops of messages, no ICMP messages are sent about other ICMP messages. Functionality provided by ICMP is used in several common network utilities, such as traceroute or ping. Ping uses echo request and echo reply ICMP messages to measure the round-trip time for messages sent from one device to another. Traceroute is a tool used to determine the path datagrams follow to reach a specified host. For that, it utilizes ICMP time exceeded messages sent by routers when received datagram's TTL value reaches zero.

ICMP messages are sent using the normal IP header. The information that they convey is indicated by the message's type and code. Type represents the main piece of information, while code can be used for further clarification. For example, let's have an ICMP message with type 3 and code 3. The type tells us the destination is unreachable and the code tells us that it is so because the port in the destination is unreachable [13]. Note that this information does not always have to be completely correct due to usage of firewalls and other systems interacting with the traffic.

ICMPv6

The Internet Control Message Protocol for the Internet Protocol Version 6 (ICMPv6) is an integral part of IPv6 and must be fully implemented by every IPv6 node. Just like its predecessor, ICMPv6 is used to report errors encountered in processing datagrams [2]. It was created by taking the original ICMP and applying it to IPv6 with a number of changes. ICMPv6 messages are grouped into two classes. These are error messages, with types from 0 to 127, and informational messages with types from 128 to 255. ICMPv6 has an IPv6 next header value of 58.

2.3 OpenFlow

OpenFlow is a protocol that allows remote administration of switches and routers of different vendors. It does that by programming flow tables in these devices. Even though each vendor's flow tables are different, the devices have a set of common functions that run in them. One of main motivations behind OpenFlow was to allow a practical way for experimentation with new network protocols in realistic settings [10]. This is needed to give new ideas a chance to break through and be widely deployed. With OpenFlow, researches can control their own network traffic flows and experiment with them without influencing other traffic.

The remote administration of switches and routers is done using another device called controller. A controller is a process running on a computer that is connected to the networking device via a secure channel. Packets sent between them are using the OpenFlow protocol, which provides an open and standard way of communication. There can be dedicated OpenFlow switches that do not support normal Internet Layer and Network Interface Layer processing, but also general-purpose commercial switches and routers interfacing with the OpenFlow protocol [10].

The original version of the OpenFlow protocol worked with twelve header fields [11] that are shown in Figure 2.9. From Ethernet, these include source and destination MAC addresses and the type field. From IP, it utilizes source and destination IP addresses, type of service, and upper layer protocol fields. The Transport layer provides source and destination port numbers. On top of these, OpenFlow also works with the ingress port number and VLAN information, specifically VLAN id and VLAN priority. Ingress port is simply the number of the port on which a packet arrived. VLAN information is included in special IEEE 802.1Q Ethernet frames which are discussed in subsection 2.1.1. Newer versions of OpenFlow include more header fields. The generator of network traffic that is the product of this thesis works only with those from version 1.0.0, as they are the ones that ClassBench-ng works with.

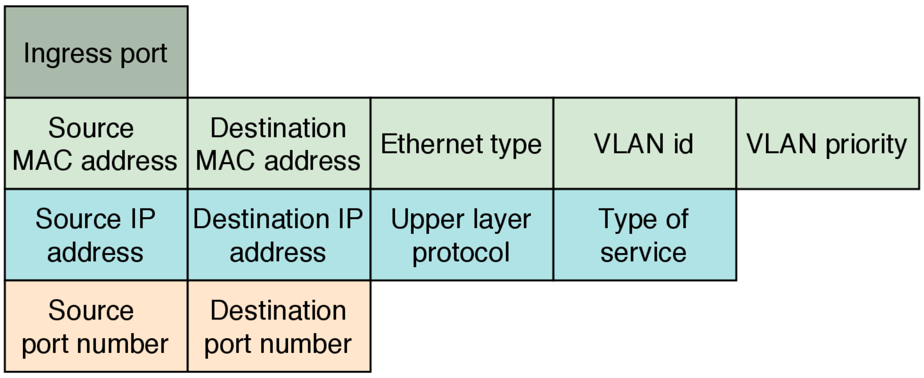

Figure 2.9: OpenFlow 1.0.0 header fields

Chapter 3

Packet Classification

This chapter serves as an introduction to packet classification. First, it explains the term and presents reasons why it is needed. That is followed by a short list of some of the most common approaches to packet classification in section 3.1. Finally, section 3.2 focuses on tools that can be used for testing of packet classification algorithms. These tools are capable of creating synthetic rule sets that imitate properties of real rule sets.

The process of packet classification is determining a class that a packet belongs to [8]. The class is determined using selected header fields of the packet and a set of used classification rules. Each rule represents one class. Packet belongs to a class that is represented by a rule whose conditions are satisfied by the packet. As classes may overlap, it is possible for one packet to satisfy conditions of multiple rules. In that case, the rule with the highest priority is usually selected. Matching conditions of header fields can be defined as one concrete value, a range of allowed values, or as a wildcard. Wildcard means that any value is accepted.

Packet classification is vital for proper functioning of the Internet. It is, in some way, implemented in most networking devices serving several key purposes. The most common use cases of packet classification are routing and packet filtering.

Routing is a process of determining packet's route to its destination. It can be done simply by matching IP addresses, also known as IP routing [8]. Alternatively, more header fields can be used to determine the route in order to allow routing based on more complex policies.

Packet filtering is mostly used in the area of networking security. It is used to decide which traffic is allowed. Malicious or unwanted traffic is filtered out and prevented from going through. The most common example of this is firewall, which is often present on a border of a network and also at individual stations.

3.1 Approaches

There are many ways to approach packet classification, just like there are many different algorithms that can be used for this purpose. Only some of those approaches and algorithms are discussed in this thesis. The simplest, naive approach to packet classification is to compare values from the packet sequentially against classification rules. The first matching rule is selected as the output. That is correct behavior in case the rules are ordered according to their priority (also called implicit priority).

Another approach to packet classification is to use a special kind of memory called Ternary Content-Addressable Memory (TCAM). It is based on Content-Addressable Memory (CAM), which consists of rows addressed by their content. The output of CAM is the address of a row that matches the input value. TCAM extends matching functionality to so-called ternary matching. Apart from the bit values 0 and 1, it introduces the third value "X". Comparing this value to another one always results in a match. This is useful when creating rules that do not care about a certain value. Without this functionality, it would be necessary to specify the same rule for both options (0 and 1). Using the same naive approach as before, but with TCAM, results in significantly smaller number of required entries for the same functionality [8]. Parallelism can be used to compare values from the packet against multiple, or even all, present rules. While this can lead to very fast search time, it also requires more resources and consumes a lot of power. TCAM also suffers from limited scalability to longer search keys due to its exhaustive search approach [16].

It is possible to represent classification rules using tuples. Elements of a tuple represent the number of bits used for the specification of corresponding rule's conditions. Real rule sets often contain only a few combinations of specification lengths, which makes the tuple approach viable. The number of bits used in the value, wildcard, and prefix specifications is clear, but the value of tuple elements corresponding to the range specification is not. The representation of utilized ranges is based on a hierarchy of non-overlapping ranges. In the hierarchy, ranges are organized in levels from the most general to the most specific. Each range is then represented by a pair consisting of nesting level and range ID. This pair characterizes position of the range within the hierarchy [8].

The packet classification problem can be represented in multidimensional space, where each dimension corresponds to one header field. Each condition of a classification rule can be represented as an interval in the corresponding dimension. The whole rule is then represented as a geometric body formed by those intervals [8]. It has as many dimensions as there are utilized header fields. A packet is represented as a singular point in the space, as it has a single value in each dimension. If the point is inside a body representing a rule, then the packet satisfies conditions of that rule. Which rule is selected as the output, if the packet matches more than one of them, is decided by priority, which can be implicit (given by order of the rules).

3.2 Existing Tools for Packet Classification Testing

Packet classification is crucial networking technology and often a performance bottleneck in routers. There are some publicly available performance evaluation tools for packet classification, but their functionality is still quite limited. The most notable ones are discussed further in this section.

3.2.1 ClassBench

ClassBench is a set of tools for benchmarking packet classification algorithms and devices. Its overview is shown in Figure 3.1. There are three tools in total. The first is Filter Set Analyzer, which analyzes real rule sets and produces filter set parameter files. These files contain statistical properties of the analyzed data. The created parameter files are then used in the next tool called Filter Set Generator, which is probably the most important piece in the tool suit. It produces synthetic rule sets. Apart from a parameter file, it also takes as input a set of parameters that allow to adjust high-level parameters of the generated rule set by a user. The final tool in the chain is Trace Generator that produces a sequence of packet headers with respect to the given filter set. ClassBench is probably the most used tool for these purposes. The greatest weakness of this tool suite is that it only operates with IPv4 headers. Specifically, a 5-tuple consisting of source and destination IP addresses, source and destination port numbers and the upper layer protocol field in an IPv4 header [17]. That was sufficient for the research community at the time of ClassBench's creation, but it is not anymore. More recent tools, that are discussed in the following sections, have expanded rule set generation to include other header fields and protocols.

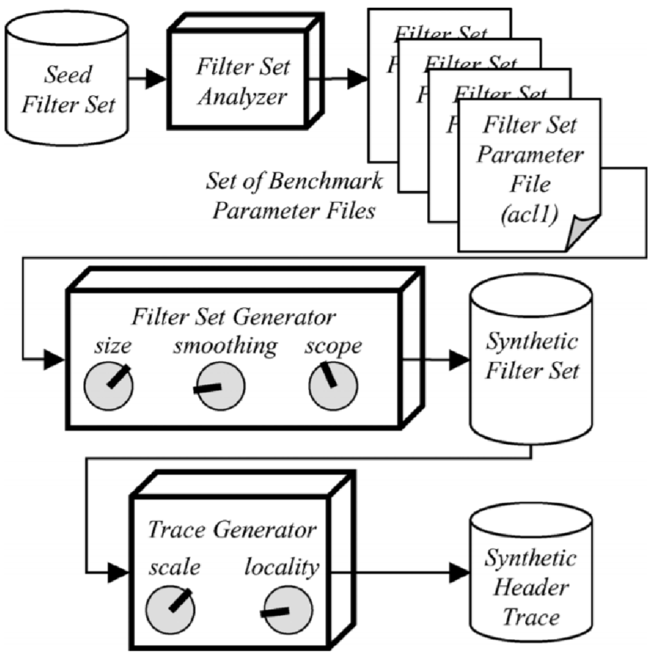

Figure 3.1: Block diagram of the ClassBench tool suite, taken from [17].

3.2.2 FRuG

Flexible Rule Generator (FRuG) is an entirely user-controlled benchmarking tool for evaluating forwarding algorithms. It allows its users to define distributions, composition, size and all other parameters related to the rule set generation. What makes the tool even more flexible is that rule generation is not restricted to a fixed number of fields, which makes it potentially useful even for future algorithms that the authors could not predict. IPv6, which has a different header structure, is not supported by this tool.

FRuG consists of the IPv4 prefix analyzer and generator, the MAC address analyzer and generator, the configuration file parser, and the FRuG engine. The entire structure is shown in Figure 3.2. Input parameter files provide an interface for the user to interact with the tool. There are three types of input files: a configuration file, a class file, and a descriptor file. The configuration file allows the user to define how each protocol field should appear in the generated rule set and the composition of each class of rules. In the class file the user specifies required parameters for the fields that will appear in the generated rules. The descriptor file can be used to further alter the structure of the generated rule sets.

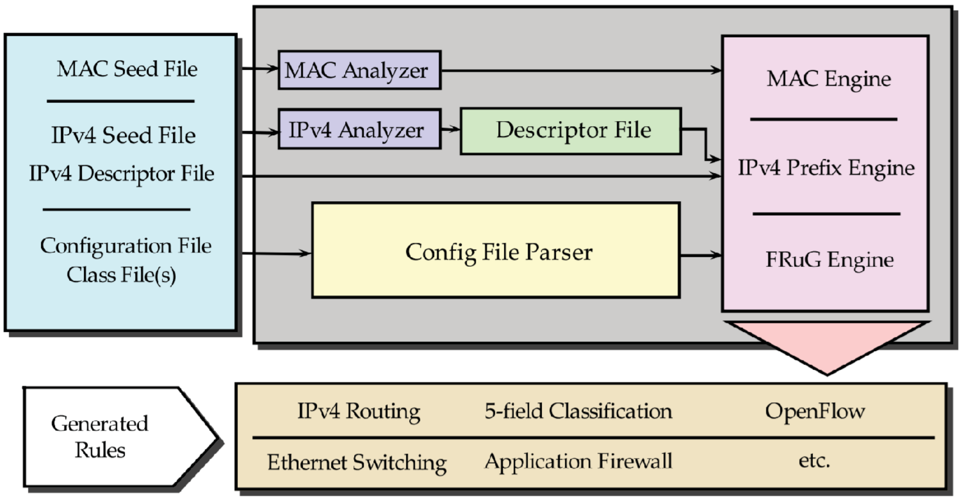

Figure 3.2: FRuG framework overview, taken from [4].

There are two modes of operation in FRuG. One is the IPv4 prefix generation mode, which runs the IPv4 prefix analyzer and the generator. The other mode is flexible rule generation, which is used when a configuration file and class files are specified. However, it might also require assistance of the IPv4 generator [4]. The output of the FRuG engine is a table of rules that follow the properties described in the input parameter files. Unlike ClassBench, FRuG does not provide a tool capable of generating network traffic.

3.2.3 ClassBench-ng

ClassBench-ng is an open source tool that follows in footsteps of ClassBench. Compared to the original ClassBench, the rule set generation is improved and it is capable of generating synthetic rule sets not only for IPv4, but also for IPv6 and OpenFlow 1.0.0.

It utilizes one input file that can specify the statistical behavior for all fields that need to be generated. The authors of ClassBench-ng analyzed real rule sets to understand their properties. Based on this analysis, they defined the structure of input seeds that accurately reflect characteristics of different operational scenarios. Some input seeds are provided with the tool. ClassBench-ng is capable of creating input parameter files from real rule sets as well. Input parameter files created in this way can then also be used to generate synthetic rule sets. Since the parameter files only contain statistical properties of the original real rules, they can be shared among researchers while keeping their anonymity. For that reason, no sensitive information regarding the original real data is revealed to the public.

ClassBench-ng tries to improve the rule set generation process by iteratively building an output rule set with characteristics as close as possible to the input seed. Just like FRuG, ClassBench-ng currently does not provide a tool for network traffic generation [9]. Considering the precision of generated synthetic rule sets, ClassBench-ng on average outperformes both ClassBench and FruG [8].

Comparison of functionality of the three mentioned tools is shown in Table 3.1. ClassBench provides all functionality, but only for IPv4 5-tuples. FRuG is capable of generating rule sets with up to 12 header fields chosen by a user. Therefore, it can be used to generate OpenFlow1.0. rules, but it is capable of analyzing only some of the header fields. Overall the characteristic of generated rule sets is more user driven. FRuG does not work with IPv6 addresses and does not have a trace generator. ClassBench-ng can analyze rules consisting of 5-tuples (only those with IPv4 addresses) and OpenFlow1.0 header fields. It can create rule sets even for 5-tuples with IPv6 addresses, but does not contain a header trace generator.

	Analysis	Rule set generation	Trace generation	Note
ClassBench	✓	✓	✓	IPv4 5-tuples only
FRuG	partly	✓	X	no IPv6
ClassBench-ng	partly (no IPv6)	✓	X	IPv4, IPv6, OF

Table 3.1: Functionality comparison of tools for packet classification testing

Chapter 4

Network Traffic Generation

Based on the description of existing tools, it is clear that the most current of them, ClassBench-ng, provides the best functionality. However, compared to the original ClassBench, it lacks the network traffic generator. In general, there is no existing trace generator for IPv6 5-tuple and OpenFlow header values. The lack of such tool is the motivation behind this thesis.

As input, the generator must take all classification rules that can be created by ClassBench-ng. That is, rules consisting of OpenFlow1.0.0 headers and 5-tuple headers made up of source and destination IP addresses (either IPv4 or IPv6), source and destination port numbers, and the upper layer protocol field for IPv4 or the next header field for IPv6. Those fields are marked with green color in figures that display structure of a protocol in Chapter 2. Header fields present in OpenFlow1.0.0 that are not included in the 5-tuples are marked with yellow.

The simplest way to generate network traffic is to generate random headers with values within the permitted ranges. This technique is undesirable, because it does not ensure that every header is covered by at least one rule. This problem can be solved by using random values within a rule, i.e., for each generated header, use random values that are within ranges specified by a rule's conditions. This method is further discussed in section 4.1. The opposite approach is to perform a complete analysis of the rule set, find all intersections between rules, and then generate a header for each unique case. Depending on the size of a rule set and the amount of overlaps, this calculation may become virtually impossible due to its memory requirements and time complexity. Therefore, it is necessary to find a reasonable compromise between optimizing for rule coverage and optimizing for performance. For example, the authors of the original ClassBench tool decided to use "corner" values. That means selecting either the smallest or the largest value from the range specified by each condition of a rule [17]. Headers generated in this manner should have higher chance of matching multiple rules, which means a higher chance of covering an overlap.

4.1 Simple Network Traffic Generator

Before creating an optimized network traffic generator that would be able to adequately test classification algorithms, it is useful to create its simplified version. Since the generator is designed with ClassBench-ng in mind, it should be compatible with the rule sets generated by this tool. That automatically makes it compatible with ClassBench's rule sets as well.

In order to keep backward compatibility, the structure of generated IPv4 headers is the same as in the original ClassBench.

The first version of the generator could also be described as a naive version. Implemented in Python3.8, it works with all required protocols. In a way, the naive version implements all desired functionality, but no optimization for smarter generation, which would lead to better testing of classification algorithms, is performed. The way this tool works can be seen in Figure 4.1. When generating a packet header, a random rule from the input rule set is selected (line 3). Then the generator assigns values matching that rule to the newly created header (line 5). If a field in the rule is a range, a completely random value within this range is picked. Finally, the rule is sent to output (line 7). This repeats until required number of headers is generated.

```
1: function TRACEGEN($rules, size$)
2:     for $i \leftarrow 0$ to $size$ do
3:         $rule \leftarrow$ GETRANDOMRULE($rules$)
4:         for each $field$ in $rule$ do
5:             $header[field] \leftarrow$ GETRANDOMVALUE($rule[field]$)
6:         end for
7:         PRINTHEADER($header$)
8:     end for
9: end function
```

Figure 4.1: Pseudocode of Naive Trace Generator

Some results of testing this generator's version using IPv4 5-tuple rule set can be seen in Table 4.1 and using OpenFlow 1.0.0 in Table 4.2. Both rule sets consist of 1 000 rules. All presented data are averages from 10 runs. The results clearly show how increasing the number of generated headers does not increase memory consumption. This is due to the fact that the headers are sent to output immediately after creation and not stored anywhere. Rules have to be stored, therefore the memory consumption is driven almost exclusively by the size of the rule set. Execution time is not increasing exactly in proportion to the number of generated headers. This is because a considerable portion of execution time is taken by loading the rule set. Since the loading time remains static, generating more headers leads to more efficient generation (larger proportion of execution time is spent on the actual process of generating headers). This is why more headers per second are generated when more headers are required. The generator creates IPv4 5-tuples faster than OpenFlow, because of the smaller number of header fields.

`Number of headers`	10	100	1 000	2 000	5 000
`Rule coverage [%]`	0,98	9,55	63,01	86,19	99,26
`Overlap coverage [%]`	0,18	1,27	9,22	12,32	14,13
`Region coverage [%]`	0,78	7,41	49,13	67,13	77,30
`Peak real memory [kB]`	151 288	151 544	151 800	152 188	152 188
`Peak virtual memory [kB]`	9 136	9 396	9 667	9 969	9 994
`Execution time [ms]`	43	52	96	143	178
`Headers per second`	233	1 928	10 367	13 966	28 169

Table 4.1: Results of testing naive trace generator using IPv4 5-tuple rule set with 1 000 rules.

Number of headers	10	100	1 000	2 000	5 000
Rule coverage [%]	1,00	9,52	62,72	86,34	99,31
Overlap coverage [%]	0,00	0,00	0,00	0,00	0,00
Region coverage [%]	0,02	0,23	1,50	2,07	2,38
Peak real memory [kB]	151 288	151 288	151 800	151 932	151 932
Peak virtual memory [kB]	9 136	9 136	9 683	9 696	9 856
Execution time [ms]	75	91	165	199	308
Headers per second	133	1 097	6 049	10 044	16 247

Table 4.2: Results of testing naive trace generator using OpenFlow rule set with 1 000 rules.

4.2 Analyzing Generated Network Traffic

A mandatory step in creating an efficient network traffic generator is to analyze its outputs. There are several criteria that can be used to evaluate the results. The overarching goal is to provide the best possible coverage of the provided rule set.

A decent portion of the work on this thesis was spent on creating a Python script to analyze the generated network traffic. In order to do that, the script has to analyze the rule set first. Therefore, two most important parameters of this tool are paths to files containing generated headers and the used rule set. These classification rules are generated by ClassBench or ClassBench-ng. Another parameter specifies if the rules are tuples or OpenFlow 1.0.0 rules. Tuples can be either IPv4 or IPv6. Since ClassBench does not support IPv6 or OpenFlow, those rules will always come exclusively from ClassBench-ng. The final parameter can be used to make the script run in an experimentation mode, which suppresses normal output and provides results in a way that allows for further easy automatic processing.

The first step of the evaluation tool is to load rules from the provided file into a list. For each rule an instance of class representing a rule is created. The rules are analyzed after all of them are loaded. Several different approaches can be taken to do this. The ones that were utilized are further discussed in subsection 4.2.1. The next step is to check coverage of those rules and overlaps by the generated header trace. That is done as the header fields are being loaded, one by one, without saving them. This means that only one header at a time is loaded in memory and it gets overwritten by the next one when it is no longer needed. This significantly improves memory consumption of the script, especially for large header traces. The final step is to calculate and output the results.

4.2.1 Analyzing Rules and Evaluating Rule Coverage

The first approach that was taken detects all overlaps between individual rules and keeps track of overlaps as a list of pairs of rules. This is the simplest solution. It does not require any explicit specification of the overlapping region, thus it is not necessary to compute its dimensions. It allows for analyzing the coverage of given rules, rules using implicit priority, and overlap pairs. When implicit priority is taken into account, then only the first matching rule is covered by a header. In the opposite case, all rules that match the header are considered to be covered. That is, if the header falls into an overlap, it covers both rules from that overlap. Both of these variants could be useful for certain applications.

The main problem with the overlap pairs approach is that it does not really take into consideration the possibility of more than two overlapping rules. Of course, there are pairs representing overlaps for all involved rules, but that is not the same as differentiating between all distinct regions of such overlap. To better explain this, we can represent the problem graphically, as described in section 3.1. Using two dimensions and three rules we can get a situation shown in Figure 4.2. Each of the three rules have a part that is not overlapping with others (red, green, blue), and parts that are overlapping with one of the other two rules (yellow, cyan, purple). One part is shared by all three of them (white). Ideally, when determining rule coverage, we want to be able to distinguish to which one of those seven distinct regions a packet belongs to. That is the best way to test the classification algorithm. The pairs approach is not able to do that. Presented with the situation in Figure 4.2 the pairs approach would discover that rule 1 overlaps with rule 2 and rule 3, and that rule 2 also overlaps with rule 3, but there would be no information about the distinct region where all of them overlap. Similarly, this approach does not provide a clue whether the overlaps are just partial (like in the example) or if one rule forming the overlap pair is a subset of the other rule.

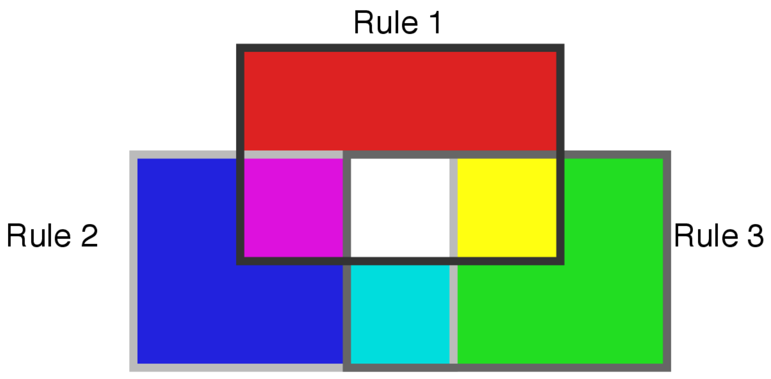

Figure 4.2: Example of possible overlaps among three rules

To achieve a desired level of analysis, it is necessary to not only discover overlaps between rules, but also to be able to define them. That is why the final version of the evaluation tool, described in following paragraphs, does not simply look for overlaps. Instead, it finds all distinct regions and creates a rule for each of them.

Since some of rule's conditions are ranges (IP prefixes, port numbers, etc.), they can be specified by their maximum and minimum values, which makes it easy to test if a packet matches the rule. This becomes impossible after splitting the rules into regions, as they might take shapes that cannot be specified like this. An example of such shape is the "L" shape of blue and green regions in the example shown in Figure 4.2. This problem is solved in the evaluation tool by what could be described as layering the rules representing regions on top of each other using explicit priority. Explicit priority can be used because implicit priority becomes irrelevant for evaluation using distinct regions (every packet matches at most one region rule). This way conditions can still be specified by their maximum and minimum values.

The evaluation tool finds all distinct regions in rounds. First round begins with round number zero. Only rules with explicit priority equal or higher than the rule number participate in each round. Default rule priority is zero. When a partial overlap is found, the rules that are forming it keep their values and the newly created rule (representing the overlap) is assigned priority one higher than the round number (i.e., it goes to the next round). This can be demonstrated on the following example where, for simplicity, rules have just a single integer condition. Let's have two rules with the default priority (zero): rule A with condition 0 to 10 and rule B with condition 5 to 15. The evaluation tool will detect the overlap and create a new rule C with condition 5 to 10 and priority 1. The original rules are not changed, which means that there could be a header that could satisfy all three of the rules (e.g., with field value corresponding to the condition set to 8), but only rule C would be marked as covered due to the priority. Therefore it works as if each rule's condition was specified by a range of numbers that is unique to it. If one rule is discovered to be a subset of another, priority of the inner rule is set to one higher than the round number (it becomes the overlap). When two rules are found to be the same, one of them is removed. Newly created rules are added to the list of all rules at the end of the round in which they were created (they do not participate in this round). Next round is started only if at least one new rule has been created in the current round or if at least one rule's priority has been changed due to it being a subset of another rule.

In summary, five different metrics of rule set coverage have been implemented as part of the evaluation tool. The first is the percentage of covered rules, where each packet covers only the first rule that it matches. Next is the percentage of covered overlap pairs. The third is essentially the combination of the previous two. It is the percentage of covered rules, where each packet covers every rule it matches (i.e., if a packet hits an overlap, it covers all rules forming this overlap). The fourth metric is the percentage of covered distinct region overlaps. It is a subset of the final metric, the most important one, which is the percentage of covered distinct regions.

Comparison of all rules with each other can be done at best with quadratic time complexity. Each rule has to be loaded into memory during this process. Note that each overlapping region is represented by a separate rule that is added to the original rule set. Therefore, it is clear that performance is a major issue for large rule sets or rule sets with a large number of overlaps. From a certain point, memory consumption and execution time become infeasible. Due to the need for better performance, all methods have eventually been stripped from the tool, except the distinct regions one, which provides the most informative data.

Other improvements have also been made. One of the most significant yet simple ones, which leads to almost five times shorter execution time using the same data, is changing the order of different header fields' comparisons when looking for an overlap. For example, comparing network prefixes is much more expensive operation than comparing protocol numbers. Therefore, checking protocol numbers is done earlier and if they do not match, it is clear that the rules cannot overlap. In such scenario, the comparison of prefixes is not needed, and therefore is never invoked.

Since many algorithms rely on prefix matching only, the generator is also tested with rule sets containing only prefixes (other fields are essentially wildcards). For such cases, a lightweight version of the evaluation tool for evaluating generated traffic has been created. It ignores all the irrelevant fields which significantly decreases memory consumption and also improves performance.

Chapter 5

Optimizing Network Traffic Generator

After the completion of the simple generator and the tool for evaluating results, work on improving the generator could begin. Starting with the simplest and least accurate solution possible, it was necessary to begin by introducing techniques to improve region coverage. Many different methods were tried and experimented with. The more useful ones are listed in this chapter. The generator and its results are dependent on its input, specifically the provided rule sets. Using realistic rule sets is crucial for benchmarking all the different ways of creating header traces.

The next section, section 5.1, discusses the used rule sets. Following sections describe the different generator versions and examine their results. Final parts of this chapter are devoted to performance of the generator in section 5.6 and possible future improvements in section 5.7.

5.1 Rule sets used for testing

All rule sets used for testing the generator were generated using ClassBench-ng. They were created according to the parameters extracted from real rule sets and therefore should be an accurate representation of rules used in real applications. All of them were analyzed in a way described in subsection 4.2.1. The number of overlaps and distinct regions of each used rule set can be seen in Table 5.1. An exception is the OpenFlow rule set of1_gen_b_10000, which contains too many overlaps to be analyzed in a reasonable time. For this reason, only a part of it was used for testing rule coverage, rule set of1_gen_b_x1000 represents 1000 rules out of the total 10000. Rule sets consisting of rules using only prefixes are also commonly used. One for each IPv4 and IPv6 were used for testing. The fifth, and final, rule set acl4_gen_1000 consists of rules represented as IPv4 5-tuples. All sizes were chosen based on sizes of real rule sets that were used for generating the synthetic ones.

When performing tests with the generator, the number of generated headers was often established as a certain percentage of the size of the rule set. It is reasonable to expect a good coverage when generating for instance five times more headers as there are rules, but in case of the OpenFlow rule set, it is necessary to generate way more headers, as there are more than forty times more distinct regions than rules.

Type	Name	Size	Overlaps	Regions
`IPv4 prefixes`	2015_rrc00_ipv4_gen_b_100000	100 000	12 095	100 000
`IPv6 prefixes`	2015_rrc00_ipv6_gen_b_10000	10 000	1	10 000
`IPv4 5-tuples`	acl4_gen_1000	1 000	332	1 287
`OF rules`	of1_gen_b_10000	10 000	?	?
`OF rules`	of1_gen_b_x1000	1 000	40 800	41 800

Table 5.1: Number of overlaps and regions of used rule sets

Note that the only overlaps in prefix rule sets are caused by one rule being a subset of another, therefore no new regions are created. This stems from the way IP prefixes work. They can never overlap without one being fully encapsulated by the other.

5.2 Corners Version

The original ClassBench paper acknowledges that there needs to be a compromise between generating the perfect trace and a completely random one [17]. The authors came up with the idea of generating packet headers representing the “corners” of rules. Through their analyses, they discovered that such headers are more likely to be in an area of overlapping rules [17]. When generating header trace, first, a random rule is selected. Then, values of individual header fields are selected as either the smallest or the largest value of the range specified by the rule for a corresponding dimension. The trace generator from ClassBench also creates a variable number of copies of the created packet headers. Their amount is sampled from Pareto distribution with two input parameters.

This version was recreated and extended to work with OpenFlow 1.0.0 as a part of this thesis in order to experiment with it and use it as a benchmark for future versions. It was discovered that the Corners version of the generator does not provide any better coverage of overlaps or regions than the naive version as can be seen in Figure 5.1, which shows comparison of coverage of the IPv4 5-tuple rule set by those two versions. The results were nearly the same for all tested rule sets. It is no surprise when it comes to prefixes, since each rule is either a subset/superset of another one, or it does not overlap at all. The only improvement that corners provide is that when a header is generated based on a rule that is a superset of another rule, the header is more likely to not belong to the inner rule. Experiments have shown that this is not enough to make a noticeable difference.

Where an improvement of coverage could have been expected, but did not happen, is the case of IPv4 5-tuples. This is exactly what the original ClassBench worked with. It is possible that the characteristics of used rule sets have changed since then in such a way that this approach is no longer effective. Table 5.2 shows results of some of the experiments.

The results of testing the Corners version of the generator using OpenFlow are shown in Table 5.3. OpenFlow rules contain a large number of wildcards. Those are the main contributors to overlaps. Choosing only the most extreme values of a field can actually be counterproductive, as the absolute minimum and maximum values of specific fields are rarely used, but end up in a generated packet header every time a rule containing a wildcard is used. Overall the results are nearly identical with the naive version of the generator.

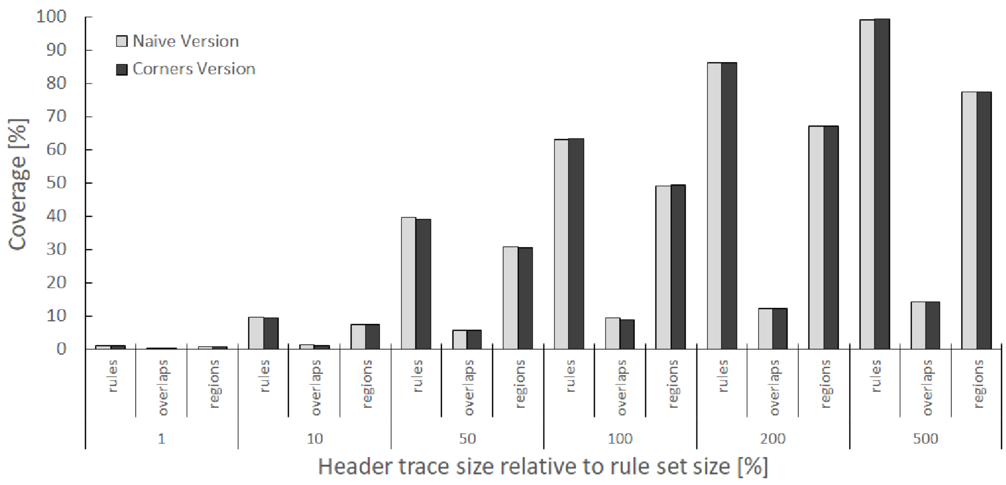

Figure 5.1: Coverage of individual rules, overlaps and regions of IPv4 5-tuple rule set by naive and Corners versions of the generator.

Number of headers	10	100	1 000	2 000	5 000
Rule coverage [%]	0,97	9,46	63,50	86,39	99,34
Overlap coverage [%]	0,18	1,12	8,89	12,11	14,07
Region coverage [%]	0,77	7,30	49,41	67,23	77,34

Table 5.2: Results of Corners version experiments using IPv4 5-tuple rule set with 1 000 rules.

5.3 Prevent Rule Reusage

The idea behind this version is to keep track of the rules that have already been used to generate a header and to not reuse them until a certain percentage of all rules had been used. From the concept, it is clear that this approach only guarantees a good coverage of individual rules, but not overlaps. Therefore, a proper coverage of regions is achieved only if a given rule set contains a small amount of overlaps. After some testing it was determined that ninety percent is a good amount to use before rules are allowed to be reused. It guarantees a decent amount of coverage and it does not hinder performance.

Nearly perfect rule coverage could be achieved by going through the rules in some orderly manner, such as from the first to the last, but that would introduce unwanted regularity into the process. It is possible to get around this problem by removing the element of order from the process. Following version of the generator, Prevent Rule Reusage 2, goes through the whole rule set in a random order. There is a reason why the coverage is still only nearly

Number of headers	10	100	1 000	2 000	5 000
Rule coverage [%]	1,00	9,55	63,14	86,54	99,36
Overlap coverage [%]	0,00	0,00	0,00	0,00	0,00
Region coverage [%]	0,02	0,23	1,51	2,07	2,38

Table 5.3: Results of Corners version experiments using OpenFlow rule set with 1 000 rules.

perfect. It is because of situations when one rule is a subset of another. In such cases, it is possible that the header generated based on the superset rule will actually belong to the subset rule. As a result, the superset rule will not be covered even though a header was created based on it.

5.4 Smart Random

Previously mentioned method is useful for covering individual rules, which in itself is not a very complicated problem. The real issue is achieving a high degree of overlap coverage with reasonable performance demands. One of the first ideas to achieve that was to keep the random system, but also exploit sizes of different rules. Larger rules should, in theory, be involved in more overlaps than the smaller ones, because they simply take up more space. Using larger rules more often to generate headers should then lead to greater overlap coverage.

The Size of a rule can be defined as the number of possible header field combinations that satisfy the rule's conditions. Calculating the exact value is not necessary to sort the rules by size though. It is satisfactory to count how many wildcards there are in each rule and calculate the span of ranges for IP prefixes and also port numbers in case of 5-tuples. Remaining conditions are defined as a single value and therefore not relevant in the sorting process.

Different levels of preference of the larger rules have been experimented with, in both the frequency of preference and the portion of rules to be preferred. Nevertheless, none of the various setups lead to any significant improvement of overlap coverage.

Further inspection of various rule sets revealed that overlaps can be as small as a single value, meaning that the rules which form them intersect in a single point. This happens, for example, when two rules differ in just two conditions, where the first condition is a wildcard in one rule and a specific value in the other rule, and vice versa for the second condition. This situation is more common for OpenFlow rule sets, which contain many wildcards. As an attempt to tackle this, the next approach was to keep a portion of values from already generated headers and then reuse them in another header. Only values that originated from a non-wildcard field were kept. They were saved in a list and sometimes used again in another header if the alternative was to generate a random value, i.e., when a wildcard was present.

This quickly turned out to be insufficient. Due to the low probability of reusing the fitting value at the right time, the improvement of overlap coverage was noticeable, but meager. Creating a generator capable of creating headers that would cover even such overlaps without a full analysis of the rule set requires a more complex approach, certainly much more complex than randomly selecting values. Yet, it is necessary to not introduce too much order into the process, which would create a bias that could skew testing of classification algorithms in a certain direction.

Smart Random 2.0

Generating the values for each header based on a rule seemed to be the right way to go from the beginning. It guarantees that the header will always match at least one rule and no downsides of using it manifested themselves. Further development was mostly based around this original idea.

```
1: function TraceGen($rules, size, par\_a, par\_b$)
2:     $wc\_masks \leftarrow$ GenerateMasks($rules$)
3:     $search\_limit \leftarrow$ Min($1000, size$)
4:     while $gen\_num < size$ do
5:         $prim\_i \leftarrow$ GetRandomRule($rules$)
6:         $prim\_rule \leftarrow rules[prim\_i]$
7:         $sec\_i \leftarrow$ SelSecondary($prim\_i, rules, search\_limit, wc\_masks$)
8:         $sec\_rule \leftarrow rules[sec\_i]$
9:         $header \leftarrow$ GenHeader($prim\_rule, sec\_rule$)
10:        $gen\_num$ += PrintHeaders($header, par\_a, par\_b$)
11: end function
```

Figure 5.2: Pseudocode of the Smart Random 2.0 header trace generator.

It is possible to use more than one rule at a time to generate a header. In case of this version, there is one extra rule used. The pseudocode of the Smart Random 2.0 header trace generator is shown in Figure 5.2. For each rule a mask is created, where each condition is represented by zero if it is a wildcard or one in the opposite case (line 2). The rest of the process is repeated until the generated trace reaches the specified size. First, a random primary rule is selected (line 5). A secondary rule is also selected at random, but if it does not meet certain conditions in regards to the primary rule, another random rule is selected (line 7). This could go on forever, as there might be no rules to satisfy the conditions, which is why an upper limit must exist. This is called the search limit. After some amount of experimentation, the search limit is set to one thousand, or the number of rules in a rule set, for rule sets smaller than a thousand (line 3).

The conditions that must be satisfied play a key role in the whole process. The first condition that has to be met by the secondary rule is that the logical operation *and*, between its mask and the primary rule's mask, results in a mask that consists of a maximum amount of ones equal to three for 5-tuples and four for OpenFlow. In other words, the number of fields where both rules have a non-wildcard value is limited to a maximum, which, if exceeded, results in not using this secondary rule candidate. This serves as a quick filter of rules that have an extremely low chance of overlapping with the primary rule. If a rule passes this test, it is checked whether the two rules overlap (masks are used to check only the necessary fields). If the secondary rule overlaps with the primary rule, a fitting secondary rule is found. If a suitable secondary rule is not found in the maximum search limit of attempts, then the last failed candidate is used. A header is then generated employing both rules (line 9). The primary rule is used first. All of its non-wildcard conditions are applied. For the rest of them, values from the secondary rule are used. If a given field is wildcard for both rules, then a random value is generated. The header is then sent to output. Multiple copies of it may be inserted into the trace, where the number of repetitions is sampled form a Pareto distribution taken from the original ClassBench (line 10).

Smart Random 3.0

If there were some universally shared properties between rules that form overlaps, they could be used to guess which rules overlap. Then, using such rules more frequently would lead to better region coverage. In search of such properties, it was discovered that there are not any, but the rules that are a part of at least one overlap, often participate in more of them.

Version 3.0 builds on the previous one. It does not change how secondary rule is selected. Instead, improvements target the selection of the primary rule. A new parameter called overlap focus is introduced. It determines how often should a primary rule be selected from rules that are known to overlap instead of being chosen completely at random.

```
 1: function TraceGen($rules, size, overlap\_focus, par\_a, par\_b$)
 2:     $overlap\_indices \leftarrow \emptyset$
 3:     $wc\_masks \leftarrow$ GenerateMasks($rules$)
 4:     $search\_limit \leftarrow$ Min($1000, size$)
 5:     while $gen\_num < size$ do
 6:         $prim\_i \leftarrow$ SelPrimary($rules, overlap\_focus, overlap\_indices$)
 7:         $prim\_rule \leftarrow rules[prim\_i]$
 8:         $sec\_i \leftarrow$ SelSecondary($prim\_i, rules, search\_limit, wc\_masks$)
 9:         $sec\_rule \leftarrow rules[sec\_i]$
10:         $header \leftarrow$ GenHeader($prim\_rule, sec\_rule$)
11:         $gen\_num$ += PrintHeaders($header, par\_a, par\_b$)
12:         if Overlap($prim\_rule, sec\_rule$) then
13:             $overlap\_i \leftarrow overlap\_i \cup \{prim\_i, sec\_i\}$
14: end function
```

Figure 5.3: Pseudocode of the Smart Random 3.0 header trace generator.

The pseudocode of this version of the generator is shown in Figure 5.3. Overlapping rules are discovered during the process of selecting a suitable secondary rule (line 8) and their indices are added to the set of indices of known overlapping rules. The overlap focus is a floating point number ranging from 0 to 0.9. If not specified, the default value of 0.5 is used. Multiplied by one hundred, the overlap focus directly corresponds to the percentage of how often the primary rule is chosen from the list of known overlapping rules.

Experiments regarding the overlap focus were mostly done with the 5-tuple rule set acl4_gen_1000. Figure 5.4 shows effects of varying overlap focus on overlap coverage for several trace sizes. Note that the maximum overlap focus is very useful when not too many headers are generated, but things begin to change when there are about as many headers as there are rules in the used rule set. Eventually, a higher overlap focus becomes counterproductive, as can be seen in the figure. This happens because the generator ends up rediscovering the same overlaps instead of searching for those that are made out of rules that have not been found to overlap yet. It is clear that there is no ultimately best value of the overlap focus. It depends on the situation, but overall, the value 0.5 is a good middle-of-the-road solution, which is why it was selected as the default value. Similar experiments with OpenFlow rule sets did not bring any useful results, because almost all of their rules participate in overlaps. For such rule sets, this version is effectively not different from the previous one.

Smart Random 3.1

Previously mentioned versions struggle with covering overlaps that are made up of more than just two rules. This is especially noticeable for OpenFlow rule sets, where such overlaps

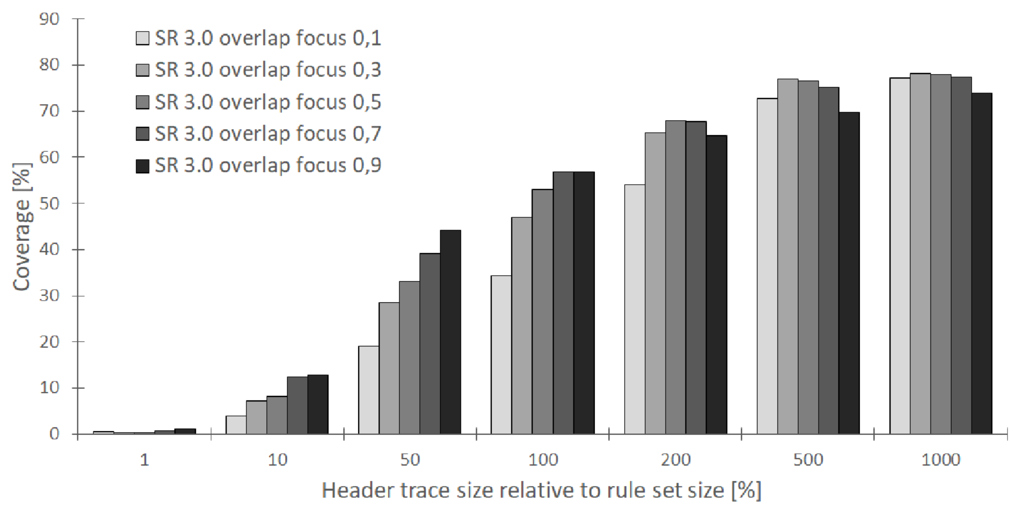

Figure 5.4: Overlap coverage of IPv4 5-tuples using Smart Random 3.0 with various overlap focus.

commonly appear. Version 3.1 set out to address this. It strictly focuses on overlaps and to a certain extent neglects covering individual rules.

When a pair of overlapping primary and secondary rules is found, it is not only used to generate a header (and its copies), as described in section 5.4, but the rules are also merged to create a new temporary rule. Merging the rules creates a rule describing their overlap. This temporary rule is used as a new primary rule and a fitting secondary rule is looked for in the same manner as previously. If it is found, then a combination of three overlapping rules is discovered and used to create another header and, optionally, its copies. The overlap focus has to be set to at least 0.2 for this to happen. This process is not repeated again, even though it could continue forever, or at least until no more suitable rule pairs could be found. Through experiments, it was discovered that repeating it does not lead to better overlap coverage. It is possible that it could be advantageous for some other, specific rule sets, but not in general.

Comparison of overlap coverage of the OpenFlow rule set of1_gen_b_x1000 by different Smart Random versions can be seen in Figure 5.5. As noted earlier, version 3.0 does not bring any improvement over 2.0 due to the nature of the rule set. The graph clearly demonstrates how those two versions “cap” when a certain amount of headers is generated. Further increases in trace size lead to very little increase in overlap coverage. Version 3.1, on the other hand, keeps improving. Used trace sizes were chosen based on the total number of distinct regions present in this rule set, which is 41 800. The overlap focus was set to its default value of 0.5 for these and all the following experiments. Parameters of the Pareto distribution are set to not create any redundant headers.

The corners method of packet header generation, proposed by the original ClassBench, has zero percent coverage of overlaps in the OpenFlow rule set. That method was designed for IPv4 5-tuples and not OpenFlow rules, as OpenFlow did not even exist back then.

The results for 5-tuples are shown in Figure 5.6. Even there, generating corner values provides significantly worse overlap coverage than the other versions. Smart Random 3.0 and 3.1 achieve similar results, because the rule set does not contain that many overlaps, especially those that are formed by more than two rules.

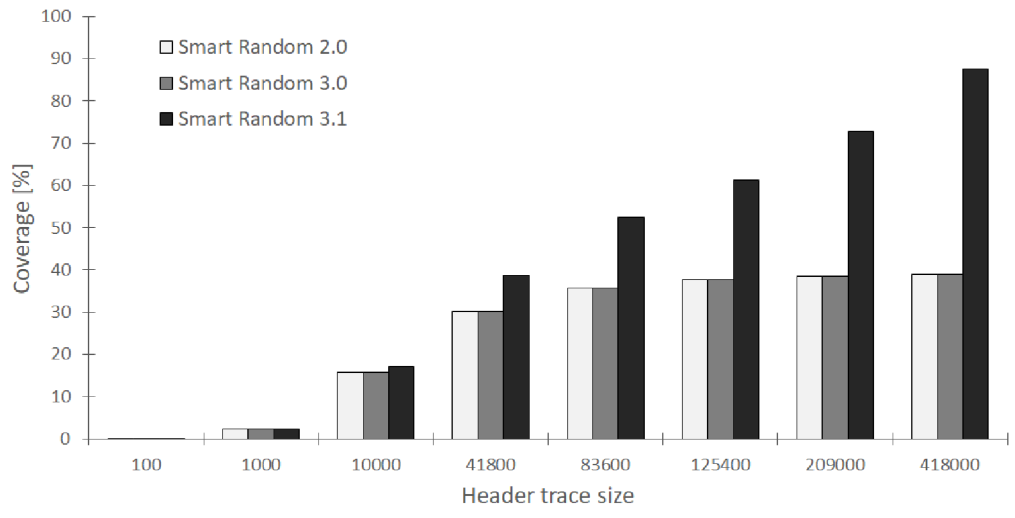

Figure 5.5: Overlap coverage of an OpenFlow rule set using various versions of the generator.

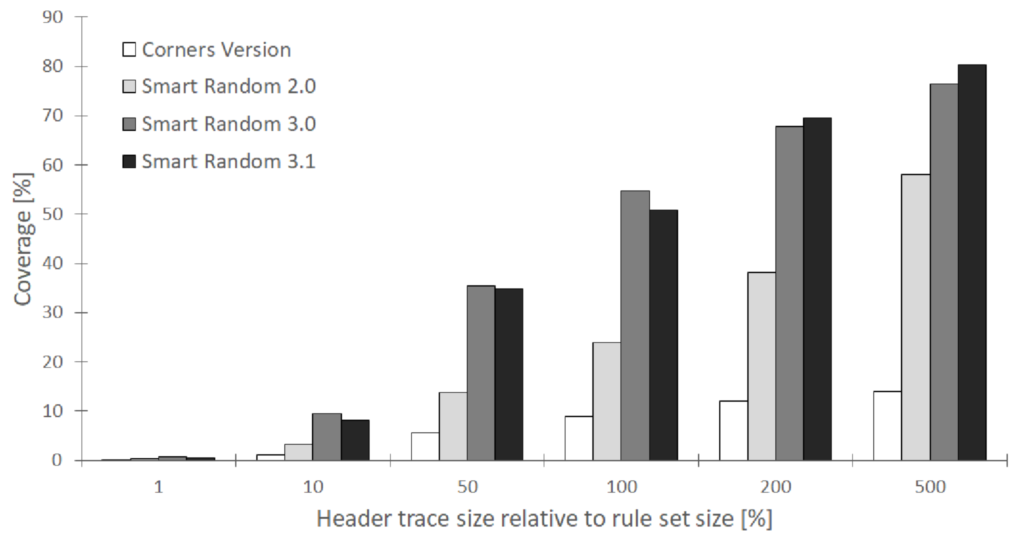

Figure 5.6: Overlap coverage of IPv4 5-tuples using various versions of the generator.

5.5 Combined Version

The overarching goal is to provide the best coverage of all regions. Ideas described in section 5.3 lead to a good coverage of individual rules, while methods presented in section 5.4 focus on achieving the best overlap coverage. Naturally, the next step was to take the advantages of both approaches and combine them together.

The combined version of the trace generator first applies principles from the Prevent Rule Reusage 2 generator. It randomly iterates over the rules provided as input and generates a matching header (and its copies) for each of them, or stops earlier if the required trace size is reached. If the required trace size has not been reached, then the rest of the headers is generated using Smart Random 3.1.

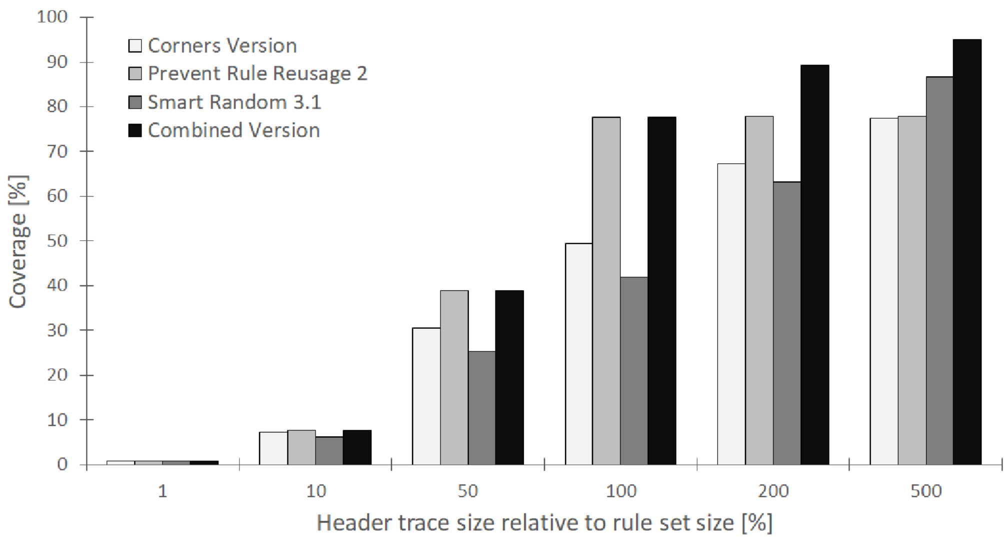

Figure 5.7: Region coverage of IPv4 5-tuples using different versions of the generator.

Comparison of region coverage of the IPv4 5-tuple rule set acl4_gen_1000 achieved by different versions of the generator is shown in Figure 5.7. The Combined version provides the best region coverage for all trace sizes. Smart Random 3.1 trails behind the other versions at first, as it focuses on overlaps and struggles to cover individual rules. At header trace size equal to 500 % of the rule set's size, all versions, except Smart Random 3.1, have covered almost all individual rules. The Corners version and Prevent Rule Reusage 2 provide poor coverage of overlaps and therefore also worse coverage of all regions than the Combined version.

Experiments with OpenFlow rule sets resulted in the greatest disparities between different generators. This is because all versions of the generator that use only one rule's conditions to create a header provide extremely poor coverage of overlaps, often none at all. Figure 5.8 shows region coverage of the OpenFlow rule set of1_gen_b_x1000 by different generator versions. The number of generated headers was chosen based on the total number of regions present in the rule set. The Corners version and Prevent Rule Reusage 2 are not able to cover any overlaps, but they do cover all individual rules, even with just 41 800 headers, which results in region coverage of 2,39 %. Smart Random 3.1 covers only 8.6 % of the individual rules when generating 418 000 headers. That is what makes it lack behind the Combined version which provides the best overall coverage.

Since the IPv6 prefix rule set 2015_rrc00_ipv6_gen_b_10000 contains only a single overlap, the set of distinct regions is almost equal to the set of individual rules. That results in the difference between region coverage by the different versions being rather small, as shown in Figure 5.9. The header trace size equal to 500 % of the rule set's size is enough to almost always fully cover regions for all versions. In case of Prevent Rule Reusage 2, and thus also for the Combined version, generating as many headers as there are rules is usually enough to cover all regions.

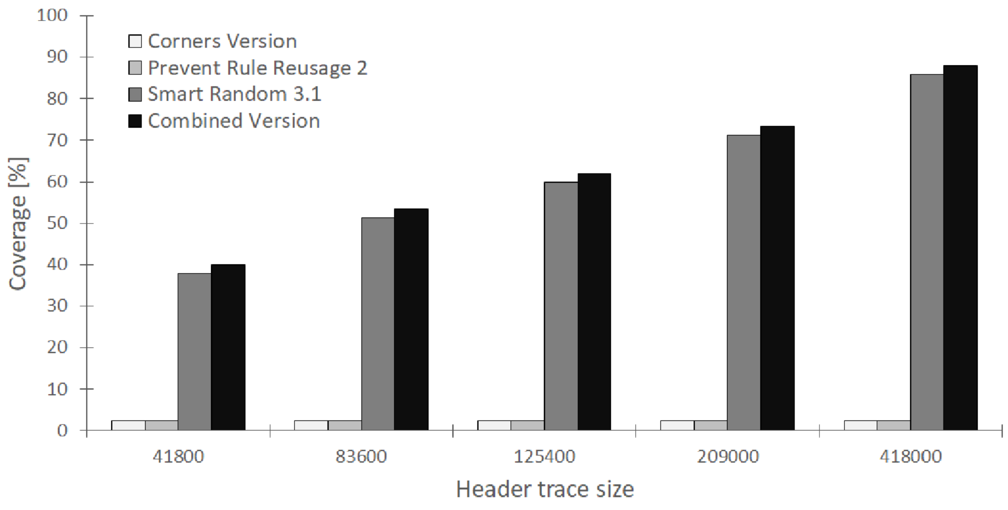

Figure 5.8: Region coverage of the OpenFlow rule set using different versions of the generator.

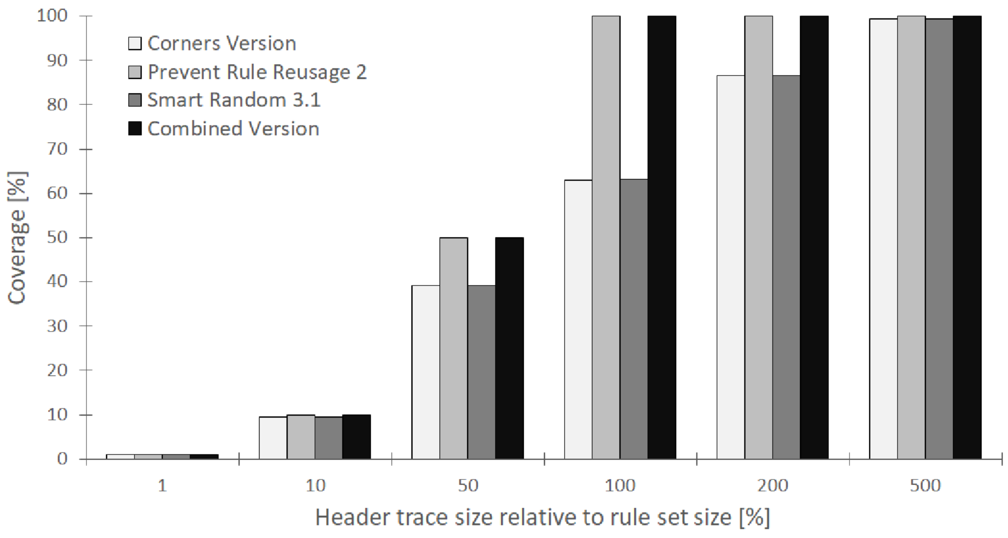

Figure 5.9: Region coverage of the IPv6 prefix rule set using different versions of the generator.

Results of region coverage for IPv4 prefixes are shown in Figure 5.10. They are nearly identical to those of IPv6 prefixes. The 2015_rrc00_ipv4_gen_b_100000 rule set contains more overlaps. Due to their nature, that was explained earlier, it is more likely for this rule set that a header generated based on a superset rule will actually belong to the subset rule. Over the 10 runs, this resulted in a few hundredths of percent lower coverage than was achieved with the IPv6 prefixes.

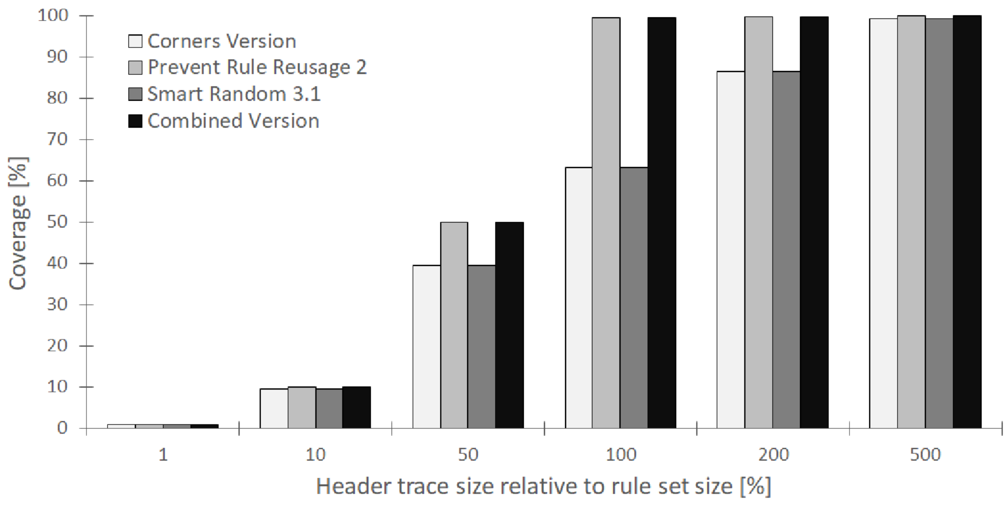

Figure 5.10: Region coverage of the IPv4 prefix rule set using different versions of the generator.

5.6 Performance

The evaluation of trace generators' performance is based on two metrics: memory consumption and run time. Run time can be used to calculate how many headers are generated per second, which allows for comparison of performance between rule sets of different sizes. Introducing a similar metric for memory consumption, such as headers per kB of memory, would not be useful, because memory consumption is not dependant on the size of the generated header trace.

All performance experiments were conducted on a Scientific Linux server with Intel Xeon E5-2670 processing unit running at a base frequency of 2,6 GHz. A total of about 300 MB of RAM were available for the experiments. Just like the experiments with rule set coverage, all experiments were run ten times and the provided values are averages of those runs.

Table 5.4 shows the number of headers that each version generated per second for different input rule sets. In these experiments, the generator was instructed to create a header trace ten times larger than the input rule set's size and to not create any redundant headers (which would lead to shorter execution times). Used rule sets are presented in section 5.1. Results for the two OpenFlow rule sets were almost identical. To prevent repetition, only results obtained using the of1_gen_b_10000 rule set are displayed in the table.

The advanced versions of the generator that provide very good coverage are, as expected, significantly slower than the simple generators. Smart Random 3.0 is faster than Smart Random 2.0, most likely because it randomly selects some rules from the list of known overlapping rules, which is smaller than the list of all rules. This appears to be the case, because 3.0 is evidently faster only when the input rule set contains some overlaps, but at the same time not all rules participate in them. Prevent Rule Reusage 2 is faster than its previous version, because it prepares the random order in which it uses rules to generate headers and then continues generating completely randomly. The previous version is selecting rules completely at random from the start, but until a certain point outputs

	5-tuples	OpenFlow	IPv4 prefixes	IPv6 prefixes
`Naive Version`	24 969	18 947	27 510	28 130
`Corners Version`	27 964	18 432	29 004	28 418
`Prevent Rule Reusage`	12 305	5 276	870	5 043
`Prevent Rule Reusage 2`	34 438	19 999	34 005	30 553
`Smart Random 2.0`	264	1 032	126	146
`Smart Random 3.0`	335	1 018	132	147
`Smart Random 3.1`	73	124	138	145
`Combined Version`	80	138	131	156

Table 5.4: Number of generated headers per second by different generators.

only the headers generated based on a rule that has not yet been used. The Combined version is slightly faster than Smart Random 3.1, because it generates some headers using the much faster Prevent Rule Reusage 2.

Getting a high number of headers generated per second was not the goal of this thesis. Instead, it was important to achieve the best possible coverage of all present regions, while keeping the performance at a level which allows even common machines to generate a header trace in a reasonable time. This means that the actual number of headers per second is not that important, but the time complexity of the algorithm is. Smart Random 3.1, which is also used in the Combined Version, is the slowest of the algorithms. It is still satisfactory, because it has a linear time complexity $O(n)$, where n is the required trace size. The complexity is linear due to the fact that the iteration of the main loop has a constant worst-case time complexity and it is repeated at most once per each generated header. This has been confirmed through experiments.

	5-tuples		OpenFlow		IPv6 prefixes	
	real [kB]	virt [kB]	real [kB]	virt [kB]	real [kB]	virt [kB]
`Naive Version`	151 800	9 667	161 948	19 844	164 240	22 098
`Corners Version`	151 800	9 668	161 948	19 830	164 240	21 984
`Prevent Rule Reusage`	152 060	9 928	163 740	21 700	166 774	24 699
`Prevent Rule Reusage 2`	152 188	9 978	164 680	22 449	167 952	25 844
`Smart Random 2.0`	152 056	9 945	163 970	21 930	165 264	23 250
`Smart Random 3.0`	151 800	9 656	163 919	21 876	165 264	23 258
`Smart Random 3.1`	151 800	9 660	163 484	21 428	165 264	23 250
`Combined Version`	152 364	10 156	165 832	23 684	168 636	26 496

Table 5.5: Memory consumption of different generators.

Memory consumption of all versions of the generator is mostly driven by the size of a provided rule set. Peak values of real and virtual memory were measured. The reference machine, on which all performance tests were conducted, has insufficient main memory for the largest used rule set (IPv4 prefixes 2015_rrc00_ipv4_gen_b_100000). That lead to extensive virtual memory usage and, therefore, completely meaningless results when it comes to observing real memory consumption. This rule set is thus not included in Table 5.5, which shows real and virtual memory consumption of each generator's version. The more sophisticated versions have slightly higher demands on memory due to some extra variables that they use. The 5-tuple rule set consists of 1 000 rules and therefore requires less memory than the other two rule sets, which have 10 000 rules.

5.7 Possible Future Improvements

The Combined version has proven itself to be the most useful. Therefore, all future work would focus on it, unless an overall better algorithm would be found.

When writing the code of the generator, it was important to implement proposed algorithms quickly and without errors, so that they could be tested and evaluated as fast as possible. Time played a key role, because the results of experiments with each algorithm were needed before I could come up with new algorithms or improve the existing ones. During development, effectiveness of the code was secondary. For all of those reasons I decided to implement the algorithms in Python. Since the code is finished now, it could be further optimized to be more effective. More importantly, it could be rewritten in a different language, like C++, which should lead to faster execution speed and significantly more efficient memory management.

Advancing the code itself is not the only way to improve the generator. Currently, the execution time mostly hinges on the *search limit* constant. Decreasing this constant would lead to better execution time, but worse region coverage. The worsening of coverage is largely dependent on the rule set. While the current default value of *search limit* was chosen based on experiments with various values, it was selected rather conservatively in regard to preserving a high coverage. Finer tuning of the constant could potentially achieve a better compromise between execution time and coverage. Alternatively, it could be turned into a parameter of the generator and different users could use values that work best for them.

Similar balancing act could be done within the function that selects a secondary rule. Specifically, altering the maximum limit of fields where primary and secondary rules must have a non-wildcard value for the secondary rule to be selected. Just like the *search limit*, the current limit has been chosen based on a series of experiments with the main motivation to improve execution time, but to not hinder region coverage. Decreasing the limit would lead to faster run time, but worse coverage. Another possibility is to remove this condition entirely, which would guarantee the best possible coverage with this algorithm, but also the worst execution time. Depending on the circumstances, both approaches could be seen as improvements.

Last but not least, it is also possible to achieve better performance without sacrificing coverage through parallelization. The entire main loop of the Combined version could be fully parallelized. If the generator was rewritten in C++, I believe that it would be possible to create an effective parallel solution in a short period of time by using OpenMP.

Chapter 6

Conclusion

Packet classification is one of the most common operations in computer networks. It is often a performance bottleneck in routers. Creation of new, more efficient packet classification algorithms is being slowed down by a lack of proper rule sets and packets to test with.

ClassBench-ng is a synthetic rule set generating tool capable of generating realistic IPv4, IPv6 and OpenFlow 1.0.0 rules. However, there is no easy way to obtain packet header traces that are also needed for proper evaluation of the classification algorithms and devices that employ them. Main goal of this thesis is creation of a tool that would generate such header traces based on rules provided by ClassBench-ng.

An ideal header trace for such testing should contain at least one header values combination for every rule and each overlap of rules in a provided rule set. It is possible to achieve this, but only for rule sets up to a certain size. Memory consumption and computational demands become unfeasible for larger rule sets or rule sets containing many overlaps. The opposite approach is to generate the header values randomly, which is guaranteed to be very fast, but the resulting coverage of rules and their overlaps is very poor. The core of my work was to find a good compromise between those two approaches, that is, a compromise between performance and coverage.

As a part of this thesis I devised several different approaches to header trace generation, implemented them, and evaluated their properties. The evaluation was performed using a tool I had made specifically for that purpose. Creation of the different generator versions was an iterative process, where obtained results usually determined the direction of further improvements. Final version, the Combined version, achieved the best properties of all the implemented generators. It combines methods from two previous versions.

The two main contributions of this thesis are the evaluation tool and the final version of the generator.

There was no specific target value that the thesis should reach. Instead, the Corners version served as a benchmark which should be surpassed in terms of coverage, as it is the only previously used method of this kind of header trace generation. That was successfully achieved. Coverage provided by the final version of my generator is significantly better compared to the Corners version. The difference is most severe with OpenFlow rule sets. Performance-wise, the Combined version generator is considerably slower, but it has linear time complexity, which is satisfactory. Memory consumption of the two is comparable.

Bibliography

[1] Bradner, S. O.: The Internet Standards Process – Revision 3. BCP 9. RFC Editor. October 1996.
Retrieved from: https://tools.ietf.org/html/rfc2026

[2] Conta, A.; Deering, S.; Gupta, M.: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443. RFC Editor. March 2006.
Retrieved from: https://tools.ietf.org/html/rfc4443#page-2

[3] Deering, S.; Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. STD 86. RFC Editor. July 2017.
Retrieved from: https://tools.ietf.org/html/rfc8200

[4] Ganegedara, T.; Jiang, W.; Prasanna, V.: FRuG: A benchmark for packet forwarding in future networks. In *International Performance Computing and Communications Conference*. Dec 2010. ISSN 1097-2641. pp. 231–238. doi:10.1109/PCCC.2010.5682304.

[5] Hinden, R.; Deering, S.: IP Version 6 Addressing Architecture. RFC 4291. RFC Editor. February 2006.
Retrieved from: https://tools.ietf.org/html/rfc4291

[6] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-2011 (Revision of IEEE Std 802.1Q-2005)*. 2011.

[7] Kurose, J. F.; Ross, K. W.: *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson. 6 edition. 2012. ISBN 0132856204, 9780132856201.

[8] Matoušek, J.: *Addressing Issues in Research on Packet Classification in Core Networks*. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. 2019.
Retrieved from: https://www.fit.vut.cz/study/phd-thesis/642/

[9] Matoušek, J.; Antichi, G.; Lučanský, A.; et al.: ClassBench-ng: Recasting ClassBench After a Decade of Network Evolution. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. ANCS '17. Piscataway, NJ, USA: IEEE Press. 2017. ISBN 978-1-5090-6386-4. pp. 204–216. doi:10.1109/ANCS.2017.33.

[10] McKeown, N.; Anderson, T.; Balakrishnan, H.; et al.: OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*. vol. 38, no. 2.

March 2008: page 69–74. ISSN 0146-4833. doi:10.1145/1355734.1355746.
Retrieved from: https://www.net.t-labs.tu-berlin.de/teaching/ss09/IR_seminar/papers/openflow-wp-latest.pdf

[11] OpenFlow Switch Specification Version 1.0.0. 2009.
Retrieved from: https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[12] Parziale, L.; Britt, D. T.; Davis, C.; et al.: *TCP/IP Tutorial and Technical Overview*. IBM Redbooks. 8 edition. 2006. ISBN 0738494682.

[13] Postel, J.: Internet Control Message Protocol. STD 5. RFC Editor. September 1981.
Retrieved from: https://tools.ietf.org/html/rfc792

[14] Postel, J.: Internet Protocol. STD 5. RFC Editor. September 1981.
Retrieved from: https://tools.ietf.org/html/rfc791

[15] Stallings, W.: *Data and Computer Communications (8th Edition)*. Prentice-Hall, Inc.. 2006. ISBN 0132433109.

[16] Taylor, D. E.: Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*. vol. 37, no. 3. 2005: pp. 238–275.

[17] 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. doi:10.1109/TNET.2007.893156.