Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

FIIT-XXXX-XXXXX

Tomáš Belluš

Network traffic capture and analysis

Bachelor's thesis

Study program: Internet Technologies
Field of study: 9.2.4 Computer Engineering
Worked out at: Faculty of Informatics and Information Technologies, FIIT STU, Bratislava
Supervisor: Ing. Dušan Bernát, PhD.

December 2018

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

ZADANIE BAKALÁRSKEHO PROJEKTU

Meno študenta: **Belluš Tomáš**
Študijný odbor: Počítačové inžinierstvo
Študijný program: Internetové technológie
Názov projektu: **Zber a analýza dát o sieťovej premávke**

Zadanie:
Poznať rôzne parametre premávky uzlov v sieti môže mať význam z pohľadu administrácie, bezpečnosti alebo aj z teoretického hľadiska pri vypracovaní modelu, ktorý môže slúžiť na predikciu správania sa siete. Vypracovanie praktických nástrojov pre získavanie relevantných dát je základom pre ich ďalšie vyhodnotenie.
Analyzujte možnosti získavania údajov o sieťovej premávke v rôznych vrstvách, v OS typu Unix, v reálnom čase. Navrhnite systém pre zber a uchovávanie týchto dát (ako napríklad čas, IP adresy, porty, veľkosti a pod.) pre zvolený systém. Ďalej spôsob ich analýzy, vyhodnotenia a prezentácie, pričom systém by mal dať odpovede na otázky ako: v ktorých časoch sú využívané ktoré služby, v ktorých krajinách sa najčastejšie nachádzajú zdrojové a cieľové uzly komunikácie, aké objemy dát sa prenášajú, vytvoriť štatistický model dĺžky paketov, časové priebehy, trendy a predikcia uvedených charakteristík a pod. Systém implementujte a overte v reálnej premávke.

Práca musí obsahovať:

Anotáciu v slovenskom a anglickom jazyku
Analýzu problému
Opis riešenia
Zhodnotenie
Technickú dokumentáciu
Zoznam použitej literatúry
Elektronické médium obsahujúce vytvorený produkt spolu s dokumentáciou

Miesto vypracovania: Ústav počítačového inžinierstva a aplikovanej informatiky, FIIT STU, Bratislava
Vedúci projektu: Ing. Dušan Bernát, PhD.

Termín odovzdania práce v zimnom semestri 11. decembra 2018
Termín odovzdania práce v letnom semestri 07. mája 2019

SLOVENSKÁ TECHNICKÁ UNIVERZITA
V BRATISLAVE
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4
1

Bratislava 17.09.2018

Ing. Katarína Jelemenská, PhD.
riaditeľka UPAI

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Dušan Bernát, PhD.. All the relevant information sources, which were used during the preparation of this thesis, are properly cited and included in the bibliography.

in Bratislava, signature

.....
Tomáš Belluš

Contents

Chapter 1

Introduction

Ever since the first two network devices, whether in telecommunications or computer networking, exchanged data in some manner, we began to urge ourselves to send, receive and analyze data. Sending and receiving for communicating, and analyzing for understanding. Nowadays, these concepts are very actual in network security, network statistics and network devices such as routers, switches, firewalls, servers or any end-point devices. Applications require fast packet capture mechanisms for further processing, to keep up with the world's ever-increasing trend of transfer rates.

The Internet Protocol version 4 (IPv4) address range is, for a long time now, not enough for all devices connected to the largest network - the Internet. It implies that it is directly proportional to data being transfered over the Internet. This raises the need for applications, products or complex infrastructure solutions for keeping up with today's technologies.

With increasing network transfers, number of network devices, new frameworks and transfer protocols, improved algorithms and overall network security comes the threat of data theft, (distributed) denial of service attacks, compromised system or network and many other. Therefore, real time network traffic analysis is of utmost importance for national security authorities (mainly cybernetic security department), security or network companies and even end-point users. Network traffic analysis includes understanding and depicting various indications of either unlawful actions for security reasons or network transfers for statistical purposes. For both it means monitoring the correct functionality of network at hand.

Furthermore, each packet provides different decisive protocols that indicate information about the connection nodes and service used. A full understanding of the TCP/IP model and its layers is crucial for analyzing these services. Application, transport, network and link layer comprise the TCP/IP stack. Application layer is where applications exchange raw data, transport

layer connects sockets for data transfer, network layer forwards packets to destination and link layer checks credibility and handles the closest physical device connections over medium.

What are existing solutions to packet capturing mechanisms with network traffic analysis? How can we utilize existing packet capture mechanisms, reuse captured data analysis and identifying trends, threats and possibly predict future traffic? With packet capture and data analysis come storage systems, which need to be examined and evaluated. What known databases provide fast search process for real time network analysis and at the same time fast insertion rate, which would be sufficient for wire speed captures?

This thesis aims to answer and evaluate these questions. In addition, consider frameworks, systems and mechanisms explicitly for UNIX systems. Implement a system for packet capturing, store captured data to chosen database and finally analyze it with a framework or tools. Create visualizations of how the network behaves, depict trends, make predictions of given characteristics and analyze network packets on different TCP/IP layers.

Chapter 1 analyzes packet capturing process, distinctions among various mechanisms in performance and techniques, storage systems, data for analysis and existing solutions. Chapter 2 focuses on system specifications, system design including the system architecture and implementation including a data model.

Chapter 2

Analysis

2.1 Packet capture mechanisms

The most critical part is real time data capture in network traffic, meaning capturing network packets using a framework or interpreted language library. It is a dependent process of hardware components like Network Interface Controller (NIC), processor performance and its properties (e.g. number of processor cores). Capture of every packet, which can be processed by arbitrary mechanism ordinarily takes place in the higher system layer - the user space. More efficient mechanisms have access to data, which fall under the jurisdiction of lower system layer - kernel space. Though, why is it more effective and efficient to access data in lower system layers? Kernel space is the operation system's (OS) core and an interface to hardware for the OS. More precisely, it is the access to shared kernel space, which altogether bypasses the additional copying and processing present with basic frameworks not utilizing efficient features.

Received data on NIC is stored in NIC buffers. NIC registers keep track of whether the buffers are full and ready for reading or sending. This indication is handled by interrupts from the NIC to the OS. OS copies the buffer content to kernel space buffers (called *m_buffs* or *sk_buff*[1]). These buffers are not accessible by processes, but can be copied to user space or the process accessible memory. This is a typical scenario - two copies and interrupts needed for a process to access received data on the NIC. Possible improvements is to use, as mentioned above, buffers shared between kernel and user space, which eliminates one or more copies and interrupts from user to kernel space, leaving only one copy between NIC buffers and shared buffers. Other

[1]These kernel buffers are expensive to create and vary in size. They are complex and include large amount of metadata. [20]

possibility is to have a NIC supporting multiple buffers, which split the load and each buffer is handled by different core simultaneously [19].

A problem arises with the ability to capture all traffic on wire with no increasing delay resulting in packet loss due to lack of buffer space. Solution may be a zero-copy mechanism which utilizes a NIC dependent direct NIC access (DNA) or a one-copy mechanism, which utilizes the shared buffers. This section analyzes packet capture mechanisms, with various efficiency improvements at high packet rates in network traffic.

2.1.1 Scapy and RSS

Scapy[2] for python programming running on Linux systems operates in user space with no access to shared memory. At its core it utlizes Libpcap library and it is mainly a packet crafting library, packet decoder and a network sniffing mechanism on given interface [4]. In addition, it features filtering, detecting request responses and supports multiple existing protocols from all layers [10]. Even though, it has complex packet parsing, which makes it simple to extract any field, it is a huge bottleneck. It is present due to known python complexity in lower layers of its implementation. While using scapy is a fast solution from development point of view, the execution time is crucial here, therefore its packet capturing performance is insufficient (see section 2.2).

Additional resolution to the bottleneck, despite not using scapy, may be applying improvements to data processing on NIC driver level. It would require enabling CPU to process incoming packets by dividing received data among multiple CPU cores. On Linux platforms it is referred to as *Receive-Side Scaling* (`RSS`) and it "distributes network receive processing across several hardware-based receive queues" [17]. More precisely, each queue will be handled by single core independently and simultaneously. Unfortunately, this feature is dependent of the NIC driver.

2.1.2 Packet socket

Creating a file descriptor *packet socket*[3] provides receiving and sending packets at datalink layer. It means that packets are captured before any processing in the Linux network stack as raw frames including all protocol headers [2]. A packet socket with raw network protocol access (raw socket) is opened by a system call (see Listing 2.1), where the first argument must be

[2] https://scapy.net/

[3] http://man7.org/linux/man-pages/man7/packet.7.html

AF_PACKET[4] [3], which indicates the protocal family. *SOCK_RAW* paramenter identifies that it is a raw socket providing a whole encapsulated Ethernet frame. Macro parameter *ETH_P_ALL* specifies the socket protocol, which is expected on receive so in this case we require all protocols to be passed to socket interface [2].

```
1 int fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Listing 2.1: Raw socket system call

All in all, the packet socket is the interface receiving raw frames, or packets in case of trimming out link layer header, directly form the NIC and efficiently bypassing the network stack processing. This is an advantage concerning performance, because it produces only one copy of received data (from NIC driver to packet socket file descriptor) [28] [27]. The shared memory between kernel and user space is the packet socket file descriptor. In contrast with basic address family *AF_INET*, packet socket provides socket option for packet capture statistics (PACKET_STATISTICS), which is crucial for performance testing. In comparison with scapy it is the opposite in development and execution time point of view - it is presumably faster than scapy, but more complex to implement.

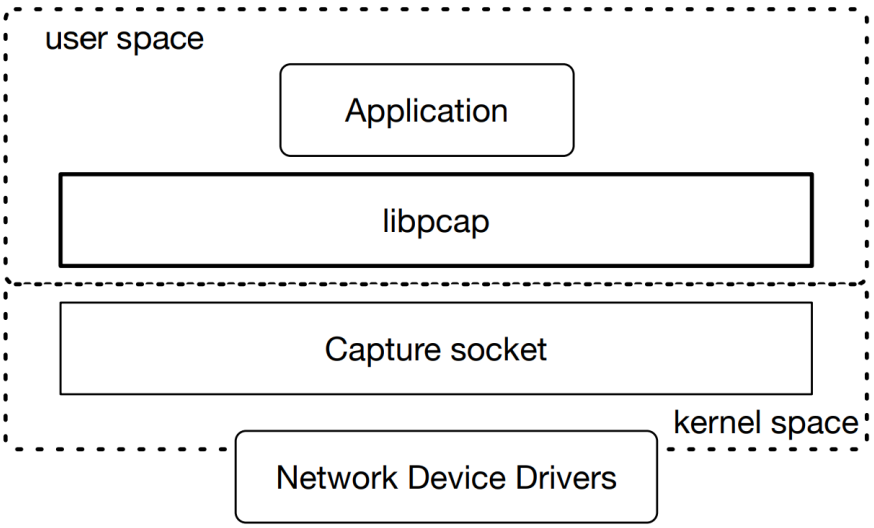


Figure 2.1: UNIX packet capture architecture for Libpcap or packet socket

[4] AF_PACKET is interchangable with PF_PACKET and they stand for Address Family and Protocol Family respectively specified the communication domain. [3]

2.1.3 Libpcap

Libpcap[5] is an user space library designed mainly for capturing Ethernet frames. It is a cross platform library for UNIX systems not only for C/C++ programming (e.g *tcpdump*[6] and *wireshark*[7]), but for other more abstract languages (Python, C#, Java, etc.) it exists in form of wrappers [9]. An user space library does not utilize kernel network stack, rather it reads raw Ethernet frames from opened socket queue in kernel space. Therefore, frame decapsulation is handled by libpcap and any protocol data, of parsed frame at hand, is accessed by libpcap's Application Programming Interface (API) (see Figure 2.1). Libpcap uses PF_PACKET [21] [8] since version 0.6 release (year 2001) with kernel 2.2 support and later [23].

Moreover, as of libpcap release 0.9.5, this library supports capture statistics for protocol family PF_PACKET by a *getsockopt()* system call [24]. In reference to scapy and packet socket, libpcap provides more abstraction to implementations by wrapping socket operations to library functions and similar performance to packet socket.

2.1.4 PF_RING

Moving from user space library to the kernel space modules and possible zero-copy mechanisms, brings *PF_RING*[8]. On socket layer it is a protocol family, replacing PF_PACKET, but it requires loading a kernel module. On application layer, PF_RING has an API for accessing received packets. PF_RING lacking extra enhancements and independent of NIC driver is referred to as PF_RING Vanilla. As of performance, through Linux New API (NAPI)[9] the kernel module copies polled packets from the NIC to shared memory ring buffers accessible by user application (see Figure 2.2) [15]. This process bypasses Linux network stack and the standard NIC driver. On the other hand, the performance depends on multiple hardware-based queues (RSS) in form of shared memory ring buffers (see Listing 2.2). Nevertheless, at least one shared memory ring buffer is created after loading the module to kernel. [16]

```
insmod driver_module.ko RSS=4,4
```

[5] http://www.tcpdump.org/manpages/pcap.3pcap.html
[6] http://www.tcpdump.org/manpages/tcpdump.1.html
[7] https://www.wireshark.org/
[8] https://www.ntop.org/products/packet-capture/pf_ring/
[9] NAPI is a performance increase for packet capturing at higher packet rates, by enabling packet polling, therefore decreasing number of interrupts which otherwise would overwhelm the CPU [5]

```
# enables 4 queues per interface (in this case two)
# the driver_module is the PF_RING enabled NIC driver
```

Listing 2.2: Enabling RSS [14]

Worth mentioning is the PF_RING_ZC, which is a zero-copy alternative to PF_RING Vanilla. It is strictly NIC driver dependent framework, which neglects NAPI, kernel modules and standard NIC driver to maximize efficiency by directly accessing NIC (DNA) (see Figure 2.2). Inserting the correct PF_RING-provided NIC driver enables an interface to be opened in zero-copy mode[10]. Any application can access packets through its API on 1-10 Gbit links at wire speed [12], if the kernel is bypassed. Otherwise the driver replaces the standard NIC driver and is faster compared to PF_RING Vanilla [13]. Disadvantages are, that while accessing NIC in zero-copy mode, standard networking is on hold until the device at hand is closed [13] and due to bypassing kernel packet filtering is missing [12].

Most valuable is the modified libpcap library[11] (pfring-libpcap) provided by the PF_RING framework. Pfring-libpcap requires the PF_RING module inserted and applications need to be recompiled with linking the new libpcap and specifically linking PF_RING library (libpfring) [11].

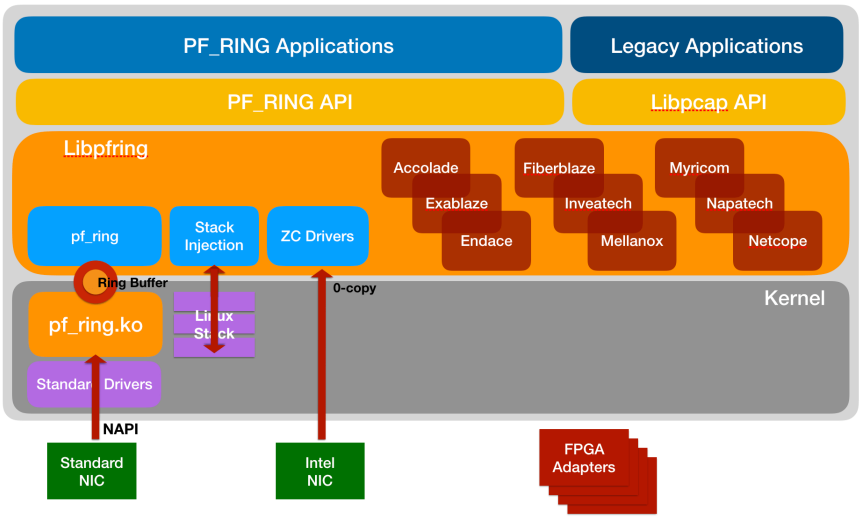


Figure 2.2: PF_RING variants and architecture overview

[10]Interface in the zero-copy mode has the "zc:" prefix (e.g. eth0 in zero-copy mode is accessed by zc:eth0). [12]

[11]https://github.com/ntop/PF_RING/tree/dev/userland/libpcap-1.8.1

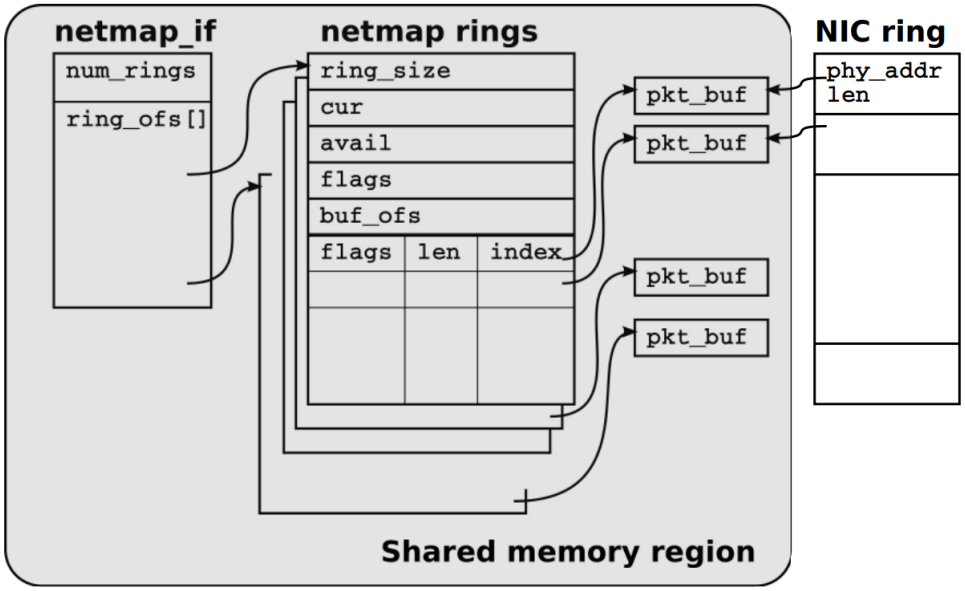


Figure 2.3: Netmap ring buffer design

2.1.5 Netmap

Last, but not least capture mechanism with an API is *netmap*[12]. The current paragraph is derived from a report on Luigi Rizzo's netmap [19]. "It is a framework to reduce the cost of moving traffic between hardware and the host stack" [19]. Netmap targets the processing bottleneck by utilizing, similarly as PF_RING or packet socket, the shared memory consisting of netmap packet buffers and netmap rings (buffer descriptors). The netmap packet buffers are shared between NIC rings and netmap rings, meaning that application processes access the packets in cost of one copy. In addition, the buffers are circular (ring buffers) and are designed (see Figure 2.3) to eliminate most of the processing time. Standard NIC uses ring buffers for received data and the implementation of netmap rings is a replica of those buffers. The *netmap_if* is a descriptor table of netmap rings mainly used when multiple netmap rings are used for load balancing. In contrast with standard Linux packet capturing (I/O operations) process, where the buffers holding packets (*sk_buffs* or *m_buffs*) are allocated and deallocated throughout the capturing process, the netmap framework preallocates *pkt_bufs* (see Figure 2.3) to bypass this time consuming operation.

Moreover, it requires a modified driver to be loaded to kernel, which provides maximum performance at wire rate if the netmap native API is used.

[12] http://info.iet.unipi.it/~luigi/netmap/

Just as PF_RING, netmap provides a netmap-based libpcap by creating a modified library, which maps libpcap standard calls to netmap calls [19]. Even though, the netmap libpcap has weaker performance in comparison with its native API, it does show improvement to the libpcap library [20]. For compatibility purposes, opposing to the native API the netmap libpcap may be the most efficient mechanism variant.

2.2 Performance testing

All of the above are potentially effective packet capture mechanisms, but only few are efficient enough to produce expected results in reasonable time. Performance testing of any mechanism means to simulate expected network traffic environment and mark down elapsed time or other performance output based on specific mechanism. Requirements for efficient capturing mechanism are minimal or no packet drop rate combined with wire rate capture, bypassing kernel processing and minimizing data copies. Packet drop rate is the ratio of dropped packets and total packets receievd by a NIC during a given period of time. At wire rate capture, packets are received with no delay. Bypassing kernel altogether (except for handling interrupts) and minimizing data copies results in performance increase and it is a crucial mechanism feature. These are the three aspects for testing and comparing all mechanisms analyzed.

This section focuses on performance testing process and its results. Both the drop rate of received traffic and time elapsed are measured and evaluated. Brief focus on testing automation, followed by the specific tests of both mentioned aspects and finally providing test results.

2.2.1 Automation

A bash script is used for efficient performance testing (see Appendices A and B) of both packet capture drop rate and elapsed time of captured traffic. For most mechanisms the implementation is straightforward, but running multiple executables (binary files) of single c program with various libpcap versions requires additional variables. It was the most crucial part of testing, since only one libpcap library version can be installed on a system. Appending the */etc/ld.so.conf* (see Listing 2.3) with directory paths of other libpcap versions partially solved the problem. Although, only one extra libpcap version can be added this way, since the linker works in a "first match" sense. For example, the target library was pfring-libpcap, but it was linked to netmap libpcap.

```
# paths to multiple libpcap libraries
/home/tomas/libpcaps/netmap-libpcap/
/home/tomas/libpcaps/PF_RING/userland/libpcap-1.8.1/
```

Listing 2.3: Contents of /etc/ld.so.conf.d/libpcaps.conf

Solution is to link the library directly with the source file and on execution, prefix the binary with overriding the LD_LIBRARY_PATH [1] environment variable (see Appendix C). This variable may contain library paths to be searched before the library configuration file is used (*/etc/ld.so.conf*). Setting the variable in line with execution makes it temporary, rather than sourcing it every other execution for different libpcap version. Using the *ldd* command, which prints the shared object dependencies, validates this method a success.

2.2.2 Tests

Figure 2.4: Diagram of testing scenario

Testing the drop rate of packet capture required a traffic generator to simulate a high frequency of packets. For this purpose, Kali Linux has a generator tool *hping3*, which is capable of sending 179 000 packets per second (179 Kpps). Even though, it is not a perfect real life traffic simulator it is sufficient enough to depict differences between some mechanisms. The network simulation testing environment consisted of two computers as shown in Figure 2.4. Drop rate is the best indicator of an efficient capture mechanism. Each test was constructed by the *hping3* command generating raw IP packets with 120-byte payload at different frequencies (see Appendix D). Each mechanism was implemented as a simple C program or python script and tested 5 times in 30 second time blocks (see Table 2.1). Refer to the Appendices for bash script testing drop rate.

In contrast, testing capture elapsed time proofs the wire speed capture of a mechanism. Similarly, *hping3* tool was used for generating traffic at maximum rate (179 Kpps) using flood mode (see Appendix D). Each mechanism was tested 5 times with various number of packets to be captured subse-

Drop rate test					
Packet rate [pkt/s]	scapy	packet socket	pcap	pfring	netmap
20 194 2 010 20 050 179 000					

Table 2.1: Testing for drop rate in various mechanisms

quently returning elapsed time (see Table 2.2). Refer to the Appendices for bash script testing elapsed time.

Elapsed time test					
NO. of packets	scapy	packet socket	pcap	pfring	netmap
10 100 1 000 10 000 100 000 1 000 000					

Table 2.2: Measuring the elapsed time of capturing various packet batches

2.2.3 Results

Graphs displaying results of the performance tests are included in the Appendices E and F.

Packet capture drop rate

Results have proven that scapy is not effective at higher packet rates (see Figure E.1), which is due to the python complexity. It makes the drop rate increase rapidly, until more packets are dropped than captured (see Table 2.3). Therefore, scapy is insufficient for capturing real time network traffic.

Moreover, other mechanisms were efficient and captured all the traffic (no packets were dropped due to lack of buffer space). Referring to Figure E.2,

Elapsed time test			
Packet rate [pkt-s/s]	received	captured	dropped
20	600	591	9
194	5 820	5 726	94
2 010	60 300	59 413	887
20 050	601 500	87 711	513 789
179 000	5 370 000	88 012	5 281 988

Table 2.3: Scapy measured drop rate

packet socket is significantly faster at capturing packets at small rates, because it captures packets in a flow (one by one), rather than in batches[13]. As the rate increases, all mechanisms have zero drop rate and captured all packets. Deciding for a mechanism accords for potential zero-copy feature (netmap and PF_RING), since all are equally efficient. Even though, netmap and PF_RING could be compared as equally efficient, there are some major aspects analyzed in their respective sections, which breaks the tie (e.g. NIC independence of the netmap mechanism). Finally, the netmap mechanism is the most suitable choice, because it provides minimum copies of data received, it has no measured drop rates and it is utilized in netmap libpcap library, which makes the program compatible with other libpcap versions.

Capture elapsed time

Measuring the elapsed time of a function comprises of marking the start time, execute target commands and mark end time. This way would be reliable, if the system's time would not change due to Network Time Protocol (NTP) bad update. Alternative is the monotonic clock, which goes forward independently of system's time. For python scripts monotonic clock is included in *time* module and for C programming it is included in *time* library by *clock_gettime()*[14] system call.

As it was for testing drop rate, elapsed time of packet capture with scapy was radically slow. In addition, scapy was not tested for the maximum of million packets because of the predicted exponential increase (see Figure F.1). In comparison scapy was processing more than 30 seconds, while other mechanisms processed the same load in half a second. All other mechanisms captured packets in wire speed - as the packet arrived on NIC it

[13]Capturing packets in batches means to copy or access multiple packets within one interrupt or read

[14]https://linux.die.net/man/3/clock_gettime

was processed with no delay. Figure F.3 compares average elapsed time for all considered mechanisms and netmap has processed million packets the shortest time. Even though, the bar graph shows differences between packet socket, netmap, PF_RING and libpcap, their performance could be the same due to inconsistent packet rate on wire. Nevertheless, netmap did not fail this performance test, so it remains as the most efficient choice.

2.2.4 Summary

There are multiple mechanisms considered and analyzed, but finally only netmap remained as the most efficient choice for both implementation and performance reasons. Therefore, netmap libpcap version in C/C++ programming language could be utilized for this thesis for compatibility and performance reasons.

2.3 Data storage

Initially, as it may seam, the most critical part is real time packet capture. Although, the real bottleneck in whole data processing is data storing in real time with acceptable time delay. Storing captured data includes parsing raw data and send them to a remote database server for further data analysis. This section analyzes considered database systems and options.

2.3.1 SQL versus NoSQL

Using a Structured Query Language (SQL) database requires knowing the data model of the expected data - Relational Database Management System (RDBMS). This would require a complex data model of all possible fields of network protocols. SQL database store create full entries of data despite multiple empty fields, which produces extra disk space usage.

However, Non-relational Structured Query Language (NoSQL) database provides high flexibility with its data and a faster search. Data is stored in JSON format, which makes it directly compatible with numerous parsers and any data manipulation. NoSQL databases are also characterized as key-value pair database systems, such as Redis database, which is a simple caching database, MongoDB or Elasticsearch (ELK stack).

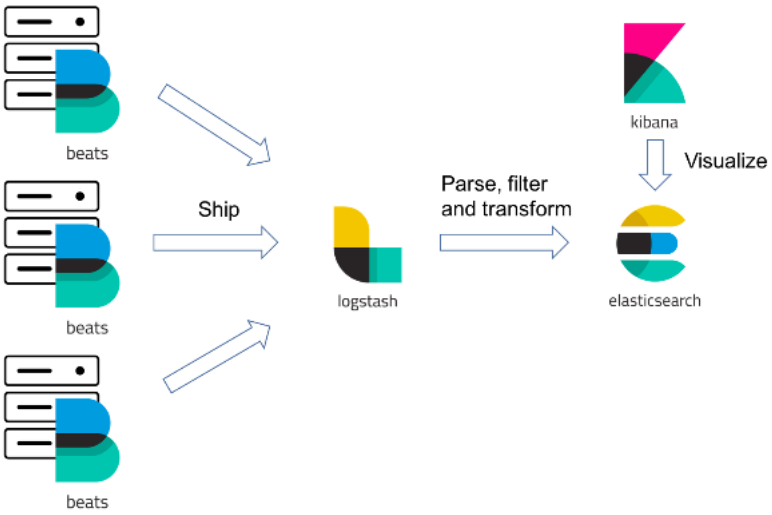


Figure 2.5: ELK stack set up including Filebeat

2.3.2 ELK stack

ELK stack[15] stands for *Elasticsearch*, *Logstash* and *Kibana*, but at this point there are more possible components of the ELK stack (e.g. *Filebeat*). The database component Elasticsearch, is a NoSQL database with no fixed data model. In comparison with a SQL database, where data is stored in records and tables, Elasticsearch stores data in documents and indices. Communication with an Elasticseach database is through various APIs using well known HTTP methods GET, POST, PUT and DELETE is in JSON query format.

An Elasticsearch index is populated by documents in JSON format and is defined by its settings API and a mapping API. Via settings, the default number of shards[16] and the number of replicas[17] [6] is defined. The mapping defines the document expected field's value type, but in case of lacking a concrete field mapping a document insertion does not fail.

Logstash provides parsing, filtering and queuing data streams from multiple sources to be inserted into Elasticseacrh database (see Figure 2.5). It is useful for data enrichment and modification before it is inserted in bulk[18] requests, rather than single documents. In its configuration are data input

[15] https://www.elastic.co/elk-stack

[16] Indices may be split into multiple sub-indices called shards.

[17] Shard copies, functioning as a failover. The number of replicas represents the number of copies of each shard.

[18] Documents wrapped in batches sent via one HTTP request [7].

streams' specifications, filter clauses including transformations and output stream, which is usually the Elasticsearch database. It reads various input streams like Filebeat, Elasticsearch or raw TCP data stream.

In contrast, Filebeat is a lightweight component reading raw inputs, such as system log files or TCP raw data stream, which are redirected either directly to Elasticsearch or Logstash for more transformations and queuing purposes.

Kibana is an optional component of the ELK stack capable of visualizing and monitoring data in Elasticseach (see Figure 2.5). It is an user friendly Elasticsearch interface, which abstracts the request queries when creating a visualization. For example, Kibana could serve as the Graphical User Interface (GUI) of the data stored in database for a network monitoring team of specialists. In addition, it recognizes a compatible timestamp formatted field for complex time period selections from any index or a specific data set. Moreover, to extend a field value, or creating a new field (visible only for Kibana) is possible by using scripted fields feature. All these features, creating visualizations or querying data in a NoSQL fashion are all advantages of using Kibana. Similarly as other ELK stack components, Kibana has a configuration file for specifying the ELasticsearch database.

In comparison with a SQL database, Elasticsearch documents are populated with only provided fields, despite being defined in the mapping or not. Whereas, SQL database does always insert into all fields even if they are empty. This Elasticsearch advantage saves space and makes the search elastic and fast. Utilizing ELK stack would require an external system connected to the sensor, or it could be residing on the same physical machine.

2.3.3 Redis

As an in-memory solution, where data analysis would take place as a part of the program and not as an external system connected to it, *Redis*[19] could be the solution. Redis is widely used as a caching database [18], which would require analyzing and implementing the visualization of data as a part of the final program. In addition, the data received would not be stored, rather cached in Redis database.

2.3.4 Sumamry

Since SQL databses are not suitable and Kibana provides the visualization tool for received data at arbitrary time period from the past, the ELK stack

[19] https://redis.io/

could be the right database system for this thesis.

2.4 Data visualization

Data visualization will be done by the ELK stack component Kibana. It meets all visualization requirements, such as selecting a specific time period, creating bar graphs, heat maps or trend lines and updating all as new data is received. This section discusses the data worth visualizing for further analysis. We will consider all layers of the TCP/IP model - Network layer, Transport layer and Application layer.

2.4.1 Network layer

Collected data at this layer include endpoint nodes, source and destination IP address, nested protocol of the higher layer, the the size of the packet at hand and the Time To Live (TTL) field. IP address is a valuable field, containing multiple properties of the endpoint. The IP address provides the location of the node, Autonomous System Number (ASN) of a specific Internet service provider, whether its an unicast, multicast or broadcast or it being a private or public IP address (regarding IPv4). Moreover, the Total length field (16-bit packet length) indicator can be used to detect large data transmissions. If the length is and very frequent, it could indicate unlawful activity or predict link failure for it would not withstand that high packet rate.

OS	TTL	Packet Size	DF Flag	SackOK Option	NOP Flag Option	Timestamp Option
MAC OS X	64	60	1	0	1	1
Linux	64	60	1	1	1	1
Windows	128	48	1	1	1	0
Cisco IOS	255	44	0	0	0	0

Figure 2.6: OS fingerprinting with known IPv4 fields

Additionally, according to an article on operating system (OS) fingerprinting with packets [22], the TTL, packet size, flags and other option fields in the IPv4 protocol header can be used to determine the OS of the source endpoint (see Figure 2.6). Based on this assumption, the network traffic is analyzed from more security concerned point of view, by grouping hosts by OS. Other frequent network layer protocol Internet Control Message Protocol (ICMP), as the name defines, is used in various network use-cases for notification purposes. For example, *traceroute* protocol utilizes a property of a network device responding with an *ICMP Time Exceeded* message, when

the TTL counter reaches zero. ICMP is also used in routing protocols, such as RIPv2, and is most recognized as protocol utilized by *ping* program used for debugging connectivity issues.

2.4.2 Transport layer

Transport layer provides more information on the communication, specifically port numbers, which identify the the service over which data is transmitted. Most recognizable Transport layer protocols are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is used for services requiring reliable transmissions and do not require fast delivery, whereas UDP provides fast transmit rates at cost not being reliable.

TCP

Services like SSH, FTP, Telnet, HTTP(S), BGP, SMTP and more are transfered by the TCP protocol. TCP header includes the source and destination port numbers, where usually one is higher (client public port 49152 - 65535) and the other is lower (server designated port 0 - 1023). According to this, it is possible to monitor when a well known service is being used in the network or when two endpoints are communicating with public ports. There are various indicators considering port numbers and the client-server model.

Furthermore, the TCP flags may be used to detected a session has opened or closed with the SYN and FIN flags respectively. When following the standardized three-way handshake, the SYN flag stands for synchronize and should indicate a session start. The FIN referring to final, finish or finalize indicates the session close. This is only a brief explanation and of course there are more TCP flagged packets to be sent in between.

UDP

UDP is much simpler protocol with minimum header fields. It is the source and destination port numbers, length of corresponding UDP header with data and an optional checksum field. UDP is used by DNS, DHCP, TFTP, NTP and more services, which in comparison with TCP, provide fast file transfer service (e.g. TFTP) or generic service transparent for basic user (e.g. DNS, DHCP or NTP). Like all services, these have specific port numbers too, which can be read from packet capture even if the communication is encrypted[20]. The UDP datagram length can be used for UDP datagram

[20]Communication is encrypted on the application layer.

size histograms and statistics or detecting a high data transmission on the network.

2.4.3 Application layer

Since most of network traffic on this layer is encrypted (e.g. SSH, HTTPS, etc.), there are minimum unencrypted protocols, which data could be analyzed or visualized. Unencrypted services include HTTP, Telnet over port 23, DNS, DHCP, FTP, TFTP and more. Raw data of these protocols could be used for deeper analysis.

Unencrypted and encrypted

In scope of this thesis, DNS packets may be the most valuable UDP-related application service for analysis. Instead of acquiring sole IP address, the prior DNS communication (if any) could provide the reversed host address registered on the IP address at hand.

Furthermore, HTTP, FTP and TFTP are unencrypted data transmitting protocols, which could carry potentially valuable informations. For example HTTP protocol could be used as the insecure communication between client and web server, database or any API. Intercepting this traffic could provide statistics of successful or failed requests ergo connections. File transfers over TFTP or FTP discloses all data, which is part of the transmitted message. For statistical reasons, it is not beneficial to capture this data, but detecting these protocols indicates usage of insecure transmission.

Others protocols (e.g. SSH and Telnet) are communication protocols acquiring connection with a host device for remote control. SSH is a secure communication protocol, which could be analyzed even when encrypted [25], but it is complicated. Nevertheless, the SSH communication is analyzed by detecting TCP protocol with source or destination port 22. In contrast, Telnet may be used for the same purposes as SSH, but it is not advised for it is an unencrypted communication channel. All sensitive data, such as user names, passwords or file system contents must be transmitted over to remote control terminal and thereby disclosing it all on the way.

2.5 Existing solutions

Existing software solutions Moloch, Wireshark and SolarWinds partially depict the problem at hand.

2.5.1 Wireshark

Wireshark[21] is a tool using standard libpcap library used for live deep packet analysis, pcap file analysis and following connection streams. It includes protocol hierarchy statistics, I/O graph view of received packets per second, endpoint and network session statistics and complex filtering options. Wireshark understands multiple protocols, outputs live packet capture and parses packets in hierarchical manner. Wireshark does not provide the required data visualizations.

2.5.2 Moloch

Moloch[22] is similar to wireshark in a way of reading pcap files and life network traffic. It utilizes Elasticseach cluster for fast search operations and captures data with PF_RING module. Moloch provides network sessions statistics, unique value occurrence in sessions (SPI view or graph), network connections graph view of search results and Elasticsearch cluster and Moloch capture node statistics. Although, various visualizations of received data is not included.

2.5.3 SolarWinds

Specifically *SolarWinds Deep Packet Inspection and Analysis tool*[23], processes each received packet, but summarizes the gathered information to protocol statistics and packet classifications. Includes network latency testing in network segment for troubleshooting purposes. Does not provide deeper packet analysis (e.g. various protocol header fields) or arbitrary visualizations of captured data.

[21] https://www.wireshark.org/
[22] https://molo.ch/
[23] https://bit.ly/2FUWBRn

Chapter 3

Problem solution

The solution to the analyzed components and merging it to one functioning system is described in the following chapter. The system specification, including its properties and use cases is outlined in section 3.1. The design of the solution and the overall system architecture is introduced in section 3.2. The concrete implementation techniques and class model is described in section 3.3.

3.1 System specification

The final system must include the network sensor implanted in network for capturing packets and the ELK stack. The sensor must then process and parse the received packets into a compatible JSON format for storing it to the database system via an API call.

The key processes that need to be included are as follows. Choosing an interface as the source of received traffic and specifying the direction (outgoing, incoming or promiscuous mode), specifying the database destination with an IP address and a port and provide filters to be applied on received packets. Furthermore, it must notify the user with potential errors or successful tasks (such as initialization of the sensor) in standard log files in designated system directory. The packet capture and further processing must not create a bottleneck and be ideally processing at wire speed. The system must have control over used space in the database. In such cases, the system finalizes the received data with summarized data and clear the disk space.

Considering the data visualization, various line graphs, bar graphs, heat maps and trend lines will be periodically updated as new data is received. These visualizations must be modifiable and it must be possible to create new visualizations. The final data set must include all desired protocol header

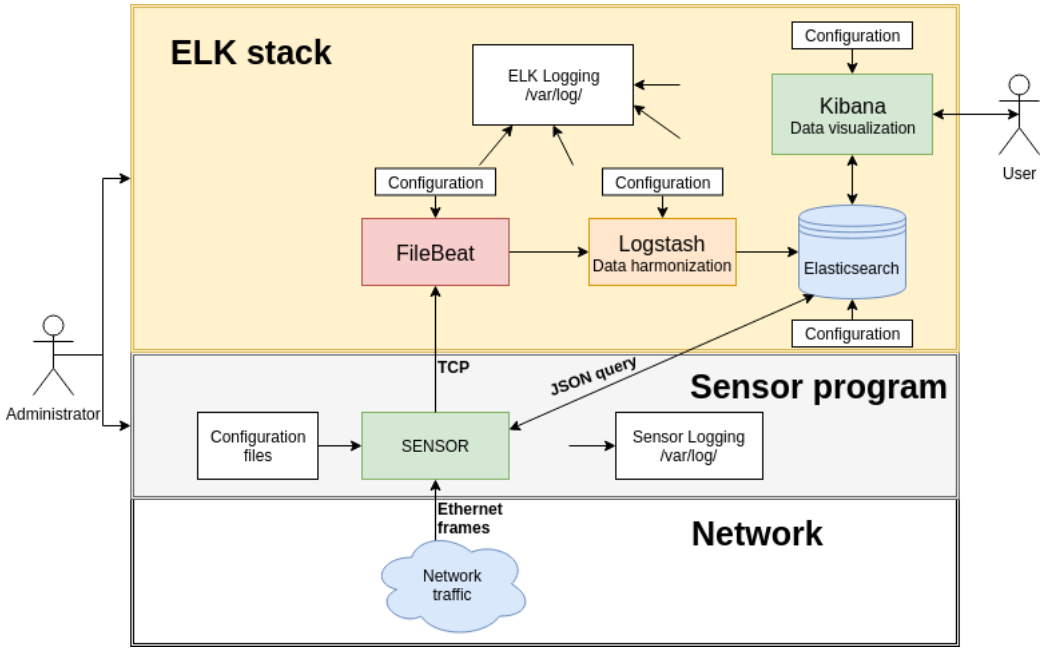


Figure 3.1: System architecture model

fields and packet meta data such as, timestamp of the received packet, source host of the received data and source interface of that host. The database system should not delay the data stream. The following visualizations must be possible to setup:

- What services in use at arbitrary time range.
- Show activity for a specific service at arbitrary time range.
- Network sessions' (host_1 <-> host_2) activity.
- What is the geographical location of network communication nodes.
- Packet rate or bytes transmitted per arbitrary time unit.
- Statistical packet length models or histograms.
- Trend lines for predicting future traffic.

3.2 Design

The sensor will run on top of netmap libpcap (supported by test results) and the chosen visualization tool and storage system is the ELK stack. Capturing packets on specified interface is part of the library. A suitable JSON

library will be used for parsing the packets to JSON structures, which are send to the ELK stack for storage and visualization (see Figure 3.1). In between Kibana and the sensor, Filebeat will accept TCP data flow on an open port and forward it to Logstash service. Then Logstash will add harmonization data, remove unwanted fields and store data via the Elasticsearch bulk API.

3.2.1 Data processing and storing

The sensor will be designed as a Linux program utilizing a configuration file, configured by the user. Both interface specification and filters will be included in these configuration file. Configuration file validation will report to log files success or error. Similarly, the capture will be logged on initialization and log error occurrence. For parsing packet structures, the most suitable library regarding conformance, parsing time, parsing memory and stringify time is *RapidJSON*[1] [26].

Before any packet capture, an Elasticsearch index will be created with a given mapping in JSON format. Keeping consistent protocol field types, such as integers, strings or IP addresses is crucial to bypass Elasticsearch field recognization. Filebeat will accept TCP data flow on an open port and forward it to the Logstash service. Filebeat additionally appends some requested meta data, such as source host of the data and timestamp. Logstash has full control over this data and it will remove some extra meta data and add arbitrary static fields like overall data type identification (e.g. "sensordata").

The sensor will periodically monitor the database disk usage, until it detects low free space. In such case, the data will be summarized by multiple query requests and a new identical index will be created. Similarly, it will monitor the sensor log files and periodically archive large log files.

3.2.2 Visualization

Requested visualizations are executed by Kibana. When a visualization is configured, it is periodically refreshed as new data is stored. Visualizations are grouped in dashboards (see Appendix G), which serves as the main page for further data analysis. In addition, Kibana includes *Developer tools* to request data with a JSON formatted query for fast search. Lastly, the *Discover* page shows the latest received data and basic statistics of incoming

[1] http://rapidjson.org/index.html

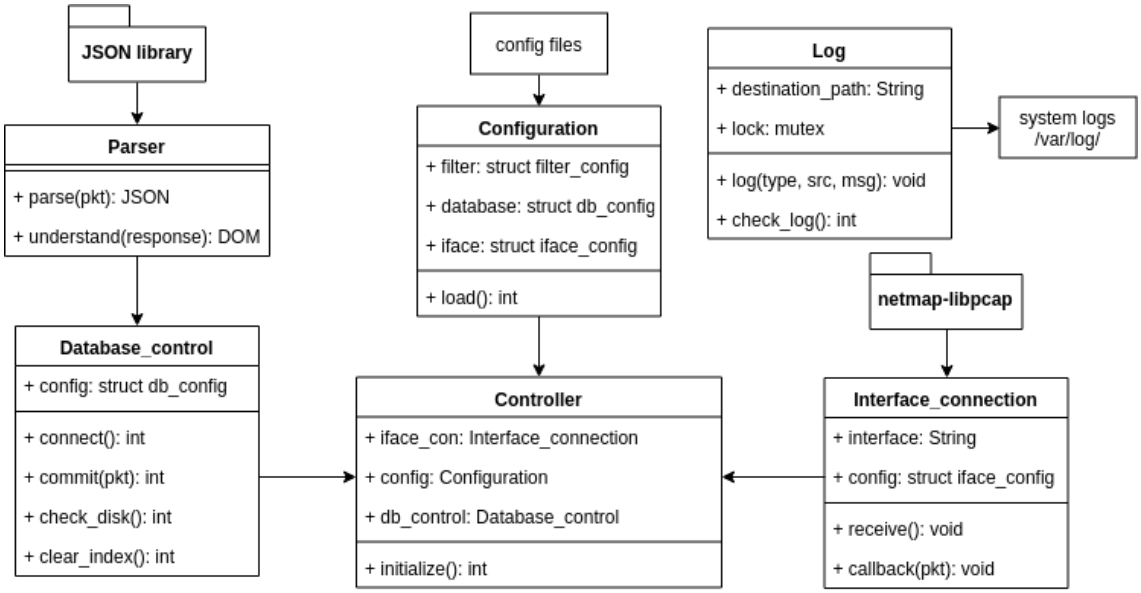


Figure 3.2: Class model of the implemented sensor

data to the Elasticsearch database and is used to monitor the received data frequency.

3.3 Implementation

According to all specifications and speed requirements, the programming language will be C++, well compatible with netmap libpcap. The sensor will consist of a specific interface connection, packet parsing, logging and database connection. Referring to Figure 3.1 the implemented component will be the sensor with external logging to files. The ELK stack is configured by respective configuration files and index creation via API call from Kibana developer tool or directly from remote host machine.

3.3.1 Sensor

Sensor implementation requires efficient multi-threaded solution, where each component will be viewed as an object (see Figure 3.2). The main requirement - configuration file, will have a standard Linux white space separated format (e.g. */etc/hosts* file) with allowed bash style comments. The Configuration class loads the configuration file from the */etc/sensor/* directory to specific C++ structures - *filter_config, db_configa, iface_config* and validates the file's format. Components Interface_connection and Database_control

are created with corresponding configuration structures, which contents are verified by connection attempt to database system, opening given interface file descriptor with specified filters. If any errors occur the message is logged and process terminated.

Configuration file

Configuration file must be located in the designated directory for program configurations (*/etc/sensor/conf.d/test.conf*). The main configuration file */etc/sensor/sensor.conf* will hold path to configuration files (see Listing 3.1).

```
1 /etc/sensor/conf.d/*.conf
```

Listing 3.1: Contents of main configuration file

The configuration will be loaded on sensor startup and each line is loaded to corresponding data structure. Validations of configuration file will be logged to specific sensor log file. The configuration file must include the following fields. The source interface name must correspond to the name listed by *ip link* or *ifconfig* commands. The possible capture direction modes are "out" for outgoing packets, "in" for incoming packets and "promisc" for promiscuous mode. Packet capture filters are not compulsory, because they slow down the capture process. Nevertheless the allowed syntax is specified by the Berkeley Packet Filter syntax[2]. The Filebeat and Elasticsearch connection specification must be a resolvable host IP address or host name and port must be a numbering ranging 0-65535. Archive path, must be an existing path on the local file system, where all summarized data will be archived. Archivation is limited by the *archive_limit* specified in mega bytes, when the index will be archived and cleared.

In case of missing component, the system will log it as an error and request correction. For sample configuration and default configuration refer to the Listing H.1 in the Appendices section.

Packet capturing

Netmap does not change the modified libpcap from programmers point of view. Packet capturing implementation consists of opening a file descriptor, applying filters to each received packet and opening a loop capture session (see Listing 3.2). After initialization of the *Interface_connection* class, the main packet receiving method will be a running thread in the *Controller* class.

[2] https://docs.extrahop.com/7.2/bpf-syntax/

```
1  struct bpf_program bpf;
2  bpf_u_int32 ip_addr, netmask;
3
4  pcap_lookupnet("eth1", &ip_addr, &netmask, error_buf);
5  pcap_t *handle = pcap_open_live("eth1", BUFSIZE, 1,
       1000, error_buf);
6  pcap_compile(handle, &bpf, "dst 8.8.8.8", 0, ip_addr);
7  pcap_setfilter(handle, &bpf);
8
9  pcap_loop(handle, 0, callback_fnc, NULL);
```

Listing 3.2: Libpcap capture code sample

All packet capture relying data is provided by the configuration file specifying the interface used, capture direction and filters.

Packet parsing

RapidJSON library provides the *Simple API for XML* (SAX API), which includes a JSON generator *Writer*. Writer converts keys and values into JSON in a procedural manner (see Listing 3.3). The parsing is part of *Parser* class and its main parse function takes a libpcap packet structure provided by the caller (*Database_control* class). Keys specified in the output JSON must correspond to the Elasticsearch database mapping field names. Parsing is expected to execute without errors, since the input is a libpcap packet structure and the output is a JSON string.

```
1  using namespace rapidjson
2
3  StringBuffer = json_pkt
4  Writer<rapidjson.StringBuffer> writer(json_pkt)
5
6  writer.StartObject();
7  writer.Key("ip.src");
8  writer.String("100.154.25.6")
9  writer.Key("ip.dst");
10 writer.String("78.56.1.236")
11 writer.EndObject();
12 // json_pkt == {"ip.src": "100.154.25.6", "ip.dst":
       "78.56.1.236"}
```

Listing 3.3: Packet parsing to JSON format

Committing packets to ELK stack

Connecting to the database system is part of the *Database_control* class. The connection parameters, including the Filebeat public IP address and the target port, are specified in a configuration file (see Listing H.1). The established connection with the ELK stack component Filebeat, will provide a TCP stream of raw JSON data. Committing is expected to fail if the connection with Filebeat is lost, which will be properly logged. Successful commit will not be logged, because of the high frequency of commits. Sending data to Filebeat needs to be as fast as it was received and it will require another live libpcap session to send raw TCP data packets. The *Parser* class provides the JSON data of the received packet to the *Database_control* class.

This class receives packet structures from the packet capturing thread, running in the *Controller* class, which has a running thread of committing packets to the database system. That is the multi-threaded part of the sensor.

Checking disk usage

In a separate thread, running in the *Controller* class, the *Database_control* class has a function monitoring the disk usage directly on the Elasticsearch node, whose connection specifications are read from the configuration structure. It will log the current database state (number of documents and disk usage) and check for low free space.

```
GET "elastic.bp:9200/_cat/shards?v"
[Sample output]
index   shard rep state   docs  store   ip        node
bp_test 2     r   STARTED 76661 415.6mb 10.0.0.1 Andromeda
```

Listing 3.4: Get number of documents and disk usage

When the free space gets low, specifically under a specified limit, a periodic function (see simplified code sample in Listing 3.5) checks for such scenario and executes specific tasks.

```
1 while (1) {
2   elastic = db_config.host + ":" + port;
3   // mark the latest document's date
4   latest_date = get_latest_date();
5   current_size = get_current_disk_size(elastic);
6
7   // check disk space
8   if (current_size < db_config.size_limit) continue;
9
```

```
10    // execute number of static and dynamically provided
          requests
11    // stores the request responses to specified path
12    summarize_index(elastic, db_config.archive_path);
13
14    // delete index by query
15    clear_index(elastic, latest_date);
16    sleep(CHECK_RATE);
17  }
```

Listing 3.5: Simplified *check_disk* function

Provided requests can be modified or created as a valid JSON formatted Elasticsearch GET API request query. The main configuration file */etc/sensor/requests.conf* will hold path to request files (see Listing 3.6).

```
1  /etc/sensor/requests.d/*.conf
```

Listing 3.6: Contents of main requests configuration file

The validity of GET requests is checked by Elasticsearch and received errors are logged. Additional GET requests must be tested on Elasticsearch beforehand, because there will be no warnings, besides the logs.

Deleting documents by query, queries the index and removes all matched documents. The deleted portion of index frees up space without interfering with frequent data storing. The summarized data in request responses will be stored in the archive path in parsed structure (parsing JSON to readable data using *Parser* class). According to the class model, the *understand* function returns a Document Object Model (DOM) object, which is a JSON parsed to C++ compatible object. Each archive file name will comprise of a timestamp and brief contents description distinguishing all archives.

Logging

Logging to */var/log/sesnor/* directory will be handled by the *Log* class. Logs will be structured including the event's timestamp, log message level, log's source and the raw message (see *log* function in Figure 3.2). The timestamp will be marked in the *Log* class and other parameters will be passed by the caller object. Message level will be statically listed in the *Log* class as an enumeration. For scalability purposes, each class utilizing logging will have a custom source identifier distinguishing it in the log file.

The existing message levels are infromative, warnings, errors or debugs.*INFO_LOG* notifies successful operations or messages of informative manner. *WARN_LOG* includes warning messages, which are avoidable, but a fix is advised. *ERROR_LOG* message often comprises of the exception message caught at sen-

sor start up and object initializations. *DEBBUG_LOG* describes an optional output in the log file.

Since, multiple sources will log to the same file, the *Log* class has a *mutex* lock object to handle mutual exclusion of the shared log file. On sensor start up or every 24 hours the log file size will be checked (see Listing 3.7). If the size exceeds 10 MB, it will be archived in the same directory and compressed using the *gunzip*[3] compression tool by an OS system call (log rotation). The *check_log* function will be running in a thread under the *Controller* class.

```
1  while (1) {
2    current_log_size = get_log_size(destination_path);
3    if (current_log < LIMIT_SIZE) continue;
4
5    // using ctime library
6    time_t now = time(0);
7    tm *ltm = localtime(&now);
8    current_date = std::to_string(ltm->tm_year + 1970)
9           + "-" + std::to_string(ltm->tm_mon)
10          + "-" + std::to_string(ltm->tm_mday);
11
12   // use mutex to block logging while archiving
13   unique_lock<mutex> lock(mutex);
14   // archive current log file
15   logfile = destination_path  + "/current.log";
16   archivefile = destination_path + "/current_" +
        current_date + ".log";
17   system("mv " + logfile + " " + archivefile);
18   system("gzip " + archivefile);
19   system("touch " + logfile);
20   lock.unlock();
21   sleep(WHOLE_DAY)
22 }
```

Listing 3.7: Simplified *check_log* function

3.3.2 ELK stack setup

The deployment of the ELK stack on a server will be handled by series of *Ansible*[4] scripts. It will install each ELK stack component, including setting up required configuration files. Every component configuration is described in the following subsections.

[3] https://www.gzip.org/
[4] https://www.ansible.com/

Elasticsearch

Configuring Elasticserach requires setting up a new index for data (see Listing 3.8) and filling out the configuration file located in */etc/elasticsearch/elasticsearch.yml* (see Listing 3.9). It can be create after correctly configuring Elasticsearch via *curl* program or via Kibana developer's tools.

```
PUT new_index
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1
  },
  "mapping": { "_doc": { "properties": {
        "source": {
          "properties": {
            "ip": { "type": "ip" },
            "asn": { "type": "integer" }
          }
        },
        "calssification": { "type": "keyword" }
  }
}}}
```

Listing 3.8: Sample of creating an index in Elasticsearch

The mapping of the index needs to include all fields. The proper way is to design it in a object form (e.g. *source* object mapping in Listing 3.9). Mapping will correspond to real life data types, meaning an IP address has the type *ip* or a geographical coordinates is of type *geo_point*. All mapping data types are listed in the Elasticsearch documentation[5].

```
cluster.name: "sensor_cluster"
node.name: "main_db"

path:
  logs: "/var/log/elasticsearch"
  data: "/var/data/elasticsearch"

network.host: "0.0.0.0"
http.port: "9200"
```

Listing 3.9: Elasticsearch configuration in YAML format

[5] https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-point.html

Path.logs and *path.data* specified the logging base directory and the path to archoved index data respectively. Most important is the *network.host* in the example, which specifies that any remote system can access the database at specified port. This security vulnerability will be resolved by a firewall setting on the ELK stack server, allowing only the sensor host and localhost connections.

Filebeat

Filebeat will be configured with a host IP address and a port accepting TCP data. The JSON decoder in Filebeat is notified of receiving JSON formatted data via a tcp stream. All received data will be logged to corresponding log file. The output stream will be forwarded to Logstash socket specified by *output.logstash.hosts* (see Listing 3.10). In the background, filebeat harmonizes the data by various fields.

```
1  filebeat.inputs:
2  - type: tcp
3    max_message_size: 20MiB
4    host: "0.0.0.0:12000"
5
6  json.message_key: tcp
7  logging.to_files: true
8
9  output.logstash:
10   hosts: "localhost:12345"
```

Listing 3.10: Filebeat configuration in YAML format

Logstash

Refering the to the configuration, Logstash will be set up to listen on an arbitrary port and tagging the input. The filter clause removes undesired fields, and renames fields for data field misconception elimination. The original *source* field holds the host address of the the source system sending data to Filebeat. The nested json clause creates a JSON object of the *message* in JSON string type. Once the JSON object is created, the *message* field may be deleted and a final field will be added to mark the future document with an identification of the filter clause used. The output clause sets the forwarding destination, which will be Elasticsearch database API, target index and document type.

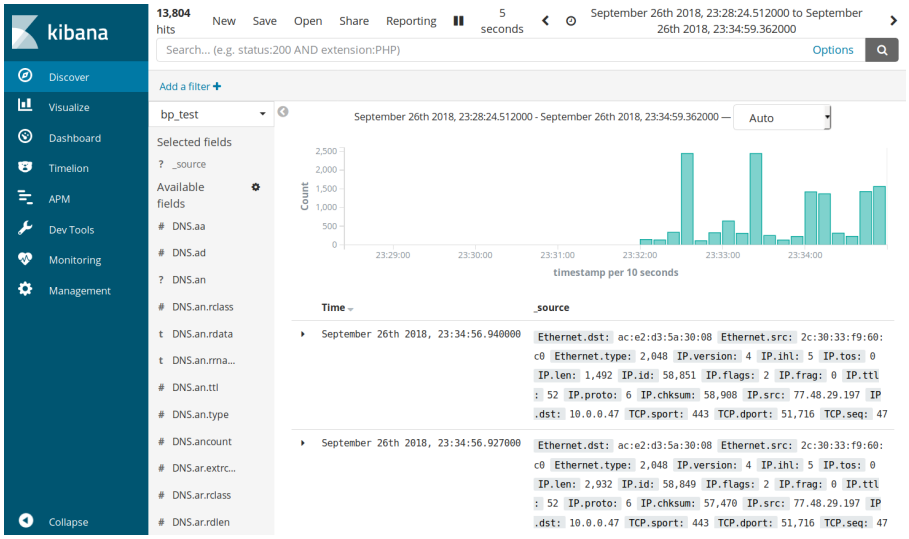


Figure 3.3: Kibana discover page of index *bp_test* with auto refresh and specific time period

Kibana

Kibana is configured by a configuration file */etc/kibana/kibana.yml*. The default configuration (see Listing 3.11) from the installation is correct, since both Kibana and Elasticsearch will coexist on the same physical machine. After all configuration files are correctly set and the Elasticsearch index is created, ELK stack is ready to receive data. The received data can be recognized and visualized by Kibana after setting up the index pattern, timestamp field and optionally adding script fields seen only by Kibana. New index pattern will be created under Management -> Kibana -> Index Patterns only after the index occupies at least one document. The index pattern timestamp fields is set in the second step. Afterwards, under Discover choosing the correct index pattern and setting up auto refresh (top right corner) will show new received data (see Figure 3.3).

```
1 # Deafult Kibana server settings
2 server.host: "0.0.0.0"
3 server.port: "5601"
4
5 elasticsearch.url: "http://127.0.0.1:9200"
```

Listing 3.11: Kibana sufficient default configuration

Bibliography

[1] 3.3. environment variables. http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html. [27.11.2018].

[2] *PACKET(7) Linux Programmer's Manual*, September 2017. http://man7.org/linux/man-pages/man7/packet.7.html [6.11.2018].

[3] *SOCKET(2) Linux Programmer's Manual*, September 2017.

[4] Philippe Biondi. About scapy. https://scapy.net. "[2.11.2018].

[5] corber. Napi. https://lwn.net/Articles/30107/, April 2003. [21.11.2018].

[6] elastic. Basic concepts. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html. [30.11.2018].

[7] elastic. Introduction to redis. https://www.elastic.co/guide/en/logstash/current/plugins-outputs-elasticsearch.html. [30.11.2018].

[8] Nicola Bonelli et. al. Enabling packet fan-out in the libpcap library for parallel traffic processing. Technical report, Universita di Pisa and CNIT. http://dl.ifip.org/db/conf/tma/tma2017/tma2017_paper65.pdf [20.11.2018].

[9] Luis Martin Garcia. Programming with libpcap - sniffing the network from our own application. *HAKING*, 3(2):39–46, February 2008. http://recursos.aldabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf [20.11.2018].

[10] Dirk Loss. Introduction. https://scapy.readthedocs.io/en/latest/introduction.html. [2.11.2018].

[11] ntop. Libpfring and libpcap installation. https://www.ntop.org/guides/pf_ring/get_started/git_installation.html?highlight=libpcap#libpfring-and-libpcap-installation. [21.11.2018].

[12] ntop. Pf ring zc (zero copy). https://www.ntop.org/guides/pf_ring/zc.html. [20.11.2018].

[13] ntop. Pf ring zc (zero copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/. [20.11.2018].

[14] ntop. Rss (receive side scaling). https://www.ntop.org/guides/pf_ring/rss.html. [20.11.2018].

[15] ntop. Vanilla pf ring. https://www.ntop.org/products/packet-capture/pf_ring/. [20.11.2018].

[16] ntop. Vanilla pf ring. https://www.ntop.org/guides/pf_ring/vanilla.html#. [20.11.2018].

[17] RedHat. Receive-side scaling (rss). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss. [19.11.2018].

[18] redis. Elasticseach output plugin. https://redis.io/topics/introduction. [30.11.2018].

[19] Luigi Rizzo. netmap: a novel framework for fast packet i/o. Technical report, Universita di Pisa. https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf [17.11.2018].

[20] Luigi Rizzo. netmap: A novel framework for fast packet i/o. https://www.youtube.com/watch?v=la5kzNhqhGs, July 2012. USENIX conference presentation [24.11.2018].

[21] Rami Rosen. Sockets in the kernel. Technical report, Haifux, 2009. http://haifux.org/lectures/217/netLec5.pdf [5.11.2018].

[22] Chris Sanders. Operating system fingerprinting with packets (part 1). http://techgenix.com/operating-system-fingerprinting-packets-part1/, August 2011. [1.11.2018].

[23] The tcpdump group. Summary for 0.6 release. https://github.com/the-tcpdump-group/libpcap/blob/libpcap-0.6.1/CHANGES. [19.11.2018].

[24] The tcpdump group. Summary for 0.9.5 release. https://github.com/the-tcpdump-group/libpcap/blob/libpcap-0.9.5/CHANGES. [19.11.2018].

[25] vivek. Traffic analysis of secure shell (ssh). https://www.trisul.org/blog/analysing-ssh/post.html, July 2017. [2.11.2018].

[26] Milo Yip. Native json benchmark. https://github.com/miloyip/nativejson-benchmark/blob/master/README.md, 2016. [4.11.2018].

[27] "Yusuf. Raw socket, packet socket and zero copy networking in linux. https://yusufonlinux.blogspot.com/2010/11/data-link-access-and-zero-copy.html. [4.11.2018].

[28] Jozef Zuzelka. Network traffic capturing with application tags, 2017. https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=159387 [12.11.2018].

Appendix A

Bash script for drop rate

```
1  #! /usr/bin/env bash
2
3  test "$#" -lt 2 && exit 1
4
5  time="$1s"
6  iterator=$2
7
8  pcap=()
9  pfring=()
10 netmap=()
11 socket=()
12 scapy=()
13
14 while [ $iterator -gt 0 ]; do
15   echo "$iterator more loop tests.."
16   # PCAP
17   result=$(timeout $time ./pcap)
18   pcap+=("$result")
19   echo "$iterator:   $result - pcap"
20
21   # PFRING
22   result=$(LD_LIBRARY_PATH=/home/tomas/pfring/PF_RING/
       userland/libpcap-1.8.1 timeout $time ./pfring)
23   pfring+=("$result")
24   echo "$iterator:   $result - pfring"
25
26   # NETMAP
27   result=$(LD_LIBRARY_PATH=/home/tomas/netmap-libpcap
       timeout $time ./netmap)
```

```
28    netmap+=("$result")
29    echo "$iterator:   $result - netmap"
30
31    # RAW SOCKET
32    result=$(timeout $time ./test_rawsocket)
33    socket+=("$result")
34    echo "$iterator:   $result - socket"
35
36    # SCAPY
37    result=$(timeout $time ./test_scapy.py)
38    scapy+=("$result")
39    echo "$iterator:   $result - scapy"
40
41    iterator=$(expr $iterator - 1)
42  done
```

Appendix B

Bash script for elapsed time

```
1  #! /usr/bin/env bash
2
3  test "$#" -lt 2 && exit 1
4
5  count=$1
6  iterator=$2
7
8  pcap=()
9  pfring=()
10 netmap=()
11 socket=()
12 scapy=()
13
14 while [ $iterator -gt 0 ]; do
15   # PCAP
16   result=$(./pcap $count)
17   pcap+=("$result")
18   echo "$iterator:   $result - pcap"
19
20   # PFRING
21   result=$(LD_LIBRARY_PATH=/home/tomas/pfring/PF_RING/
        userland/libpcap-1.8.1 ./pfring $count)
22   pfring+=("$result")
23   echo "$iterator:   $result - pfring"
24
25   # NETMAP
26   result=$(LD_LIBRARY_PATH=/home/tomas/netmap-libpcap ./
        netmap $count)
27   netmap+=("$result")
```

```
28    echo "$iterator:   $result - netmap"
29
30    # RAW SOCKET
31    result=$(./test_rawsocket.py $count)
32    socket+=" $result"
33    echo "$iterator:   $result - socket"
34
35    # SCAPY
36    result=$(./test_scapy.py $count)
37    scapy+=" $result"
38    echo "$iterator:   $result - scapy"
39
40    echo ""
41    iterator=$(expr $iterator - 1)
42  done
```

Appendix C

Linking various libpcap versions to one C source

```
1  root# gcc -o pfring test_libpcap.c /home/tomas/pfring/PF
      _RING/userland/libpcap-1.8.1/libpcap.so.1.8.1 -
      lpfring
2  root# gcc -o netmap test_libpcap.c /home/tomas/netmap-
      libpcap/libpcap.so.1.6.0-PRE-GIT
3  root# gcc -o pcap test_libpcap.c -lpcap
4
5  root# LD_LIBRARY_PATH=/home/tomas/netmap-libpcap ldd ./
      netmap
6    linux-vdso.so.1 (0x00007fff8fd0b000)
7    libpcap.so.1 => /home/tomas/netmap-libpcap/libpcap.so.
       1 (0x00007fc345fb2000)
8    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
       x00007fc345bc1000)
9    libnl-genl-3.so.200 => /lib/x86_64-linux-gnu/libnl-
       genl-3.so.200 (0x00007fc3459bb000)
10   libnl-3.so.200 => /lib/x86_64-linux-gnu/libnl-3.so.200
        (0x00007fc34579b000)
11   /lib64/ld-linux-x86-64.so.2 (0x00007fc3463f8000)
12   libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so
       .0 (0x00007fc34557c000)
13   libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0
       x00007fc3451de000)
14 root# LD_LIBRARY_PATH=/home/tomas/pfring/PF_RING/
      userland/libpcap-1.8.1 ldd ./pfring
15   linux-vdso.so.1 (0x00007ffe22335000)
16   libpcap.so.1 => /home/tomas/pfring/PF_RING/userland/
```

```
        libpcap-1.8.1/libpcap.so.1 (0x00007f1fcaf42000)
17    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
        x00007f1fcab51000)
18    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so
        .0 (0x00007f1fca932000)
19    librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0
        x00007f1fca72a000)
20    libnl-genl-3.so.200 => /lib/x86_64-linux-gnu/libnl-
        genl-3.so.200 (0x00007f1fca524000)
21    libnl-3.so.200 => /lib/x86_64-linux-gnu/libnl-3.so.200
         (0x00007f1fca304000)
22    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0
        x00007f1fca100000)
23    /lib64/ld-linux-x86-64.so.2 (0x00007f1fcb40c000)
24    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0
        x00007f1fc9d62000)
25  root# ldd ./pcap
26    linux-vdso.so.1 (0x00007fff365ee000)
27    libpcap.so.0.8 => /usr/lib/x86_64-linux-gnu/libpcap.so
        .0.8 (0x00007f393b7b0000)
28    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
        x00007f393b3bf000)
29    /lib64/ld-linux-x86-64.so.2 (0x00007f393bbf4000)
```

Appendix D

Hping3 traffic generator

```
1  hping3 command setup:
2  hping3 -I eth0 --rawip -d 120 -i u<interval> 18.x.x.x --
       rand-dest --rand-source
3
4    -d 120        --> include 120 bytes of data
5    -i <interval> --> set the interval in microseconds
6    --flood       --> generate maximum number of packets
7    --rand-source --> set random source IP address
8    --rand-dest   --> route packets to random destination
       IP address
9    18.x.x.x      --> specifies range of destination
       addresses
10
11 testing script execution:
12 ./test_mechanisms_droprate.sh 30 5
13 30            --> 30 second timeout for each mech
14 5             --> 5 iterations
15
16 intervals used:
17 50000         -->  20      pkts/s
18 5000          -->  194     pkts/s
19 450           -->  2010    pkts/s
20 32            -->  20050   pkts/s
21 --flood       -->  179000  pkts/s
```

Appendix E

Packet capture graphs

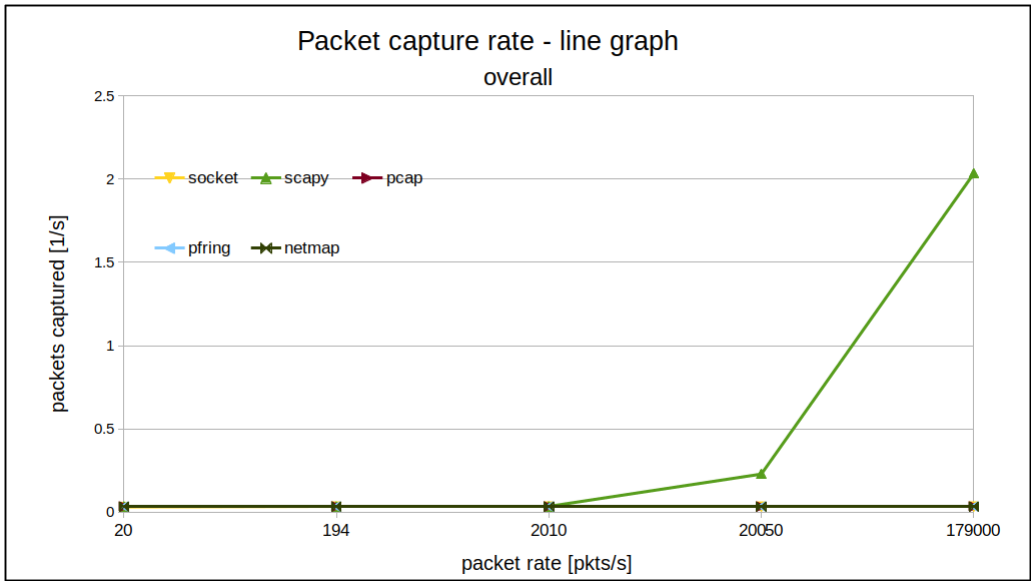


Figure E.1: Capture performance for testing drop rate

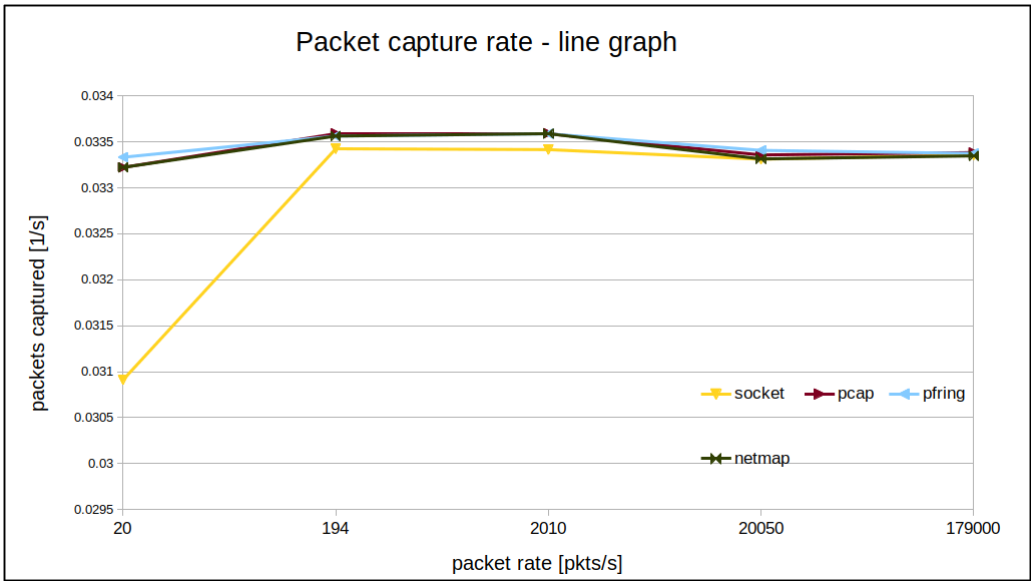


Figure E.2: Capture performance for testing drop rate. Number of packets captured in a second

Appendix F

Packet capture elapsed time

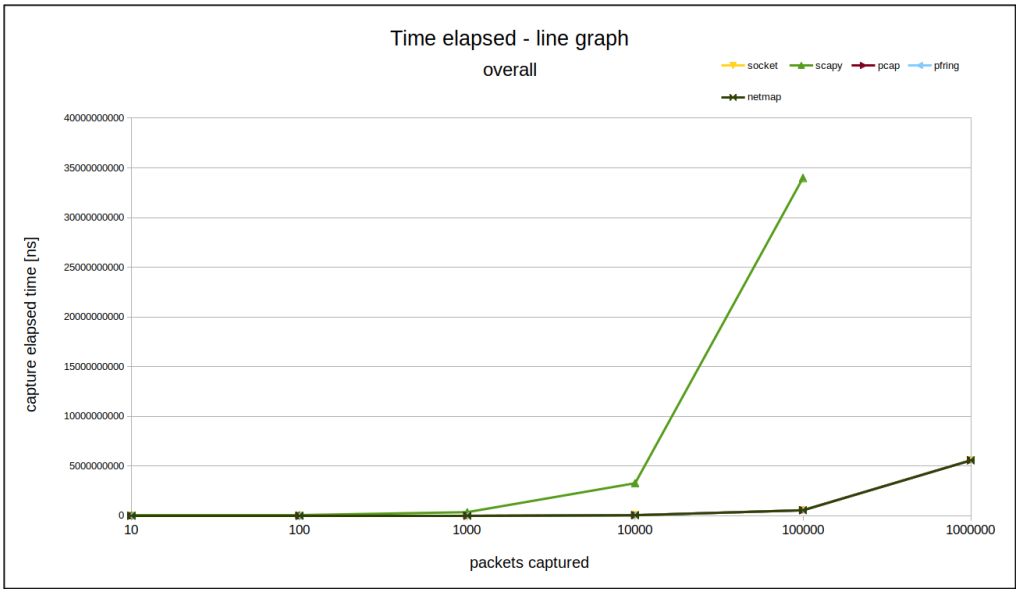


Figure F.1: Capture performance for testing elapsed time

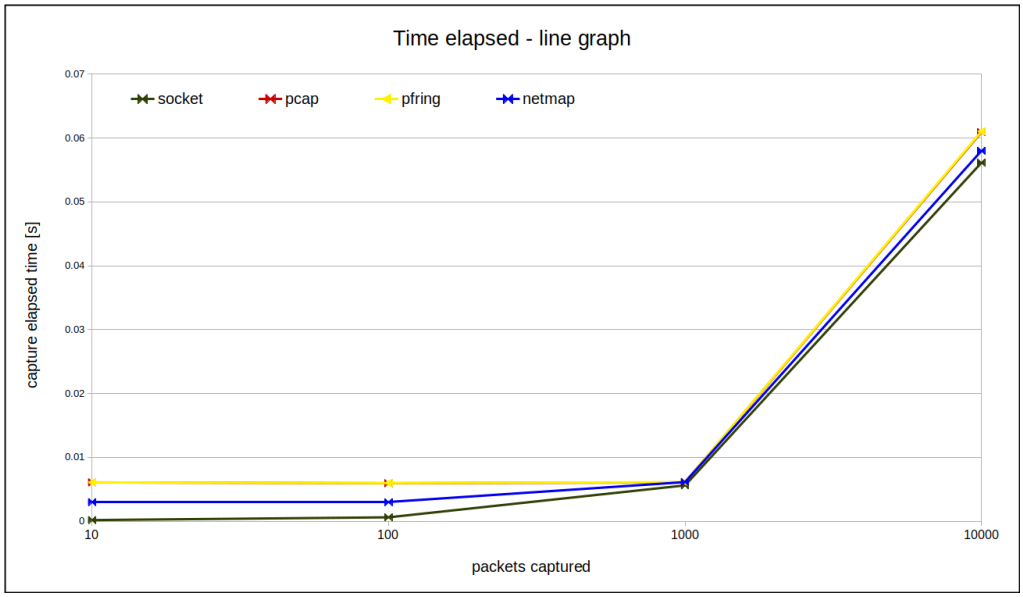


Figure F.2: Close up graph of capture performance for testing elapsed time and discarding scapy.

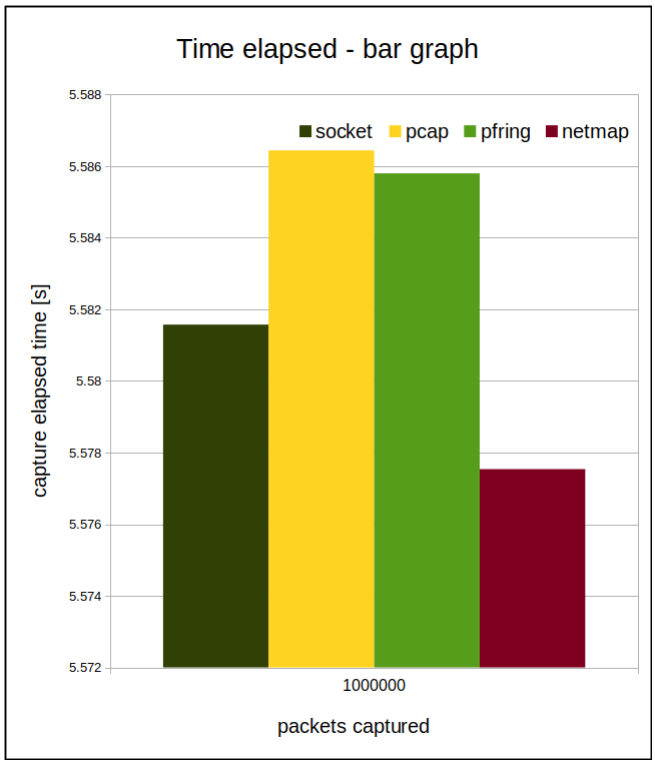
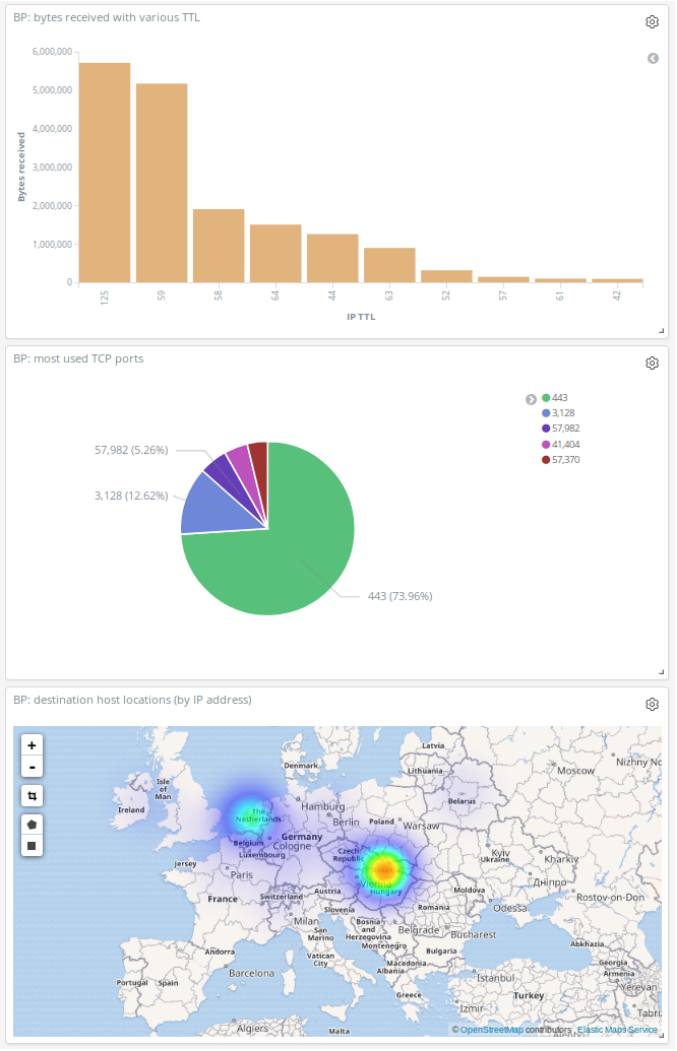


Figure F.3: Best performance at million sent packets

Appendix G

Kibana dashboard



Appendix H

Sensor configuration

```
1  # contents of the /etc/sensor/conf.d/example.conf
2
3  # capture interface and mode
4  #default interface => not specified
5  interface      eth1
6  #default direction => promisc
7  direction      promisc
8
9  # filters as allowed traffic
10 # default => not specified
11 filter         src 10.0.100.56, dst 8.8.8.8
12
13 # Filebeat connection specification
14 # default host => localhost:12000
15 beats.host     192.148.56.4
16 beats.port     12000
17
18 # Elasticseach connection specification
19 # default => localhost:9200
20 elastic.host   192.148.56.4
21 elastic.port   12000
22
23 # archive output path and archivation condition
24 # default => /etc/sensor/
25 archive_path   /home/user/sensor_archive/
26 # default => 10000 [MB]
27 archive_limit 50000
```

Listing H.1: Configuration example

Appendix I

Logstash configuration

```
input {
  beats{
    port => 12345
    type => "bp_sensor"
  }
}

filter {
  if [type] == "bp_sensor" {
    mutate {
      remove_fiueld => ["type", "beat", "input_type", "
          offset", "fields"]
      rename => ["source", "_host_source"]
    }
    json {
      source => "message"
    }
    mutate {
      remove_field => "message"
      add_field => { "_logstash_input_flow_id" => "bp_
          sensor" }
    }
  }
}

output {
  if [_logstash_input_flow_id] == "bp_sensor" {
    elasticsearch {
      hosts => "localhost:9200"
```

```
28        manage_template => false
29        index => "bp_test"
30        document_type => "_doc"
31      }
32    }
33  }
```

Appendix J

Plan of work in winter semester

```
1  ===12.6.2018===
2  - My supervisor and I decided on using python.
3  - Look up a suitable library.
4  - Think about zero-copy (minimum number of copies)
       mechanims.
5  - Look up existing solutions.
6  - Test scapy + Redis and scapy + Elasticsearch.
7
8
9  ===20.9.2018===
10 - We discussed the structure of teh thesis.
11 - Redis is not a good choice for its efficiency decrease
       at data increase.
12 - Produce a prototype of scapy + Elasticseach + Kibana
       visualizations.
13
14
15 ===1.10.2018===
16 - The prototype was a success and i can continue on
       developing Elasticsearch + Kibana.
17 - We discussed the possible visualizations.
18   - I continued imporving the prototype towards that end
         .
19 - Search for fast capture mechanisms.
20
21
22 ===8.10.2018===
23 - Begin performance testing of various capture
       mechanisms with hping3 traffic generator tool
```

```
24    - tcpdump, netsniff-ng, pfring-tcpdump
25  - Write thesis structure.
26
27
28  ===15.10.2018===
29  - Add other mechanisms for performance testing.
30    - raw socket, libpcap
31  - Both pfring-tcpdump and netsniff-ng have no drop rate
       oposing to tcpdump.
32  - Proper performance testing with capturing elapsed time
       and capture statistics.
33  - Create graphs representing mechnisms' performance.
34
35
36  ===22.10.2018===
37  - Test pfring libpcap and standard libpcap libraries.
38  - Read about netmap framework.
39  - Scapy is not usable, since its performance it very
       poor.
40  - Netsniff-ng is also not usable, since its only a
       capture utility with no API.
41  - Raw socket is best so far (especially for low packet
       rate)
42
43
44  ===29.10.2018===
45  - Work on ELK stack (the database interface) and its
       performance.
46  - Implementa simple single libpcap program and compile
       it with all libpcap libraries.
47    - netmap-libpcap, pfring-libpcap, standard libpcap.
48  - Python is no longer the target programming language =>
       C/C===.
49
50
51  ===05.11.2018===
52  - Linked multiple complied libpcap versions to a single
       libcpap program.
53  - Their performance is mostly comparable, but netmap is
       by far the best.
54  - We discussed the contents of multiple chapters of the
       thesis.
55  - Started writing analysis.
```

```
56
57
58 ===12.11.2018===
59 - Netmap provides a fast traffic generator (pkt-gen),
      but wasn't able to reach its full potential (9 Mpps).
60 - ELK stack performance is very low, but Kibana is very
      suitable program for visualizations and Elasticsearch
       has fast search.
61 - Possible theoretical improvemnet is to enlarge the
      Elasticseach cluster by several nodes.
62
63
64 ===22.11.2018===
65 - Finished mechanism analysis.
66 - Started with performance testing description.
67
68
69 ===29.11.2018===
70 - Finished Analysis.
71 - Started with solution design.
72
73
74 ===06.12.2018===
75 - Finishing up with Design.
76 - Started on implementation outline
```

Appendix K

Plan of work in summer semester

```
===11-02-2018===
- Start deploying functional ELK stack.
- Begin scripting Ansible automation


===18-02-2018===
- Test automated (Ansible) deployment of the ELK stack.
- Begin programming the sensor (Controller class) and
   setting up its configuration file.
- Load configuration file to C++ structures.
- Configuration object verifying configuration file.


===25-02-2018===
- Add test index for checking database connectivity.
- Implement the prototype Database class and Log class.
- Redirect error messages and other to log file.


===04-03-2018===
- Start network capturing development of the Interface_
   connection class.
- Include Parser class with RapidJSON library.


===11-03-2018===
- Include multi-threading between database control
```

```
    object and capturing object.
- Insert first packet to the database.
- Add complete packet parsing in the Parser class.


===18-03-2018===
- Add monitoring features (check database disk space and
    log file disk usage).
- Parse Elasticsearch response JSON of summarized data.
- Test delete_by_query API.
- Test archiving log files and logging continuity.


===25-03-2018===
- Add initialization logging and release alpha version.
- Create examplary visualizations in Kibana.
- Version testing with various sensor configurations.


===01-04-2018===
- Fix misconceptions and optimize program.


===08-04-2018===
- Structure technical documentation.
- Possible changes in program (optimizations).


===15-04-2018===
- Final testing and documentation.
- Start writing missing thesis sections.


===22-04-2018===
- Empty week for corrections and finalization.


==29-04-2018===
- Release version 1.0.0.


===06-05-2018===
- Project deadline 07-05-2018.
```