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

FIIT-104199-82385

Tomáš Belluš

Network traffic capture and analysis

Bachelor thesis

Study program: Internet Technologies
Field of study: 9.2.4 Computer Engineering
Training workplace: Institute of Computer Engineering and Applied Informatics
Supervisor: Ing. Dušan Bernát, PhD.

May 2019

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

Annotation

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Degree Course: Internet Technologies

Author: Tomáš Belluš
Bachelor thesis: Network traffic capture and analysis
Supervisor: Ing. Dušan Bernát, PhD.
May 2019

Goal of this thesis is to implement an efficient packet capture tool with storage and analysis of the received data. Key features are fast capture mechanism utilization, storage system and network traffic analysis tool.

Anotácia

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Internetové Technológie

Autor: Tomáš Belluš
Bakalárska práca: Zber a analýza dát o sieťovej premávke
Vedúci bakalárskej práce: Ing. Dušan Bernát, PhD.
May 2019

here

Acknowledgements

Most of all, I am pleased to thank my supervisor for his every advice and maintaining the work on its course. Furthermore I would like to thank my coworkers Ing. Tibor Csóka, PhD., Bc. Ján Skalný and por. Ing. Ján Doboš for their consulting and specialistic advices.

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 2 analyzes packet capturing process, distinctions among various mechanisms in performance and techniques, storage systems, data for analysis and existing solutions. Chapter 3 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. [27]

possibility is to have a NIC supporting multiple buffers, which split the load and each buffer is handled by different core simultaneously [26].

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 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 [10]. In addition, it features filtering, detecting request responses and supports multiple existing protocols from all layers [17]. 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" [24]. 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 [6]. 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] [7], 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 [6].

```
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) [35] [34]. 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.

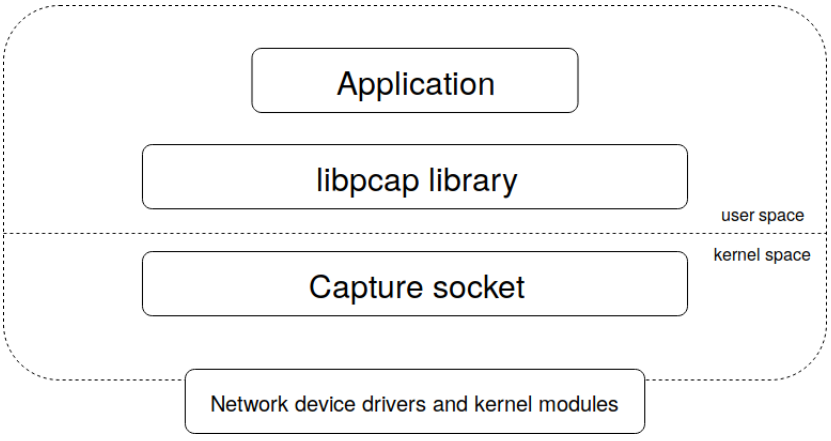


Figure 2.1: UNIX packet capture architecture for Libpcap or packet socket.

[4] AF_PACKET is interchangable with PF_PACKET and they stand for Address Family and Protocol Family respectively specified the communication domain. [7]

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 [15]. 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 [28] [14] since version 0.6 release (year 2001) with kernel 2.2 support and later [30].

Moreover, as of libpcap release 0.9.5, this library supports capture statistics for protocol family PF_PACKET by a *getsockopt()* system call [31]. 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) [22]. 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. [23]

```
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 [11]

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

Listing 2.2: Enabling RSS [21].

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 [19], if the kernel is bypassed. Otherwise the driver replaces the standard NIC driver and is faster compared to PF_RING Vanilla [20]. Disadvantages are, that while accessing NIC in zero-copy mode, standard networking is on hold until the device at hand is closed [20] and due to bypassing kernel packet filtering is missing [19].

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) [18].

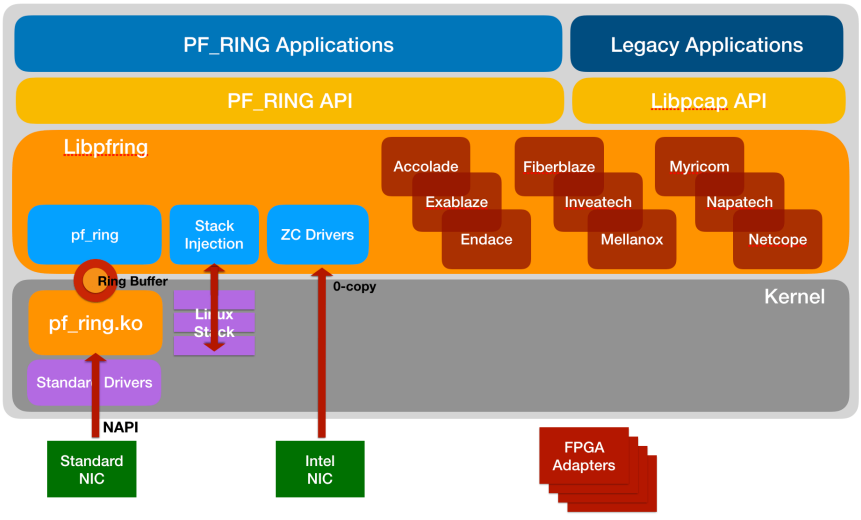


Figure 2.2: PF_RING variants and architecture overview [4].

[10]Interface in the zero-copy mode has the "zc:" prefix (e.g. eth0 in zero-copy mode is accessed by zc:eth0) [19].

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

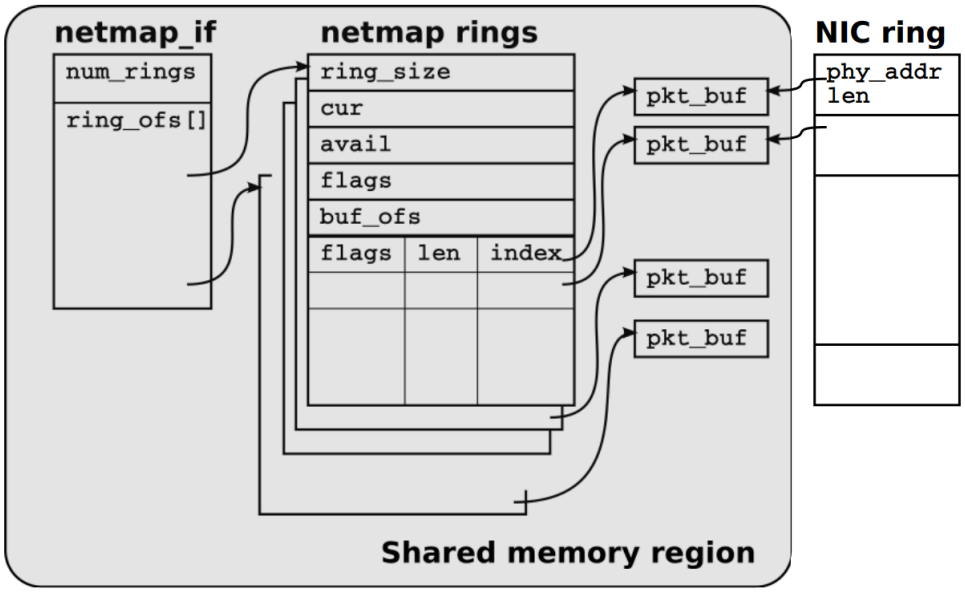


Figure 2.3: Netmap ring buffer design [5].

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 [26]. "It is a framework to reduce the cost of moving traffic between hardware and the host stack" [26]. 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 [26]. Even though, the netmap libpcap has weaker performance in comparison with its native API, it does show improvement to the libpcap library [27]. 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 received 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. Packet capture efficiency (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 C and D) of both packet capture efficiency 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* 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 (see Listing 2.3).

```
# 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 program source file, and on execution prefix the binary with overriding the LD_LIBRARY_PATH [1] environment variable (see Appendix E). 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

Machine	CPU	RAM	NIC	OS
Laptop A	Intel i5-7300HQ, 2.50GHz x4	16 GB	Realtek 1Gb	Ubuntu 18 64bit
Laptop B	Intel i5-2430M, 2,40Ghz x2	6 GB	Realtek 1Gb	Kali Linux 64bit

Table 2.1: Computer hardware specifications

Testing the packet capture efficiency 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) on a Realtek[13] adapter. Even though, it is not a perfect real life traffic simulator it is sufficient enough to depict differences between some mechanisms. The testing environment consisted of two computers as shown

[13]RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller

in Figure 2.4 with specifications in Table 2.1. 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 F). Each mechanism was implemented as a simple C program or Python script and tested 5 times in 30 second time blocks. Refer to the Appendices for bash script for testing packet capture efficiency.

In contrast, testing capture elapsed time proves 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 F). Each mechanism was tested 5 times with various number of packets to be captured subsequently returning elapsed time. Refer to the Appendices for bash script testing elapsed time.

In addition to this specific performance testing, a more precise and valuable tests were performed as part of the IIT.SRC 2019 conference included in Appendix G.

2.2.3 Results

Additional results conducted as part of the IIT.SRC 2019 conference are included in Appendix G.

Packet capture efficiency

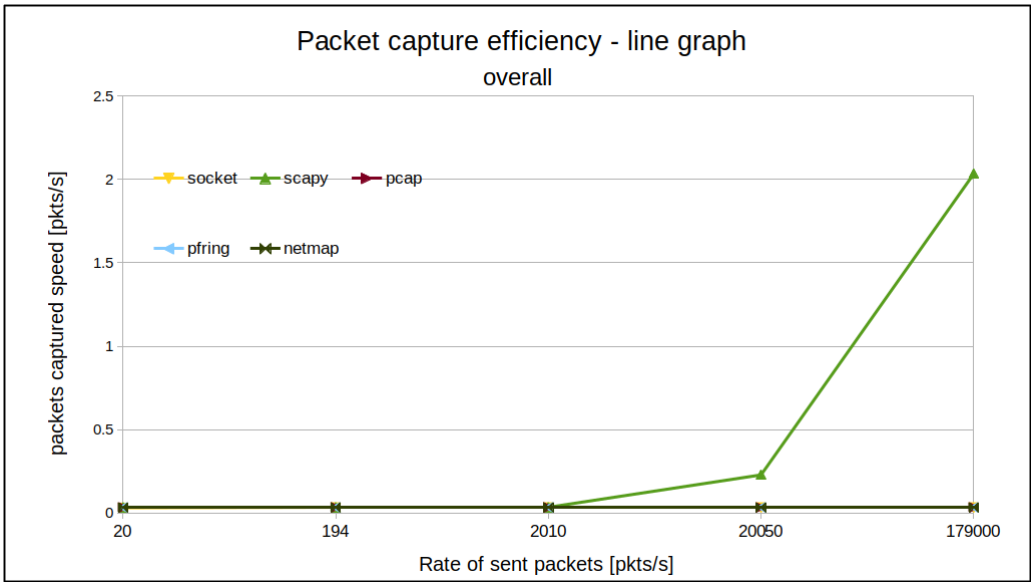


Figure 2.5: Packet capture performance and efficiency.

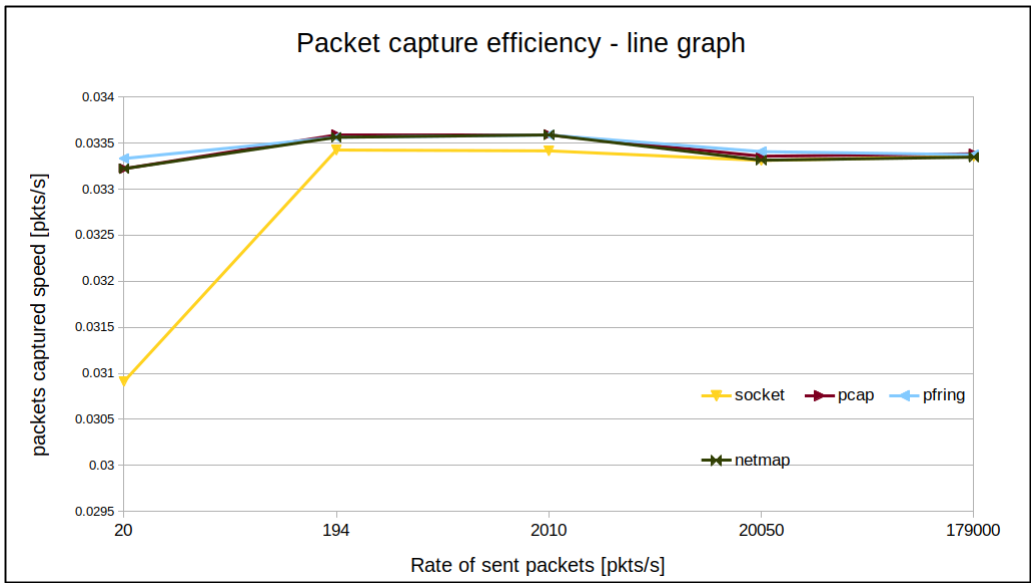


Figure 2.6: Packet capture performance and efficiency. Number of packets captured in a second.

Results have proven that scapy is not effective at higher packet rates (see Figure 2.5), which is due to the Python complexity. It makes the drop rate increase rapidly, until more packets are dropped than captured (see Table 2.2). Therefore, scapy is insufficient for capturing real time network traffic.

Elapsed time test			
Packet rate [pkts/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 2.2: Scapy measured drop rate.

Moreover, other mechanisms were efficient and captured all the traffic (no packets were dropped due to lack of buffer space). Referring to Figure 2.6, 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[14].

[14]Capturing packets in batches means to copy or access multiple packets within one

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.

With respect to this test, netmap is the most suitable choice, because it provides minimum copies of data received, it has no measured drop rates and it is utilized in a modified netmap-based libpcap library, which makes the program compatible with other libpcap versions.

Capture elapsed time

Measuring the elapsed time of set of operations comprises of marking the time before and after the execution of target commands. This would be reliable if the system's time would not change due to Network Time Protocol (NTP) bad update or other unpredictable changes. 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 it is included in *time* library by *clock_gettime()*[15] system call.

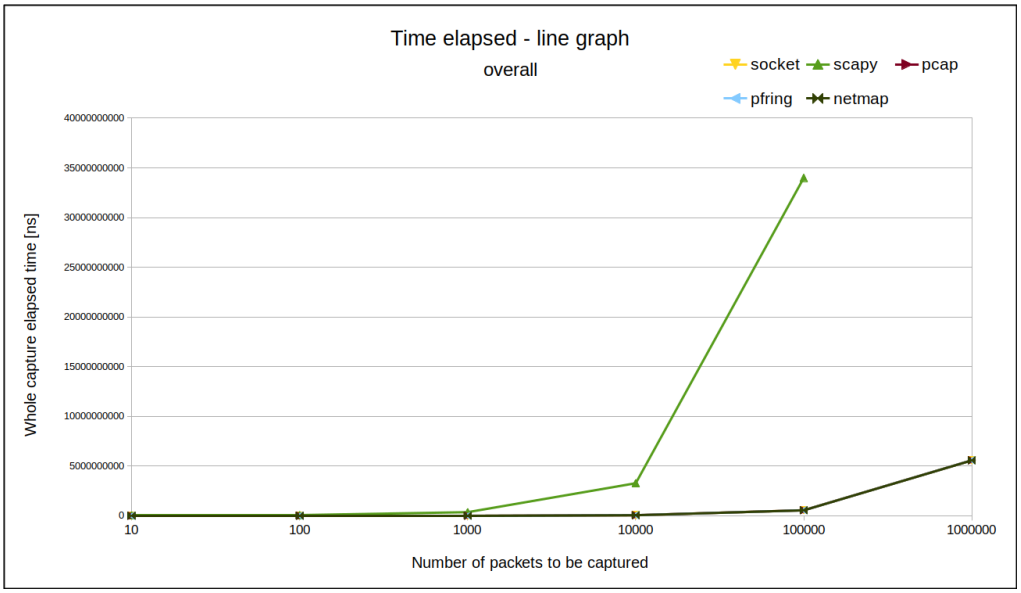


Figure 2.7: Capture performance for testing elapsed time. Each vertical step is 5 seconds.

interrupt or read.

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

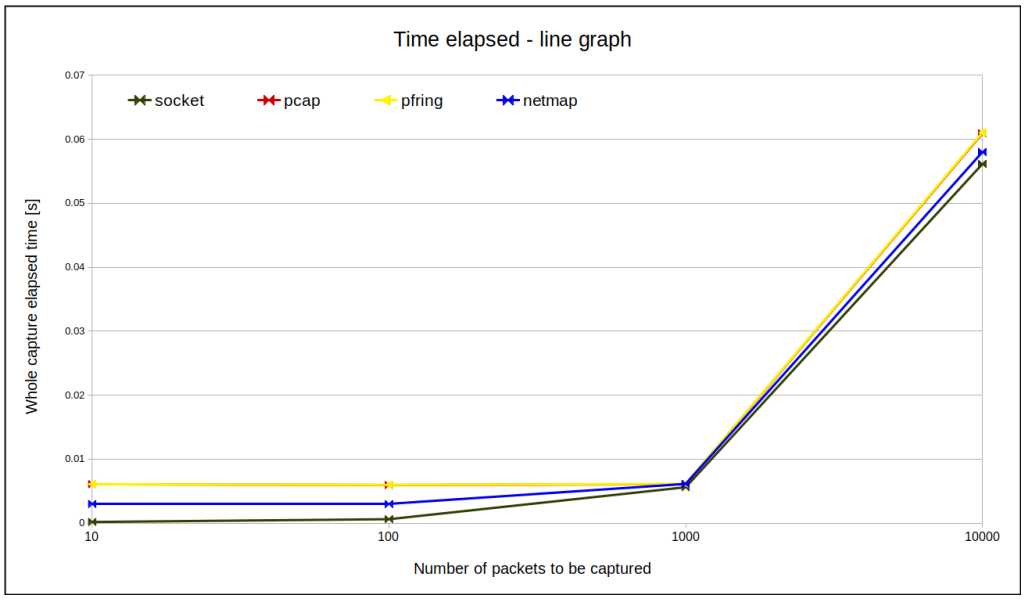


Figure 2.8: Close up graph of capture performance for testing elapsed time and discarding scapy.

As it was for testing packet capture efficiency, elapsed time of packet capture with scapy was radically slow. As a result, scapy was not tested for the maximum of million packets because of the predicted exponential increase (see Figure 2.7). In comparison scapy was processing more than 30 seconds, while other mechanisms processed the same load in 500 ms. All other mechanisms captured packets in wire speed. Figure 2.9 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-based libpcap could be utilized for this thesis for compatibility and performance reasons.

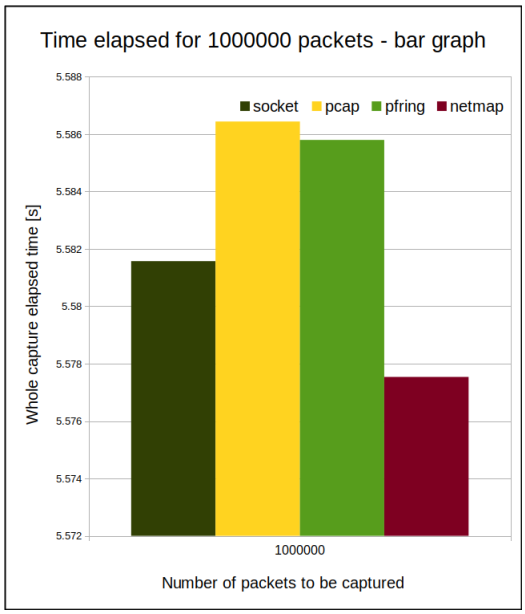


Figure 2.9: Best performance at million sent packets.

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 stores 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 can be 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.

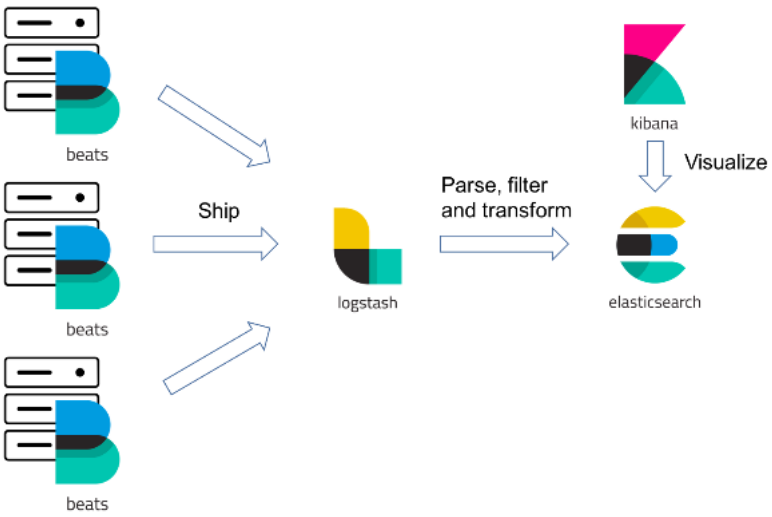


Figure 2.10: ELK stack setup including Filebeat [2].

2.3.2 ELK stack

ELK stack[16] 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 mapping API. Via settings, the default number of shards[17] and the number of replicas[18] [12] 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 2.10). It is useful for data enrichment and modification before it is inserted in bulk[19] requests, rather than single documents. In its configuration are data input

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

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

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

[19] Documents wrapped in batches sent via one HTTP request [13].

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 2.10). 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*[20] could be the solution. Redis is widely used as a caching database [25], 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

[20] `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 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 2.11: OS fingerprinting with known IPv4 fields [29].

Additionally, according to an article on operating system (OS) fingerprinting with packets [29], 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 2.11). 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[21]. The UDP datagram length can be used for UDP datagram

[21] 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. Detecting and monitoring a DNS communication has high security reasons. A possible amplification attack is realized by DNS packets.

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 [32], 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[22] 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[23] 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*[24], 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.

[22] `https://www.wireshark.org/`
[23] `https://molo.ch/`
[24] `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 chapter 4.

3.1 System specification

The final system must include the network sensor implanted in network for capturing packets and the ELK stack, which as the sensor is on a Linux machine. 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. Furthermore, it must notify the user with potential errors or successful tasks (such as initialization of the sensor) in standard log file in designated system directory. The packet capture and further processing must not create a bottleneck and be ideally processing at wire speed.

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

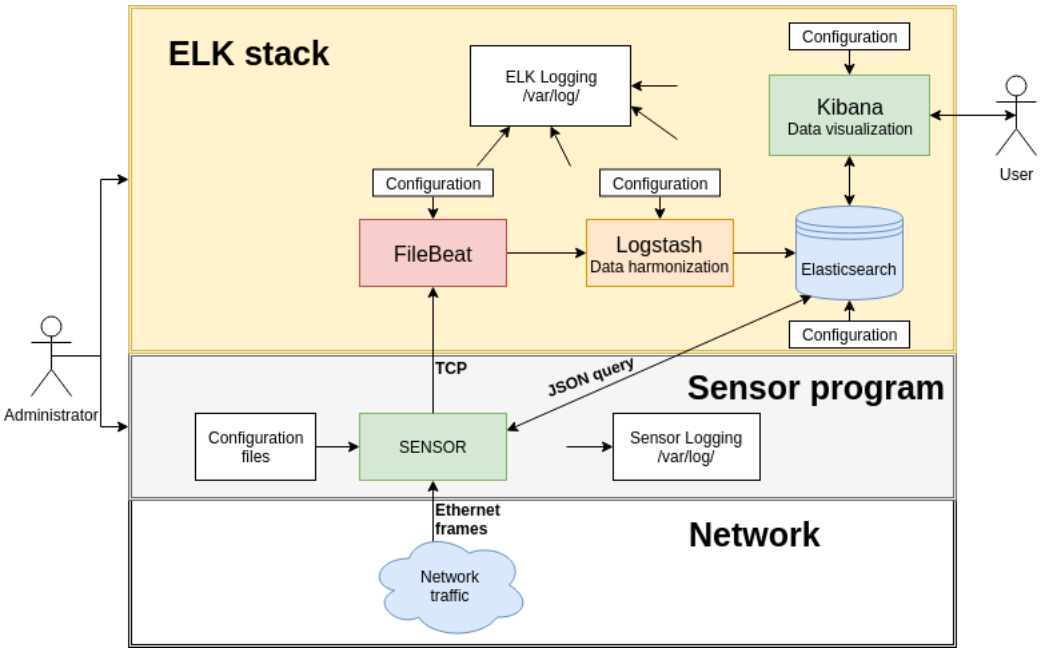


Figure 3.1: System architecture model.

not delay the data stream. The requested visualizations are listed and briefly explained in section 3.2.1 part *Visualizations and database setup*.

3.2 Design

3.2.1 ELK stack initialization

Starting from the top of the system architecture model, the ELK stack setup is a prerequisite for functional sensor sending data to database. The deployment of the ELK stack on a remote or local server is handled by series of *Ansible*[1] roles. It will install each ELK stack component, including setting up required configuration files. Every component configuration is described in the following subsections.

Elasticsearch

Configuring Elasticserach requires setting up a new index for data and filling out the configuration file located in */etc/elasticsearch/elasticsearch.yml*

[1] `https://www.ansible.com/`

(see Listing 3.2). After correctly configuring Elasticsearch, the index template is created via *curl* program or via Kibana developer's tools (see Listing 3.1).

```
PUT _template/new_template
{
  "index_patterns": ["my_index-*"]
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1
  },
  "mappings": {
    "_doc": {
      "properties": {
        "ip": {
          "properties": {
            "source": { "type": "ip" },
            "ttl": { "type": "integer" }
          }
        },
        "host": { "type": "keyword" }
      }
    }}}
```

Listing 3.1: Create index template with index field mapping.

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 3.1). 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[2].

```
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 3.2: Elasticsearch configuration in YAML format

[2] 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 is 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 3.3). In the background, filebeat harmonizes the data by filebeat specific fields. This example configuration sets a functional Filebeat receiving and forwarding JSON objects separated by *new-line*.

```
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 3.3: Filebeat configuration in YAML format.

Logstash

Refering the to the configuration (see Appendix J), 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.

Kibana

Kibana is configured by a configuration file */etc/kibana/kibana.yml*. The default configuration (see Listing 3.4) from the installation is correct, since both Kibana and Elasticsearch will coexist on the same physical machine. The received data can be recognized and visualized by Kibana after setting up the index pattern and timestamp. New index pattern is created under Management -> Kibana -> Index Patterns only after the index occupies at least one document. This limitation can be bypassed by using the Kibana REST API.

```
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 3.4: Kibana sufficient default configuration.

Visualization and database setup

With the index pattern set the requested visualizations are executed by Kibana on indices matching the index pattern. Each visualization is periodically refreshed as new data is stored. Visualizations are grouped in dashboards (see Appendix H), which serves as the main page for further data analysis and monitoring. In addition, Kibana includes *Developer tools* to request data with a JSON formatted query for fast search and the *Discover* page which shows the latest received data. Other sections are not in scope of this thesis.

Following the required visualizations, each is visualized by the most suitable representation:

- *"Services in use"* are two line graphs (source and destination) displaying the count of packets based on the service (port number). Each line representing a service independent of transport protocol. The x-axis is the time at which the packets were captured. By filtering the services this visualization is for monitoring *"specific service(s)"* too.

- *"Geographical location of end nodes"* are two maps (source and destination of traffic) distinguishing multitude by color. The country is resolved from IP address either by the sensor or at the visualization tool.

- *"Packet size ranges"* displays UDP, TCP, ICMP and etc. protocol multitudes as a heat map, pie graph and line graph. Heat map groups by packet size ranges on one axis and protocol on the other. Pie graph depicting percentage ratio. Line graph specifically showing the multitude of protocols at a given time, where each line represents single protocol.

- Time based *"packet size model"* is a bar graph with time axis split by packet size ranges. The ranges specify the small packets, middle sizes and large data packet sizes.

- *"Packet flow"* is a line graph with time axis. Dots represent the total count of packets at particular time and a line representing the moving average of the values. It effectively shows the growth or descent of packet flow.

- *"Visualization filters"* include the protocol filter, packet size range select and differentiating between receiving and transmitted (RX and TX) packets. This is not the filter limitation, but it depicts the most useful ones and other are possible.

- *"Nodes' communications"* is a histogram or tag cloud displaying the most active connections between two hosts (IP addresses).

- *"Statistical model"* of packet sizes is a line graph showing various statistical metrics, such as count, median, minimum, maximum or standard deviation.

To completely automate the initialization process and ready the visualizations all must be functioning when the visualization tool is opened. For Elasticsearch and Kibana it requires loading the index template, create visualizations, index patterns and dashboards. The reason for automation is if the system is duplicated or moved on different machines. It requires to preserve all settings, configurations and visualizations.

Solution is to use Ansible for deploying the ELK stack components on the target machines. These Ansible roles deploy configured ELK stack components with functioning pipeline ready to receive data on Filebeat, which forwards them to Logstash and Elasticsearch. Missing are the visualizations, which need additional deploying mechanism to ready the visualizations and filters in Kibana. Kibana has its own REST API used for importing and exporting of saved objects[3]. Listing 3.5 briefly depicts the necessary steps for initializing the Kibana environment.

[3]Saved objects include all index patterns, visualizations, dashboards and saved searches created in Kibana.

```
# Create template for indices with 'bp-*' pattern.
PUT _template/custom_bp_tmeplate
{
  "index_patterns": ["bp-*"],
  "settings": {"index": "number_of_shards": 2},
  "mappings": {"_doc": {"properties": "FIELD_NAME": {"
     type": "FIELD_TYPE"}}}
}

# Export saved objects from Kibana. Returns JSON.
GET kibana_host:5601/api/saved_objects/_find?type=
   visualization&type=dashboard&type=search&type=index-
   pattern&page=1&per_page=20

# Import saved objects. Oversimplified example.
POST kibana_host:5601/api/saved_objects/_bulk_create
[
  {"type": "visualization", "id": 1, "attributes": {}},
  {"type": "visualization", "id": 2, "attributes": {}},
  {"type": "dashboard", "id": 3, "attributes": {}}
]
```

Listing 3.5: Initialize Elasticsearch and Kibana.

3.2.2 Sensor

The sensor is a Linux program utilizing netmap-based libpcap (supported by test results in section 2.2) capturing network traffic, storing parsed packets to database and logging progress to local or remote rsyslog server.

Utilizing libpcap, the sensor captures traffic in simple receive, transmit or promiscuous mode and distinguish each packet by this indication. Each received packet is be parsed in scope of required protocols up to the Transport layer excluding the data payload (headers only). IP addresses are resolved in geological location database for country distinguishing. Parsed and harmonized packet data are converted to JSON objects. A suitable JSON library is used for parsing the packets to JSON structures, which are sent out as raw TCP data to Filebeat. Whole process is simplified in Figure 3.2.

Main process

The class model briefly depicts the most crucial parts of the sensor class design (see Figure 3.3). The sensor is configured by a configuration file, specified by the user. Both interface specification and database connectors

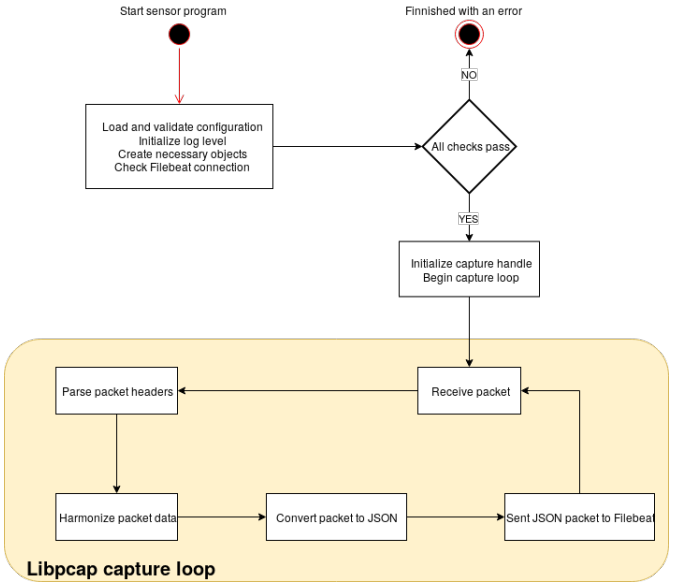


Figure 3.2: Sensor packet parsing and processing

will be included in this configuration file. Configuration file is validated before any initialization is performed. More precisely the configuration file specifies:

- target interface and the capture direction
- target Filebeat host and port

The format is in YAML for clear interpretation, which requires a YAML parsing library reading the preferences to program structures. Configuration file is read on start of the program and any further modification do not affect the running process.

Setting up a libpcap capture handle comprises of the following preferences and is followed by opening the handle for active capture processing each packet in a callback function.

- resolving the target interface to existing interface and retrieving MAC and IP address
- attempt to set snapshot length, immediate mode and specific buffer size
- set capture direction

In the callback function the packets are broken down to protocols using a packet parsing library. Each packet is harmonized and converted to JSON string. A suitable library for producing JSON regarding performance [33],

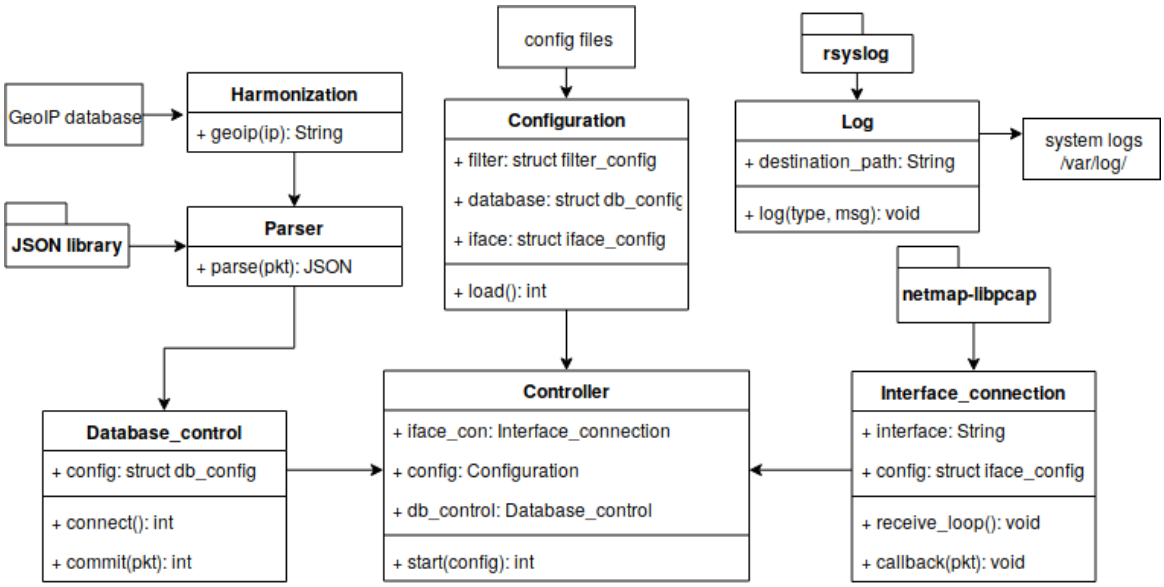


Figure 3.3: Sensor class model.

such as parsing time, memory in use and time it takes to stringify is *RapidJSON*[4]. After packet harmonization and parsing the JSON strings are sent through TCP socket to Filebeat.

Configuration file

Configuration file must be located in the designated directory for program configurations - */project_root/resources/traffcol.yml*, where *traffcol* is an abbreviation for traffic collector. It is the main configuration file loaded by the program.

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. The Filebeat connection specification must be a resolvable host and a port number.

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 I.1 in the Appendices section.

[4] `http://rapidjson.org/index.html`

Initialization

For sensor initialization, it requires a configuration file with all mandatory preferences. Of course the prerequisite is the ELK stack initialized and ready with visualizations waiting for new data. Optionally a running *rsyslog*[5] server or syslog accepting local logs. Setting up (r)syslog server can be simple as configuring a destination file and accepting logs from local source if the its on the same machine.

[5]Rsyslog stands for rocket-fast syslog and it is the decedent of syslog.

Chapter 4

Implementation

According to all specifications and speed requirements, the sensor is programmed in C++, well compatible with netmap-based libpcap. The sensor consists of a specific interface as source of network capture, packet parsing, logging and database connection. Referring to Figure 3.1 the implemented component will be the sensor with logging to files. The ELK stack components are configured by respective configuration file. Kibana and Elasticsearch is initialized by a script via specific APIs.

4.1 ELK stack deployment and setup

Both deployment and setup is designed as series of Ansible scripts and a single bash script. All ELK stack components are deployed on Debian-based Linux machines (e.g. Ubuntu Server).

4.1.1 Deployment

Ansible a remote command execution automation tool consisting of various modules (e.g. shell, package, copy, etc.). It utilizes SSH connection for remote deployment, which is the only prerequisite of target machine. A series of Ansible roles[1], for each ELK stack component, are executed to fully configure and enable ELK stack. Each component is a Java application, so a requirement is to install a specific version of Java. An exemplary role for deploying Elasticsearch is listed in Listing 4.1. All roles are documented in the repository.

```
1 --- # Elasticsearch tasks
2
```

[1] https://github.com/tomas321/ansible-roles/tree/master/common/elk

```
3  - name: install java 8 jre
4    # apt module to install "openjdk-8-jre-headless"
5
6  - name: ensure elasticsearch is installed
7    # apt module to install "elasticseach"
8
9  - name: load configuration
10   # template module to copy file
11
12 - name: remove memory limitations
13   # template module to copy file
14
15 - name: ensure /etc/systemd/system/elasticsearch.service
       .d exists
16   # file module to ensure existance of a path
17
18 - name: override elasticsearch systemd memlimit
19   # template module to copy file
20
21 - name: restart and enable elasticsearch
22   # systemd module to enable and restart elasticsearch
```

Listing 4.1: Elasticsearch Ansible role overview.

4.1.2 Setup

ELK stack components are configured as shown on subsection 3.2.1. All configurations are derivatives from default configurations, because no extended plugin features are required. For monitoring the data in Kibana an index template with field mapping is created in Elasticsearch and all visualizations in section 3.2.1 were configured directly in Kibana. Just as described in section 3.2.1, the Kibana objects are retrieved and ready for importing to an existing instance of Kibana via a script (see Listing 4.2).

```
1  #!/usr/bin/env bash
2
3  host="$1"
4  function readfile() {
5      local data=""
6      while read -r line; do
7          data="$data$line"
8      done < "$1"
9      echo "${data}"
```

```
10 }
11
12 curl -X PUT "$host:9200/_template/my_template" \
13      -H "Content-Type: application/json" \
14      -d "$(readfile /path/to/tempalte)"
15 curl -X POST "$host:5601/api/saved_objects/_bulk_create?
      overwrite=true" \
16      -H "kbn-xsrf: true" \
17      -H "Content-Type: application/json" \
18      -d "$(readfile ${objects_path})"
19 curl -X POST "$host:5601/api/kibana/settings/
      visualization:regionmap:showWarnings" \
20      -H "kbn-xsrf: true" \
21      -H "Content-Type: application/json" \
22      -d '{"value": false}'
```

Listing 4.2: Initialize Elasticsearch and Kibana custom prerequisites.

Initialization script puts the index template to Elasticsearch and imports the visualizations via the Kibana REST API. Kibana region map visualizations is not fully equipped to mark all known country codes[2], therefore the script issues to disable a possible warning about not being able to mark non-country codes.

The *Nodes' communications* histogram visualization required extra script (see Listing 4.1.2) to create in the Kibana environment. This scripted field is part of the index template inserted by the kibina initialization script.

```
1 if (doc['ethernet.ethertype'].value.compareTo('ipv4') ==
     0) {
2     if (doc['ip.source.keyword'].value.compareTo(doc['ip
         .destination.keyword'].value) > 0) {
3         return doc['ip.source'].value + "<->" + doc['ip.
             destination'].value;
4     }
5     return doc['ip.destination'].value + "<->" + doc['ip
         .source'].value;
6 } else if (doc['ethernet.ethertype'].value.compareTo("
    ipv6") == 0) {
7     if (doc['ipv6.source.keyword'].value.compareTo(doc['
         ipv6.destination.keyword'].value) > 0) {
8         return doc['ipv6.source'].value + "<->" + doc['
             ipv6.destination'].value;
9     }
```

[2]Regarding country codes include also continent or other non-country codes [3].

```
10      return doc['ipv6.destination'].value + "<->" + doc['
            ipv6.source'].value;
11  }
```

It creates a new field representing a connection between two nodes independent of the packet direction (source to destination and vice versa) by comparing the IP address string representations to maintain the order.

4.2 Sensor

Sensor is implemented as a single thread program, where each component will be viewed as an object (see Figure 3.3). The Configuration class loads a YAML configuration file from the *"project_root/resources/"* directory to specific C++ structures - *db_config, iface_config* and validates (see subsection 4.2.1) the file's format. Configuration file is parsed utilizing *yaml-cpp*[3] library. 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 preferences. Any errors are logged and the main process is terminated.

4.2.1 Configuration file validation

A dynamic validation algorithm for YAML configuration file requires another YAML file specifying the configuration file structure and available fields. The algorithm crawls through the specification file and accordingly validates the fields in the actual configuration file. The only thing it checks is whether mandatory fields are not left out and if it is structured the way it should be. The algorithm is simplified in Listing 4.3.

```
1  void validate(specs_config, key) {
2      foreach (iterator : specs_config) {
3          string current_key;
4          if (key.empty()) current_key = iterator.first.
               scalar();
5          else current_key = "{key}.{iterator.first.scalar
               ()}";
6          if (iterator.second.IsMap()) {
7              # validate if value is a map
8              validate(iterator.second, current_key);
9          } else if (iterator.second.IsScalar()) {
```

[3] https://github.com/jbeder/yaml-cpp

```
10              # checks if the key is mandatory or optional
11              validate_key(current_key, iterator.second.as
                    <bool>());
12          }
13      }
14  }
```

Listing 4.3: Algorithm for configuration file validation utilizing the YAML parsing library.

Calling *validate_key* (see line 11 in Listing 4.3) takes the current key of the specification file and a boolean value specifying if it's mandatory. Failed test throws an exception and the program initialization stops.

4.2.2 Packet capturing

Capture handle in libpcap is well configurable, though it is system dependent and not all preferences are available. All function calls are listed in Listing 4.4 and worth mentioning are performance oriented settings according to Libpcap manual page [9]. Snapshot length of 300, minimizes the CPU time and evades the addition packet reassemble and it should be enough to bound the maximum size of packet headers even in case of tunneling such as *IP in IPv6*[4] and vice versa. Arriving packets are processed as soon as they arrive with setting the immediate mode. To minimize the packet loss, the buffer size is set to a maximum of 2 GB. The capture direction is set using the libpcap filters by filtering by source and destination MAC address (e.g. *ether dst 10:e3:aa:00:00:12*).

```
1  pcap_t *handle;
2  pcap_findalldevs((pcap_if_t **) alldevs, errbuf); #
       followed by retrieving the MAC and IP address
3  pcap_set_snaplen(handle, 300);
4  pcap_set_immediate_mode(handle, 1);
5  pcap_set_buffer_size(handle, INT_MAX); # 2GB
6  pcap_set_tstamp_type(handle, PCAP_TSTAMP_ADAPTER);
7  pcap_set_promisc(handle, 1); # in case of promiscuous
       mode
8  pcap_activate(handle);
9
10 # filter traffic if direction is one of ['in', 'out']
```

[4]Size of such packet header might be 24B Ethernet + 40B IPv6 + optional IPv6 extension headers multiple of 8B (e.g. 40B) + tunneled IPv4 of maximum 60B + maximum 60B TCP = 224 [16].

```
11 pcap_compile(handle, (bpf_program *) prg, filter_str, 0,
       (bpf_u_int32) mask);
12 pcap_setfilter(handle, (bpf_program *) prg);
13
14 pcap_loop(handle, 0, (pcap_handler) fnc, (u_char *) user
      );
```

Listing 4.4: Packet capture handle setup API calls.

Correct settings are followed by opening a capture session and looping for infinite number of packets. Each packet is handled in a callback function passing the packet to the parsing process.

4.2.3 Packet parsing

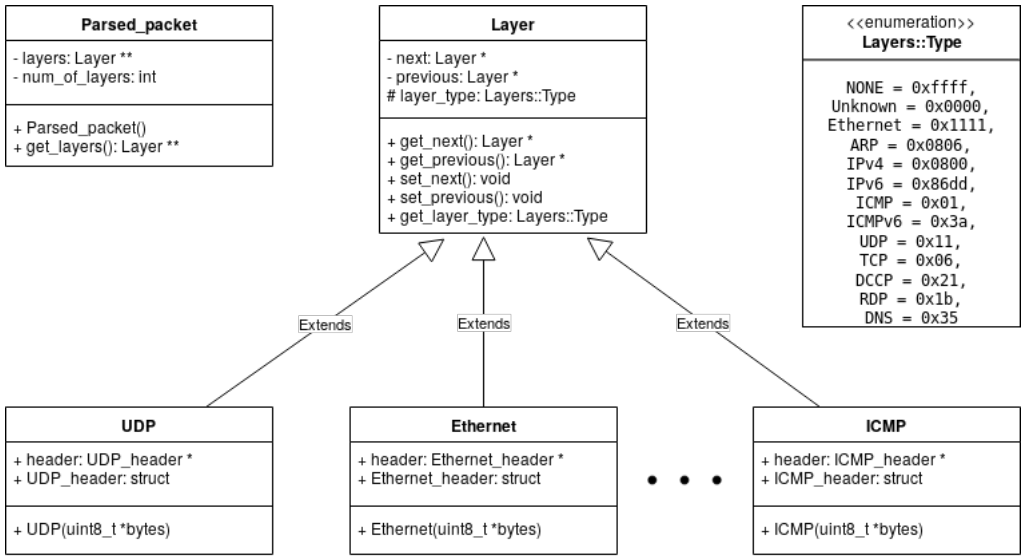


Figure 4.1: Packet parsing library class model.

Firstly, each packet is parsed based on this class model (see Figure 4.1) by simple algorithm (see Listing 4.5). Parsing packet to *Parsed_packet* object goes through supported layers one by one until acceptable last protocol header is reached (e.g. TCP header). Using such custom parsing library is minimalistic, because it does no extra object creation and is independent of third party libraries. This parsing process is the backbone of understanding raw packet bytes. Called function *parse_header* parses a given layer and increases the offset from the first byte of raw packet dynamically (IHL field in IPv4) or statically.

```
1  int offset = 0; // next packet header offset from 0th
       byte.
2  int done = 0;
3
4  num_of_layers = 0;
5  Layers::Type layer_type = Layers::Ethernet; // initial
       layer
6  layers = (Layer **) malloc(sizeof(Layer *));
7
8  do {
9      layers = (Layer **) realloc(layers, sizeof(Layer *)
           * (num_of_layers + 1));
10     layers[num_of_layers] = (Layer *) malloc(sizeof(
           Layer));
11
12     done = parse_header(&layers[num_of_layers], bytes, &
           layer_type, &offset);
13
14     if (Layers::layer_string(layer_type) == Layers::
           layer_string(Layers::Unknown)) {
15         done = 1;
16     }
17
18     if (num_of_layers == 0) {
19         layers[num_of_layers]->set_previous(nullptr);
20     } else {
21         layers[num_of_layers]->set_previous(layers[
               num_of_layers - 1]);
22         layers[num_of_layers - 1]->set_next(layers[
               num_of_layers]);
23     }
24     layers[num_of_layers]->set_next(nullptr);
25     ++num_of_layers;
26 } while (! done);
```

Listing 4.5: Raw packet parsing process.

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 4.6). Keys specified in the output JSON correspond to the Elasticsearch index mapping. Parsing is expected to execute without errors, since the input is a Parsed_packet object structure and the output is a JSON string. The sensor has a higher layer API for constructing JSON to support multiple value types. JSON parsing solution

sampled in Listing 4.6 encapsulates, even the minimum, complexity in a small Json class. It's based on utilizing the C++ function templating as seen in lines 15 through 20.

```
1  /* Now an abstraction hidden in Json class
2  StringBuffer = json_pkt;
3  Writer<rapidjson.StringBuffer> writer(json_pkt); */
4
5  Json json; // includes above abstraction and opening a
      JSON object
6  json.start_object("ipv4");
7  json.add<string>("source", "string_value");
8  json.add<uint16_t>("length", ip.total_length);
9  json->end_object();
10 // ...
11
12 json_string = json.stringify() + "\n"; // ends the
      master object and uses the RapidJSON library for
      conversion
13
14 /* templated Json.add function
15 template<typename T>
16 int add(string key, T value) {
17     if (Json::add_specific(key, value)) return 0;
18     else return 1;
19 }
20 */
```

Listing 4.6: Packet parsing to JSON format.

In this section the packet data is harmonized by geological location from static *GeoIP* database file utilizing a library[5] for querying such files with source or destination IP addresses. In addition, when knowing the IP addresses the data is compared to the capture interface address and store the direction of the packet (either RX or TX).

4.2.4 Committing packets to ELK stack

Committing to database is realized by a TCP socket. Prerequisites are connection to Filebeat and setting a keep-alive session. The destination host is specified in the configuration file in a host name format or an IP address. In case of host name, the IP address is resolved by *gethostbyname* system call. A successful address resolution the socket attempts a connection to

[5] `https://github.com/maxmind/geoip-api-c`

a specific port. This is the initialization process and on success the newly created object holds the active socket file descriptor. Sending the JSON string requires new-line terminated data (line 12 in Listing 4.6).

4.2.5 Logging

Logging is realized by local *rsyslog* server configured either manually or utilizing Ansible[6]. Rsyslog and generally syslog server distinguishes sources by facilities [8] and sensor logs to facility LOG_LOCAL1 (see Listing 4.7).

```
1 Logging::Logging(int level) {
2     openlog("traffic_collector", (LOG_CONS | LOG_NDELAY
          | LOG_PID), LOG_LOCAL1);
3     setlogmask(LOG_UPTO(level));
4 }
5
6 void Logging::log(int level, const char *msg) {
7     syslog(level, "%s", msg);
8 }
```

Listing 4.7: Opening syslog and API function for logging.

LOG_CONS - Logs syslog failures to system console.
LOG_NDELAY - No delay opening the syslog connection.
LOG_PID - Log process ID (PID) with each message.

Rsyslog is configured in a configuration file (*/etc/rsyslog.d/custom.conf*) with specifying the facility, log levels logged, log message format and a destination file (see Listing 4.8).

```
1 $template message_template,"%timegenerated% [%
     syslogseverity-text%] [%PROCID%] %msg%\n"
2 $template log_filename_fmt,"/var/log/traffcol/traffcol.
     log"
3
4 local1.* log_filename_fmt;message_template
```

Listing 4.8: Rsyslog configuration file.

To resolve ever-increasing log file size, rsyslog provides log rotation (see Listing 4.8). Logs are rotates only if not empty, each log is rotated 4 times and the rotation is executed when the file size exceeds 10k bytes. After rotation the new log file is created and the old is compressed with no delay (otherwise it compresses the file after next rotation). To reset the file descriptor of the log file for rsyslog the systemd service is restarted right after the rotation.

[6] https://github.com/tomas321/ansible-roles/tree/master/common/rsyslog

```
1  /var/log/traffcol/traffcol.log {
2      notifempty
3      rotate 10
4      size 10k
5      create
6      compress
7      nodelaycompress
8      postrotate
9          /usr/sbin/service rsyslog restart > /dev/null
               2>/dev/null || true
10     endscript
11 }
```

Listing 4.9: Rsyslog log rotation configuration.

Chapter 5

System tests

Subject of testing were mostly all features and functionalities the program has.Expected behavior is to be able to configure capture direction, start packet capture and log all progress to a local rsyslog server. Log files should rotate as specified in the log rotation configuration file. Finally, the ELK stack should be configured for storing captured packets, including visualizations in Kibana with minimal hands on configuring.

5.1 Whole system test run

Testing the system as a whole consisted of starting the packet capture, and monitoring for real time statistics and analysis in Kibana.

The sensor was capturing packets and successfully sent the data in correct format and structure to Elasticsearch. The Logstash component has correctly parsed the JSON string to JSON and sent the data to Elasticsearch index with the expected suffix (current date). The Elasticsearch index template correctly templated the data types of the destination index. Finally, the data was visualized in the dashboard (see Figure 5.1) as more data was arriving (real time analysis). From top to bottom and left to right the dashboard includes filters, overall count of captured packets, trend of the packet flow, time characteristics of protocols encapsulated by IP layer followed by its percentage representation. Next are the region maps displaying the source and destination of traffic[1]. Below is the distribution of packets to range of sizes represented as a heat map with IP protocols and time distributed. Next pair is the packet count with source and destination service (in port numbers). On the very bottom are the top 10 node communications arranged

[1]Since the capture was set to promiscuous mode the maps are similar.

in a way to not take in account which is source and which destination aside the statistical model of the packet lengths.

5.2 Deployment

On a newly installed Linux system (Ubuntu server 18.04) Ansible deployed the ELK stack preconfigured components, rsyslog server with log rotation and sensor dependencies excluding netmap and netmap-based libpcap library. After that, Filebeat was ready for accepting data and sensor was ready for manual compilation with netmap download and installation.

5.3 Logging

Log file was successfully created in the specified right when the sensor started. After the log file was big enough the log rotation system cron job would rotate the log file. To issue the log rotation the *logrotate* either forces or attempts the log rotation. As a result the log file has rotated, created a new file, compressed the old log file and restarted the rsyslog service.

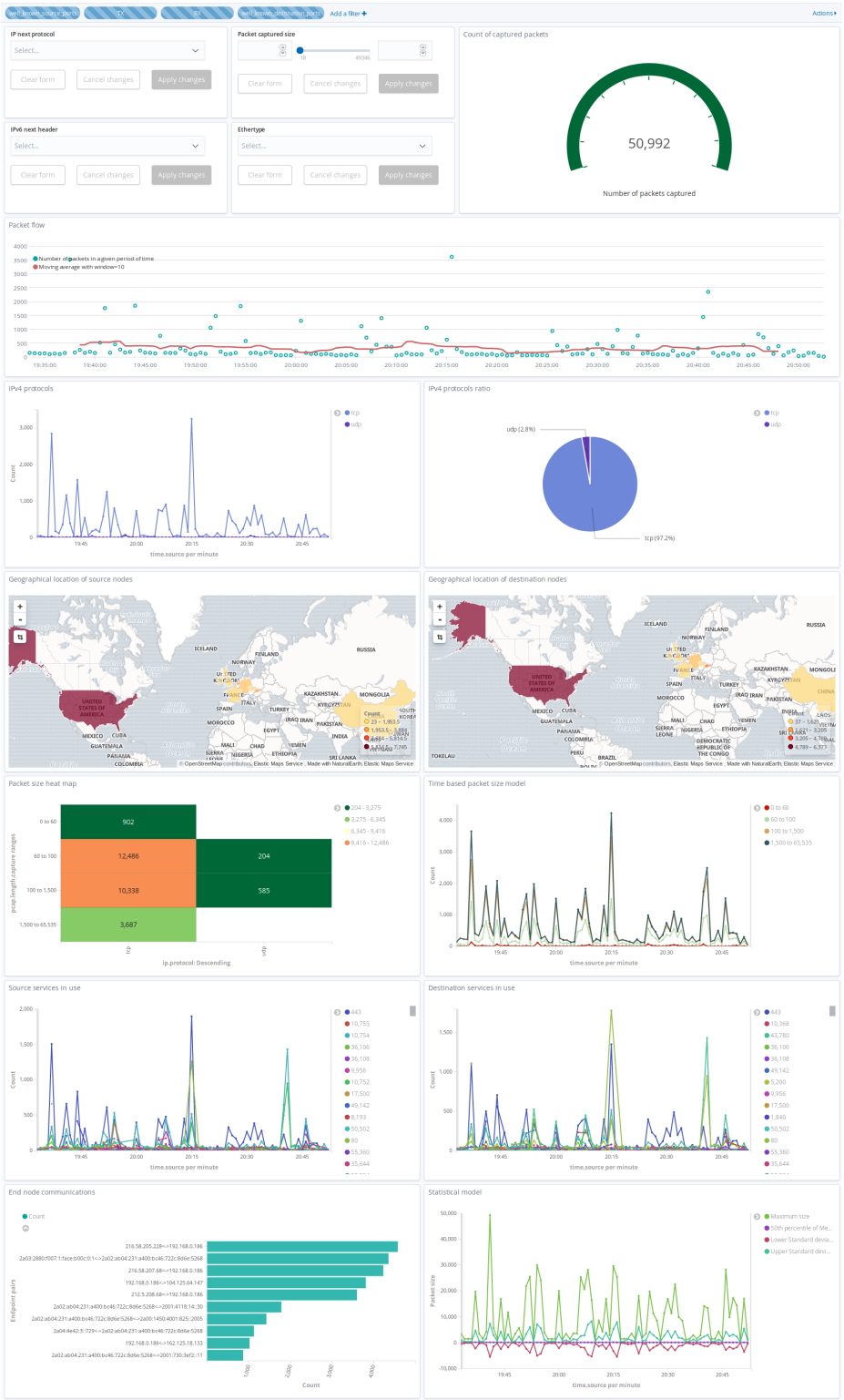


Figure 5.1: Screenshot of the final dashboard.

Chapter 6

Conclusion

Multiple factors are to be taken to account, since the system is a functional capture mechanism with a connected existing monitoring tool. Even though the packet capturing is well analyzed and multiple mechanisms are compared, the solution is implemented with libpcap library. The reason was to make the system more compatible across platforms. Sensor program may be linked to various libpcap versions, such as netmap or PF_RIING based libraries. Down side is the performance compared to the native API of both of these frameworks. Nevertheless, utilizing the netmap-based libpcap is fare compromise to compatibility and performance.

As for the other part of the system - database and visualization tool alone brings fast search from the database and vast visualization features. All captured network traffic is visualized in real time. However, insertion to Elasticsearch is not as fast as the capture mechanism can be, which raises the issue of bottlenecks and losing data. This is most certainly where the system can be improved by either enlarging the Elasticsearch cluster to more nodes and splitting the load. Other solutions may be to implement buffered monitoring tool with additional external storage system. This way every new packet would simple update each visualization and be stored to the database. Nevertheless, Kibana is the right tool for visualizing data in Elasticsearch even in real time, to detect possible network abnormalities or attack attempts.

Overall, the system is well organized and documented in source code as well in installation and user guides. In addition most of the system is deployable by Ansible to simplified the installation on target systems. As is it for most programs, sensor logs its progress to local syslog server. For future work the sensor should be fully configurable, meaning more capture options specified by the user, such as filter packet capture, logging to remote syslog server or adding more IP address harmonization data (ASN, city or

geological coordinates). In addition, sensor has no install target configured in its make file, so for future work the could be fully installed on system as a system wide executable or even as a systemd service.

In comparison to mentioned existing solutions (see section 2.5) this solution has more of the visualization part, which is the most important part in security even with acceptable delay. A proper visualization targeting the right data can detect most of network traffic spikes. Most of existing solutions are not dependent on external database system, but work as a single tool for monitoring and searching network packets. This sensor may be theoretically deployed multiple times on different machines and with minor modifications of the Filebeat configuration scaling up the capturing mechanism to distributed packet capture system with centralized database and monitoring tool.

Chapter 7

Resumé

Zber a analýza o sieťovej prevádzke má dávať odpoveď na otázky štýle, kedy, kto, koľko, kde a od kiaľ komunikoval v cieľovej sieti. Cieľom je implementovať systém zberu sieťovej prevádzky v systému typu UNIX/Linux s úložiskom dát a následnou vizualizáciou a analýzou dát. Zber dát spočíva v čitaní sieťových rámcov zo sieťového adaptéra (NIC) a v zmysle zadanie je potrebné zanalyzovať a porovnať rôzne mechanizmy. S cieľom jednoduchosti a praktického využitia bola zvolená pytónová knižnica *scapy*, kompatibilita a štandard reprezentovala knižnica *libpcap*, rýchle a efektívne riešenia prestavovali rámce *PF_RING* a *netmap* a nízkoúrovňový soket *PF_PACKET*.

Každý z týchto mechanizmov poskytoje iné výhode ale aj nevýhode. Scapy je vhodný pre komplexné skladanie paketov, ale chýba mu rýchlosť a jednoduchosť. Libpcap je kompatibilný a často nainštalovaný naprieč väčšiny platform, ale pre vyššie toky paketov zahadzuje pakety. PF_RING je rámec pre rýchly zber sieťovej prevádzky závislý od Intelových sieťových adaptérov a vlastných jadrových modulov. Netmap je rámec presadzujúci rýchlosť nizkoúrovňových fundamentálnych zmien v štruktúre zásobníkov a je závislí od jadrového modulu. PF_RING aj netmap obsahujú vlastnú vylepšenú knižnicu libpcap mapujúcu natívne API volania na štandardné libpcap volania. Na záver PF_PACKET je protokolová rodina využívaná libpcapom zabezpečujúca priame posielanie či príjmanie čistých Eternetových rámcov.

V jednoduchej sieti s dvoma priamo prepojenými počítačmi, pričom jeden bol generátor paketov v rôznych frekvenciách a druhý bol zberač obmieňajúci spomínané mechanizmy. Prvá sada testov bola zameraná na efiktívnosť zberu paketov, pričom mal každý mechanizmus zozbierať čo najviac paketov za daný čas s meniacou sa frekvenciou generovania. Druhá sada bola zameraná na rýchlosť zozbierania variabilného počtu paketov s konštatnou frekvenciou generovania pre každú iteráciu. V prípade PF_RING a netmapu boli testované ich špecifické libpcapové verzie. Výsledky ukázali, že netmap boli

najefektívnejším mechanizmom, aj keď ostatné, okrem scapy, boli skoro rovnako efektívne. Výhodou netmapu bolo množstvo zahodených paketov, čo bolo výrazne malé hlavne s porovnaním štandarného libpcapu.

Analýza zozbieraných dát spočíva vo výbere vhodného nástroja pre vizualizáciu dát, úložisko a analýzu. Do úvahy boli brané databázové systému type *SQL* a *NoSQL*. SQL databáza paketov by mala buď veľa vzťahov medzi tabuľkami pre každý protokol alebo jednu velkú tabuľku všetkých protokolov a príslušných polí. NoSQL poskytuje vhodnejšiu formu dát, a to bez predvolenej štruktúry. Vhodná databáza by poskytovala aj možnosť vizualizovať dáta vo vhodnom nástroji. Taký nástroj, respektíve databázový systém je zoskupenie *Elasticseachu*, *Logstash* a*Kibana* (ELK koponenty) spolu s *Filebeat* komponentom. ELK pokytuje rýchlo prehliadateľné úložisko - Elasticsearch a vizualizačný nástroj - Kibana.

Návrh riešenia popisuje spôsob realizácie senzora spolu s úložiskom dát a ich následnej vizualizácie. Senzor je C++ program zachytávací sieťovú prevádzku pomocou libpcap knižnice vylepšenej netmapom, rozkladajúci pakety na rozpoznateľné protokoly a ich obohacovanie. Obohatené pakety sú zaielané v JSON formáte do Filebeat komponentu ELK. Prijaté dáta sú preposlaná do Logstashu, kde sú rozložené z reťazcovej formy JSONu na JSON objekt a následné vložené do Elasticsearch databázy. Predpripravené vizualizácie v Kibane slúžia na špecifické zobrazenie dát a odpoveď na hlavné otázky zadania práce. Dodatočne výsledný systém z veľkej väčšiny automatizovateľný prostredníctvom nástroja *Ansible* a sledovanie funkčného správania cez logové súbory cez *syslog* server.

Výsledný systém je funkčný zberač paketov s externe pripojeným databázovým systémom ELK. Aj analýza opisuje aj potenciálne efektívnejšie mechanizmu, bola zvolená knižnica libpcap namapovaná na natívne netmap API volania z dôvodu širokej kompatibility. Kompilácia senzora je možná s lubovoľnou verziou libpcap knižnice, ako aj v prípade nefunkčnosti netmapu na cieľovom zariadení štandarný libpcap. ELK komponenty pokytujú úložisko a zobrazovanie dát v reálnom čase, avšak vklad do databzy nie je rovnako rýchly ako potencionálna rýchlosť zberu sieťovej prevádzky. Ako dôsledok môže dochádzť k strate dát alebo oneskorenému zobrazeniu v Kibane. Vylepšenie práce je možné práve v zrýchlení ukladaní paketov do databázy.

Systém je celkovo správne zorganizovaný a zdokumentovaný spolu s manuálnou inštalačnou príručnou s Ansible nasadením a príručkou pre používateľa. V zmysle budúcej práce by mal byť senzor plne konfigurovateľný, a to vytiahnuť programom predefinované preferencie libpcap, širšie obohacovanie IP adries (ASN, mestá alebo konkrétne koordináty lokality), logovanie do vzdialeného syslog servera. V zmysle plného nasadenia senzor na zariadenie, chýba inštalácia spotuteľných súborov a senzora ako celok do

operačného systému. Prípade rozširiť spúšťanie na *systemd* službu, ktorá číta konfiguračný súbor z *"/etc"* adresára.

Na porovnanie s existujúcimi riešeniami, toto riešenie poskytuje vylepšenie v zobrazovaní dát, čo je jedna z najdôležitejších súčastí systému zabezpečúci monitorovanie siete z bezpečnostného hladiska. Aj v prípade menšieho zdržania, zobrazenia sú cielené na správne časti zachytených paketov, ktorú môžu indikovať potenciálne hrozby alebo nezrovnalosti. Väčšina existujúcich riešení funguje ako jednotný systém s databázovým úložiskom, narozdiel od tohto riešenia s externým úložiskom. V prípade menších úprav vo Filebeat konfigurácii je možné nasadiť systém na viaceré zariadenie s distribuovanými zberačmi a centralizovanou databázou s zobrazovacím nástrojom.

Bibliography

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

[2] Elk stack flow and architecture diagram. https://arender.io/doc/current/_images/ELK_architecture.png.

[3] Iso 3166 country codes. https://dev.maxmind.com/geoip/legacy/codes/iso3166/.

[4] Pf ring modules image. https://www.ntop.org/products/packet-capture/pf_ring/.

[5] Shared memore view of netmap. https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf.

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

[7] *SOCKET(2) Linux Programmer's Manual*, September 2017. http://man7.org/linux/man-pages/man2/socket.2.html [6.11.2018].

[8] *syslog(3) - Linux man page*, September 2017. https://linux.die.net/man/3/syslog [28.03.2019].

[9] *Manpage of PCAP*, July 2018. https://www.tcpdump.org/manpages/pcap.3pcap.html [18.03.2019].

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

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

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

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

[14] 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].

[15] 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].

[16] *Generic Packet Tunneling in IPv6*, December 1998. https://tools.ietf.org/html/rfc2473 [25.04.2019].

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

[18] 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].

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

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

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

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

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

[24] 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].

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

[26] 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].

[27] 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].

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

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

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

[31] 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].

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

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

[34] "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].

[35] 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

Installation guide

```
A full installation guide for dependencies and the traffic_collector program.

############################
## INSTALL dependencies ##
############################

Either use Ansible or manually install the dependencies. Not all libraries are
installed by Ansible, so the rest requires manual installation.

### Ansible deployment ###
Before running Ansible playbooks set the target hosts in the 'hosts' file to
something like this:
    elk.bp.local ansible_connection=ssh ansible_user=tomas ansible_python_interpreter="/usr/bin/
         python3"
    sensor.bp.local ansible_connection=ssh ansible_user=tomas ansible_python_interpreter="/usr/bin/
         python3"

    [elk]
    elk.bp.local

    [sensor]
    sensor.bp.local

Note: For each Ansible playbook change the hostname in the playbook according
the host group as from the above example (name in square brackets).

Ansible playbook 'traffcol/traffcol.yml' in the ansible directory provides the
deployment these libraries:
  - cmake
  - make
  - gcc
  - g++
  - yaml-cpp
  - geoip-api-c
  - libnet1-dev
Deploy it by running:
  'ansible-playbook -K playbooks/traffcol/traffcol.yml'
from the root Ansible directory.

To deploy the ELK stack use the 'traffcol/elk.yml' playbook to install and
configure:
  - Filebeat
  - Elasticsearch
  - Logstash
  - Kibana
Deploy it by running:
  'ansible-playbook -K playbooks/traffcol/elk.yml'
from the root Ansible directory.

## yaml-cpp installation ##
Installation guide is in detail on 'https:Clean//Cleangithub.CleancomClean/CleanjbederClean/CleanyamlClean-
     CleancppClean'.
Simple install on Unix/Linux like systems:
  1. Direct download or clone the repository.
  2. Locate to the repository root directory.
  3.
     mkdir build && cd build
```

```
     cmake ..
     make
     make install


## rapidjson ##
Rapidjson is a header only library and it should be included in the lib/
directory. In scope of this program, there is no need for full installation.


## geoip-api-c ##
Intallation guide is in detail on 'https:Clean//Cleangithub.CleancomClean/CleanmaxmindClean/CleangeoipClean-
     CleanapiClean-CleancClean'.
according to the installation guide the building requires GNU make utility.
Simple install on Unix/Linux like systems:
  1. Direct download or clone the repository.
  2. Locate to the repository root directory.
  3.
     ./bootstrap # dependent on autoreconf (autoconf, libtool, automake)
     ./configure
     gmake
     gmake check
     gmake install

Via Ubuntu PPA:
  1. sudo add-apt-repository ppa:maxmind/ppa
  2. sudo apt update
  3. sudo apt install libgeoip1 libgeoip-dev geoip-bin


## netmap ##
Dependency for installation are the kernel source files which are passed as
configarion option --kernel-sources

Installation guide is in detail on 'https:Clean//Cleangithub.CleancomClean/CleanluigirizzoClean/Cleannetmap
     Clean'.Clean CleanThe
process varies depending on machine NIC used for capturing. The list of
supported NIC drivers can be found in official repository. This is an example
for Intel 1Gb Adapter with e1000e driver:
  1. Direct download or clone the repository.
  2. cd path/to/netmap/LINUX
  3.
    ./configure --no-ext-drivers --kernel-sources=/usr/src/linux-source-4.15.0
                --drivers=e1000e
     make
     make install
  4. Disable multiple device offloadings:
      ethtool -K <device_name> tx off rx off gso off tso off gro off
  5. Disable pause frames:
      ethtool -A <device_name> autoneg off tx off rx off


## netmap-libpcap ##
Dependencies are flex and bison packages. If the configuration fails, pass
--without-flex --without-bison to configuration.

Full installation guide is in official repository
'https:Clean//Cleangithub.CleancomClean/CleanluigirizzoClean/CleannetmapClean-CleanlibpcapClean/CleanblobClean/
     CleanmasterClean/CleanINSTALL.CleantxtClean'.
Simple install on Unix/Linux like systems:
  1. Direct download or clone the repository.
  2. Locate to the repository root directory.
  3.
     ./configure
     make
     make install (optional)
        - run 'make install' only if you want to replace your current libpcap.


#############
## INSTALL ##
#############

## traffic_collector ##
After successfully installing the listed dependencies this is the main guide
to build traffic_collector on Unix/Linux like system to following dependecies
are required:
  - cmake
  - make
  - gcc
  - gmake -> for building geoip library
  - netlib -> gethostbyname
```

```
Installation process:
  1. Locate to the repository root directory.
  2.
     mkdir build && cd build
     cmake ..
     make


## elk stack ##
Traffic_collector sends captured data to ELK stack infrastructure. Installation
guide in detailed is on the official elastic website
'https:Clean//Cleanwww.Cleanelastic.CleancoClean/CleanguideClean/CleanenClean/CleanelasticClean-CleanstackClean/
      CleancurrentClean/CleaninstallingClean-CleanelasticClean-Cleanstack.CleanhtmlClean'.
Requirement is to install:
  - Filebeat
  - Logstash
  - Elasticsearch
  - Kibana
All components are configured with specific configuration files. To speed up
the use the ansible for elk stack deployment:
'https:Clean//Cleangithub.CleancomClean/Cleantomas321Clean/CleanansibleClean-CleanrolesClean'

After installing ELK stack components to set up Kibana visualizations for
immediate network monitoring use the 'initialize_kibana.sh' script which
requires a direct connection to Kibana and Elasticsearch.

Possible problems:
  1. If the script was unable to connect, ensure the following passes:
    - nc -v <elastic.host> 9200
    - nc -v <kibana.host> 5601

## rsyslog ##
To enable rsyslog use the Ansible playbook 'traffcol/rsyslog.yml' with modified
target host arccording the 'hosts' file. The playbook sets the target log file
and configures the log rotation.
Deploy it by running:
  'ansible-playbook -K traffcol/rsyslog.yml'
from the root Ansible directory.
```

Appendix B

User's guide

```
##################
## Usage Guide ##
##################

### Start Traffcol ###

1. Refere to the installation guide file for compilation and installation.
2. Modify `<REPO_ROOT>/resources/config.yml` accordingly.
3. Run any of the built targets from <REPO_ROOT>.
    Example:
        - locate to <REPO_ROOT>
        - run `./build/traffcol-validate`
        - this validates your configuration file

NOTE: If netmap-based libpcap is not your installed libpcap library:
For running 'traffcol' main executable you must have super-user priveleges
and prepend the execution with LD_LIBRARY_PATH variable setting the path to
netmap-based libpcap directory containing the library file.

# Configuration validation #
1. Locate to <REPO_ROOT>
2. Run `./build/traffcol-validate`

# Print configuration file #
1. Locate to <REPO_ROOT>.
2. Run `./build/traffcol-print-config`.

# Start sensor for packet capture #
1. Locate to <REPO_ROOT>.
2. Run `sudo ./build/traffcol`.


### Start Kibana ####

After successfully compiling the sensor, installing and configuring the ELK
stack with initialized index template and loaded visualizations and dashboards,
you are ready to use the Kibana for monitoring your network (if sensor is
running and data is flowing).

1. Open Kibana in your favorite browser by going to <host>:<port> address (if
   the port was not change during installation its 5601)
2. Kibana offers to try their sample data for experimenting. Choose the other
   option for exploring your own data. (applies to first open)
3. Locate to the `Dashboard` from the side bar and select the dashboard from
   list.
4. Possible errors will pop-up on the top of the page if no data is yet
   present.
5. Monitor your network traffic by manipulating filters and time ranges.

## Filters ##
- Use predefined filter locate on top of the dashboard.
- More specific filters are possible by clicking an area or a point on graph
  representing values.
- To filter by source or destination service (port number) there are 2 options:
  1. Issue 'service.source:<port>' or 'service.destination:<port>' commands to
     the top search box.
  2. Use the 'Add a filter' feature, which is more user friendly settings.
```

Appendix C

Bash script for packet capture efficiency

```
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 D

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 E

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 F

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 G

IIT.SRC research paper

Differentiating packet capture mechanisms in Proceedings of IIT.SRC 2019

Tomáš BELLUŠ*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xbellust@fiit.stuba.sk

Abstract. This research paper, as a part of Bachelor's thesis, deals with analyzing various network traffic capture tools or libraries for Linux-based systems and choosing the most efficient one for real time network traffic. The main goal is to differentiate between native libraries (e.g. Libpcap), low-level protocol families (e.g. PF_PACKET/AF_PACKET, PF_RING) and other capture mechanisms with possible zero-copy/one-copy features, because efficiently keeping track of network traffic indications, is crucial for security and administrative reasons.

1 Introduction

The key difference in efficient network capture is

- the count of redundant copies of received packet data
- packet post-processing bypass
- buffers shared between kernel and user space.

Distinguishing the most efficient capture mechanism consisted of testing `the elapsed time` of capturing a static count of packets at consistent packet rate and testing for any drop rate of captured packets within a predefined time at various packet rates (`packet capture efficiency`). Testing network topology (see Figure 1) consists of two equal machines with following specifications.

- Operating system: `Ubuntu server 18.04.2 LTS`
- Ethernet Controller: `Intel(R) 82579LM Gigabit Network Connection`
- Process: `Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz, boost 3.70GHz, 4 cores`
- Memory: `8 GB DDR3, 1333 MHz`

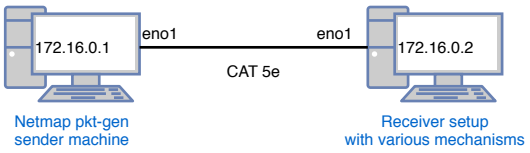


Figure 1. Testing topology

2 Packet capture

The most critical part is real time data capture in network traffic, meaning capturing network packets using a framework or capture library. It is a dependent process of hardware components like Ethernet Controller (NIC), processor performance and its properties (e.g. number of processor cores). Packet capture takes place in the NIC, but packet processing takes place in higher system layers, like the user space for Libpcap library. More efficient mechanisms share data from memory, which falls under the jurisdiction of lower system layer - the 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 buffers, 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 kernel. Kernel copies the buffer content to kernel space buffers (called *m_buffs* or *sk_buff*[1]). These buffers are not

---

* Bachelor study programme in field: Computer Engineering
Supervisor: Ing. Dušan Bernát, PhD., Institute of Computer Engineering and Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

[1] These kernel buffers are expensive to create and vary in size. They are complex and include large amount of metadata. [4]

accessible by user processes, but can be copied to the user space or the process accessible memory. This is a typical scenario

- two copies
- interrupts between kernel and user process to access received data on the NIC.

Possible improvements is to use, as mentioned previously, buffers shared between the kernel and the user space, which eliminates one or more copies and interrupts from kernel to user space, leaving only one copy between NIC buffers and shared buffers. Additional improvement is to have a NIC supporting multiple buffers, which splits the load and each buffer is handled by different core simultaneously [3].

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) feature or a one-copy mechanism, which utilizes the shared buffers.

3 Mechanisms

Tested mechanisms in scope of this research paper are

- *PF_PACKET*[2] protocol family
- standard Linux library *Libpcap*[3]
- open source framework *Netmap*[4]
- *PF_RING*[5] protocol family by *ntop*

3.1 PF_PACKET

A protocol family or file descriptor (FD) introduced to utilize the shared buffer is bypassing the Linux network stack and the packet socket FD receives full Ethernet frames. This packet socket is utilized by Libpcap since v0.6 release [5].

3.2 Libpcap

Standard Linux packet capture library utilized by Wireshark and Tcpdump. On datalink layer it captures packets with PF_PACKET and in its user space library it breaks down the packet to structures for further processing by the user process.

3.3 Netmap

Netmap targets the processing bottleneck in copying data from NIC to shared buffers. Netmap has its own *pkt_buffs* replacing standard *sk_buffs*, which are preallocated in comparison with *sk_buffs*' constant reallocations [3]. It includes *netmap* kernel module utilizing modified NIC driver kernel module.

3.4 PF_RING

On the socket layer it replaces PF_PACKET. It utilizes ring buffers in shared memory and modified NIC driver kernel modules and the *pf_ring* kernel module. Through Linux New API (NAPI)[6] the kernel module copies polled packets from the NIC to shared ring buffers accessible by user application [2] Enhanced packet capturing is dependent of Intel NICs, but it works for other NICs too.

4 Performance testing

Setting up the testing environment consisted of two bootable Ubuntu server USB sticks with persistent storage. Both with installed Netmap framework with Netmap kernel module utilizing Netmap's patched NIC kernel module matching the original (see Listing 1). Machine generating packets (sender) was setup for running the Netmap's *pkt-gen*. Machine capturing packets (receiver) had, in addition to the Netmap framework, a compiled Netmap-based Libpcap and additionally had configured PF_RING, its modified Libpcap and installed corresponding library *libpfring*.

Listing 1. Netmap setup list of commands for NIC kernel module e1000e

```
sudo apt install linux-source
git clone https://github.com/
    luigirizzo/netmap dev/netmap
cd dev/netmap/LINUX
./configure --no-ext-drivers --
    kernel-sources=/usr/src/linux-
    source-4.15.0 --drivers=e1000e
make && sudo make install
sudo rmmod e1000e
sudo insmod ./netmap.ko
sudo insmod ./e1000e/e1000e.ko
```

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

[3] http://www.tcpdump.org/manpages/pcap.3pcap.html

[4] http://info.iet.unipi.it/~luigi/netmap/ or https://github.com/luigirizzo/netmap

[5] https://www.ntop.org/products/packet-capture/pf_ring/ or https://github.com/ntop/PF_RING

[6] 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 [1]

Using *ethtool*, the network device features are disabled (-K option) for Netmap to correctly interact with its ring buffers. Secondly, the pause frames (option -A) are also disabled, which negotiate the allowed packet rx/tx frequency and therefore limits the receiving of transmitting process. (see Listing 2)

Listing 2. Machine network setup to maximize capturing for both machines

```
sudo ethtool -K eno1 tx off rx off
    gso off tso off gro off
sudo ethtool -A eno1 autoneg off tx
     off rx off
```

After setting up the sender, it could generate up to 1.389 million packets per second (Mpps) with exact options shown in Listing 3.

Listing 3. pkt-gen command executed on sender machine

```
sudo pkt-gen -i eno1 -c 4 -f tx -l
    60 -w 3 -d 172.16.0.2 [-R <rate
    >pps]
```

Table 1. pkt-gen options

option	description
-i	interface used
-c	number of CPUs
-f	function
-l	packet size
-w	wait for link time
-d	destination IP address
-R	packet rate

The receiver was setup with PF_RING as listed in Listing 4 and for both Netmap and PF_RING with compiled modified Libpcaps in their respected directories.

Listing 4. PF_RING setup list of commands for NIC kernel module e1000e

```
git clone https://github.com/ntop/
    PF\_RING dev/pfring
cd dev/pfring/kernel
make && sudo make install
sudo insmod ./pf_ring.ko
cd ../drivers/intel && make
cd e1000e/e1000e-*-zc/src
sudo ./load_driver.sh
# set up libpfring and libpcap
cd dev/pfring/userland/lib
./configure && make && sudo make
    install
cd ../libpcap && ./configure &&
    make
sudo make install
```

4.1 Packet capture efficiency

Testing the packet capture efficiency comprised of sender executing the pkt-gen command (see Listing 3) with

- rate of 1000 pps (see Table 2)
- and maximum rate of 1.389 Mpps (see Table3).

The receiver was capturing packets for 120 seconds and returning basic statistics of captured and dropped packets. For every capture mechanism the process was tested 5 times.

Table 2. Packet capture efficiency median results for 1000 pps

mechanism	packets captured	packets dropped
Libpcap	119963	0
PF_PACKET	120640	0
Netmap	119971	0
PF_RING	119964	0

Table 3. Packet capture efficiency median results for 1.389 Mpps

mechanism	packets captured	packets dropped
Libpcap	86880719	1003
PF_PACKET	117737210	9879085
Netmap	92077024	1143
PF_RING	21476253	0

Dropped packets is read from the mechanism's statistics output (e.g. *getsockopt* or *pcap_stats*), although considering that total packets sent was

$$1.389\ Mpps * 120\ s = 166.68\ Mpkts \quad (1)$$

4.2 Elapsed time of captured packets

Testing the elapsed time (measured in nanoseconds) comprised of sender executing the pkt-gen command (see Listing 3) with no rate provided (resulting rate 1.389 Mpps) and on the receiver side changing the number of packets to be captured. The results are shown in Table 4.

Table 4. Elapsed time of captured packets with median results

mechanism	1K pkts [ms]	100K pkts [ms]	10M pkts [s]
Libpcap	2.292	140.531	13.847
PF_PACKET	5.060	505.313	50.768
Netmap	2.112	129.603	12.965
PF_RING	6.814	388.881	38.673

5 Conclusion

Referring to Packet capture efficiency (see Section 4.1) for the rather small packet rate all mechanisms captured 100% of generated packets. For the maximum packet rate the most packets were captured by PF_PACKET, which are raw Ethernet frames and would still require post processing. On the other hand Netmap-based Libpcap captured the most packets from the Libpcap-based mechanisms.

Although PF_PACKET captured to most packets in the previous test, it was not the fastest mechanism. Netmap managed to capture any number of packets tested in the shortest period of time (see Table 4). PF_PACKET could have been the fastest, but since measuring elapsed time was implemented in python, it might have influenced its performance.

Altogether, Netmap-based Libpcap has shown improvement to its native counterpart in both tests. PF_RING was surprisingly inefficient, which could have been due to improper opening of the network device in Libpcap. Worth mentioning are the native APIs of both Netmap and PF_RING, which are much more efficient and worth further testing.

Acknowledgement: This work was supported by the Ministry of Education, Science, Research and Sport of the Slovak Republic within the Research and Development Operational Programme for the project "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the European Regional Development Fund. The work was also partially supported by the Slovak Research and Development Agency (APVV-15-0789).

References

[1] corber: NAPI. https://lwn.net/Articles/30107/, 2003, [21.11.2018].

[2] ntop: Vanilla PF RING. https://www.ntop.org/products/packet-capture/pf_ring/, [20.11.2018].

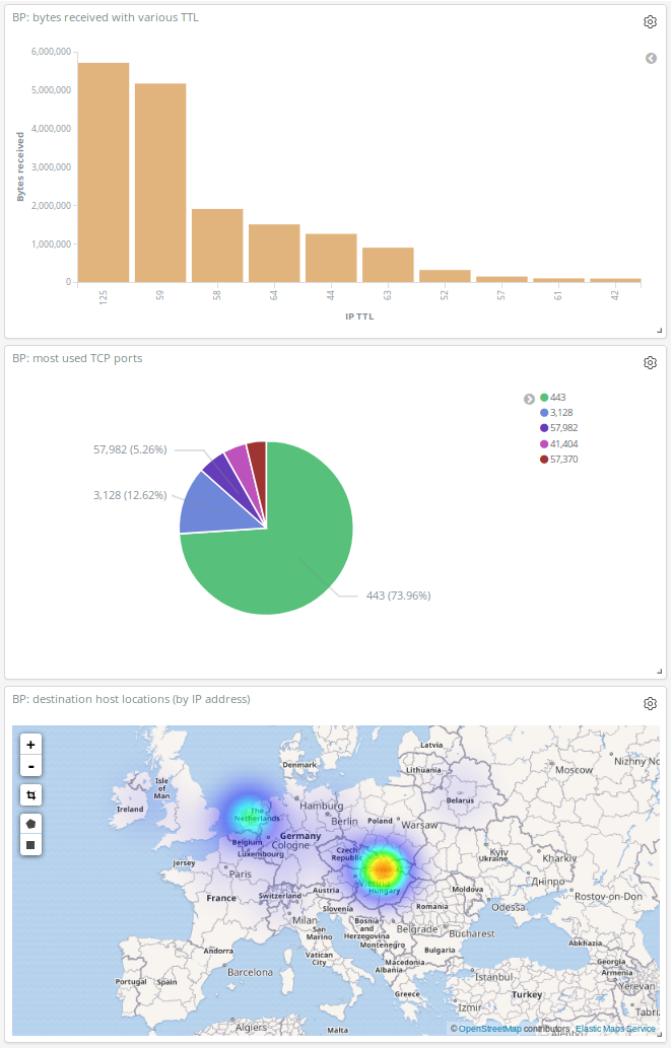
[3] Rizzo, L.: netmap: a novel framework for fast packet I/O. Technical report, Universita di Pisa, https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf [17.11.2018].

[4] Rizzo, L.: netmap: A Novel Framework for Fast Packet I/O. https://www.youtube.com/watch?v=la5kzNhqhGs, 2012, USENIX conference presentation [24.11.2018].

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

Appendix H

Kibana dashboard



Appendix I

Sensor configuration

```
1  # capture interface and mode
2  # default interface => not specified
3  sensor:
4      interface: eth1
5      # one of [in, out, promisc] default direction =>
           promisc
6      direction: promisc
7
8  # Filebeat connection specification
9  # default host => localhost:12000
10 beats:
11     host: 192.148.56.4
12     port: 12000
```

Listing I.1: Configuration example. File is located in *project_root*/resources/traffcol.yml.

Appendix J

Logstash configuration

```
1  input {
2    beats {
3      port => 12345
4    }
5  }
6
7  filter {
8    json {
9      source => "message"
10   }
11   mutate {
12     remove_field => ["@version", "message", "input", "
         log", "tags", "prospector"]
13   }
14 }
15
16 output {
17   elasticsearch {
18     hosts => "localhost:9200"
19     manage_template => false
20     document_type => "_doc"
21     index => "bp-%{+YYYY-MM-dd2}"
22   }
23 }
```

Listing J.1: Configuration file on machine with installed Logstash (/etc/logstash/conf.d/sensor.conf).

Appendix K

Plan of work

K.1 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
```

K.2 Winter semester evaluation

Staring in the summer before the semester I met with my supervisor and discussed the vision of the project with main target aspects. The plan for winter semester was to properly analyze the problem and design a solution with everything documented in this thesis. I made progress each week and kept my supervisor posted with current changes and further work. I analyzed the problem in depth and briefly designed the final system. Although, the design was not complete and the thesis lacked correct implementation sections. Altogether, the thesis was handed over with finalized analysis of the problem and mostly finished design.

K.3 Summer semester

```
1  ===11-02-2019===
2  - Start deploying functional ELK stack.
3  - Begin scripting Ansible automation
4
5
6  ===18-02-2019===
7  - Test automated (Ansible) deployment of the ELK stack.
8  - Begin programming the sensor (Controller class) and
       setting up its configuration file.
9  - Load configuration file to C++ structures.
10 - Configuration object verifying configuration file.
11
12
13 ===25-02-2019===
14 - Add test index for checking database connectivity.
15 - Implement the prototype Database class and Log class.
16 - Redirect error messages and other to log file.
17
18
19 ===04-03-2019===
20 - Start network capturing development of the Interface_
       connection class.
21 - Include Parser class with RapidJSON library.
22
23
24 ===11-03-2019===
25 - Include multi-threading between database control
       object and capturing object.
26 - Insert first packet to the database.
27 - Add complete packet parsing in the Parser class.
28
29
30 ===18-03-2019===
31 - Add monitoring features (check database disk space and
       log file disk usage).
32 - Parse Elasticsearch response JSON of summarized data.
33 - Test delete_by_query API.
34 - Test archiving log files and logging continuity.
35
36
37 ===25-03-2019===
38 - Add initialization logging and release alpha version.
39 - Create examplary visualizations in Kibana.
```

```
40 - Version testing with various sensor configurations.
41
42
43 ===01-04-2019===
44 - Fix misconceptions and optimize program.
45
46
47 ===08-04-2019===
48 - Structure technical documentation.
49 - Possible changes in program (optimizations).
50
51
52 ===15-04-2019===
53 - Final testing and documentation.
54 - Start writing missing thesis sections.
55
56
57 ===22-04-2019===
58 - Empty week for corrections and finalization.
59
60
61 ==29-04-2019===
62 - Release version 1.0.0.
63
64
65 ===06-05-2019===
66 - Project deadline 07-05-2019.
```

K.4 Summer semester evaluation

The plan for finalizing the system included full implementation and documentation. More or less, my progress was following the specified plan, but after few week I diverged from the original plan and was falling back. Nevertheless, after fixing issues that slowed me down (programming misconceptions and barriers and most fitting solutions) I came back on track and followed the plan. Main changes in the plan include logging component from custom class implementation to utilizing the syslog server, the program is not multi-threaded but single. I managed to finish the all sections in time and add deployment options as extra, which makes it a good evenly distributed plan of work.

Appendix L

Digital submission

Registration number from AIS: FIIT-104199-82385
Contents of the digital submission (ZIP archive):

File	Description
sensor/	contains the C++ project structure
bin/	all executable source files
lib/	local library files
include/	all C++ header files
src/	all C++ source files
resources/	configuration and data files
elk/	kibana objects and index template
geoip/	geolocation data
config_specs.yml	
traffcol.yml	main configuration file.
dependencies	list of sensor dependecies
CMakeLists.txt	cmake file
initialize_kibana.sh	kibana initialization script
ansible/	main ansible directory
ansible.cfg	ansible configuration file
ansible.log	predifined ansible log file
hosts	list of ansible target hosts
playbooks/	all playbooks
? elk/	ELK stack specific playbooks
rsyslog.yml	rsyslog deployment playbook
traffcol/	traffcol specific playbooks
elk.yml	ELK stack deployment playbook

```
    traffcol.yml          traffcol dependecies playbook
  roles/                  all ansible roles

thesis/
  thesis.pdf              thesis pdf version
```