BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

USING NETFLOW DATA TO CREATE FILTERING RULES

VYUŽITÍ NETFLOW DAT PRO TVORBU FILTRAČNÍCH PRAVIDEL

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **DOMINIKA PLOČICOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

146752

Institut:	Department of Information Systems (UIFS)
Student:	**Pločicová Dominika, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Computer Networks
Title:	**Using NetFlow Data to Create Filtering Rules**
Category:	Networking
Academic year:	2022/23

Assignment:

1. Get familiar with NetFlow technology for monitoring network traffic.
2. Study libnf library that allows processing NetFlow data.
3. Design a system that can detect an ongoing attack for selected rules defined by the administrator and create a filtration rule for effective attack mitigation. Focus on DDoS attacks.
4. Implement and deploy the proposed system in the BUT test environment.
5. Evaluate the achieved results and discuss possible extensions.

Literature:

- Claise B., "Cisco Systems NetFlow Services Export Version 9", RFC 3954, DOI 10.17487/RFC3954, October 2004, https://www.rfc-editor.org/info/rfc3954>
- Hofstede R. et al., "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX," in IEEE Communications Surveys & Tutorials, vol. 16, no. 4, pp. 2037-2064, Fourthquarter 2014, doi: 10.1109/COMST.2014.2321898.
- Mirkovic J., and Reiher P. "A taxonomy of DDoS attack and DDoS defense mechanisms." *ACM SIGCOMM Computer Communication Review* 34.2 (2004): 39-53.
- Zargar S. T., Joshi J. and Tipper D. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks." *IEEE communications surveys & tutorials* 15.4 (2013): 2046-2069.

Requirements for the semestral defence:
1., 2.

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

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	26.10.2022

Abstract

The goal of this work is to design a system that will be able to detect an ongoing DDoS attack and create filtering rules to stop it. Upon getting the input definition of chosen traffic profiles from the network administrator, it should examine the ongoing traffic. In the case of an eventual DDoS attack from one or more defined profiles, it shall create a filtering rule to eliminate traffic belonging under the said profile (or profiles, respectively).

Abstrakt

Cieľom práce je navrhnúť a implementovať systém, ktorý bude schopný detegovať prebiehajúci DDoS útok z dát NetFlow a vytvoriť filtračné pravidlo na jeho zastavenie. Po načítaní definície profilov danej administrátorom by sa mala skúmať sieťová prevádzka. V prípade prebiehajúceho DDoS útoku z jedného alebo viacerých definovaných profilov by sa malo vytvoriť filtračné pravidlo pre elimináciu prevádzky spadajúcej pod daný profil (prípadne profily).

Keywords

NetFlow, DDoS, flow monitoring, libnf, DDOS attack detection

Klíčová slova

NetFlow, DDoS, monitorovanie tokov, libnf, detekcia DDOS útokov

Reference

PLOČICOVÁ, Dominika. *Using NetFlow Data to Create Filtering Rules.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Cieľom tejto práce je navrhnúť systém, ktorý by bol schopný detegovať prebiehajúci DDoS útok na administrátorom definovanom profile z dát NetFlow a vytvoriť filtračné pravidlo pre jeho zastavenie.

Prvá kapitola sa teda zaoberá protokolom NetFlow a monitorovaním siete. Popisuje, čo samotné monitorovanie je a ako sa klasifikuje. Následne rozoberá protokol NetFlow - jeden z protokolov, ktoré je možné použiť na exportovanie informácií a atribútov siete pre optimalizáciu a charakterizáciu prevádzky siete, a nastavenie služieb pre dané pripojenie. Posledná časť tejto kapitoly rozoberá, čo je monitorovanie tokov a ako je štruktúrované.

Popisu a klasifikácii DDoS útokov je venovaná tretia kapitola. Keďže práca sa zaoberá detekciou týchto útokov, v prvej časti kapitoly sa nachádza krátke vysvetlenie toho, čo vlastne DDoS útok je. Následne je popísané, ako sa prejavuje DDoS útok, spolu s poukázaním na niektoré z najznámejších (a najväčších) útokov - Mirai útok z roku 2016, DDoS útok na GitHub z roku 2018, a najnovší útok nazvaný Mantis z júna 2022. V poslednej časti kapitoly sa rozoberajú typy DDoS útokov.

V štvrtej kapitole sa nachádza stručný pohľad na testovacie prostredie a množstvo dát, ktoré sú na danom stroji ukladané. Druhá polovica kapitoly sa venuje popisu knižnice `libnf`.

Piata kapitola predstavuje návrh a implementáciu cieľového systému. Najskôr predstavuje všeobecný návrh systému - ako bude daný systém pracovať a aké budú jeho súčasti. Následne je v kapitole predstavený algoritmus, ktorý je využitý pre vyhľadávanie útočníka (útočníkov) v prípade detegovaného možného DDoS útoku. Súčasťou popisu algoritmu sú aj pseudokódy najpodstatnejších častí algoritmu a popis spôsobu agregovania dát pri detekcii, ako aj spôsob vyhodnocovania počtu výsledných pravidiel.

Ďalšia časť kapitoly sa venuje samotnému programu. Popisuje jeho štruktúru a najdôležitejšie funkcie a úložisko všetkých spracovaných dát (MySQL databáza). Taktiež predstavuje vstupy a výstupy programu. Vstupom je konfiguračný súbor, ktorý aplikácia dostane od administrátora. V danej konfigurácii musí byť definícia aspoň jedného profilu, ktorý bude skúmaný - analyzujú sa len dáta patriace práve tomuto profilu (alebo profilom v prípade, že ich je viac). Výstup programu je k dispozícii v databáze v tabuľke `filters`. Kapitola pokrýva aj spustenie a preklad programu.

V poslednej, šiestej, kapitole, sa nachádza zhodnotenie aplikácie a popis jej testovania. Kapitola sa venuje najmä testovaniu samotnej detekcie útokov v prípade, že sa nejakú útok odohral, a tiež výkonnostnému testovaniu programu, kde sú spomenuté aj problémy, ktoré sa počas testovania objavili. Celkovo je aplikácia funkčná podľa požiadaviek, je schopná detegovať útok pri jeho výskyte a vie bežať a analyzovať prichádzajúce dáta aj v reálnom čase

Using NetFlow Data to Create Filtering Rules

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Matěj Grégr, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dominika Pločicová
May 16, 2023

Acknowledgements

I would like to thank my supervisor Ing. Matěj Grégr, Ph.D. for advice, patience, understanding and all the time spent on consultations.

Contents

List of Figures

Chapter 1

Introduction

In the age of the Internet and network communications, the need for data analysis and network monitoring is constantly increasing. With the rapid increase of Internet users and devices connected and communicating over the network, it is becoming more and more attractive for attackers to carry out malicious actions - whether for personal gain or to cause trouble for their gratification.

Network or flow monitoring and data analysis can help immensely in detecting such malicious actions and uncovering network vulnerabilities or security risks. Vulnerable network nodes, their abnormal behavior, or atypical network flows can be revealed through export network protocols. They allow us to find both internal and external incidents, help improve transmission and balance the network load.

One of the malicious actions mentioned above could be a Denial of Service (DoS) or Distributed Denial of Service (DDoS) attack. It can be inconvenient to lose access to some important data or devices due to them no longer being able to communicate, and this can be caused, among many other reasons, by DDoS attacks. These attacks can be detected by monitoring and analyzing the network flow.

This paper first looks at network monitoring. It describes what monitoring itself is and how it is classified. Then it describes one of the protocols that can be used for exporting the information and attributes of a network flow, for traffic optimization, characterization of network operation, and setting of the services for the given connection - the NetFlow protocol. Moreover, it can detect weak real-time communication points, identify dominant traffic sources of traffic or ensure better network setup. It also gives the network administrator a better and more visible overview of the entire network. At the end of this chapter, it is defined in more detail what flow monitoring is and how it is structured.

The next chapter is dedicated to one of the most common threats on the Internet - DDoS attacks. It explains what DDoS attacks are, how they work and can be identified, what types of attacks there are and which are most common. They may be simple to carry out, but they can still be very effective.

A DDoS attack can sometimes be very annoying but not necessarily harmful. For example, in a (non-competitive) game, one of the players is really good, and one of his opponents might not like it so much, so the opponent might hire a botnet to carry out an attack that makes the game more difficult or even impossible for the good player. So it is annoying for the good player, but it is not something that would do much damage. However, for critical infrastructure like hospitals, which are a popular target for DDoS attacks, it can have a catastrophic impact.

For that reason, it is necessary to be able to detect such an attack and, if it is possible, to stop it as well. The fourth chapter shows a preview of the way how NetFlow data can be used to detect the attack. Firstly, it shows the architecture and provides statistical information on the amount of data that passes through the testing environment. Then it introduces a library that can be used for the proccessing of the data itself.

The goal of this work is to design a system that would try to detect an incoming attack (or outcoming). Upon getting the input definition of chosen traffic profiles from the network administrator, it should examine the ongoing traffic. In the case of an eventual DDoS attack from one or more defined profiles, it shall create a filtering rule to eliminate traffic belonging under the said profile (or profiles, respectively).

The fifth chapter therefore provides a closer look at the designed system and its implementation. The introduction to the chapter begins with the overall description of how the whole system has been designed and how it works. Second part of the chapter covers the detection algorithms used for the work purpose. The chapter then describes the input and output of the program. Lastly, it provides a short documentation of the implementation itself.

Last chapter then evaluates the design of the implemented system. It provides a closer look at the testing cases that have been done on the model as well as the results that the system was able to collect. It also covers possible future plans for the system and its weaknesses.

Chapter 2

Network Monitoring

The continuous growth of the network system, the number of users with various requests, and different types of services and applications require some tools for intelligent network management. Besides other things like installing cables, connecting computers, or configuring IP addresses, network management also includes tracking and monitoring activities on the network, detecting errors and attacks, balancing the network performance, planning, etc.

To make network management effective, it is crucial to know the current state of the network. To do this, we need information about the network topology (which devices are connected to it, what are their addresses), the current state of the devices (running interfaces, processes and available services), statistical data about the transmission (the number of bytes transmitted, packets, broadcast packets) and sometimes the content of the transmitted data as well. The process of gathering information can be divided into two categories - data analysis and monitoring. Data analysis is a process that deals with the detailed examination and evaluation of the collected data.

On the other hand, monitoring provides information about the current state of network devices and services [11]. Over the years, various approaches to network monitoring have been proposed and developed, each serving a different purpose. They can be divided into two categories [8]:

- active approaches - implemented by tools such as Ping or Traceroute, which inject traffic into a network to perform different types of measurements.
- passive approaches - they observe existing traffic generated by users. One of these approaches is packet capture, which generally provides better insight into network traffic, as complete packets can be captured and further analyzed. However, in high-speed networks, packet capture requires expensive hardware and extensive infrastructure for storage and analysis. Another approach to passive network monitoring that is more scalable for use in high-speed networks is flow export, where packets are aggregated into flows and exported for storage and analysis.

The definitions of a flow may differ. According to RFC7011, a flow is a set of packets or frames that pass through an observation point in the network during a specific time interval. Packets belonging to a particular flow must share a set of common properties. The most commonly mentioned properties are the source and destination IP addresses, source and destination ports, and IP protocol, but sometimes packet contents and meta information can also be included. It should be noted that the flows are usually unidirectional and that the set of packets belonging to a flow may be empty - it may contain zero or more packets.

Flow export protocols and technologies offer several other advantages over packet capture [8]:

- they are widely used because they are integrated into devices such as routers, switches, and firewalls, therefore no additional devices are needed, and it is less expensive
- flow export is well understood as it is often used for security analysis, accounting, planning, and others.
- a significant reduction in data can be achieved by aggregating the packets once they have been captured
- since it usually only considers packet headers, there are fewer privacy issues

Representatives of flow export protocols are, e.g., NetFlow or IPFIX (IP Flow Information eXport).

2.1 NetFlow

NetFlow is an application patented by Cisco IOS in 1996 that provides statistics about packets flowing through routing devices on the network.

No connection-setup protocol is involved between routers, other networking devices, or end stations, and no external changes are required. It is transparent to the existing network, including end stations, application software, and network devices such as LAN switches.

It identifies packet flows for both incoming and outgoing IP packets. As stated in [4], NetFlow defines a flow as a unidirectional stream of packets between a given source and destination, specifically characterized by the combination of the following seven key fields:

- source IP address
- destination IP address
- source port number
- destination port number
- IP protocol type
- type of service (ToS)
- input logical interface

These seven (or, respectively, five without ToS and the interface) key fields define a unique flow - if a packet has one or more key fields that differ from those of another packet, it belongs to another flow. In addition, depending on the configured export record version, a flow can also contain other accounting fields, e.g., the autonomous system (AS) number in the NetFlow export Version 5 flow format. Flows are stored in the NetFlow cache, which will be described in more detail later [4].

The architecture of NetFlow consists of three pieces[13]:

- flow exporter - an appliance or device that normally collects flow information and exports it to a flow collector; multiple exporters can export flows to multiple collectors

- flow collector - an appliance or server that receives exported flow information
- flow analyzer - an application that analyses the flows collected by the flow collector; the analysis can be both manual or automatic

Each internetworking device performs capture and export independently. The set of traffic statistics captured by NetFlow has a wide spread of uses [4]:

- network application and user monitoring - the detailed view of time and application-based usage of a network allows us to plan and allocate network and application resources and offers almost real-time network monitoring capabilities. That can be used to display traffic patterns, providing proactive problem detection and decent troubleshooting. What is more, we can also use it to detect potential policy and security violations.
- network planning - as a result of data capture over a long time, it is possible to track and anticipate network growth and to plan upgrades. Network planning, including peering, backbone upgrade planning, and routing policy planning, can be optimized, and the total cost of network operations and capacity and reliability can be minimized while maximizing network performance. NetFlow also detects unwanted WAN traffic, validates bandwidth and quality of service (QoS) usage, enables the analysis of new network applications, and offers valuable information for the reduction of the cost of operating the network.
- accounting and billing - NetFlow data provide metering for detailed and highly flexible resource utilization accounting
- traffic engineering - NetFlow-captured traffic data can be used to understand source-to-destination traffic trends, which is convenient for load-balancing traffic across alternate paths or forwarding traffic to a preferred route.
- NetFlow data storage and data mining - NetFlow data can be stored and later retrieved and analyzed in support of marketing and customer service programs.
- DoS and security analysis - NetFlow data may also be used to identify and classify DoS attacks and other threats in real time. Changes in network behavior indicate anomalies. The data is also a valuable forensic tool that could be used to understand and replay the history of security incidents.

2.1.1 NetFlow Versions

NetFlow UDP datagrams can be used to export data in one of the following formats (versions)[4]:

- version 1 - the initially released export format, the original version of NetFlow, very rarely, almost never, used today
- version 2-4 - working NetFlow versions that were never released
- version 5 - a version that adds Border Gateway Protocol (BGP) autonomous system (AS) information and flow sequence numbers, most commonly deployed version today, but supports only IPv4

- version 6 - working NetFlow version that has never been released
- version 7 - only used on some Cisco Catalyst switches
- version 8 - never widely adopted, but it was added to support data export from aggregation caches - it allows export datagrams to contain a subset of the usual Version 5 export data if that data is valid for a particular aggregation cache scheme
- version 9 - next-generation flow formatting, flexible and extensible format, which provides the versatility needed to support new fields and record types. This format accommodates new NetFlow-supported technologies such as multicast, Multiprotocol Label Switching (MPLS), and BGP next hop. It is template based. Templates offer a means of extending the record format, a feature allowing future enhancements to NetFlow services without requiring concurrent changes to the basic flow-record format
- IPFIX - Internet Protocol Information eXport based on the Version 9 export format is sometimes considered as NetFlow's version 10 format

2.2 Flow Monitoring Architecture

Typical flow monitoring architecture setups consists of several stages: packet observation, flow metering and export, data collection, and data analysis [8].

Packet Observation

Packet observation is the first stage in which packets are captured on an observation point (for example, line cards or interfaces of the packet forwarding devices) and then pre-processed. The journey of the packet in this stage could be divided into a few steps[8]:

1. Firstly, packets have to be read from the line before any packet pre-processing - this part is called packet capture and is usually taken care of by Network Interface Card (NIC).
2. Upon entering the card, each packet must pass several checks, such as checksum error.
3. After a packet is captured and has passed the check, the timestamping follows. It can be done in hardware upon packet arrival to avoid delays due to forwarding latencies to software. However, hardware-based timestamping is typically only available on special NICs using Field Programmable Gate Arrays (FPGAs), so it is mostly performed in software.
4. Both packet capture and timestamping are performed for all packets. The next steps are only optional:
    - packet truncation - cuts off all bytes of a packet that exceed the preconfigured snapshot length
    - packet sampling and packet filtering - only packets matching some sampling and filtering rules are selected for measurement; the remaining packets are dropped

Flow Metering and Export

Flow metering and export stage consists of both a metering process and an exporting process. Within the metering process, packet aggregation is performed based on Information Elements (from now on IE) that define the layout of a flow. However, the metering process configuration has yet to be standardized and varies from exporter to exporter. For that reason, the exporters instruct the collectors about which IEs have been used for which flows utilizing templates.

Both IPFIX and Netflow v9 use this approach. In contrast, Netflow v5 is fixed in its initial specification since it does not have template support, and therefore its applicability is limited. For that reason, Netflow v5's evolution is impossible and cannot be used to monitor IPv6 traffic. It is still so widely deployed though that it is still considered a relevant source of information.

After the aggregation, an entry is stored in a flow cache until the flow is considered terminated. Flow cache is a table within the metering process that stores information about all active network traffic flows. The entries are composed of key or non-key fields of IEs. The set of key fields, or flow key, defines the properties that distinguish the flows - it is used to determine whether a packet belongs to a flow already present in the cache or if it is supposed to add a new entry with a new flow.

The size of the cache depends on the memory available and the expected number of flows, as well as selected key and non-key fields and expiration policies. So how long does a flow record stay in the cache? A flow is considered terminated when it matches one of the following conditions[8]:

- active timeout expiration - a flow has been active for a specified time, but the cache entries usually are not removed as the counters are reset and start and end times are updated
- idle timeout expiration - no packets belonging to the specified flow have been captured for a given time
- resource constraint - some flows might be expired prematurely because of resource constraints
- natural expiration - FIN or RST flag in TCP
- emergency expiration - cache becomes full, so some flows have to be removed
- cache flush - when a significant change in the flow exporter system occurs, all flow cache entries expire

After the metering process, flow record sampling and filtering should be performed to reduce the processing requirements for the following stages and the exporting process. While filtering and sampling in the previous stage were only considering packets, in this stage, we take whole flows into consideration - either all packets of a flow or none at all are accounted[8].

Data Collection

The main task of the data collection stage is the reception, storage and pre-processing of flow data generated by the previous stage. Pre-processing operations include aggregation, filtering, data compression and summary generation.

This stage is performed by flow collectors and one or more collecting processes within them. They are an integral part of flow monitoring setups as they receive, store and pre-process flow data from one or more flow exporters.

One of the main tasks of flow collectors is data storage. There are two types of possible storage formats [8]:

- volatile - performed in-memory and therefore very fast, but only temporary
- persistent - used for storing data for a longer time, usually has a larger capacity but is significantly slower than volatile storage type

It is often best to keep the data in memory, especially in the case of on-the-fly data analysis when only results have to be stored. However, sometimes data has to be stored longer than just for processing. In that case, persistent storage would be needed. We distinguish three types of persistent storage [8]:

- flat files - fast, but provides limited query facilities
- column-oriented databases - data is stored in columns instead of rows
- row-oriented databases - data is stored in tables and therefore are accessed by reading the full rows even though only a part of the data may be needed

Another task of this stage is data anonymization. Flow data traditionally has significant privacy protection as it generally does not contain any payload. As a result, the content of end-user communications is protected. However, flows can still be used to identify individuals and track individual activity. For that reason, the collection and analysis of flow data can pose severe risks to the privacy of end users [8].

Data Analysis

Data analysis is often of an exploratory nature (i.e., manual analysis). In contrast, the analysis functions in operational environments are often integrated into the data collection stage (i.e., both manual and automated). Typical analysis functions include correlation and aggregation, traffic profiling, classification, and characterization, anomaly and intrusion detection, and search of archival data for forensic or other research purposes.

Generally, it is the last stage in a flow monitoring setup. There are three application areas for data analysis [8]:

- performance monitoring - aimed at observing the status of services running on the network, with typical metrics including Round-Trip-Time (RTT), delay, jitter, response time, packet loss and bandwidth usage. Performance monitoring applications post-process flow data and show a set of metrics per target service, to verify Service-Level Agreement (SLA) compliance and reveal network events and their impact on end-user experience. Flow measurements sometimes provide only a coarse approximation of standard performance metrics since such metrics are generally not directly measured and exported.
- flow analysis and reporting - usually provides browsing and filtering of flow data, statistics overview, and reporting and alerting functions. Considering the strategic locations of exporting devices, the resulting data presents a comprehensive set of connection summaries. The usual start of data analysis is graph observation if one

is available. From the graph, it is possible to take a closer look at the data in those timeframes where some anomaly has occurred. This reveals which hosts have been top talkers. When top-talkers have been identified, it is not uncommon to identify sources of network scans, brute-force attacks, or even DDoS attacks, as these attacks often result in many small flows. After identification, raw flow data can be retrieved and analyzed to learn the actual nature of the abnormal traffic.

- threat detection - flow data may be used for analyzing certain types of threats. The central observation points at which flow export devices are usually deployed make flow export especially useful for detecting the following attacks and malware: DDoS attacks, network scans, worm spreading, and botnet communication. The commonality between these attacks is that they affect metrics that can be directly derived from flow records, such as the volume of traffic in terms of packets and bytes, the number of active flows in a specific time interval, suspicious port numbers commonly used by worms, and suspicious destination hosts for traffic. Identifying suspicious destination hosts for traffic is usually made using IP reputation lists or blacklists.

Chapter 3

Distributed Denial of Service Attack

DoS attack is a malicious attempt to violate the regular traffic to and from a networked system, service, website, application, or another resource by overloading the target or its surrounding infrastructure with a flood of network traffic. As a result, it keeps the resource's users from accessing it. The attack slows down the whole system, but sometimes it can deactivate it completely.

Classic DoS uses only one device to create the attack, even though these days it is nearly impossible to manage to do any significant harm with just one device. DDoS, on the other hand, uses multiple compromised computer systems (computers or other resources, e. g., Internet of Things (IoT) devices) as its source of malicious traffic. The "slaves" are being coordinated from just one central point to achieve a certain level of effectiveness, though. DDoS attacks are also larger, more devastating, and in some cases much more challenging to detect or stop [5][15].

Numerous methods of defense have been suggested to address the issue. Security systems are frequently bypassed by attackers who adapt their techniques and tools, leading researchers to adjust their methods in response to emerging threats [12].

3.1 How DDoS Attacks Work and How to Identify Them

As was mentioned before, these attacks are performed by a group of machines connected to the Internet. These groups consist of devices infected by some malware that the attacker controls remotely, while the owners of the devices are often unaware of their devices being compromised. These devices are referred to as bots, while a group consisting of bots is called a botnet.

When the botnet is created, the attacker can remotely control each of the bots and direct the attack. Once the botnet targets a victim, it sends numerous requests to the target's IP address, which results in the server or network being overwhelmed and unable to respond to normal requests, resulting in denial-of-service (DoS) [5][15].

In addition, by using a botnet, attackers' identity is hidden as the attack originates from many different systems that all appear legitimate. Most of the bots have always been infected computers. Still, it is becoming almost effortless to exploit an even higher number and different types of devices - especially with a growing number of Internet-of-Things (IoT) devices, with some of them being ideal candidates to become bots due to not having any

built-in security protections (e. g., same default password used on thousands of devices, users not updating them regularly, wide distribution).

In fact, these days, it is even easier to carry out a DDoS attack. In the past, the attackers had to build their own botnets. They had to scan the Internet for vulnerable devices and then compromise them. Nowadays, it is possible to rent DDoS-for-hire botnets for very little money from operators for short-term (but effective) attacks.

So the owner of the bot device is unaware of attacking someone else. But how can the victim or a user of the attacked service tell the server/service is under attack? While a site or service suddenly becoming slow or unavailable is the most obvious sign of a DDoS attack, several different causes can create similar performance issues; therefore, further investigation is usually required. The following indicators can also point to the possibility of a DDoS attack [5][15]:

- suspicious amounts of traffic originating from a single IP address or IP range
- a flood of traffic from users who share a single behavioral profile (e. g., device type, geolocation, web browser version, etc.)
- an unexplained surge in requests to a single page or endpoint
- interesting traffic patterns such as spikes at odd hours of the day or patterns that appear to be unnatural (e.g., a spike every 10 minutes)

There are also other, more specific signs of a DDoS attack that can vary depending on the type of attack.

3.1.1 High-profile DDoS Attacks

Collectively, previously mentioned botnets provide enough power to carry out massive attacks — far more extensive than those launched from a single source [5][15].

Mirai Botnet Attack, 2016

2016 saw the first major incident with the emergence of Mirai, a botnet composed of IoT devices such as routers and security cameras. Weak security made these devices easy pickings for attackers, who constructed a botnet big enough to carry out the most significant DDoS attack ever seen at that time, a distributed denial of service attack (DDoS) on DNS provider Dyn, which affected several high-profile websites, including Spotify, Netflix, and PayPal. The attack was estimated to have involved over 100 000 infected devices and generated peak traffic of 1.2 Tbps [10].

GitHub DDoS Attack, 2018

In 2018, GitHub was hit by 1.35 Tbps of traffic, recording the biggest attack to date at that time. No botnet was required for the attack, as a different technique has been used - memcached amplification, which exploited misconfigured memcached servers to amplify the attack traffic. The attack demonstrated the potential impact of amplification attacks and the need for proper configuration of network servers[3].

Mantis, 2022

According to [6], in June 2022, the largest HTTPS DDoS attack was allegedly reported on record - 26 million requests per second. It was carried out by a botnet called "Mantis", operating only a small fleet of approximately 5000 bots. Due to HTTPS DDoS attacks being more expensive in terms of required computational resources, the botnet size could be surprising. Unlike traditional botnets composed of Internet of Things (IoT) devices, Mantis uses hijacked virtual machines and powerful servers, resulting in more computational resources and thus allowing an attack of this size even with fewer attacking bots.

3.2 Types of DDoS attacks

Different DDoS attacks target varying components of a network connection. Although it is difficult to categorize these attacks, the most commonly recognized categories are application layer, protocol, and volumetric DDoS attacks. Sometimes the categories can even overlap.

3.2.1 Application Layer Attacks

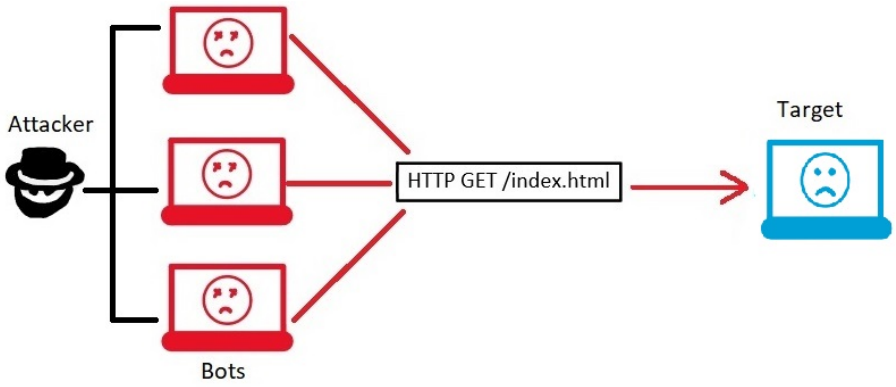

Figure 3.1: Application Layer Attack - HTTP flood

Application layer attacks are directed at web servers, web application platforms, or specific web-based applications rather than the network itself. They target known vulnerabilities in the applications and underlying business logic of the application, or they can abuse higher-layer protocols, for example, HTTP/HTTPS or SNMP. The attacker aims to crash the server and prevent the website or application users from accessing it. These attacks are usually more challenging to detect than the other kinds as they use less bandwidth and therefore do not exhibit such a sudden traffic increase.

These attacks are measured in requests per second. The most common attacks of this type include HTTP floods, low-and-slow attacks, and others [5][15].

3.2.2 Protocol Attacks

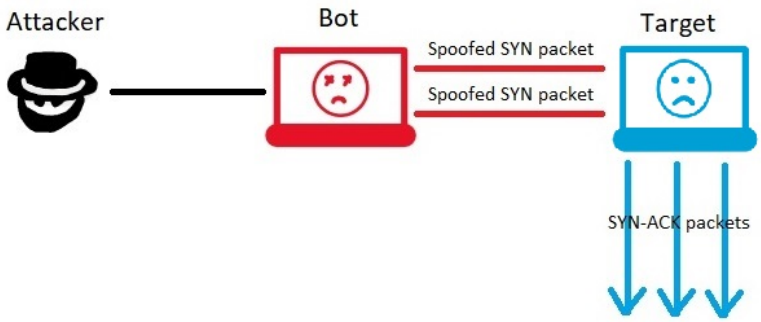

Figure 3.2: Protocol Attack - SYN flood

Protocol attacks, or state-exhaustion attacks, aim to exhaust server resources and intermediate communication equipment, thus causing service disruption. They overburden server resources with phony protocol requests to occupy available resources until there are no resources for normal requests left.

These attacks are measured in packets per second. The most common attacks of this type include SYN floods, Tsunami SYN floods, fragmented packet attacks, Ping of Death, Smurf DDoS, and more.[5][15]

3.2.3 Volumetric Attacks

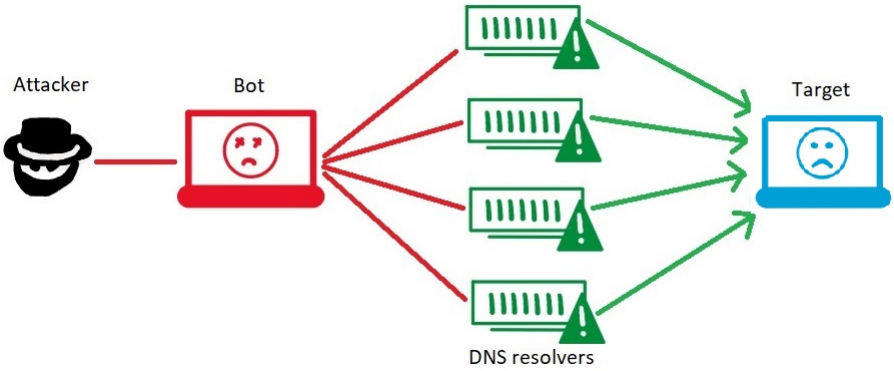

Figure 3.3: Volumetric Attack - DNS amplification

Volumetric or volume-based attacks aim at the bandwidth of a targeted website using amplification methods, halting legitimate traffic and shutting down entire websites. They are difficult to trace since they appear as authentic traffic from multiple IP addresses, even though their origins lie in the attacker's botnet.

This attack type is measured in bits per second. Whereas usual request sizes are around 100s of Gbps, some latest incidents have recorded sizes exceeding 1 Tbps. The most common attacks of this type are UDP floods, ICMP floods, and other spoofed-packet floods.[5][15]

3.2.4 Most Common DDoS Attacks

SYN Flood Attack

The SYN flood attack is the most common form of flooding attack. While classical TCP connection consists of a 3-way handshake (SYN, SYN/ACK, ACK), in this attack, the attacker sends a SYN packet with a spoofed source IP address to the target. The server reserves system resources for this potential connection, replies SYN/ACK to the spoofed IP address, and waits for the response ACK which never arrives. As a result, the server's resources are exhausted, rendering it unable to provide services to legitimate users [7]. Another type of SYN flood attack is a combo SYN flood that comprises two types of SYN attacks. One sends regular SYN packets, and the other sends large SYN packets simultaneously – the regular SYN packets exhaust server resources, while the larger packets cause network saturation [9].

UDP Flood Attack

UDP flood is an attack in which numerous UDP datagrams are sent to random ports. Under normal conditions, when a server receives a UDP datagram at one of the ports, it checks if any programs listening on the specified port are running. It responds with an ICMP message "Destination unreachable" if no such programs are found. As each datagram port has to be checked and all ICMP messages have to be sent, it uses the server's resources which can quickly cause the server to become overwhelmed, thus resulting in denial-of-service [7] [14].

ICMP Flood

In the ICMP flood attack, the attacker floods a target network with a large volume of ICMP (Internet Control Message Protocol) packets, typically ICMP echo requests (ping requests). The attacker sends ICMP echo requests to a multicast IP address of a vulnerable network, broadcast address, or a specific IP address, with the source IP address being the victim's. The nodes in this targeted network reply with an ICMP ECHO response message to the victim. The flood of ICMP ECHO responses causes an overload of the target system [7] [14].

ARP Flood Attack

The ARP flood attack floods the target with spoofed ARP (Address Resolution Protocol) requests, causing exhaustion of computing or memory resources on the victim side. These types of attacks are suitable for use in a local network [7].

Xmas Tree Attack

The Xmas tree attack generates so-called Christmas tree packets with flags such as FIN, URG, and PSH set in the TCP header. Since processing these flags is difficult, it can cause an overload of the targeted system in case of lots of incoming packets [7].

Reset Flood Attack

The Reset flood attack uses TCP packets containing an RST flag and spoofed source IP address. If the target receives a large number of fake RST packets, there is a high probability that some legitimate connections will be reset [7].

Unreachable Host Flood Attack

The Unreachable host flood attack is similar to the Reset flood. An attacker sends an ICMP message "Host unreachable" to the random ports to the victim side with a spoofed source IP address, resulting in some probability that an existing session will be canceled [7].

Ping of Death

The Ping of death attack involves the attacker sending multiple malformed or malicious pings to the targeted system. Despite the maximum packet length of an IP packet being 65 535B, it is usually split into multiple IP packets due to the maximum frame size of the link layer. The system running on the targeted node tries to reassemble the packet and ends up with a packet larger than the maximum size, which can cause a buffer overflow error, and the system may crash [14].

Teardrop Attack

The Teardrop attack is based on sending two or more packet fragments with incorrect offset setting. Fragments then cannot be correctly reassembled to the original packet, which often causes the system to crash [7].

Land Attack

In the Land attack, the attacker sends a spoofed TCP SYN packet with both the source and destination IP addresses of the target, which results in the target trying to establish a connection to itself and possibly causing the machine to reply to itself continuously, which could lead to the downfall of the system [7][2].

Chapter 4

Processing NetFlow Data for DDoS Detection

In this chapter, the focus is on the processing of NetFlow data specifically for the purpose of detecting Distributed Denial of Service (DDoS) attacks. As discussed in the introduction chapter, the goal of this work is to develop a system that can dynamically generate filtering rules when certain thresholds are exceeded on one or more of the profiles defined by the administrator. These threshold crossings serve as indicators of potential DDoS attacks.

To achieve this objective, we utilize network data obtained from the Brno University of Technology's network, which is received and stored on a collector. The architecture of the network and the collector is depicted in Figure 4.1. The data is stored on the collector in the form of flat files, as detailed in Section 2.2.

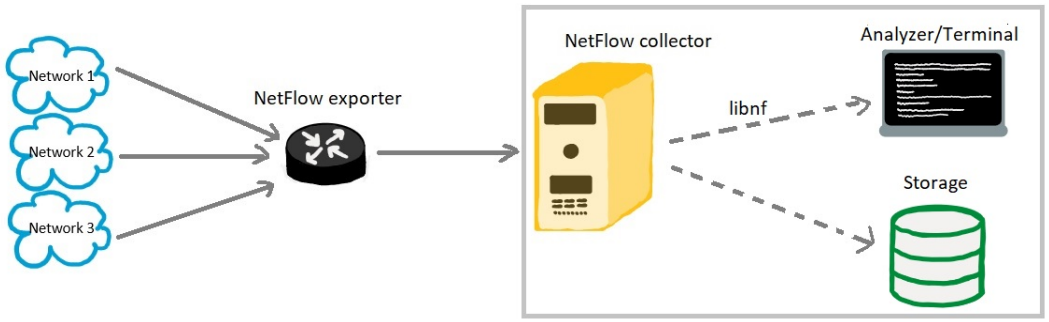

Figure 4.1: NetFlow Architecture

4.1 Incoming Data in Statistics

The provided statistical table, Table 4.1, offers some information regarding the volume of data passing through the exporters. These data are stored in individual files, with each file containing statistics recorded at five-minute intervals.

In order to identify potentially malicious traffic from these records, it is essential to retrieve the data from the stored files and perform further processing. To accomplish this task, a C library called `libnf`, described in the next section, Section 4.2, can be used. This library provides the necessary functionality to extract and manipulate NetFlow data for analysis and detection purposes.

File	Flows	Packets	Bytes
1	6.84 mil	287 mil	0.35 TB
2	6.86 mil	282 mil	0.343 TB
3	6.73 mil	285 mil	0.345 TB
4	6.71 mil	285 mil	0.341 TB
5	6.67 mil	292 mil	0.363 TB
6	6.82 mil	305 mil	0.368 TB
7	6.91 mil	305 mil	0.354 TB
8	6.98 mil	299 mil	0.347 TB

Table 4.1: Incoming data statistics table.

4.2 Library libnf

`libnf` is an open-source library that offers functions and data types for writing flow records into their storage and reading saved flow records from the storage (in this case, the flat files). It can also be used for flow records aggregation, filtration, and closer examination.

This work employs it to retrieve the records from the original saved file and, after that, to store them in a ring buffer. The records then can be taken out from the ring buffer and further processed and examined. Some of the most useful and basic functions and data structures that have been used in the work include the following [1]:

- basic file operations:
  - `int lnf_open` - initialize lnf_filep structure and open file in read or write mode
  - `int lnf_read` - read next record from file
  - `int lnf_write` - write record to file
  - `int lnf_info` - get file info
  - `void lnf_close` - close file and release resources
- record operations, fields extraction:
  - `int lnf_rec_init` - initialize empty record object
  - `void lnf_rec_clear` - zero all fields in initialized record object
  - `int lnf_rec_fset` - get the value of specific field (item) from the record object
  - `int lnf_rec_fget` - set the value of specific field (item) from the record object
  - `void lnf_rec_free` - close record and releases resources
- filter operations:
  - `int lnf_filter_init` - initialize empty filter object
  - `int lnf_filter_init_v2` - initialize empty filter object - new filter using libnf code
  - `int lnf_filter_match` - match record object against filter
  - `void lnf_filter_free` - close filter and release resources
- in memory aggregation and sorting module:

  - `int lnf_mem_init` - initialize empty memheap object
  - `int lnf_mem_fadd` - set aggregation and sort option for memheap
  - `int lnf_mem_fastaggr` - set fast aggregation mode
  - `int lnf_mem_write` - write record to memheap object
  - `int lnf_mem_write_raw` - write raw data obtained from lnf_mem_read_raw to memheap object
  - `int lnf_mem_merge_threads` - merge data from multiple threads into one thread
  - `int lnf_mem_read` - read next record from memheap
  - `int lnf_mem_first_c` - set the cursor position to the first record
  - `int lnf_mem_next_c` - set the cursor position to the next record
  - `int lnf_mem_read_c` - read next record on the position given by cursor
  - `void lnf_mem_free` - close memheap and release resources

- error functions:
  - `void lnf_error` - get error message of last error

- ring buffer operations:
  - `int lnf_ring_init` - initialize ring buffer
  - `int lnf_ring_read` - read record from ring buffer
  - `int lnf_ring_write` - write record to ring buffer
  - `void lnf_ring_free` - release all resources allocated by ring buffer

- data structures:
  - `lnf_brec1_t` - basic record type 1 - contains the most commonly used fields
  - `lnf_field_t` - text description of fields
  - `lnf_file_t` - structure representing single nfdump file
  - `lnf_ring_t` - structure representing the ring buffer
  - `lnf_rec_t` - structure representing single record
  - `lnf_ip_t` - structure representing IPv4/IPv6 address

Chapter 5

Implementation

5.1 Application Workflow

In this chapter, there will be a description of the designed system, the algorithm that was used, and its implementation.

The application is implemented in the C language.

The program first reads the configuration given by the administrator, described later in the Subsection 5.4, and gets the necessary parameters for the run. Most of the parameters have also default values set; the only requirement is the definition of the profiles. The default values can be seen in the file `default.h`.

After the configuration is read and the necessary parameters set and the definition of the profiles processed, a connection to the database is established. Then two different threads are started, the writer and the reader, which can later start another thread - the checker.

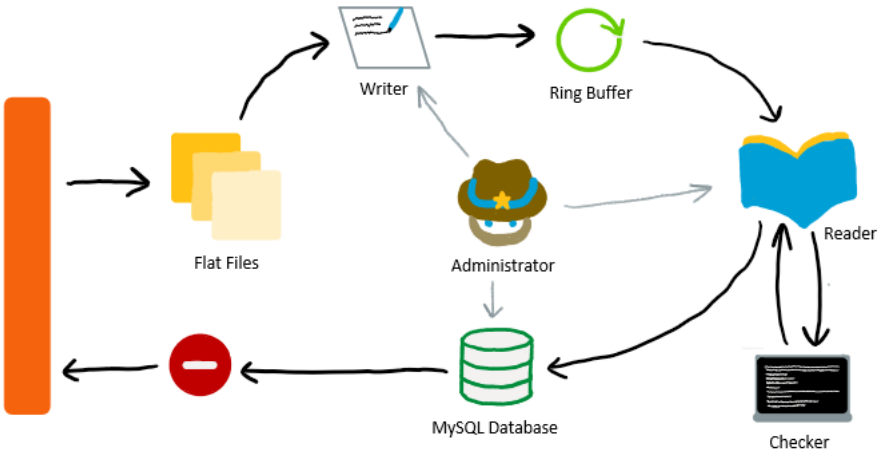

Figure 5.1: Application

The application's workflow is the following, depicted in Figure 5.1:

1. Read the configuration, set parameters, establish the connection to the database and start the two main threads.

2. The writer knows the defined rules for the profiles. It reads the data from the defined `nfcapd` file, and if the read record fits into one of the profiles, it writes it into a ring buffer.

3. The reader reads the records from the ring buffer, stores the data in its memory for each corresponding profile, and counts the numbers of packets, bytes, baselines, etc. In case of a limit crossing, it starts the checker thread.

4. If the checker thread is started, there is a chance of an attack, and the data needs to be checked. If it finds a possible suspect, it returns the result as a filter to the reader, which can write the result into the database.

5. If a filter has been inserted into the database, it means there was a possible attacker found, and the filter can be extracted from the database.

5.2 Detection Algorithms

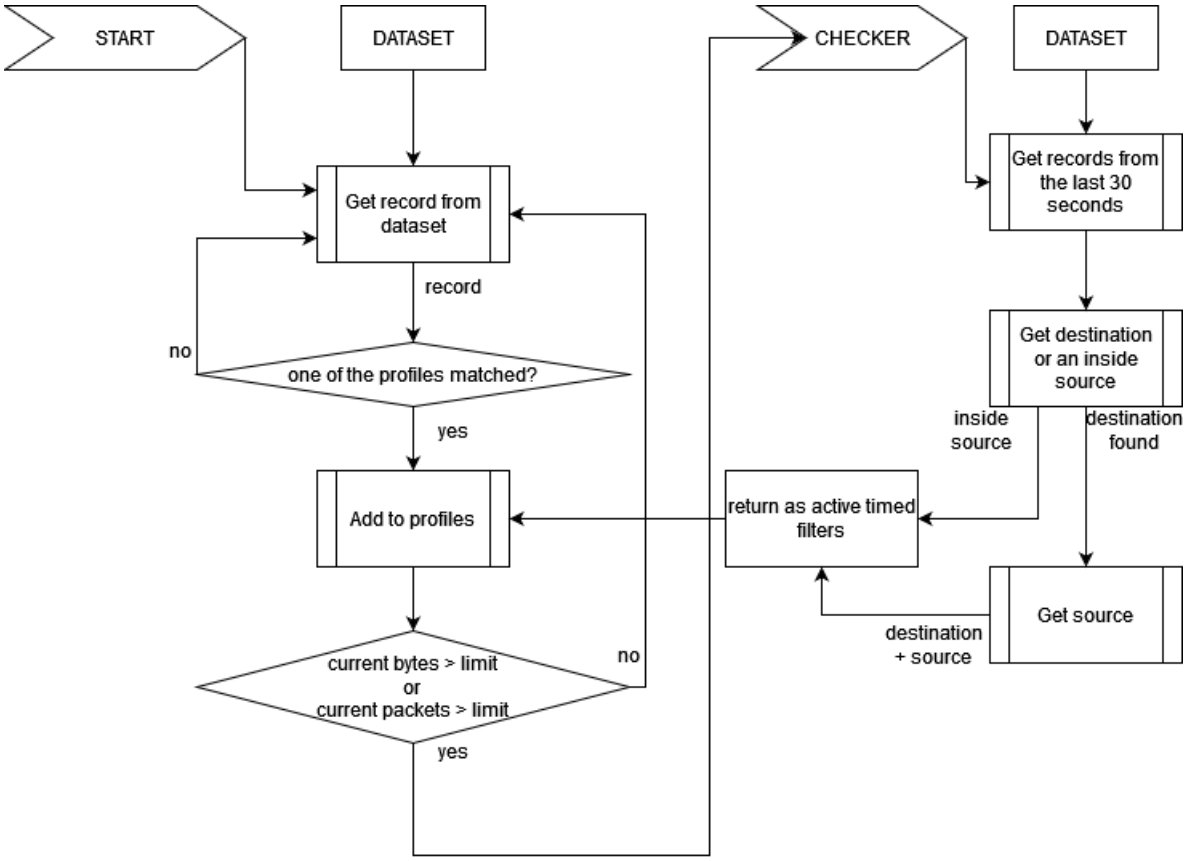

Figure 5.2: Workflow diagram

This section expands the information provided in the previous part. As shown in diagram 5.2, the threads are started (phase Start), and there is a dataset provided (`nfcapd` file). The writer reads the record, matches it against the profile filters and if one of the profiles is matched, the reader can read it from the ring buffer. If no limit has been crossed, it reads

another record. When one of the limits (whether for the bytes or the packets) is crossed, the checker thread is started.

When the checker is started, it first gets the data from the last thirty seconds that do not match any of the active filters from the previous checker runs, or as long as the active timed filter has not already been active when the data came. The records from those last thirty seconds are then stored in a dump file, from where it is fairly easy to read them whenever it is necessary.

When the algorithm has the data, the detection process can start. At first, the application tries to find the destination or destinations which are under the attack, and in case there are none, it tries to find a source address that could be responsible for the limit crossing. If the source is found, it is returned to the reader in a form of activated timed filter. On the other hand, if the destination is found, the algorithm continues and tries to find the attacker or group of attackers that is attacking the found destination. Once the source or sources are known, they are returned to the reader in the form of active timed filter.

5.2.1 Aggregation Levels

Before getting to the detection process itself, it is important to mention the levels on which the result might be returned.

As both the source and the destination may not be just one single device with a single address but rather a subnet with multiple addresses, it is necessary to take this fact into consideration. For this reason there are a few aggregation levels on which the results can be found:

- level 0 - at this level, there is one or more devices that singlehandedly cross the defined limit, and thus the result rule is in the format of `src ip` *`addr`* `and src port` *`port`* `and dst ip` *`addr`* `and dst port` *`port`*
- level 1 - at level one, it is not possible to define the exact two addresses with their ports as the attack direction, therefore the format of the resulting rule(s) is either `dst ip` *`addr`* `and dst port` *`port`* `and proto` *`protocol`* in case of the destination search and `src ip` *`addr`* `and src port` *`port`* `and proto` *`protocol`* in case of the source search
- level 2 - second level works similarly as the first level, the only difference is that the protocol is no longer taken into consideration, resulting in rule in the format of `dst ip` *`addr`* `and dst port` *`port`* for the destination and `src ip` *`addr`* `and src port` *`port`*
- level 3 - level 3 drops the port from the address, leaving just the address of the device, which returns `dst ip` *`addr`* as the destination and `src ip` *`addr`* as the source
- level 4, 5, 6 - at these levels, no single source or destination has been found that would be exceeding the limit by itself, and the whole subnets have to be considered, thus the result changes to `dst net` *`addr/n`* or `src net` *`addr/n`*, with `n` being 24 for level 4, 16 for level 5, and 8 for level 6

If one cycle of the aggregation runs through, and nothing is found, the limit that is supposed to be crossed is lowered. After the limit is lowered four times, and no result has been found, no filtration rule is returned as there is no attack found.

The number of filtration rules depends on two factors. One is the full amount of found possibilities, the other is the limit lowering. The following table, Table 5.1, shows how many results will be returned:

Limit Lowered	Results Returned
0	top 10
1	top 8
2	top 6
3	top 4
4	top 2

Table 5.1: Limits lowered vs. returned filters

Of course, not more than the full amount of found data can be returned.

5.2.2 Baseline

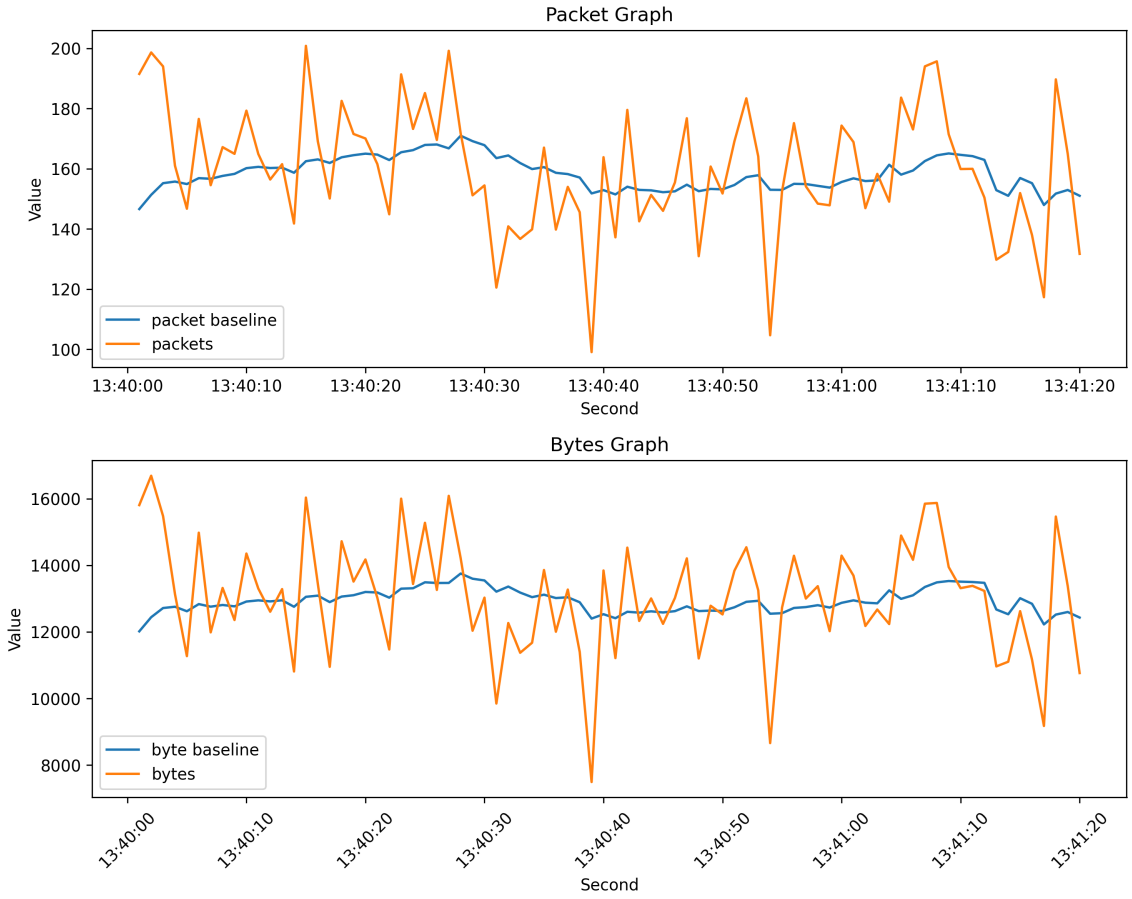

Figure 5.3: Baseline

Another important part of the detection process is the baseline. If no tresholds are given by the administrator, the detection relies on the baseline of the incoming values.

The baseline could be understood as a moving average computed on the incoming packets and bytes by the formula

$$newbsln = (oldbsln + totalTraffic * COEF)/(1 + COEF),$$

where `COEF` is the weighting coefficient of the last value, `totalTraffic` is the actual incoming traffic and `oldbsln` is the old baseline. An example of the baseline is depicted on the Figure 5.3, where the yellow line symbolizes the actual traffic and the blue line symbolizes the baseline that has been computed from the incoming data.

The baseline is counted every second of the program run, for every defined profile, for both packet and byte numbers belonging to the profile. As the baseline serves as the moving average, it is often being crossed by the packets and bytes, but that is not an indicative of a DDoS attack. For that reason, the packet or byte limit that is based on the baseline, a multiplication of the baseline has to be crossed. Several values have been tested on the program, but the best results were achieved with the multiplication of five, so that is the value that has been chosen for the application.

5.2.3 Detection Process

The algorithm described in Algorithm 1 aims to identify the destinations or sources that exceed a specific threshold. It focuses on records that have already been aggregated according to the level described in Section 5.2.1 and individually exceed the limit. The algorithm returns the top x records in a filtered format, where x is determined by the number of times the threshold has been reduced, as outlined in Table 5.1.

Algorithm 1: DETECT

Input: dataset, limits, level
Output: result array containing found sources or destinations

```
1: function DETECT()( )
2:     while record do
3:         get packets_per_second, bytes_per_second
4:         if pps > packet_limit || bps > byte_limit then:
5:             replace smallest in return array if it is bigger
6:         end if
7:     end while
8:     return result_array
9: end function
```

In Algorithm 2, the detection process begins by attempting to locate the destination of a potential attack. If no destination is found, it then searches for an internal source. If a source is found, the results are immediately returned to the reader thread, and the checking process ends. However, if a destination is identified, the detection process continues by searching for the sources that are attacking the identified destination. Once one or more sources are found, a filtering rule is created based on the findings and returned to the reader.

In case of two or more addresses being the same, with the only difference in the port, they could be joined and only the address without port would be returned. In the sixth chapter, Chapter 6, there is a comparison of how the algorithm could work in both of the cases. For the final implementation, the second case was chosen as it is more convenient when an attacker uses just a few of the ports.

Algorithm 2: CHECKER

Input: profiles, sec_crossed

Output: timed filters representing the eliminating rules

```
1: function CHECKER()( )
2:         get packet_limit, byte_limit, second_downto, second_upto
3:         get data_from_last_30_secs
4:         init destinations, sources
5:         init found = 0, level = 0
6:         while not found do
7:             //first try to find destination
8:             aggregate_by_level(DST,level)
9:             found = detect(destinations)
10:            if not found then
11:                //destination not found, trying source
12:                aggregate_by_level(SRC,level)
13:                found = detect(sources)
14:                if not found then
15:                    level++
16:                    if level = max then
17:                        level = 0
18:                        limit–
19:                    end if
20:                end if
21:            end if
22:        end while
23:        if found sources then
24:            return make_filters_from(sources)
25:        end if
26:        //destination(s) found, searching for source(s)
27:        while not found do
28:            aggregate_by_level(SRC,level)
29:            found = detect(sources)
30:            if not found then
31:                level++
32:                if level = max then
33:                    level = 0
34:                    limit–
35:                end if
36:            end if
37:        end while
38:        if found sources then
39:            return make_filters_from(sources)
40:        end if
41: end function
```

5.2.4 Parameters Affecting the Detection and its Weaknesses

There are several parameters that affect the accuracy of the detection. The first and one that has the most significant impact is the time between the checks of the limit crossing. For example, the filter might be active for 60 seconds. Still, sometimes, another examination was necessary in a shorter period of time, as the first check where the previous filter was created might not have enough data yet to find another attacker that might be present in the traffic.

Another parameter that turned out to have an effect on the detection was the multiplication of the baseline. The ideal value may vary between the profiles. For some profiles, especially higher traffic profiles, better values proved to be lower multiplications, such as the multiplication by three. But generally, in most cases, multiplication by five was low enough to get the attack detected and high enough to leave out most of the legitimate traffic.

One possible weakness of the algorithm is that if the attacker is very well hidden in the legitimate traffic (therefore not crossing any limits by himself), he will only be discovered once some of the higher limit crossings happen. In this situation, quite a few false positives could be returned (and false negatives in case the attacker goes undetected), but such a thing has not occurred during the testing.

Lastly, as the application offers the possibility of using a limit on both packets and bytes based on the baseline, a clever attacker could start to slowly raise the traffic levels, which could lead to a rise in the baseline and the limit based on it, resulting in an undetected attack.

5.3 Program Structure

The implemented project has the following structure:

```
dp/
├── tests/
├── checker.c
├── config.c
├── config.conf
├── config.h
├── default.h
├── graphs.py
├── main.c
├── Makefile
├── sqldb.c
├── sqldb.h
├── threads.h
└── writer.c
```

In the subdirectory `tests/`, there are several configuration tests. The project's compilation is handled by the `Makefile`. The example configuration can be found in the file `config.conf`, while the source code of the project is represented by the `.c` and `.h` files.

To execute the tests, use the command:

```
make tests
```

In the `main.c` file, there is the main function running the program, starting the main threads and taking care of signal handling. The files `config.*` provide configuration structures, function that parses the configuration and all the other functions that have anything to do with the configuration itself. The database connection, function and structures are handled in the `sqldb.*` files.

`threads.h`, `writer.c`, `reader.c` and `checker.c` all provide the threads environment. While the `threads.h` contains the function declarations and structures used throughout all the thread `*.c` files, the rest provides the definitions to the functions, each containing its own functions.

Last but not least, there is a simple python script used for graph plotting to visualize the results from the database.

5.3.1 Compilation

The program is compiled using the `gcc` compiler. This compiler is chosen for its user-friendly nature and seamless integration with debuggers like `gdb` and profilers such as `valgrind`.

To compile the code, run the command:

```
make dip
```

The linker command that is used for the compilation is the following:

```
gcc -g -o dip config.o writer.o reader.o main.o -lpthread -lnf
```

5.3.2 Running the Program

To execute the program normally, use the command:

```
./dip configfile
```

If you want the program to run in the background, you can use the command:

```
nohup ./dip configfile &
```

or a similar alternative. It also takes a `-h` argument, which prints help.

5.4 Input, Output and Requirements

To be able to execute the program, there are a few requirements that have to be installed. First is the `libnf` library, mentioned in 4.2 - as the whole program uses it for its basic functioning, it is not possible to run or even compile the program without it.

Another requirement is MySQL installed on the system, since the program uses it to store the data and the resulting filters. Without MySQL, there will not be a possibility to see the results.

The only input of the program is the configuration file that is described in the next section 5.4.1. The output can be found, as was mentioned above, in the defined MySQL database, and saved as `.png` images by running a python script `graphs.py`.

5.4.1 Configuration File

The configuration takes a file in a JSON-like format, commonly used for representing structured data. It follows a key-value pair format, where each key is followed by a colon and its corresponding value. The configuration is organized into a hierarchical structure, with nested objects indicated by curly braces ({}) and their properties listed within.

The style adheres to basic grammar rules, including the use of double quotes for enclosing string values, colons to separate keys and values, and commas to separate individual settings.

The configuration has two main types of settings:

- `general` - general settings, all of them have defined default values in case of them missing, but it is highly recommended to include at least the database settings as those should not be used with the default values, only one object of this type is allowed
- `rule` - profile settings, at least one profile should be defined, with two required fields - rule name (profile name) and the filter, there can be multiple objects of this type, each representing one profile

Configuration Example

```
general: {
    database: {
      name: "xploci00_db",
      user: "xploci00",
      password: "Jablko1.",
      host: "localhost"
    },
    input_file: "/data/test/13/nfcapd.202305091320",
    shm: "shm",
    verbose: true,
    recs_per_second: 1200
}

rule: {
    name: "rule1",
    filter: "proto 17"
}
```

Figure 5.4: Configuration Example

General Settings

General settings have the following properties:

- `database` - specifies the parameters necessary for the database connection, which are the following:
  - `name` - the name of the database that is going to be used (default `mysql_db`)
  - `user` - the user used for database connection (default `db_user`)
  - `password` - password that is going to be used to connect to the database (default `db_password`)

  - `host` - specifies the host address of the database (default `localhost`)

- `input-file` - the input `nfcapd` file where the records are or are being stored (default `/tmp/test_data`)
- `shm` - shared memory (default `shm`)
- `verbose` - whether the output should be verbose, and therefore log some additional information (default `true`)
- `recs_per_second` - how many records should be processed per second. When reading from the past files, it is recommended to set this value to a higher number, so that it doesn't become too slow, but it is neccessary to set it to a number - otherwise the program reads the data too quickly which can lead to inconsistencies. On the other hand, when reading data in the real-time scenario, it is neccessary to set it to zero, because otherwise it slows the program, as described in Chapter 6

Rule Settings

As for the rule, from now on called profile in this context, there must be at least one defined. It is used to filter the record to only process the data belonging to at least one of the profiles. The possible properties are the following:

- `name` - the name of the rule is used to create a dump file in case of a treshold crossing, more rules can have the same name but it is not recommended as it could cause some problems in case of a few profiles registering a possible attack simoultaneously, this is required property
- `filter` - the filter representing the profile, it should be written in the `nfdump` filter format and just as the name it is also a required property
- `limit` - a limit that can be defined by the administrator, either specifying maximum number of packets per second or bytes per second (not both), it can be omitted as there is also a baseline computed and in case of a missing filter its value is counted as the limit (baseline multiplied by 5)

5.4.2 Database

MySQL database was chosen to store the found data. There are two tables for this purpose. The first table, which is called "profiles", stores the profiles themselves and all the important values that they contain. The columns in the database are the following:

- `prof` - the column representing the profile (filter) as the table contains data from all of the profiles, it is part of the primary key for the table
- `sec` - the second part of the primary key, this column represents a second from the time in which the data has been coming, the format is in epoch time seconds so it has to be converted into human readable format when being extracted from the database
- `limp` and `limb` - limits for packets and bytes, respectively, either containing a value from the configuration or current baseline multiplied by 5 which works as the limit for the packets/bytes

- `fullp` and `fullb` - the total amount of packets and total amount of bytes that came in the given second of the time
- `filtp` and `filtb` - the total amount of packets and total amount of bytes that have been filtered by one or more of the active timed filters
- `restp` and `restb` - the amount of packets and the amount of bytes which have not been filtered by any of the active timed filters (allowed traffic)

Another table, where the filters are stored, is called "filters". There are only four columns:

- `filt` - the result filter that has been found
- `sec` - the second where limit exceeding was detected, similarly as in the `profiles` table, it is also in the format of a second in epoch time and therefore needs to be converted to a human readable format
- `filtp` - the amount of packets that the filter has filtered
- `filtb` - the amount of bytes that the filter has filtered

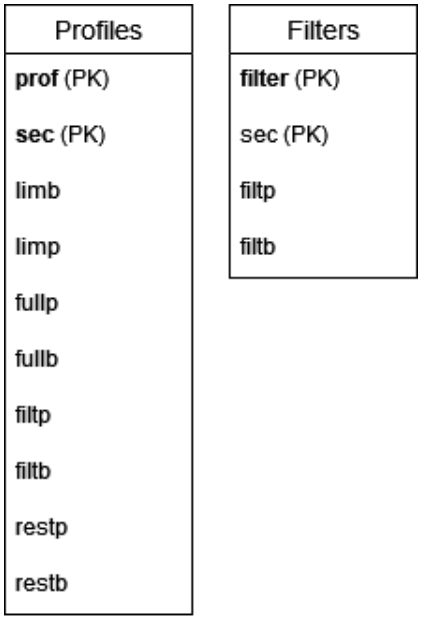

Figure 5.5: Database ER diagram

The Figure 5.5 above displays the Entity-Relationship Diagram for the database. The entities are not connected at all, as the filters are active across all of the defined profiles.

5.4.3 Program Output

As mentioned before, the output of the program is stored mainly in the database, as the result is actually the filters shown in the `filters` table.

It is possible to get a visual representation from the first table, `profiles`, which shows the progression of the traffic before and after some of the filters have been activated. For

this occasion, `graphs.py` script has been added to the project, although it serves just for the acute visualisation and its use is optional if other visualisations are available.

5.5 Application Threads, Structures and Functions

This subsection provides a closer look at the implementation itself and its most important functions and structures. Reflecting back on the earlier remark from the beginning of this chapter 5, the program starts two threads in the beginning of its run, the writer and the reader thread. But there are more functions run before they can be started.

`ctrlC(int c)`

This function serves as a signal handler for the SIGINT signal, which is typically generated by pressing Ctrl+C on the keyboard. It accepts `c`, an integer parameter that represents the signal value. When the SIGINT signal is received, this function is called to perform necessary memory freeing operations.

`void *writer_thread(void *params)` and `void *reader_thread(void *params)`

This function serves as the handler for the writer (reader, respectively) thread's activity. It takes `params` as an argument, which is a pointer to a structure that holds the required parameters for the threads's activity. This function initializes the necessary parameters and passes them to the main function of the thread.

`void set_params(parameters *ps, char *in, char *out, char *shm, int rec, int verbose, int timer, rule* filters, int cnt)`

This function populates the parameters structure with defined values obtained either from default settings or from a configuration file. The parameters structure `ps` is necessary for the threads.

5.5.1 Configuration Functions and Structures

There are a few structures neccessary for the configuration:

- `rule` - a structure representing one rule in a list of rules (of the defined profile), it contains the information like the filter itself, its name, its limit and a pointer to the next rule in a list

```
typedef struct s_rule {
    struct s_rule *first;
    char *name;
    char *filter;
    char *limit;
    lnf_filter_t *fil;
    struct s_rule *next;
} rule;
```

Figure 5.6: Rule structure

- `database` - a structure holding the database values, specifically database name, user, password, host and the port that should be used

```
typedef struct s_database {
    char *name;
    char *user;
    char *password;
    char *host;
    int port;
    int reuse;
} database;
```

Figure 5.7: Database structure

- `configuration` - a structure representing the configuration itself that will be filled with values when the configuration gets parsed, containing the values described in Section 5.4.1

```
typedef struct s_config {
    char *inFile;
    char *outFile;
    char *shm;
    database database;
    int verbose;
    int timer;
    int ruleCnt;
    int recs_per_second;
    rule *rules;
} configuration;
```

Figure 5.8: Configuration structure

The important functions from the configuration include the following:

`free_rules(rule *rules)`

This function frees all the rules in the provided list. The function takes `rules` as a parameter, which represents the list of rules to be freed.

`free_general(char *inf, char *shm)`

This is a function that frees the general settings that have been allocated. The function takes `inf`, and `shm` as parameters, representing the input file and shared memory, respectively.

`free_database(database *d)`

It is a function that frees the database settings that have been allocated. The function takes `d` as a parameter, which is a pointer to the database structure that needs to be freed.

`error(char *error_message, rule *rules, char *buffer, configuration *config, database *database, char *token)`

It is a function that handles the error state of the configuration parsing. It frees everything that has been allocated. The function takes `*error_message` as the error message that will be printed to stderr, and the rest of the values represent possibly already allocated variables that should be freed. It returns NULL, which is then passed to the main function.

`get_configuration(char *configFile)`

It is the main function that parses the configuration file into a defined struct that will be used for the rest of the program run. Firstly, it gets the configuration file and reads it into a buffer, creating a string. It then parses the configuration file into the defined structures, printing warnings and errors if something is not written correctly. Most of the keys have default values that can be found in the default.h file, just in case the values are not defined in the configuration file. The function takes `configFile` as the file from which the configuration will be read. It returns NULL on error and the configuration on success.

5.5.2 Writer, Reader and Checker Functions and Structures

There are several structures used across both of the threads:

- `parameters` - the structure representing the parameters that are passed to the reader and the writer thread, e. g. the name of the input file from which the writer is reading, the shared memory, number of records that can be read per second, whether the logging should be verbose, the number of rules and the list of rules that will now become "profiles"

```
typedef struct s_parameters {
    char *in_filename;
    char *out_filename;
    char *shm;
    int recs_per_sec;
    int verbose;
    int timer;
    int rulecnt;
    rule *filters;
} parameters;
```

Figure 5.9: parameters structure

- `limits` - a structure that represents the limits, containing a flag deciding what limit should be used (defined vs. baseline), values for both packet and byte limit which are filled with the value defined by the administrator or later by baseline value, and the baseline values for both packets and bytes

```c
typedef struct s_limits {
    int which;
    double bytes;
    double packets;
    double baseline_b;
    double baseline_p;
} limits;
```

Figure 5.10: Limits structure

- `t_records` - this structure represents the storage of the records once they have been processed by the reader, it contains a static array of `lnf_brec1_t` and a pointer to the next structure of the same type in case of more records needing to be stored that the size of the static array

```c
typedef struct s_records {
    lnf_brec1_t records[1000];
    struct s_records *next;
} t_records;
```

Figure 5.11: t_records structure

- `valTimes` - structure representing a single second in time, containing the said second's identificator, the (full) amount of packets and bytes received in the profile during that second (computed from the records), the amount of packets or bytes without the filtrated data (before any timed filters are active this value should be the same as the full amount of packets/bytes), the limits defined for this second, index of the last record important for previous structure, values of the lowest second for the records stored in the second's storage as well as highest second where this seconds records can be found, a flag saying whether this second has already triggered attack checking and a storage of the records (structure `t_records`), with the records ending in this second being stored there. The index is found by doing `sec % 60`.

```c
typedef struct s_valTimes {
    uint64_t second;
    double pps;
    double f_pps;
    double bps;
    double f_bps;
    limits lims;
    int last;
    int upto;
    int downto;
    int checked;
    t_records *list;
} valTimes;
```

Figure 5.12: ValTimes structure

This structure is kept in the `profile_rec` structure in an array of 60, covering one minute of time.

idx	second	pps	f_pps	bps	f_bps	...
0						
1						
...						
59						

Table 5.2: Table representing a single `valTimes` record

- `t_timedFilter` - structure representing a timed filter, including the filter itself, second when the filterwas created, values counting how many bytes and packets has the filter filtered and a pointer to the next timed filter in the list

```
typedef struct s_timedFilter {
    lnf_filter_t *timed_filter;
    char *filter;
    uint64_t second;
    double filtered_p;
    double filtered_b;
    struct s_timedFilter *next;
} t_timedFilter;
```

Figure 5.13: t_timedFilter structure

- `profile_rec` - the most important structure holding all of the profile's information: filter defining the profile, the numbers of bytes and packets, the dump file for the checker, a mutex for values manipulation, a flag saying whether the profile is already in the process of checking, its own checker thread that can be run when one of the limits has been exceeded, the pointer to the next profile in the list and an array of sixty `valTimes` representing a minute in time

```
typedef struct profile_rec_s {
    char *filter;
    lnf_filter_t *fil;
    t_timedFilter *timedFilter;
    uint64_t check_from;
    int count;
    double last_p_bsln;
    double last_b_bsln;
    double old_p;
    double old_b;
    char* treshold;
    char* dump_file;
    valTimes values[60];
    int checking;
    uint64_t last_checked;
    pthread_t tChecker;
    pthread_mutex_t mutex;
    struct profile_rec_s *next;
} profile_rec;
```

Figure 5.14: Profile structure

- `t_checker_vals` - a structure that represents values that are going to be passed to the checker thread, including an information about which second has crossed the limit,

the pointer to the profile where the checker belongs and a pointer to the list of all profiles

```
typedef struct s_checker_vals {
    int secondCrossed;
    profile_rec *profile;
    profile_rec *profiles;
} t_checker_vals;
```

Figure 5.15: t_checker_vals structure

The important functions regarding the threads are these:

`free_records(t_records **list)` and `free_all_records(t_records **list)`

These functions are used only in the reader. The first one is used for cleaning the records from the second when the second is about to be "rotated", therefore the one of these records can stay and does not have to be allocated again. The second function deletes all of the records.

`checker(t_checker_vals *ps)`, `writer(parameters params)`, `reader(parameters params)`

The main functions of the threads directing their whole activity, with the reader and the writer running in an endless loop and the checker running once in a while when the detection needs to be performed.

`update_baseline(profile_rec** profiles)`

It is a function that updates the baseline for both the packets and bytes for each second in each profile. The function takes `profiles` as the list of all profiles that will have their baselines updated.

`end_thread(int type, int kill_others, int free_rest)`

It is a function that ends the threads and releases their resources. The function takes `type` as the type of the thread, `kill_others` as a flag indicating whether it is supposed to kill the rest of the threads, and `free_rest` as a flag indicating whether to free the remaining resources.

`set_aggregation_function(lnf_mem_t *memp, int level, int agg)`

This function sets the aggregation rules for the memory `memp` according to the level and the flag saying whether it is supposed to be doing the aggregation on the sources or the destinations.

`get_data(char *filename, valTimes values[60], uint64_t upto, uint64_t downto)`

In this function, the data from the `values` array are stored into the dump file if their times are between `downto` and `upto` values.

`detect_src(lnf_mem_t *memp, double packets, double bytes, char srcs[10][120], int level, int count)` and `detect_dst(lnf_mem_t *memp, double packets, double bytes, char dsts[10][120], int level, int count)`

These function represent the algorithm described in the Algorithm 1. They are trying to find the destination or the source of the possible attack.

`subtract_xps(valTimes *values, int secondCrossed, t_timedFilter *filter)`

Once the target and the source are found, this function is called to subtract the values of packets and bytes per second from the seconds in `values` where the newly created filter was already supposed to be active and therefore filtrating the traffic. This is were the first differences start between previously mentioned full amount of packets/bytes vs. the filered values.

`create_timed_filter(char *filter, uint64_t second)`

The timed filter is created in this function.

Chapter 6

Application Testing

Several tests have been done on the system, whether being run over prolonged periods of time as well as on singular or multiple past files. This chapter describes how the testing went, what requires optimizations, where are the program's weaknesses and what the program does correctly.

6.1 Detection Testing

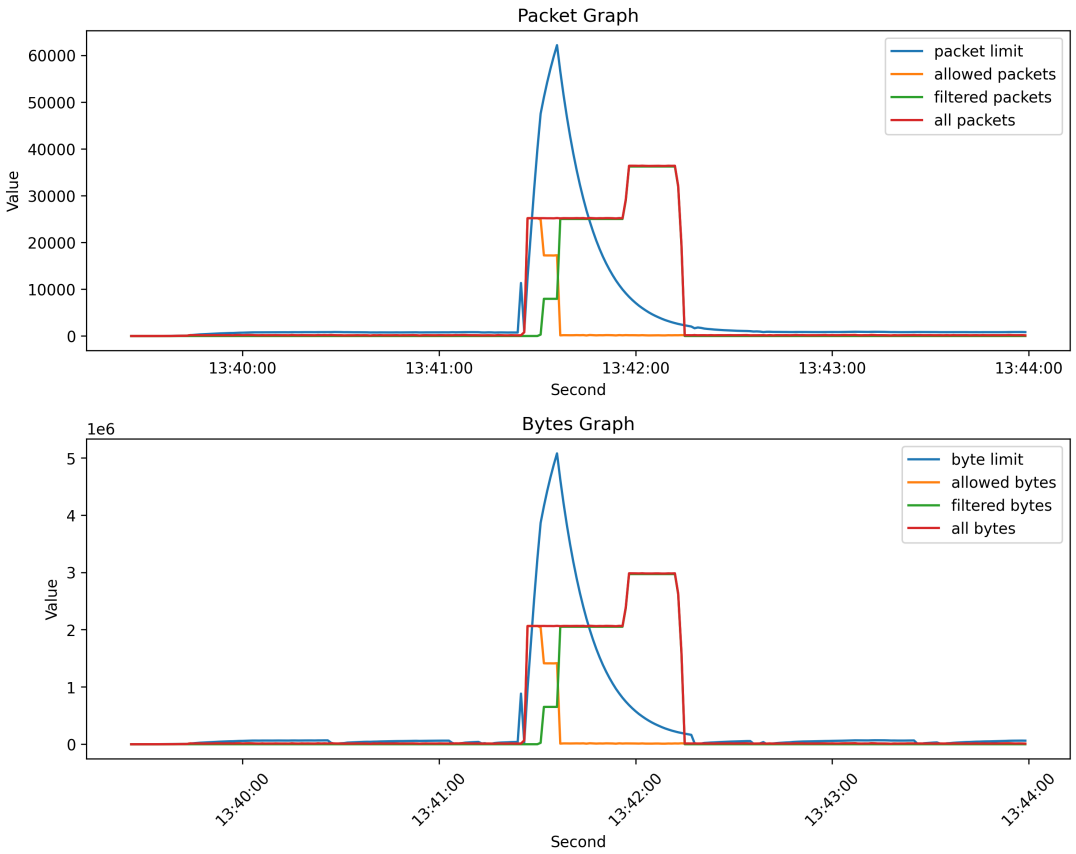

Figure 6.1: Graph from the attack data, attack registered on profile defined by `port 123`, both limits based on baseline multiplied by 5.

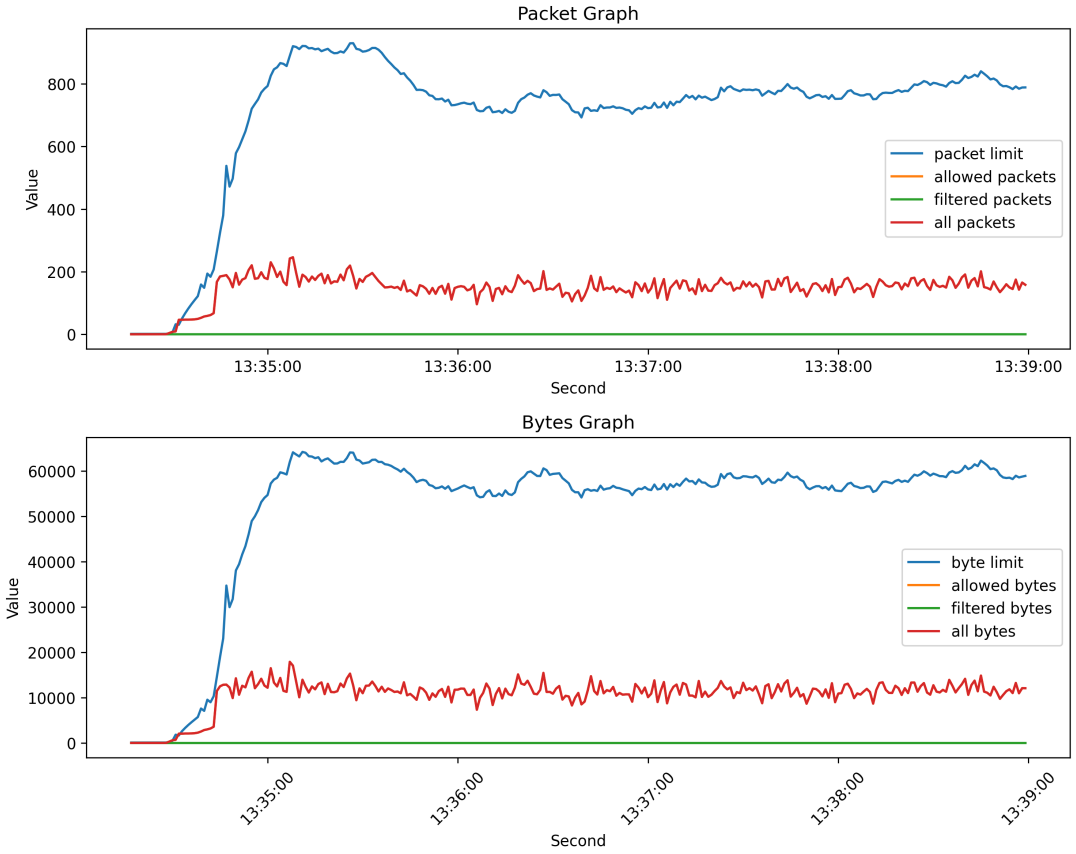

Figure 6.2: Graph from the data without an attack, profile `port 123`, both limits based on baseline multiplied by 5.

The most important question when it comes to the testing of the program was to find out whether it is able to find an attack when one occurs. This section covers exactly that - whether it is able to detect an attack, what multiplication of baseline gets the best results and how does the profile definition affect the attack detection, as all of the values play a role in the success and size of the filtering rules that are created.

The behaviour of the system and the incoming data and changing limits during an attack can be seen on Figure 6.1. The attack came to life between 13:41 and 13:42 of the local time, and the checker thread was activated when the red line (at that time yellow line was still equal to the red line) crossed the blue line. As the blue line defines the limit based on the baseline (note that it is not the baseline itself) which takes incoming traffic as one of its variables, it spikes quite high, but drops as soon as the filter takes the action and filters out the malicious traffic. The allowed traffic is then depicted by the yellow line and the green line shows the amount of blocked traffic. Please note that the graph shows the data in each of the seconds, not cumulative results.

In the Figure 6.2, the same profile's graph from the data without an attack is depicted. This graph was generated from the data minutes before the attack happened that had both limits based on baseline multiplied by 5. The crossing might have happened at the beggining of the run, but the first few seconds after the start of the program cannot be taken into consideration as the data is not in consistent state yet.

The eliminating filtering rule, shown in Figure 6.3, has been inserted into the database upon the attack, and thus defines the output of the program, as mentioned in 5.4.

```
mysql> select * from filters;
+----------------------+------------+---------+----------+
| filt                 | sec        | filtp   | filtb    |
+----------------------+------------+---------+----------+
| src net 100.91.0.0/16 | 1683632496 | 1163871 | 95437422 |
+----------------------+------------+---------+----------+
```

Figure 6.3: Filter found in the database that stopped the malicious traffic.

The testing shows that with the lower traffic profiles, such as `port 123` profile, the program is able to detect the malicious traffic fairly easily, as well as distinguish it from the regular traffic. However, the accuracy drops a bit when higher traffic profile comes to an action. With those, a few false positives have been found, but the attack remains detected still.

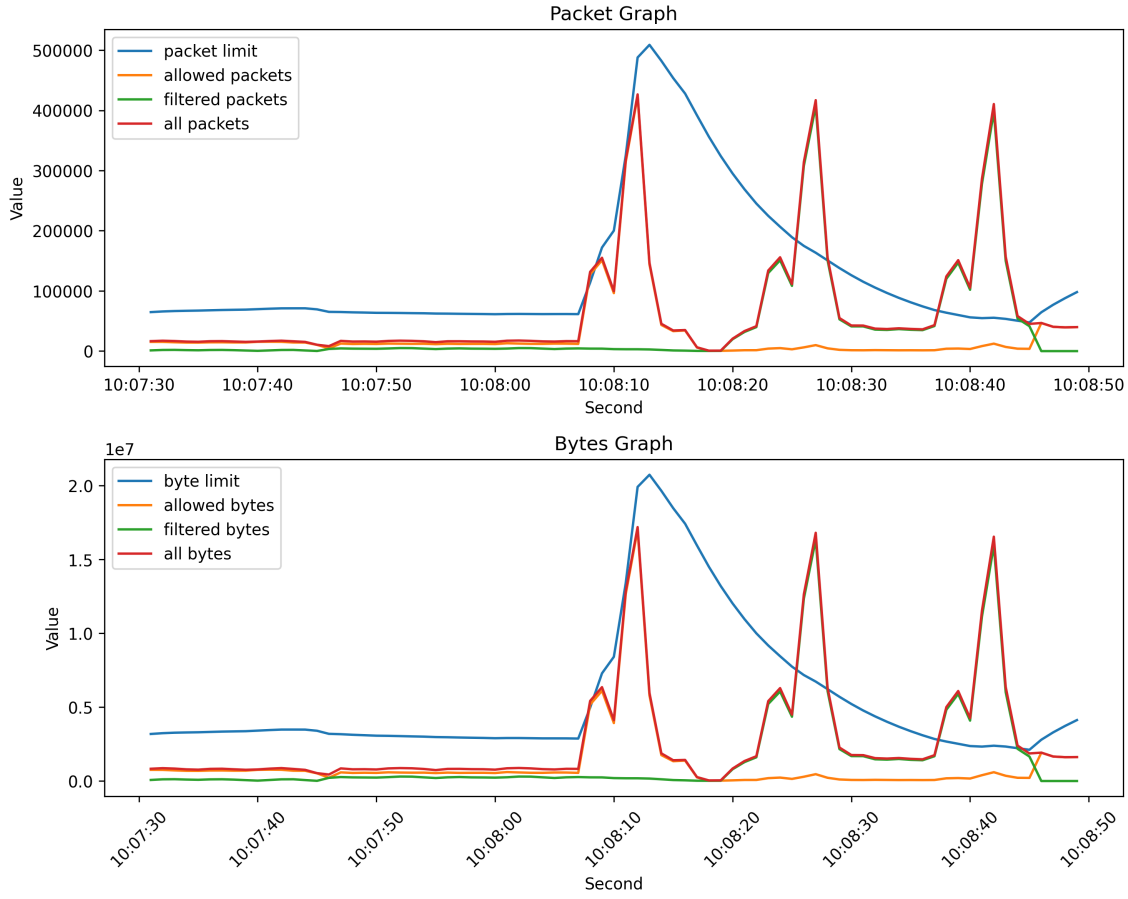

Figure 6.4: Graph of an atttack, `flags S and not flags AFRPU` profile, limits bsln * 5

In the Figure 6.4, there are the data recorded on the profile `flags S and not flags AFRPU`. As can be seen, the attack came in repetitive patterns. The first part of the attack has not been immediately filtered, but the green line, symbolizing filtered traffic, starts to dissapear under the red line - meaning that the malicious traffic is being filtered. However,

not all of the traffic has been matched, therefore at least part of the legitimate traffic could still continue, as visible from the yellow line symbolizing allowed traffic.

As mentioned before, there is a possibility of a false possitive values. In the Figure 6.5, apart from the real attacker in the second line, there is also the first line, which could possibly mean either another attacker, a false positive result, or it can be just the targetted system that was trying to communicate with the attacker and thus became a suspect itself.

```
mysql> select * from filters;
+---------------------+------------+---------------+----------------+
| filt                | sec        | filtp         | filtb          |
+---------------------+------------+---------------+----------------+
| src ip 147.229.8.240 | 1683965265 | 202089.411905 | 12125364.71429 |
| src ip 89.248.163.59 | 1683965296 |       2958071 |      118322840 |
+---------------------+------------+---------------+----------------+
```

Figure 6.5: Filters found in the database that stopped the malicious traffic.

The result above was created with the address joining. When two attacking addresses were detected, they were joined and connected to filter out all of the traffic from the found source address, without consideration of the port. The program was also tested on unjoined addresses and ports, with the filter results shown below on Figure 6.6. Unfortunatelly, there was one more false positive result found.

```
mysql> select * from filters;
+-------------------------------------------------+------------+---------------+----------------+
| filt                                            | sec        | filtp         | filtb          |
+-------------------------------------------------+------------+---------------+----------------+
| src ip 147.229.8.240                            | 1683965265 | 149569.297619 | 8974157.857146 |
| src ip 185.144.201.90                           | 1683965291 |          4237 |         254220 |
| src ip 89.248.163.59 and src port 45441 and proto 6 | 1683965302 |        398881 |       15955240 |
| src ip 89.248.163.59 and src port 45444 and proto 6 | 1683965302 |       1700879 |       68035160 |
+-------------------------------------------------+------------+---------------+----------------+
```

Figure 6.6: Filters found in the database that stopped the malicious traffic.

To sum up, the attacks that should have been found were found, and with no attacks only a minimum false positives has been found. Unfortunately, there is a way of getting even more false positives, as will be stated in the next subsection.

6.2 Performance Testing

During the testing, several optimizations needed to be done over the code. Generally, the code worked fast, managing to almost match one second of the computation time vs. real second time from the data (one second of all of the data from the `nfcapd` file that correspond with the given second), as long as the given filters were not covering most of the incoming traffic (e. g. `port 443`, `proto 6`). With the higher traffic profiles, the program still worked fast enough, but not nearly as fast as would be ideal speed to make it usable enough in "real-time" situations.

The speed problem lead to inconsistencies in the data that were being analyzed. It is easily visible in the Figure 6.7, where there are regular gaps in the data.

The gaps between data caused the droppings in the limit as well, which would sometimes lead to quite many false positives, as can be seen in the graph with tiny waverings on the green line.

As it turned out, the speed problem was caused by the records per second constraint, which was implemented by using the function `nanosleep`.

It seems that the nanosleep function was blocking the system much more than expected, thus causing huge delays in data reading, leading to the data loss described above. After the removal of the function, the program worked fast enough to be available for real-time use. The change in the graph can be seen in 6.8, where it is depicted with expected appearance.

The detection process proved to be fast to find everything that there is to be found, despite numerous aggregations sometimes having to be done, but the analysis shows that the aggregations can be hard on the system when there is just simply too much traffic to process.

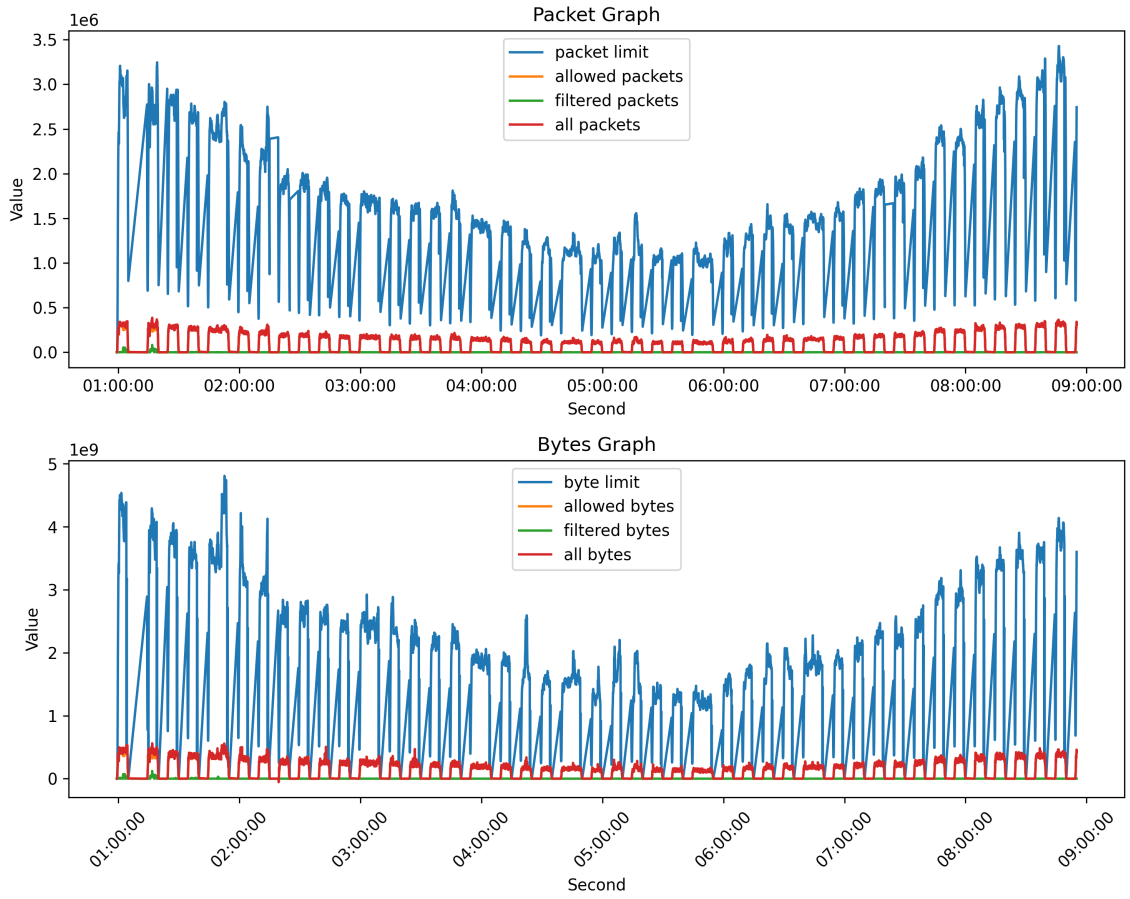

Figure 6.7: Graph with regular gaps, `port 443` profile, limits bsln * 10, part of system run

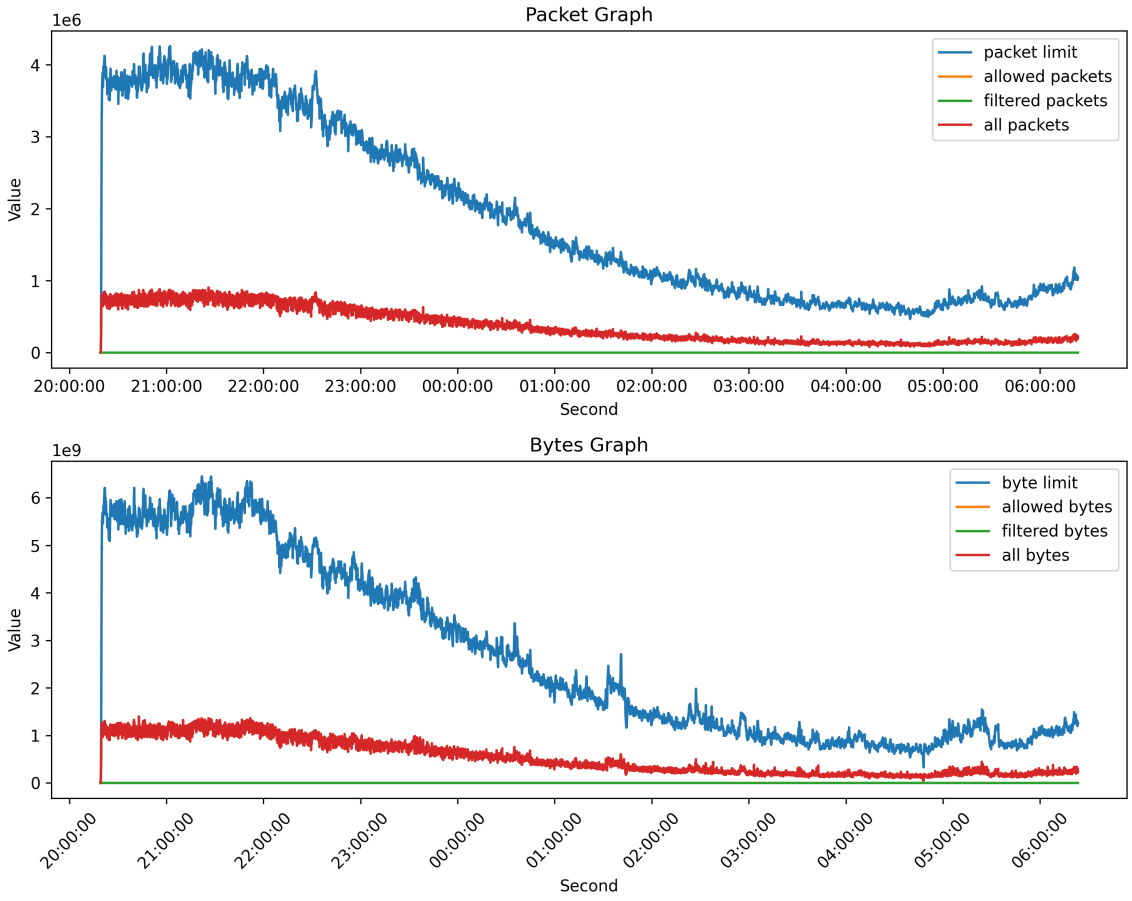

Figure 6.8: Graph with stable data, `port 443` profile, limits bsln * 5, part of system run

Chapter 7

Conclusion

The goal of this work was to design a system that would be able to detect an ongoing DDoS attack and create filtering rules to stop it. Upon getting the input definition of chosen traffic profiles from the network administrator, it was supposed to examine the incoming traffic. Then, in the case of an eventual DDoS attack from one or more defined profiles, it would create a filtering rule to eliminate traffic belonging to the said profile (or profiles, respectively).

To be able to design this system, it was necessary to study the NetFlow protocol more closely, the data formats, the storage of the data, as well as different types of DDoS attacks. It was also essential to get to know the library that provided an effective way of working with the received data - `libnf` library.

After that, it was necessary to design the system, create the algorithm that would be able to find the ongoing attack, and implement the designed system.

The system has been successfully implemented according to the design, with the details regarding its design and implementation being available in Chapter 5. The application is able to monitor cumulative network traffic according to the defined profiles. With each coming record, the record is added to the profile data of the profile(s) where it belongs. At first, the records were stored in a dump file for each profile, but it turned out that it was not an ideal solution since it would be necessary to look through the whole file with each detection process, while the only data that is needed is the last 30 seconds of the limit crossing. The limit can be read from the configuration file upon definition from the administrator, or in case of the missing limit, the baseline that is constantly being counted is used in various multiplications of it serving as the limit.

Since the dump file solution was not as successful, the data is stored in a table of sixty rows, representing the last sixty seconds of the traffic. Due to the nature of incoming data, sometimes including flows from more than just one second, the values of bytes and packets per second are counted and split between the corresponding second. This solution turned out to be more successful in terms of storage and simplicity, but it could also be one of the factors slowing the program down.

When the given limit is crossed in one or more seconds, the detection process starts and is able to find one or more attackers, provided that there is an actual ongoing attack. At first, the destination of the attack is found, and then according to that information, one or more attackers are found, and a filtration rule is created. The results then can be found in the database - in the table called `profiles`, there are the data defining the progression of the traffic (with defined full traffic, filtered traffic, and allowed traffic), and in the table

`filters`, the created filters and information about how much data has been filtered by this filter.

Lastly, the implemented system was supposed to be tested in a provided testing environment, which was also successfully done.

Testing of the application showed that the program is able to detect the attacking addresses, if there are some, according to the defined parameters and settings. Furthermore, it is able to react to the successful search by creating a correct timed filter and eliminating rule that would be able to help to stop the incoming attack. Therefore, the application should be able to run in real time, at least according to the testing environment provided.

The problem is that while for most of the attacks and regular traffic analyzed, it is enough to put a limit as a baseline multiplication by 5, especially higher traffic profiles could miss an attack or cut a lot of the regular traffic when such baseline is set. However, this situation has occurred only once during the testing process and has never been repeated.

For future work, a possibility would be to extend the configuration and configurable parameters - such as baseline multiplication to each profile, the option to define both packet and byte limit as now only one of them can be specified by the administrator, the time how long the timed filters should be active after the detection, whether to join the addresses in filtering rules when a few of them are matching, etc.

Another extension could be to connect the application and a firewall (or firewalls in case there are more of them) so that the created rules could be inserted directly into the firewall rules and therefore take action right when an attack is detected.

In regards of database extension, for now, it is a simple database with two tables. For the future work, the filters could have definitions of the profiles that created them stored as well.

The filters could also work a bit differently. Now, each of the timed filters is active for a defined period of time and after that, it deactivates. For the future, the filters could have two timers - one total timer which would define the maximum period of time it could be active, and another one, an idle timer, which would define after what period of time of inactivity on the said filter it should deactivate.

Bibliography

[1] *Libnf Documentation* [online, 19.3.2019]. [cit. 2023-01-15]. Available at: http://libnf.net/doc/api/.

[2] AL Musawi, B. Q. M. Mitigating DoS/DDoS attacks using iptables. *International Journal of Engineering & Technology*. Citeseer. 2012, vol. 12, no. 3, p. 101–111.

[3] Barrett, B. How a Dorm Room Minecraft Scam Brought Down the Internet. *Wired*. Feb 2018, [cit. 2023-05-05]. Available at: https://www.wired.com/story/github-ddos-memcached/.

[4] Cisco. *NetFlow Configuration Guide, Cisco IOS Release 15M'I&'T* [online]. 2016 [cit. 2023-01-08]. Available at: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/netflow/configuration/15-mt/nf-15-mt-book.pdf.

[5] Cloudflare. *What is a DDoS attack?* [online]. [cit. 2023-01-08]. Available at: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

[6] Cloudflare. *Mantis - the most powerful botnet to date*. 2022 [cit. 2023-01-08]. Available at: https://blog.cloudflare.com/mantis-botnet/.

[7] Dzurenda, P., Martinasek, Z. and Malina, L. Network protection against DDoS attacks. *International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems*. 2015, vol. 4, no. 1, p. 8–14, [cit. 2023-05-05].

[8] Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R. et al. *Flow Monitoring Explained: From Packet Capture to Data Analysis with NetFlow and IPFIX* [online]. [cit. 2023-01-08]. Available at: https://is.muni.cz/publication/1181098/flow-monitoring-explained-paper.pdf.

[9] Imperva. *The Top 10 DDoS Attack Trends* [Imperva. Online document]. 2015 [cit. 2023-05-05]. Available at: https://www.imperva.com/docs/DS_Incapsula_The_Top_10_DDoS_Attack_Trends_ebook.pdf.

[10] Inc., B. *Internet Security Threat Report*. Broadcom Inc., 2017 [cit. 2023-05-05]. Available at: https://docs.broadcom.com/doc/istr-22-2017-en.

[11] Matoušek, P. *Síťové aplikace a jejich architektura*. VUTIUM, 2014. ISBN 978-80-214-3766-1.

[12] Mirkovic, J. and Reiher, P. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*. ACM New York, NY, USA. 2004, vol. 34, no. 2, p. 39–53, [cit. 2023-02-20].

[13] Petryschuk, S. *NetFlow Basics: An Introduction to Monitoring Network Traffic* [online, 19.3.2019]. [cit. 2023-01-08]. Available at: https://www.auvik.com/franklyit/blog/netflow-basics/.

[14] Sonar, K. and Upadhyay, H. A survey: DDOS attack on Internet of Things. *International Journal of Engineering Research and Development*. 2014, vol. 10, no. 11, p. 58–63.

[15] Walkowski, D. *What Is a Distributed Denial-of-Service Attack?* [online, 5.6.2019]. 2019 [cit. 2023-01-08]. Available at: https://www.f5.com/labs/learning-center/what-is-a-distributed-denial-of-service-attack.