BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DISTRIBUTED SYSTEM FOR SUPPRESSION OF DOS ATTACKS

DISTRIBUOVANÝ SYSTÉM POTLAČENÍ DOS ÚTOKŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DALIBOR BENEŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ŠIŠMIŠ**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

154952

Institut:	Department of Computer Systems (DCSY)
Student:	**Beneš Dalibor, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Distributed system for suppression of DoS attacks**
Category:	Networking
Academic year:	2023/24

Assignment:

1. Study Denial of Service (DoS) attacks, how to combat them, and learn about the DDoS Protector (network traffic filtering) tool.
2. Study network monitoring capabilities with network probes providing IP flow data using the sFlow, IPFIXcol (network flow information collection), and Nemea (network flow data analysis) tools.
3. Design an approach for detecting a DoS attack and an appropriate control method to suppress them in a distributed environment.
4. Experimentally implement the selected components according to the design.
5. Demonstrate the properties of the created solution in a simulated environment.
6. Discuss the results achieved and the possibilities of further continuation of the work.

Literature:

- According to the instructions of the supervisor and consultant.

Requirements for the semestral defence:

- Fulfillment of goals 1 to 3 of the assignment.

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

Supervisor:	**Šišmiš Lukáš, Ing.**
Consultant:	Man Jakub, Ing. (CESNET)
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	24.5.2024
Approval date:	30.10.2023

Abstract

Protection against distributed denial of service (DDoS) attacks is one of the key areas of network security. One of the forms of defence is to use the DCPro DDoS Protector device developed by the CESNET association. The CESNET association also utilizes network monitoring and traffic analysis systems IPFIXcol2 and NEMEA, as well as the network monitoring protocol sFlow. The aim of this thesis was to propose and realize the integration of those systems, so that an effective distributed system for the mitigation of DDoS attacks could be created. During the work on this task, special focus was given to the effective utilization of existing solutions, reusability and the option for a future expansion of the distributed architecture.

Abstrakt

Ochrana před distribuovanými útoky odepření služby (DDoS) patří mezi klíčové oblastí síťové bezpečnosti. Jednou z možných forem ochrany je využití zařízení DCPro DDoS Protector vyvíjeného sdružením CESNET. Sdružení CESNET provozuje také systémy pro monitorování a analýzu síťového provozu IPFIXcol2 a NEMEA, a dále poskytuje možnost využít protokol pro monitorování sítě sFlow. Cílem této práce je navrhnout a uskutečnit integraci těchto systémů a vytvořit tak efektivní systém potlačení útoků odepření služby. Při vypracování tohoto cíle byl kladen důraz na efektivní využití stávajích řešení, znovupoužitelnost a možnosti budoucího rozšíření celé distribuované architektury.

Keywords

DCPro DDoS Protector, NEMEA, IPFIXcol, IPFIX, sFlow, NetFlow, CESNET, DoS, DDoS, network security, network data analysis, network monitoring, DDoS attack mitigation

Klíčová slova

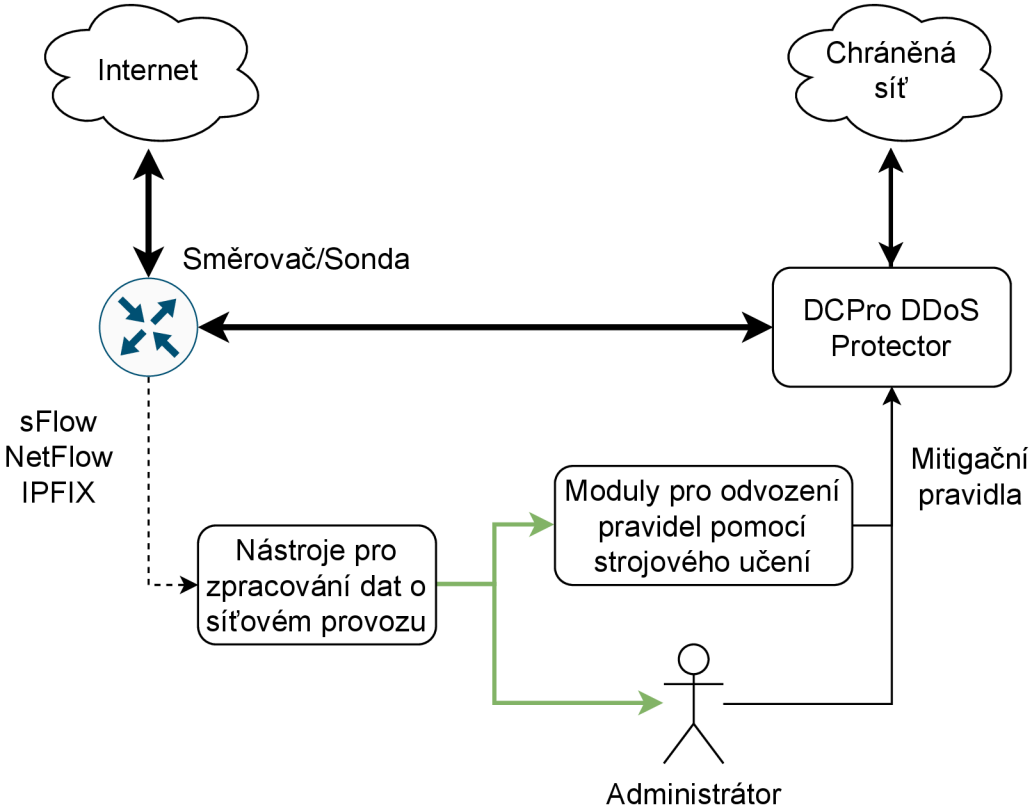
DCPro DDoS Protector, NEMEA, IPFIXcol, IPFIX, sFlow, NetFlow, CESNET, DoS, DDoS, síťová bezpečnost, analýza síťových dat, monitorování sítě, potlačení DDoS útoků

Reference

BENEŠ, Dalibor. *Distributed system for suppression of DoS attacks.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Šišmiš

Rozšířený abstrakt

Síťová bezpečnost je jednou ze základních oblastí, kterým je nutné věnovat zvýšenou pozornost při návrhu, vývoji a nasazení síťových služeb, mezi něž patří i informační systémy. V současnosti se s rozšířením systémů a aplikací, které spoléhají na přenos dat po síti, stává tato otázka o to více důležitou. Při ochraně systémů se bezpečnostní cíle typicky dělí do několika kategorií, mezi které patří integrita, důvěrnost a dostupnost. Jednou z největších hrozeb pro zajištění dostupnosti jsou v dnešní době útoky odepření služby (DoS) a jejich distribuovaná varianta (DDoS), která komplikuje detekci útoku a ochranu před ním využitím několika zdrojů útoku. Cílem těchto úkolů bývá přetížení cílového stroje nebo linky, aby se poskytovaná služba stala nedostupnou pro legitimní uživatele. Výpadek v dostupnosti služby může představovat nebezpečí zejména v oblasti kritické informační infrastruktury státu, kdy i relativně krátký výpadek důležité služby může mít za následek velký dopad.



Obrázek 1: Cílová Architektura Systému pro distribuované potlačení DDoS útoků

Ochrana proti DDoS útokům obvykle vyžaduje prvotní detekci útoku prostřednictvím technologií pro monitorování sítě. Jednou z těchto monitorovacích technologií je protokol sFlow, který slouží pro přeposílání vzorků paketů procházejících přes běžná síťová zařízení, jako jsou směrovače, nebo dedikované sondy. Další technologie zahrnují protokoly IPFIX a NetFlow. Oba tyto protokoly mohou být opět nasazené na směrovačích, které je podporují. Narozdíl od sFlow však nedochází k vzorkování jednotlivých paketů, namísto toho

jsou pakety agregovány do takzvaných *síťových toků*, které jsou identifikovány pomocí IP adres a čísel portů v paketech a obsahují agregované informace o celkovém počtu bytů a paketů, které prošly přes dané síťové zařízení v konkrétním časovém intervalu. V rámci sdružení CESNET jsou data o síťových tocích zpracovávána nejdříve kolektorem IPFIXcol, a následně modulárním systémem pro analýzu síťových toků NEMEA.

Po detekci útoku je možné útočné pakety zahodit nebo přesměrovat. K tomuto účelu slouží například směrovací protokol BGP. Použití směrovacího protokolu pro zabránění útoku nemusí být vždy efektivní řešení, protože v případě distribuovaného útoku typicky omezuje i přístup legitimních uživatelů ke službě. Z těchto důvodů je sdružením CESNET vyvíjen systém pro potlačení DDoS útoků aktivní filtrací paketů příchozích do chráněné sítě, který se nazývá DCPro DDoS Protector. DCPro DDoS Protector je z kapacitních důvodů a z důvodu zpomalení provozu obvykle nasazen tak, že v případě útoku je na něj na směrovačí přesměrován protokolem BGP podezřelý síťový provoz, ze kterého DCPro DDoS Protector následuje odfiltruje útočné pakety a legitimní navrátí přes směrovač zpět do chráněné sítě.

V současném zapojení systémů sice existuje systém pro kontinuální analýzu síťového provozu NEMEA a je možné konfigurovat směrovače pro odesílání monitorovacích zpráv pomocí protokolu sFlow, nicméně neexistuje žádné napojení na systém DCPro DDoS Protector. Cílem této práce, který je vizualizován v Obrázku 1, je navrhnou a implementovat prototyp rozhraní pro napojení systémů pro sběr dat o síťovém provozu a systému DCPro DDoS Protector tak, aby mohl administrátor systému DCPro DDoS Protector tento systém ručně konfigurovat formou takzvaných mitigačních pravidel, a napojení na moduly systému DCPro DDoS Protector, které dokáží mitigační pravidla odvozovat automaticky za využití metod strojového učení. Obrázek znázorňuje cílový stav, kde zelenou barvou je naznačeno nově vzniklé napojení zpracovaných dat získaných monitorovacimi protokoly sFlow, NetFlow a IPFIX na systém DCPro DDoS Protector. Díky integraci těchto systémů tak vzniká jeden distribuovaný systém pro potlačení DDoS útoků, který má přístup k monitorovacím datům sbíraných ze síťového provozu, a může tak efektivně potlačit příchozí útok za pomoci distribuovaného systému DCPro DDoS Protector v rámci relativně rychlé odezvy, s využitím automatizace ve formě odvození mitigačních pravidel technikami strojového učení.

Distributed system for suppression of DoS attacks

Declaration

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

. . . . . .
Dalibor Beneš
May 24, 2024

Acknowledgements

I would like to express my gratitude to my supervisor *Ing. Lukáš Šišmiš* for his enthusiastic support and great patience. Many thanks go to the people working at the CESNET DCPro DDoS Protector project, namely my technical advisor, *Ing. Jakub Man* and *Ing. Jan Kučera*. I would also like to thank my wife, family, and friends, who supported me when writing this thesis.

Contents

1 Introduction **2**

2 DDoS Detection and Monitoring **4**
2.1 Types of DDoS Attacks 4
2.2 Monitoring 6
2.3 Packet Capture Monitoring 7
2.4 Flow Monitoring 7
2.5 IPFIXcol2 11
2.6 NEMEA Framework 12
2.7 SecurityCloud 16

3 DDoS Protection and Mitigation **18**
3.1 Border Gateway Protocol (BGP) 18
3.2 DCPro DDoS Protector 19

4 Solution Design **31**
4.1 Current status 31
4.2 General Design Considerations 32
4.3 Data Format and Source Identification 33
4.4 sFlow Processor 38
4.5 DCPro NEMEA Daemon 43
4.6 Overview of the Proposed Changes 46

5 Solution Implementation **47**
5.1 Source Identifiction 47
5.2 DCPro sFlow Probe 50
5.3 sFlow Packet Parser 53
5.4 DCPro NEMEA Daemon 56
5.5 Experimentation and Evaluation 61

6 Conclusion **67**

Bibliography **68**

A Contents of the included storage media **73**

Chapter 1

Introduction

Since the introduction of the Internet, one of the most important considerations in the design, development, and deployment of services that can be accessed through it has been the issue of security. The basic cybersecurity goals have been defined using the *CIA Triad* [30], which consists of *confidentiality*, *integrity* and *availability*. Currently, Denial-of-Service (DoS) attacks, and especially their distributed variant (DDoS), are one of the most common threats to availability of services. Furthermore, as described in [45] and [13], the historical trend is that both the rate and the complexity of such attacks are increasing. According to Cisco, the number of detected DDoS attacks almost doubled between 2018 and 2023. DDoS attacks pose a significant threat to services that are part of critical infrastructure, where even a relatively short interruption in availability can cause potentially irreparable damage. Potential high-value victims include financial institutions, government agencies, healthcare facilities, and any systems and services used in the management of critical infrastructure, such as power and water supply. However, even DDoS targeted against businesses can cause significant damage to the economy. A distributed DoS or DDoS attack utilizes the greater computing power of a larger number of machines while also becoming harder to detect, as it is not possible to identify a single source of a traffic anomaly. Although historically mitigation of DoS attacks was possible by continuous analysis of statistics and blacklisting of anomalous hosts, these days it is no longer effective. Not only is the number of attacking hosts potentially quite large, but the attacker can use some form of spoofing or amplify the volume of malicious traffic using some other distributed service, such as DNS[1].

The first step in DDoS mitigation is network monitoring. Monitoring gathers important data which can be processed and analyzed. There are several protocols and technologies that can be deployed for monitoring purposes. Often, ordinary network devices, such as routers, can be configured to export network traffic information using one of the available protocols. The second step is detection, where a past, or in a better case, still ongoing, DDoS attack can be identified. The features of the detected DDoS attack can be used to prevent more such attacks in the future. However, while it is more difficult, active mitigation of the ongoing attack is more desirable. The simple option is to use the blacklisting of IP addresses, from which the attack originates, using a routing protocol. However, this approach has several drawbacks, the most important of which is that it can indiscriminately block all traffic that is on the way to the victim system. An alternative is to use a dedicated active DDoS mitigation system. One such system is being developed at CESNET[2], called DCPro DDoS

[1] Domain Name System
[2] https://www.cesnet.cz

Protector. DCPro DDoS Protector is an active DDoS mitigation system, which filters out any illegitimate traffic based on the configuration provided by the administrator. It also supports automated inference of configuration using machine learning techniques.

This thesis aims to integrate the DCPro DDoS Protector with the network monitoring tools, which are currently in use at CESNET, to form a coherent distributed system for the mitigation of DDoS attacks, which would cover all areas of protection against DDoS: monitoring, detection, and mitigation. In addition to that, integration should also affect the possibilities of inference of configuration by machine learning using the data gathered by the monitoring tools. Currently, there is no integration between the various systems and protocols, and the only way to relay data between them is by manually analyzing them and then inputting the configuration into the DCPro DDoS Protector. The thesis first discusses the basic concepts and technologies used for network monitoring and DDoS detection in CESNET in Chapter 2. Chapter 3 then describes the technologies used to actively mitigate DDoS attacks, with a special focus on the DCPro DDoS Protector and its inner structure. The solution is discussed and designed in Chapter 4. Finally, the implementation process is described in Chapter 5, together with an evaluation of the outcome.

Chapter 2

DDoS Detection and Monitoring

Denial of Service (DoS) has various definitions and many dimensions. The International Telecommunications Union (ITU-I) recommendation X.800 provides a helpful general definition:

> denial of service: 'The prevention of authorized access to resources or the delaying of time-critical operations.'

DoS can be caused by accidental or malicious actions. These actions can originate remotely or locally. Typically, damage is caused by the exhaustion of a critical resource at the target point. In this case, critical resources include network bandwidth, parts of network infrastructure, server resources, or application-level resources. In the first three cases, the attack is mostly dependent on raw computing power. Network links and devices, such as routers and switches, are effectively disabled for regular users due to the sheer size of the incoming traffic. Server resources, such as CPU computing power and memory, may be exhausted by flooding the server with requests, as every request requires some of these resources to be allocated for its processing, which eventually results in the inability of the server to handle any more requests. DDoS attacks are described in detail later in Section 2.1.

Efficient monitoring of incoming traffic makes it possible to identify the source of a DoS attack and block traffic from flooding the target and exhausting the resource. In the case of DDoS attacks, it is more difficult to distinguish the sources of the attack from legitimate users. Typically, it is possible to detect an attack by observing irregular traffic patterns, such as a sudden increase in incoming traffic, or more complex anomalies. Once an anomalous pattern is detected, sources can be blacklisted, or more sophisticated tools can be used to reduce the amount of incoming illegitimate traffic. One of those tools is the DCPro DDoS Protector described in Section 3.2.

2.1 Types of DDoS Attacks

DDoS attacks are classified into several types. According to [40], we can divide DDoS attacks into two basic categories, which are *semantic* and *volumetric* or *flooding* attacks.

2.1.1 Semantic Attacks

Semantic attacks exploit specific flaws in the implementation of communication protocols. An example of such an attack is the LAND (*Local Area Network Denial*) attack [49]. The

LAND attack sends a malformed TCP segment in which the attacker sets the destination IP address to be identical to the source IP address. When the target machine operating system receives this packet, it responds to itself, effectively sending the packet for reprocessing in an infinite loop. Another example would be the ping-of-death [25], which is based on the ICMP ECHO request, also known as *ping*. The maximum size for the request is set at 65535 bytes, which is also the maximum size for an IP packet. However, since an IP packet may be fragmented on the way, the total size of a specifically crafted ICMP ECHO request may exceed the expected maximum size, and the target system may crash when processing the request, due to the overflow of the internal 16-bit variable which holds the packet size.

Semantic attacks are best defended against by fixing the flaws in the used systems. For most users, the danger is minimized by regularly updating their systems. Defense against this kind of attack is simple and reliable, once the exploit is found and a fix is provided.

2.1.2 Volumetric attacks

Volumetric attacks are the brute-force method of disabling the target machine through the sheer volume of traffic. Although they require more resources for the attacker to execute successfully, the defense similarly requires potentially significant resources to effectively mitigate the attack. Completely reliable protection against such an attack is not possible, as the attacker may possess enough resources to overwhelm even the countermeasures. Mitigation of such an attack therefore always lies in making the attack more difficult and resource intensive than the potential gain.

Oftentimes, volumetric attacks are categorized by the location of the target in the layers of the network model. Attacks targeting the application layer target web applications and services by sending HTTP requests which overwhelm the application's capacity. Other kinds of volumetric attacks aim to exhaust the bandwidth or capacity of the target system or network device itself.

Modern DDoS attacks try to hide the identity of the attacker, making the attack more difficult to mitigate. The attacker may use a so-called botnet. A botnet is a group of machines infected by malware. A single infected machine is called a bot, which can then be controlled by the attacker without the knowledge of their owner [19, 20, 53].

Another approach is to misuse an existing Internet service as an amplifier to cause additional harm. This is called an amplification attack. The use of amplification increases the total volume of traffic and also hides the attacker behind a legitimate service that is misused to generate attack traffic [41].

Attacks Overview

The following is a simple overview of some common types of volumetric DDoS attacks. A more detailed description can be found in [55].

- **UDP flood** – A UDP flood is a sort of DoS attack in which the targeted host is flooded with IP packets containing a UDP datagram. If no applications have these datagrams, a destination unreachable packet is sent by the receiving host. As more UDP packets are received and responded, the system becomes overloaded and unavailable to other clients. In [50], several types of DDoS attacks are described, implemented, and analyzed, including the UDP flood attack.

- **ICMP flood** – A ping flood, also known as ICMP flood, is a typical DDoS attack in which the attacker causes the victim machine to fail by overpowering it with ICMP echo requests known as pings. The ICMP flood is also described in [50].

- **Smurf attack** – An amplification attack vector that increases the potential for harm by leveraging the features of the broadcast network. This type of attack is explained in [28], together with a detailed description of the effects of the amplification, and it also discusses the prevention of such attacks.

- **Fraggle attack** – involves flooding a network with faked UDP traffic directed at a router's broadcast address. As mentioned in the overview of DDoS attacks in [29], this type of attack can be considered a variant of the Smurf attack and is largely outdated now.

- **TCP SYN attack** – Makes use of a portion of the standard TCP Three-Way handshake to drain resources on the targeted server and make it unavailable. TCP SYN attacks are described in [11] in great detail. Furthermore, [15] discusses protection against TCP SYN DDoS attacks.

- **HTTP DDoS attack** – A volumetric DDoS attack that floods the target server with HTTP requests and is unable to respond to normal traffic. In [35], attacks that abuse the HTTP/2 protocol are described in detail. The study in [42] explains the protection against such attacks and proposes an optimal specification of the framework that would protect against this kind of attack.

- **XML DDoS attack** – Attacks deplete server resources and network bandwidth while handling SOAP messages. The attack is discussed in [24] in the context of cloud services.

2.2 Monitoring

In the context of DDoS detection, network monitoring is crucial, as it provides the victim with the ability to identify the sources of incoming illegitimate traffic and potentially restrict or block them completely [16]. This is most often done by collecting sample traffic from the network and then looking for anomalies by analyzing it.

An approach to monitoring DDoS attacks described in [27] is to set up a honeypot. Honeypots make it possible not only to safely identify the attacker, but also to potentially discover new methods of attack.

The other way is to use already existing network infrastructure, primarily routers, to collect traffic samples. There are two basic formats for network traffic samples: packet and flow.

A packet sample is the oldest and simplest way to store network traffic samples. The packets may be sampled at a certain rate by the network device and then collected, stored, and analyzed at a later date. The payload may be removed to save storage space and network bandwidth, as the headers are enough to detect most types of volumetric attacks targeting network layers L4 and lower.

Flow data, on the other hand, represents the flow of information or messages between two participants in a communication. Every flow is identified by several parameters, such as the IP address and port of the two communicants. A flow typically does not contain

the payloads of the packets, but instead the number of packets, the length, and other parameters.

2.3 Packet Capture Monitoring

The simplest way to monitor network traffic is to sniff packets. Recording of every IP packet is the only way to achieve complete knowledge of the information that is passed on over the network, including the protocols used, communication patterns of individual users, etc. However, as [2] states, in modern networks, it is simply not feasible to capture all packets in real time, due to the limiting impact on bandwidth and latency, as well as storage limitations. A relatively simple remedy to these problems is to sample traffic at a certain rate.

2.3.1 sFlow

One of the methods that can be used to capture packet samples is the sFlow protocol, described in RFC 3176 [39]. It defines two basic types of components, collector and exporter, called sFlow Agent. An sFlow Agent is a component or software module embedded in network devices, such as routers, switches, or standalone probes, which sends sFlow samples created by the network device to a central controller. sFlow defines the sampling mechanism, the interface for controlling sFlow Agents called sFlow Management Information Base (MIB), and the format of the sample data that the sFlow Agents forward to the central collector. The design aims to solve issues associated with the accurate monitoring of network traffic at high speeds, managing large numbers of agents, and having an extremely low cost agent implementation, so that it can be embedded in switches and routers.

sFlow implements two basic sampling methods: counter sampling and packet flow sampling. Counter sampling consists of periodically polling every data source on a network device, i.e. the counters. Although both types of samples are combined in sFlow datagrams, this thesis focuses on the use of packet flow sampling. Packet flow sampling produces samples of packets, with the restriction that the sampling mechanism ensures that any packet involved in a flow has the same chance of being sampled, regardless of the flow to which it belongs. In the context of sFlow, a flow represents a group of packets and is identified by the network device interface they were received on and the interface they were transmitted from. The mechanism involves a counter decremented with each packet. When zero is reached, a sample is taken. There is also another counter to count the total number of packets.

Taking a flow sample involves either extracting features from a packet or copying its header. The values for both counters are sent together with the samples to the sFlow Agent for processing. The format of a sFlow datagram is visualized in Figure 2.1. The sFlow protocol uses only the IP address to identify the Agent, unlike the protocols described in the following section.

2.4 Flow Monitoring

Flow monitoring does not suffer from the same issues as packet capture monitoring. Especially in high-speed networks, flow export is a relatively scalable and less costly option [18]. The IPFIX specification RFC 7011 [1] defines a flow as *'a set of IP packets passing*

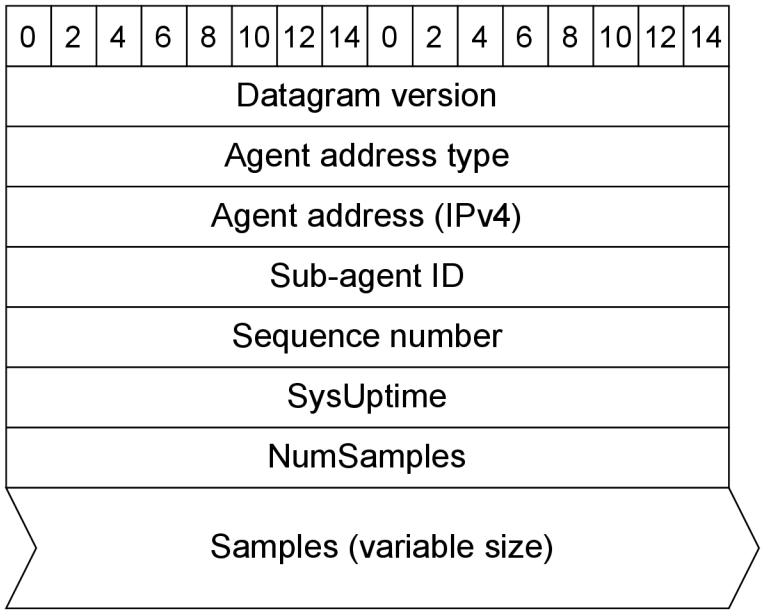


Figure 2.1: Structure of a sFlow Datagram header

an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties' These common properties can include metainformation and packet header fields, such as IP addresses and port numbers for both source and destination. However, what is recorded in a flow is mostly the headers of individual packets, not the payload. This resolves the capacity limitations, but reduces the amount of data available for analysis.

As described in [18], flow export protocols and technologies are widely used and deployed. This is due to their integration into high-end packet forwarding devices, such as routers, switches, and firewalls. In 2013, a survey among commercial and research network operators found that 70 % of participants have devices that support flow exports [47]. Secondly, significant data reduction can be achieved – in the order of 1/2000 of the original size, since packets are aggregated. Third, flow export is usually less privacy sensitive than packet export, since traditionally only packet headers are considered. Lastly, flow export makes it possible to comply with data retention laws. For example, in Europe, communication providers are required to retain connection data, such as flow export, for a period of six months to two years *'for the purpose of investigation, detection, and prosecution of serious crimes'* [38], [23].

After the aggregation of the packet header information in flows was described in [34], Cisco[1] and the Internet Engineering Task Force (IETF)[2] developed competing technologies for flow export. Cisco introduced its technology called NetFlow. The first widely available version of NetFlow was NetFlow v5, which was subsequently obsoleted by NetFlow v9 [8]. IETF on the other hand decided to standardize the IP Flow Information Protocol, or IPFIX, which is based on Cisco NetFlow v9, but offers additional features. Both the NetFlow v9 and IPFIX protocols utilize templates, which enable flexible flow definitions.

[1] https://www.cisco.com/
[2] https://www.ietf.org

In the following subsections, the two protocols are discussed in more detail. Due to the aim of this thesis, the focus is primarily on the structure of the exported flow data, rather than the specific mechanisms of capture.

2.4.1 NetFlow

NetFlow is a service provided by Cisco that allows network administrators to access information about network traffic in the form of flows. The original idea behind NetFlow came from flow-based switching, where flow information is maintained in a *flow cache* and forwarding decisions are made only in the control plane of a networking device for the first packet of the flow. The subsequent packets are switched exclusively in the data plane. The initial versions of NetFlow were based on the export of the information stored in the flow cache. The up-to-date version of NetFlow described here is NetFlow v9, defined in RFC 3954 [8].

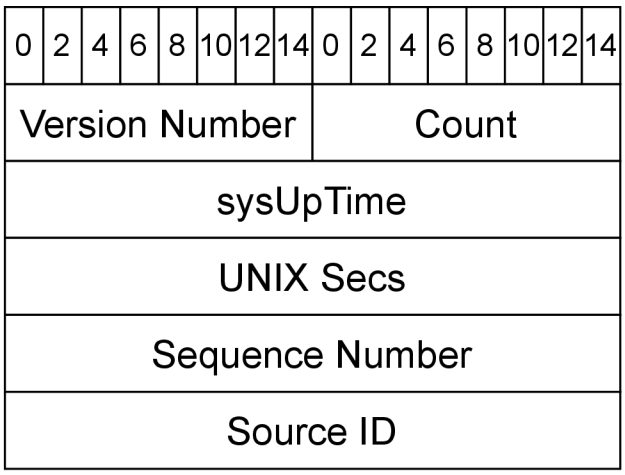


Figure 2.2: Structure of a NetFlow Export Packet header [8]

The NetFlow process running on the network device is called Exporter. The Exporter is responsible for the creation of flows from the observed IP packets and sending them to the Collector. If no packets belonging to a flow have been observed for a given timeout, the flow is considered inactive. If any packet is seen within the timeout, the flow is considered active. The Exporter can export a flow under the following conditions:

1. If the Exporter can detect the end of the flow, for example if a FIN or RST bit is detected in a TCP connection.
2. If the flow has been inactive for a given timeout.
3. For long-lasting flows, the Exporter should export the flow records on a regular basis, specified by a configurable timeout.
4. If the Exporter experiences internal constraints, such as low memory or counters wrapping, the flow may be forced to expire prematurely.

The default transport protocol is UDP, but NetFlow v9 has been designed to be transport protocol independent. Once the conditions for the export of a flow are met, the Exporter sends an *Export Packet*. An Export Packet consists of one or more FlowSets.

A FlowSet may be of three types: Template, Data or Options Template. FlowSets are collections of Flow Data Record, which is a structure containing the values a flow corresponding to a Template record. There are several ways in which the Exporter ensures that the Collector has the correct Template and Options Template Records when the corresponding Flow Data Records are received. The Exporter may interleave the Flow Data Records with Template Records to ensure they are delivered at the same time, and also periodically send the currently used Template and Options Template Records separately.

The Export Packet header is shown in Figure 2.2. Note that unlike sFlow, the NetFlow Export Packet Header contains the source ID, which should be used to identify different export streams originating from a single exporter according to [8]. Templates contain a variable number of fields. Every field is defined by type and length in bytes. The following are examples of various fields that can be used:

- `IN_BYTES` – Number of incoming bytes associated with an IP Flow. The default length is 4 B.
- `IN_PACKETS` – Number of incoming packets associated with an IP Flow. The default length is 4 B.
- `FLOWS` – Number of Flows that were aggregated
- `PROTOCOL` – IP protocol byte
- `TCP_FLAGS` – TCP flags; cumulative of all the TCP flags seen in this Flow
- `L4_SRC_PORT` – TCP/UDP source port number
- `IPV4_SRC_ADDR` – IPv4 source address

Every Flow Data Record contains the ID of the template that it uses.

2.4.2 IPFIX

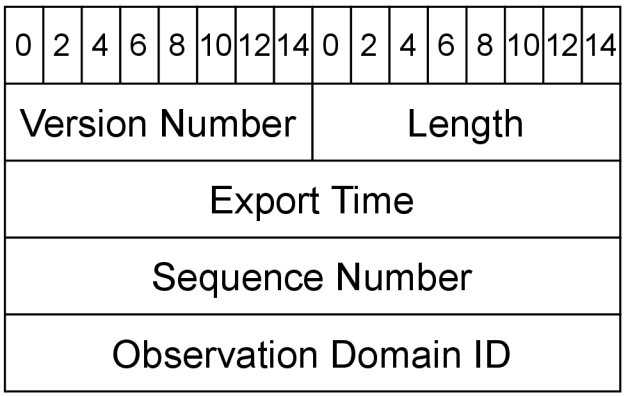


Figure 2.3: IPFIX Message header

IP Flow Information Export or IPFIX is a protocol for the exchange of flow information developed by IETF and is defined in RFC 7011 [1]. It is heavily based on NetFlow v9, which was described in Section 2.4.1. Most of the changes lie in the details of how the data

should be transfered between the Observation Point (analog to NetFlow Exporter) and the Collector. IPFIX identifies every flow by one or more Flow Keys. The traditional flow identifier is the *5-tuple* of source and destination IP addresses and ports and the transport protocol.

IPFIX Observation Point devices export Messages containing Sets, which are analogous to NetFlow FlowSets, and these have identical types; Data, Template, and Template Options. Templates are used to specify which Information Elements from the flow should be included in the Data. Unlike NetFlow, IPFIX was designed with the option to define proprietary Information Elements, which was needed by vendors.

As can be seen in Figure 2.3, IPFIX retains a similar structure to NetFlow in the Message header as well. Observational Domain ID is functionally identical to the NetFlow Export Packet Source ID. The specification recommends that each IPFIX device have a different ID.

2.5 IPFIXcol2

IPFIXcol2[3] is a flexible and high performance NetFlow v5 / v9 and IPFIX flow data collector [21]. It is fully modular and provides many input and output plugins, as illustrated in Figure 2.4. It is a new generation of IPFIXcol, which shared the same functionality but the implementation was less efficient. In the context of this thesis, all unqualified mentions of IPFIXcol refer to the IPFIXcol2 version, as the previous one is being phased out. IPFIXcol2 has been developed at CESNET and an instance is currently deployed in the CESNET network. It was created with performance in mind and, due to its parallelized design, is a perfect choice for centralized collection of flow data from the entire network.

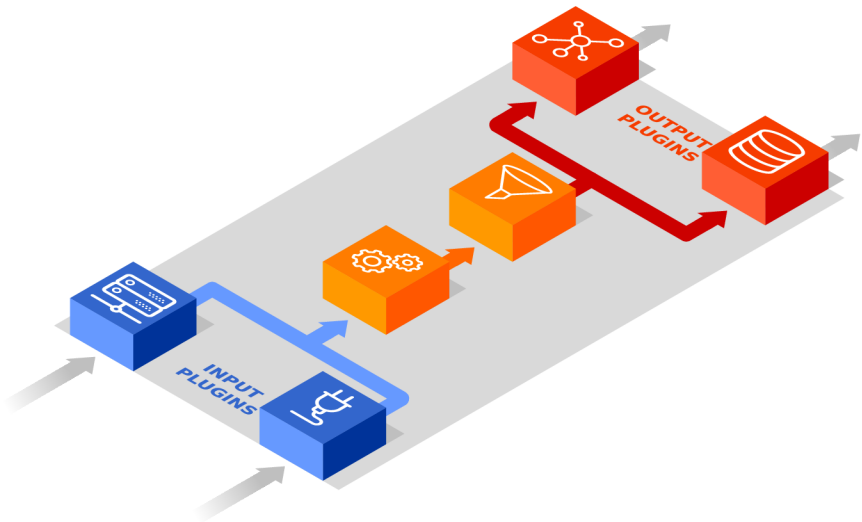


Figure 2.4: Schema of IPFIXcol2 modular design [22]

It provides the following input plugins, every plugin can listen on a given network interface and port:

- UDP – receive NetFlow v5/v9 and IPFIX over UDP

[3] https://github.com/CESNET/ipfixcol2

- TCP – receive IPFIX over TCP
- FDS File – read flow data from FDS File (efficient long-term storage)
- IPFIX File – read flow data from IPFIX File

A selection of the available output plugins is listed here:

- FDS File – store all flows in FDS file format (efficient long-term storage)
- IPFIX File – store all flows in IPFIX File format
- JSON – convert flow records to JSON and send/store them
- JSON-Kafka – convert flow records to JSON and send them to Apache Kafka
- Viewer – convert IPFIX into plain text and print it on standard output
- UniRec – send flow records in UniRec format via TRAP communication interface (into NEMEA modules)

IPFIXcol2 was implemented to match the specification of an IPFIX collector given in [1]. It is the core of a wider active network monitoring infrastructure run by CESNET. The modularity and the relatively wide selection of output plugins made it possible to integrate it with other network monitoring and traffic analysis platforms. Currently, the IPFIXcol2 instance deployed at CESNET also stores the flow records. for later manual or automated analysis. To make this possible, a flow filter was developed for use with IPFIXcol2, described in [48]. The administrator can also use visualization tools, such as FTAS or the more modern SecurityCloud, which are described in Section 2.7.

2.6 NEMEA Framework

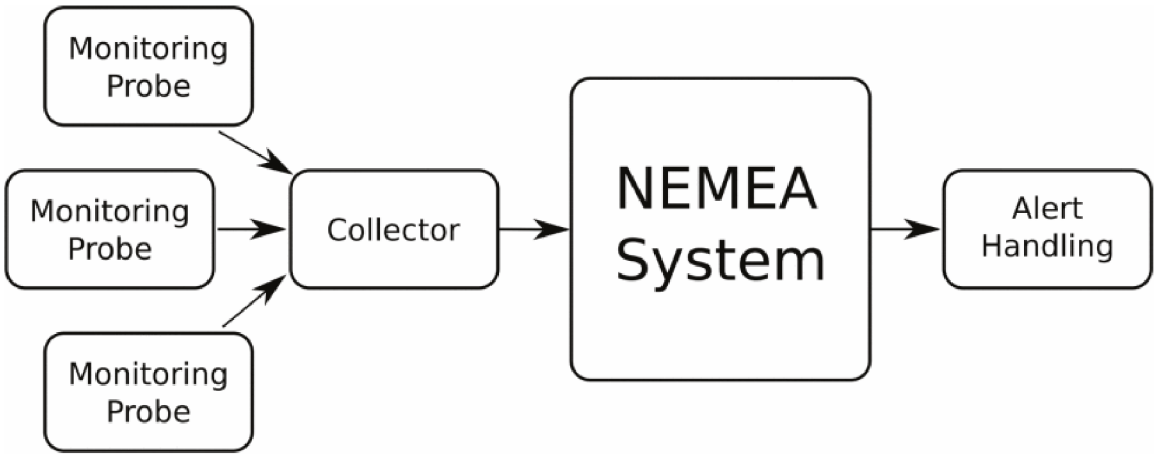


Figure 2.5: Flow monitoring infrastructure with analysis done by NEMEA [7]

Network Measurements Analysis or NEMEA is an open-source flow-based modular framework for the stream analysis of flow data [7, 9]. NEMEA has been created with the task of continuous analysis of flow data collected by IPFIXcol2. Every NEMEA instance is a collection of heterogeneous modules, which function as independent processes

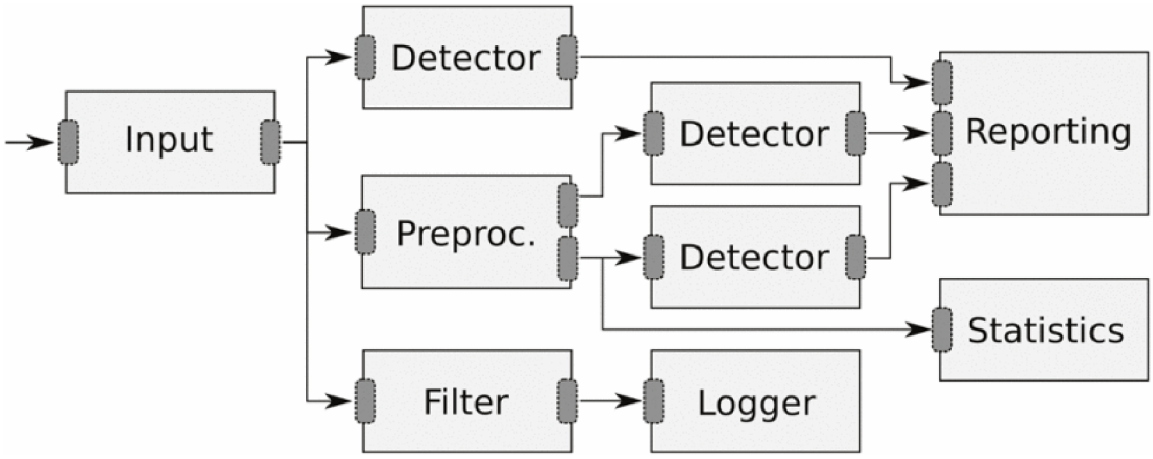


Figure 2.6: Example of several NEMEA modules and their interconnection [7]

connected by unidirectional interfaces for communication. The primary intended output of NEMEA are alerts, which are records about detected security incidents generated by detection modules.

The most important feature of NEMEA are the modules. Each NEMEA instance may be composed of a variety of modules and a number of possible interconnections between them. Different deployments of NEMEA may use completely different module configurations and thus perform different tasks. In a typical configuration, the modules are interconnected into a tree or a directed acyclic graph with a single module acting as the main input of the whole system. This module gathers or creates flow records and sends them to other modules which process them. Each module is essentially a separate program that performs a specific task on the gathered flow data, such as preprocessing, filtration, anomaly detection, or logging and reporting results. An example of a configuration of modules and their connections is visualized in Figure 2.6.

NEMEA is not only easily reconfigurable, it can also be easily extended with new functionality. The NEMEA framework is designed to allow quick and easy implementation of new modules. Although NEMEA is used in a production environment for analysis of live traffic, it is also designed to serve as a common platform for researchers in the area of network security and monitoring. It allows for fast prototyping of new traffic analysis methods, testing them on both offline and online data and comparing them with existing methods.

The framework used by the NEMEA system is developed in C language and brings support for the implementation of additional user-specific NEMEA modules in C, C++ or Python. The set of modules can be controlled and monitored by a tool called Supervisor. The intended purpose is to have every module implement an algorithm for traffic analysis or detection and communicate with other modules using the features of the NEMEA framework. The system, which consists of a whole array of modules, should be able to run on any UNIX-like operating system.

NEMEA is very well optimized for high-throughput flow data analysis, which is expected when processing backbone network traffic. It uses the binary UniRec format for data exchange, which in conjunction with the stream nature of data processing allows for efficient memory management. The modular design makes it theoretically possible for multiple identical modules to run concurrently if their design allows it. If a module requires access

to a wider or more detailed part of the traffic, there are filtering and preprocessing modules that streamline the process, so that it can still be scalable and fast enough for real-time use in high-speed networks.

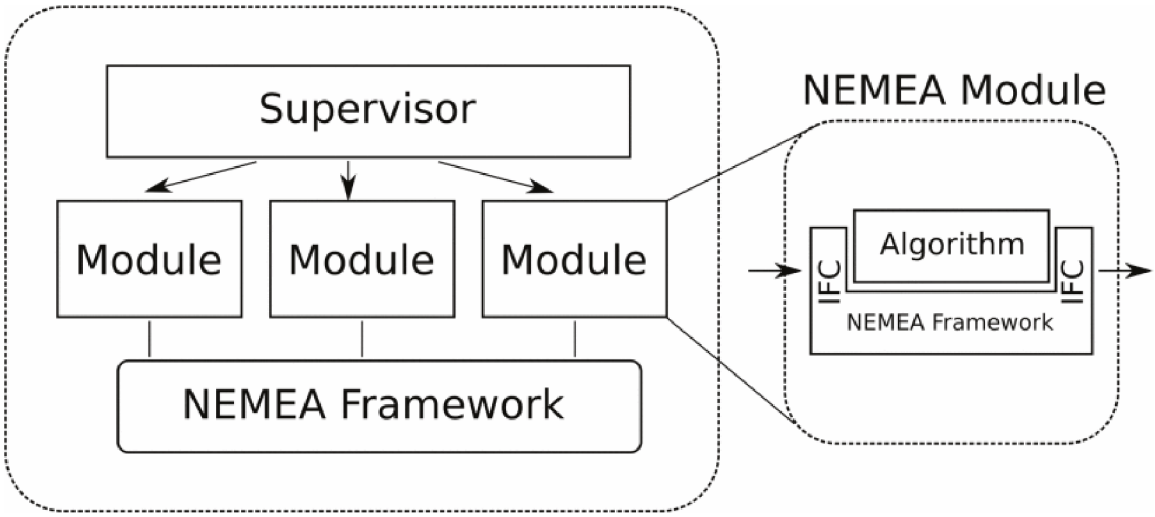


Figure 2.7: View of NEMEA module implementation and integration [7]

2.6.1 Traffic Analysis Platform (TRAP)

The most important part of the framework is the Traffic Analysis Platform (TRAP) library, which implements the communication interfaces and other basic functions that all NEMEA modules need. TRAP implements a high-level communications interface, which can be used to relay messages between the individual modules. In the code of the modules, TRAP is used in a way similar to sockets. Internally, TRAP serves to encapsulate lower-level interprocess communications interfaces, more specifically, the TCP interface, the Unix socket interface, and the SHMEM interface. TRAP also provides additional functions for the improvement of the general quality of data transmission, such as timeouts, auto-flush, multi-read operations, and buffering.

TRAP interfaces are usually initialized at the time of module launch from the parameters passed to the module instance by the user, typically using command-line arguments. A module might expect to receive the identifiers of one or more interfaces and may use them to write or read messages. There is a standardized format for passing interfaces in a command-line argument. The format requires a specification of the underlying interface type and a variable number of additional parameters, which may be required to initialize the lower-level interface. The general structure is the following: `<type>:<par1>:<par2>:...:<parN>`, where `<type>` represents the lower-level interface type and may contain the following values:

- `u` – Unix socket (for local communications)
- `t` – TCP socket (for remote communications
- `b` – blackhole (to drop all messages written to it)
- `f` – file

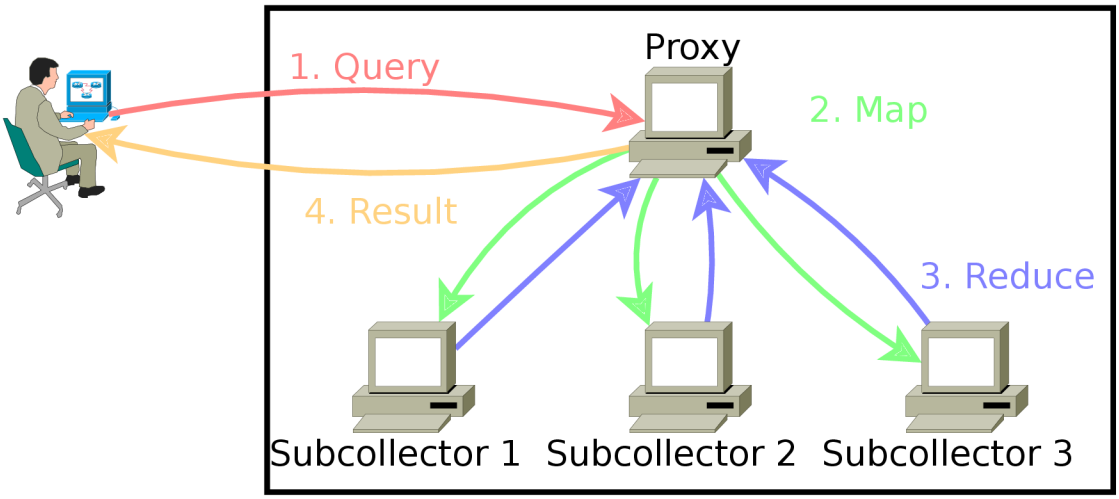


Figure 2.8: SecurityCloud architecture and query execution model [44]

The other parameters together enable the unique identification of the interface. In case of TRAP interfaces using TCP sockets, it is a combination of host address and port, in case of Unix sockets, it is a string identifier. Multiple interfaces can be passed as values of a single command-line argument, in which case they are separated by a comma (`,`). An example TRAP interface specification looks like this:

```
merger -i "t:localhost:8801,t:localhost:8802,u:DNS_out"
```

The example lists the module merger, which writes all messages received on the interfaces `t:localhost:8801` and `t:localhost:8802` to the interface `u:DNS_out`.

2.6.2 Unified Record (UniRec)

Unified Record or UniRec in short is a data format for binary representation of flow records and other information. UniRec is the default data format used for communication between modules using TRAP interfaces. A UniRec message contains binary data, which makes it efficient for storage, transmission, and processing, and a template, which describes the data. The binary data contained within the data structure is described using UniRec templates. Every UniRec message has a template attached, specifying the fields of the message and their types. There are several predefined template field types, such as `int8`, `int16`, `macaddr`, `ipaddr`. Every field must also be given a unique field name that serves as its identifier.

The NEMEA framework introduces functions for the C and Python languages, which serve to specify the expected template in the code. Templates can be dynamically modified during runtime. Most basic modules do not expect a specific template, or only require a specific field to function, such as the source or destination IP address. This is especially helpful in case an extra field is added by some other module when the whole system is already up and running.

2.7 SecurityCloud

Security Cloud is a modern distributed flow-based processing software, which is being developed as a joint effort of CESNET, Flowmon, and Masaryk University [44]. The core parts are the IPFIXcol2 collector and the fdistdump tool. Fdistdump is a command line tool that utilizes MPI[4] to communicate with subcollectors. The user can use fdistdump to execute ad hoc queries on stored data. As shown in Figure 2.8, the user sends the query to a proxy node, which then collects the data from the subcollectors using the map and reduce phases, which do not exactly correspond to the MapReduce computation model. What is also important for the topic of this thesis is not only the ability of SecurityCloud to process structured queries, but also that it includes a GUI for the visualization of graphs displaying the queried network traffic. The GUI is shown in Figure 2.9.

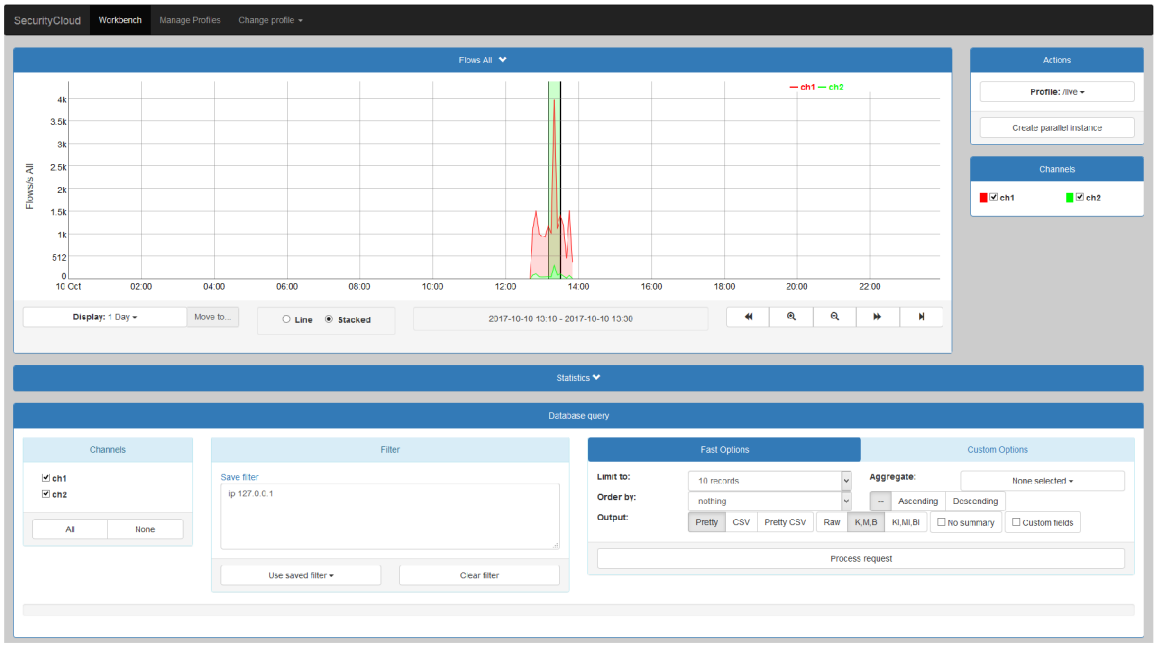


Figure 2.9: SecurityCloud GUI [43]

SecurityCloud is a relatively new successor to the system that CESNET has used before, called the Flow-based Traffic Analysis System, or FTAS[5]. FTAS, described in [17] and [52], offered similar functionality to SecurityCloud, differing mainly in implementation details.

Both mentioned visualization tools that operate with raw flow data tend to be more optimized for a detailed analysis of a single incident, rather than continuous automated monitoring, since flows are not processed in a stream. Instead, IPFIXcol2 saves the data in a file which can then be loaded and manually analyzed. This approach is also severely limited by the fact that the number of flows collected in the CESNET network every second can easily reach tens or even hundreds of thousands. Both FTAS and SecurityCloud are intended for the analysis of incidents that occurred a day or two ago at maximum. For automated traffic monitoring and analysis, it is much more effective to use a stream processing platform like NEMEA, to leave out the need to persist every single flow record. This thesis aims to provide a visualization and monitoring solution, which addresses those

[4] Message Passing Interface

[5] https://www.cesnet.cz/en/services/network-1/network-traffic-monitoring-ftas-34

concerns and which allows information to be provided in formats useful to the administrator of the DCPro DDoS Protector, which is described in Section 3.2, as well as a way to effectively aggregate flows for longer-term storage, as the storage of individual flows proves ineffective at the expected high throughput.

Chapter 3

DDoS Protection and Mitigation

Ever since the advent of DoS and DDoS attacks, protection against such attacks has been one of the primary areas of network security. However, due to the nature of volumetric DDoS, creating a completely impermeable defense is virtually impossible. Therefore, the most cost-effective approach is to minimize the effects of volumetric DDoS. The various approaches to mitigation of DDoS attacks are discussed in [3].

DDoS attacks can be prevented using passive countermeasures, such as robust load balancing and high bandwidth. However, often a more active approach is needed. Active mitigation of DDoS attacks can be broadly divided into two categories; filtering and rate-limiting. Rate-limiting mechanisms do not completely remove the attack, but reduce the incoming traffic so that the victim machine is not overwhelmed. However, rate-limiting works indiscriminately and a portion of legitimate traffic is also blocked. This effect is especially damaging if rate-limiting is done close to the victim, as a large number of legitimate users will be affected. Filtering, on the other hand, aims to identify and filter out illegitimate traffic. Although it seems to be the optimal option, filtering is significantly more resource-intensive. A common tool, which is also used in the CESNET network, is the Border Gateway Protocol. CESNET and affiliated institutions are also currently developing a more complex active DDoS mitigation tool, called DCPro DDoS Protector.

3.1 Border Gateway Protocol (BGP)

The Border Gateway Protocol (BGP) is one of the most powerful widely used routing protocols. The protocol defines the concept of Autonomous Systems (AS), which are collections of networks and routers [12, 46]. BGP is used to define routing for the connections between different AS, as well as inside them. Therefore, the BGP protocol is further classified into external BGP (eBGP), which is used in communication between different AS, and internal BGP (iBGP), which is used to route communication inside a single AS. The BPG routing protocol is one of the primary methods of DDoS mitigation, either by traffic policing or by blackholing the attack traffic. However, both approaches inevitably lead to legitimate traffic being dropped as well. In case of traffic policing, the cause is obvious. Blackholing only the attack traffic is virtually impossible. Due to the distributed nature of DDoS, the attack comes in packets with different source IPs. The only effective way to remove the attack is to drop all packets with the destination IP of the protected subnet.

Remote Triggered Black Hole (RTBH) is a technique provided by the BGP protocol, which enables the network administrator to specify which packets should be dropped by

the routers. The preconfigured policy on the router is to send the abuser packets to the nullo interface, where it becomes unreachable. The most proper way to stop a DDoS attack is as early as possible. Border routers are the suitable place to do so. A better approach might be to redirect the attack traffic to a filtering system, which can efficiently distinguish between the attack and actually legitimate traffic. An example of such a system is the DCPro DDoS Protector described in Section 3.2.

The most efficient technology based on BGP, which is currently in use, is BGP FlowSpec. BGP FlowSpec is a technology used for the effective distribution of routing information, in the form of FlowSpec rules, based on the proposal in [32]. FlowSpec rules allow for a wider variety of options to classify traffic compared to basic BGP, so the selection of the attack traffic is much more precise and less legitimate traffic is affected. FlowSpec rules always include an action, such as *accept*, *decline*, `rate-limit`, and *redirect*. The action specifies what the router should do with the traffic matched by the options in the respective rule. BGP FlowSpec allows the rules to be dynamically deployed to the routers and is an effective and scalable solution for managing the volume of incoming traffic. In the CESNET network, the ExaBGP[1] tool is used for the efficient distribution of FlowSpec rules from user-created configuration files, in plain text or JSON formats, to the routers.

3.2 DCPro DDoS Protector

DCPro DDoS Protector is a software device developed by CESNET with the aim of protecting a network from external DDoS attacks [10]. It can filter incoming traffic with bandwidth of up to 100 Gbps. The device was designed primarily to counter amplification attacks. However, since SYN flood attacks comprise the majority of today's DDoS attacks, the device also implements the corresponding mitigation methods, as detailed in [15]. As such, it is a solution to the problem of effective filtering of DDoS attacks.

The DCPro DDoS Protector is implemented using the DPDK[2] framework, so that it can directly access the network card without the need for the packets to pass through the operating system. This makes it possible to utilize the processing power of the network card much more efficiently and without any delays caused by the overhead of the different layers of the operating system that would otherwise stand in between the card and the application. As shown in Figure 3.1, the DCPro DDoS Protector acts as an L3 network device. The router that acts as the access point to the network redirects unverified traffic to the DCPro DDoS Protector. If the traffic is safe, it is redirected back to the router and then to the network. In the opposite case, the packets are dumped by the DDoS Protector and do not enter the network.

3.2.1 Architecture

The DCPro DDoS Protector as a system consists of the main application and several supporting tools. The whole system is designed to be modular and distributed. The system is divided into two types of machines, Controllers and Mitigators.

The Controller is a (optionally virtual) machine that provides a central configuration database, Web API, and additional services. It has two primary features. Firstly, it acts as a controller for Mitigator instances, storing all persistent data, such as configuration and statistics. Secondly, it hosts the tools for user interaction, such as ExaFS, which is

[1] `https://github.com/Exa-Networks/exabgp`
[2] https://www.dpdk.org

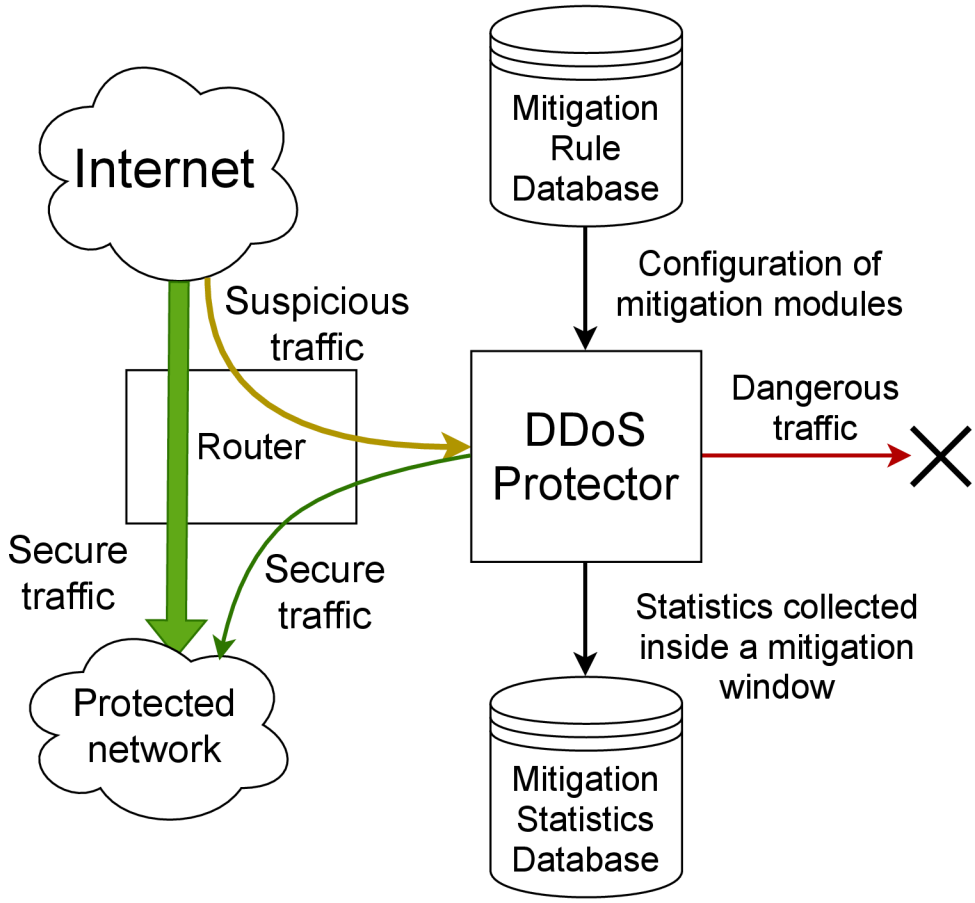


Figure 3.1: Schema of DCPro DDoS Protector connection to the network [5]

a web server with a GUI. The database is accessed by both the user tools and the Mitigator instances, which can dynamically load their configuration from there.

The Mitigator is a separate machine that is used to mitigate attacks on the wire and that should preferrably run on separate dedicated hardware. The only program running on a Mitigator instance is the main DCPro DDoS Protector DPDK application itself. In Figure 3.1, the DDoS Protector represents a Mitigator instance, while the databases represent the Controller.

The DPDK application utilizes a dependency injection framework, so that it can be fully modular. The DI configuration is done using YAML[3] files, which specify what modules should be used and how they should be instantiated. This makes it possible to distribute the application in binary format, while also leaving room for configuring the application to fulfill specific tasks. The DCPro DDoS Protector framework can be used to create a fully functioning DDoS mitigation device or a simple probe that only collects traffic data.

3.2.2 Pipeline Configuration

The DCPro DDoS Protector DPDK application consists of a pipeline which processes a stream of packets which are received by the network card. The pipeline has a few basic

[3]YAML Ain't Markup Language

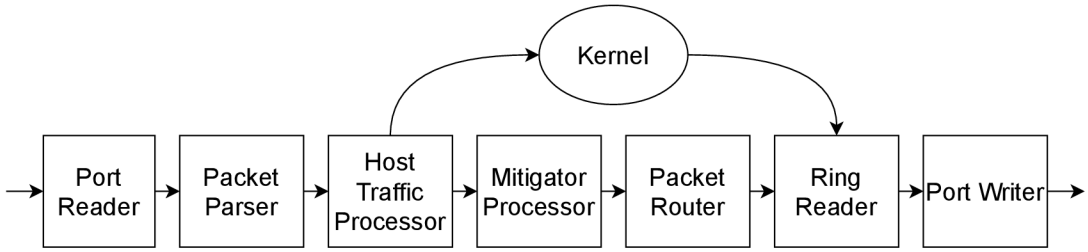


Figure 3.2: DCPro DDoS Protector packet processing pipeline

modules, each made up of a number of smaller and more specific modules. The pipeline is visualized in Figure 3.2.

The Port Reader and Port Writer stages are concerned only with reading and writing packets on the binded network card port. The Host Traffic Processor and the Ring Reader stages are primarily tasked with redirecting traffic intended for the machine itself to the operating system kernel. The bulk of the actual DDoS mitigation is done in Packet Parser and Mitigator Processor stages of the pipeline. Packet Parser parses the packet features from every packet. These include most of the items in the L3 and L4 headers. The Mitigator Processor has two main parts. The first part is the packet classifier. To understand how the packet classifier module functions, it is necessary to first discuss how the Mitigator is configured. The Mitigator Processor module is configured using so-called mitigation rules. Each mitigation rule type corresponds to a single mitigation module. A mitigation rule is essentially a data structure that contains the configuration of the mitigation module and the packet classifier. The rules share a base, which includes items like source and destination IP addresses, source and destination ports, VLAN identifier, and others. The packet classifier checks whether the incoming packet matches any of the mitigation rules, and if so, the packet is processed by the given mitigation module. The second part of the Mitigator Processor is the mitigation modules themselves. These have a common modular structure, so that the DCPro DDoS Protector may be easily extended with additional DDoS Mitigation methods.

3.2.3 Mitigation Rules

As described in Section 3.2.2, mitigation rules are data structures used by the DCPro DDoS Protector DPDK application, which hold the configuration for the packet classifier and for the respective mitigation module. There is at most a single mitigation module for every mitigation rule type. Some mitigation rules are abstract, and are used as a template for the creation rules of other types by the Controller, but are otherwise only present in the database, and are not loaded into the Mitigator itself. These would primarily be rules utilizing machine learning methods. Rules are persistently stored in the Controller database. They can be modified only through the Controller. Mitigator instances load rules into memory every time the instance is launched or when the database is modified.

There are two basic categories of mitigation rules, based on whether they produce traffic on their own or not. For better orientation, Machine Learning rules have been separated into a third category; otherwise they would also be classified as passive. The rules are listed in the order in which the methods are executed.

Passive Rules

A passive method determines whether to pass a particular packet based solely on traffic observation.

- *Packet Capture* – captures all matching packets into PCAP format.
- *Exact Match* – allows or drops all matching packets based on source or destination IP address, depending on the mode (deny/allow).
- *Malformed Traffic* – drops malformed packets.
- *Filter* – drops all matching packets.
- *SYN Drop* – mitigates SYN Flood (D)DoS attacks.
- *Amplification* – mitigates amplification (D)DoS attack.

Interactive Rules

An interactive method might also generate traffic, and it can act on behalf of a protected device or network. These crafted packets typically require additional routing configuration.

- *TCP Authenticator* – mitigates SYN Flood (D)DoS attacks.

Machine Learning Rules

Special kind of passive rules. The Mitigator does not read these rule types; instead, each of them has its dedicated tool that handles them.

- *ADT PCAP* – learning method that uses data from Packet Capture rules to generate filter rules using adaptive decision trees. The method is futher described in Subsection 3.2.6.
- *ADT Flow* – similar to ADT PCAP but using UniRec data instead of captured packets.
- *KitNET* – machine learning method that uses packet data from an external source to generate a set of IP addresses to block. The method is further described in Subsection 3.2.7.

3.2.4 Mitigation Window

In the Mitigator Processor stage, there are contrasting requirements. Firstly, the Mitigator should at all times be concerned with processing the stream of incoming packets to maximize the throughput and efficiency. To further accelerate the processing of packets, the Mitigator Processor runs in several parallel pipelines. At the same time, however, it is crucial for the administrator to be able to see how well the device and the individual mitigation rules perform, so that the configuration may be improved, and any potential problem detected and solved. Some mitigation rules also require access to the total number of packets or bytes which fit certain criteria, to know whether a threshold was exceeded or not. Because of both of these, it is necessary to aggregate the counters incremented by each of the parallel pipelines.

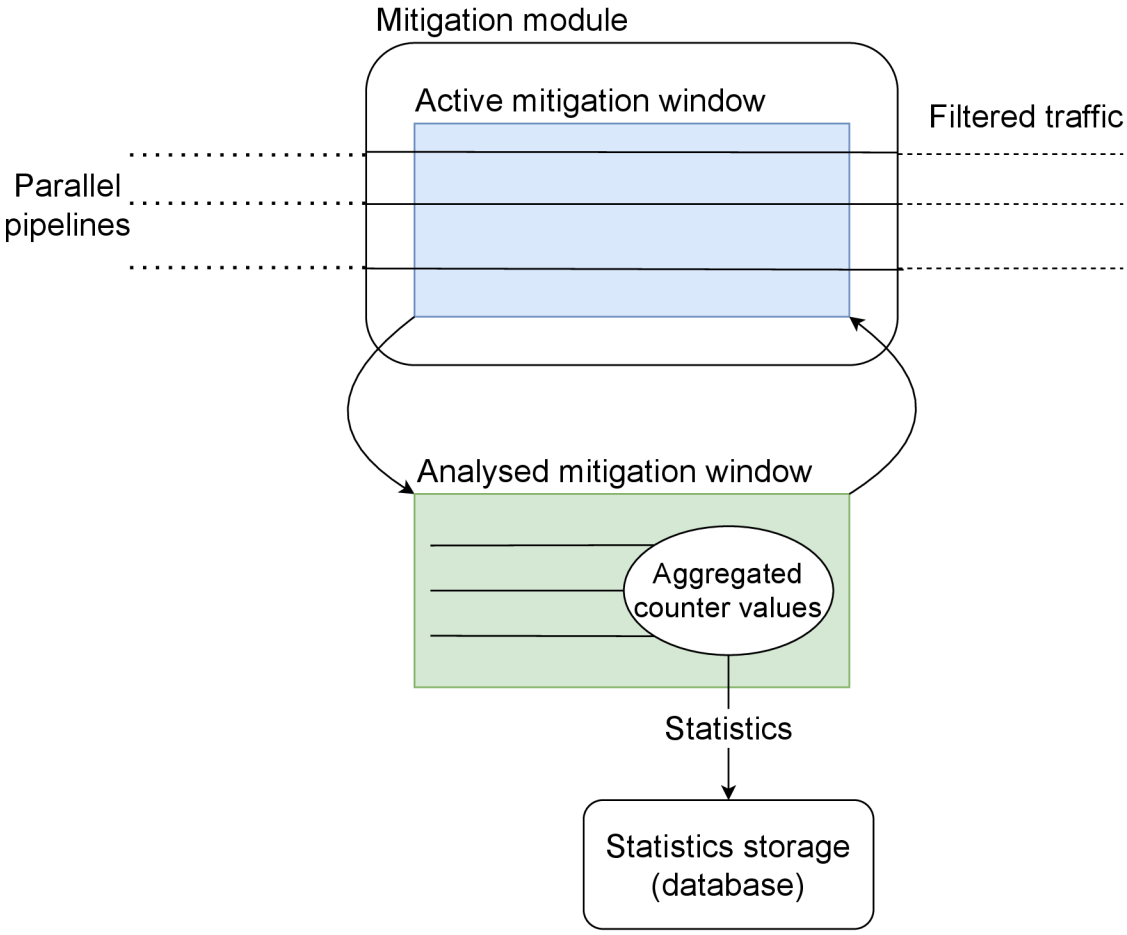


Figure 3.3: DCPro DDoS Protecto mitigation window mechanism [5]

The solution to this problem is the mitigation window mechanism, which is visualized in Figure 3.3. The Mitigator Processor keeps two structures, called mitigation windows. At any one time, one mitigation window is in the active phase, and the other is in the preparation and statistical collection phase. The two windows periodically exchange their place; the default value for the interval is one second, which is also called the duration of a mitigation window. Every time the windows are swapped, the currently inactive window aggregates data from counters from all the pipelines for every mitigation rule. The aggregates are then saved to the database in the form of mitigation statistics. The structure of the statistics is discussed later in Subsection 3.2.9. For the purposes of this thesis, it is important to note that this mechanism collects, aggregates, and periodically saves the number of bytes and packets.

3.2.5 Mitigation Rule Database

The mitigation rules represent a dynamic configuration that is expected to change many times in the lifetime of one DCPro DDoS Protector run. Although they can be preloaded from a YAML file whenever a DCPro DDoS Protector instance is launched, this is only intended for testing purposes, as rules loaded from a YAML file cannot be dynamically

changed at runtime. In a production environment, the mitigation rules are stored inside a PostgreSQL database instance, which runs on the controller machine.

The schema of the mitigation rule database is defined using Python classes in the `dcproapi` library, in conjunction with the `sqlalchemy` ORM[4] library. In this way, it is ensured that the rule structure is the same in the database schema as in the ORM classes in the configuration tools written in Python. To properly manage database migrations, `alembic` is used in conjunction with `sqlalchemy`. Since the database tables are defined with the use of Python, the fields in ER diagrams are annotated with respective Python types, such as `str`, `int` or `enum` for any kind of enumeration type. Special types for the representation of IP and MAC addresses are provided by the `sqlalchemy` extension for PostgreSQL.

Individual rule classes are mapped onto database tables in a way that for every class, even an abstract one, there is a separate table. Because of this, there is a base class which contains the common rule attributes, mostly those specifying whether the rule is enabled or in a dry-run state, and those attributes used for traffic classification, such as IP addresses and ports. Due to the fact that the structure of every rule type shares these attributes, it is possible to classify the incoming packets by whether they match the rule or not. The structure is visualized in the mitigation rule database schema in Figure 3.4. Only a few example rule types are shown in the schema; the types relevant to the purposes of this thesis are described and discussed separately in Subsections 3.2.6 and 3.2.7.

3.2.6 Adaptive Decision Tree (ADT) Rules

The Adaptive Decision Tree (ADT) is a machine learning model which is being used in the ADT Flow and ADT PCAP rule types. ADT Flow and ADT PCAP can be understood as two variants of the same rule, differing only in the type of input that is being processed by the ML model. They share all other aspects; therefore, from now on, they will be referred collectively to as ADT rules. Instead of providing configuration for an actual rule present in the DCPro DDoS Protector, it is used by a daemon, which then uses the ADT rule as a template to create additional mitigation rules. The ADT rules are used to derive a Berkeley Packet Filter (BPF) filter which can later be used in a Filter rule. The inferrence mechanism is described in detail in [6].

The daemon executes two basic kinds of processes. The first is the capture process. For ADT PCAP, this process creates a Packet Capture rule, to capture traffic on the DCPro DDoS Protector device. For ADT Flow, the process creates an instance of a simple specialized NEMEA module, which instantiates a `dataframe` class from the `pandas` library using the collected flow data. The Packet Capture rule shares the basic attribute values with the ADT PCAP rule, so it can be configured to only capture traffic to and from certain IP addresses and ports. The NEMEA module is preceded by a `unirecfilter` module, which filters the incoming UniRec records in the same way. The data, be they UniRec records or PCAP, are then converted to the `pandas dataframe` format. The data are then analyzed. If the traffic contained within the `dataframe` exceeds the threshold specified in the ADT rule, the `dataframe` is marked as containing malicious traffic. Otherwise, it is marked as safe. When a malicious `dataframe` is encountered for the first time and there is at least one safe `dataframe` exists, the model is trained. The mechanism common to both ADT rules is visualized in Figure 3.5.

[4] Object-Relational Mapping

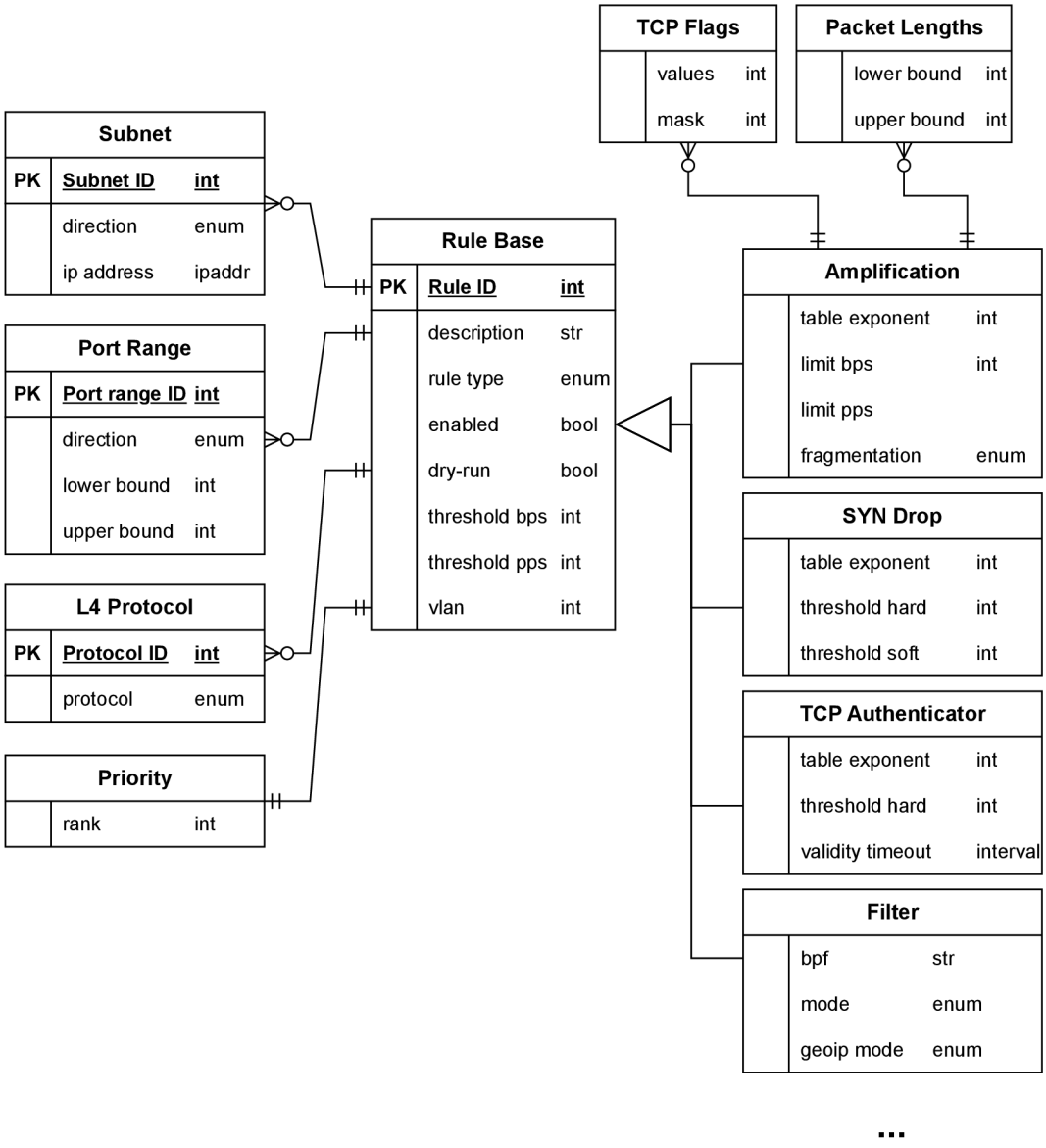


Figure 3.4: Schema of mitigation rule database

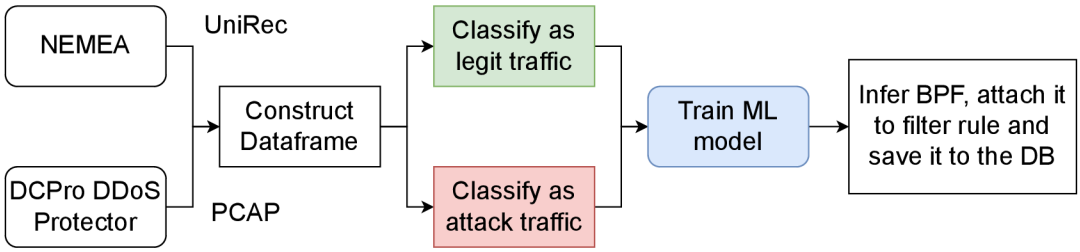


Figure 3.5: ADT rule functionality

The model for the ADT rules consists of an adaptive decision tree, which looks for the differences between the datasets and represents a BPF[5] statement, which aims to remove most of the malicious traffic while keeping as much of the safe traffic as possible. The BPF filter is then used to generate additional Filter mitigation rules, which again use the attributes of the ADT rule as a template and add the BPF generated by the machine learning model. Since the BPF is only concerned with L3 and L4 packet header values, it is possible to use it on the UniRec records as well, even if the accuracy of the model will be inherently lower than with PCAP files, as PCAP contains samples of entire packets.

3.2.7 KitNET Rule

KitNET rule is a type of machine learning rule that configures a method that uses a type of neural network called an autoencoder to properly identify and mitigate only malicious packets. The method implemented by KitNET first trains an ML model, called a KitNET instance, which may be used by multiple KitNET rules for different parts of traffic. The KitNET machine learning model is described in detail in [36], and the implementation of the KitNET module is described in [14].

As shown in Figure 3.6, each KitNET rule must be assigned a single KitNET model instance, which is then assigned a single IP set. IP sets are objects that are optimized to contain a large number of IP addresses. They can then be used by `exact_match` rules to efficiently blacklist or whitelist packets coming from the IP addresses they contain. KitNET rules have two modes, `learn` and `detect`. In `learn` mode, the KitNET rule is used by the daemon as a template for the creation of a Packet Capture rule. The daemon then uses the PCAP file produced by the Packet Capture rule to train the KitNET model instance, expecting the PCAP to contain only legitimate data. Once the model is sufficiently trained on mostly legitimate data, the KitNET rule may be switched by the administrator to the `detect` mode, in which it no longer modifies the KitNET instance, but instead uses it to detect anomalous packets from PCAP, and then fills the specified IP set with the source IP addresses of the potentially malicious packets.

3.2.8 Rule and Device Groups

The DCPro DDoS Protector is meant to be deployed as a distributed system, where multiple Mitigator instances exist with a central Controller server with a database. Since different Mitigator instances in a single distributed system may require a specific configuration, the DCPro DDoS Protector works with the concept of groups. In this context, a group refers both to a group of devices and to a group of rules. This means that we can assign a number

[5] Berkeley Packet Filter

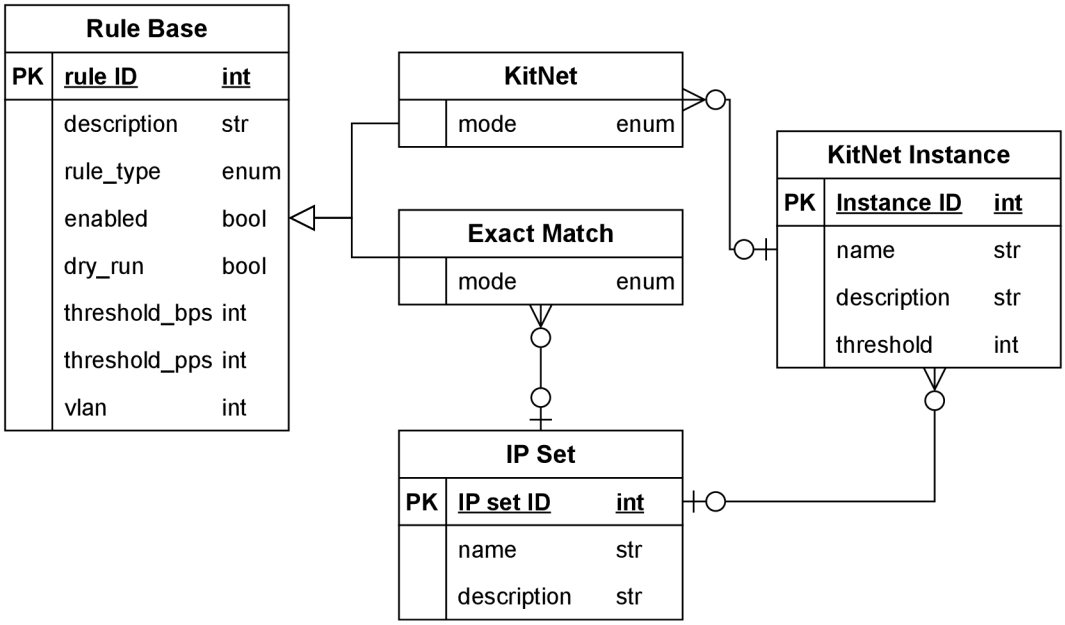


Figure 3.6: KitNET rule schema

of rules to a group which contains only Mitigator devices on the periphery of the protected network. Both rules and devices may be assigned to multiple groups at once. Each device loads all rules from every group assigned to it. The schema of the database is shown in Figure 3.7.

The group structure allows the administrator to categorize the rules into groups that are then used on a group of devices. Therefore, it is possible to specify groups of rules based on the throughput of specific deployed Mitigator instances, since some instances may use lower-capacity network cards. There may also be a difference between rules used on the periphery of a network and rules which are used to protect a specific object inside the network. The mechanism makes it possible to build a truly distributed system for DDoS mitigation, with different parts having different configuration to fully contribute to the common goal in conjunction with each other.

3.2.9 Statistics and Device Database

Other than the `mitigation_rules` schema which contains the tables that contain the mitigation rules, the Controller database contains other schemas to logically separate different types of entity. This is made possible by the PostgreSQL database management system, which supports the use of schemas as namespaces. The three other schemas are `mitigation_statistics`, which is described in Figure 3.8, and consists of a table for the statistics on how many packets and bytes were dropped or passed by individual mitigation rules on given Mitigator devices, `devices`, described in Figure 3.9, which contains records for every Mitigator instance, and `device_statistics`, described in Figure 3.10, which contains statistics on how many total packets and bytes were dropped or passed through the given device, regardless of mitigation rules.

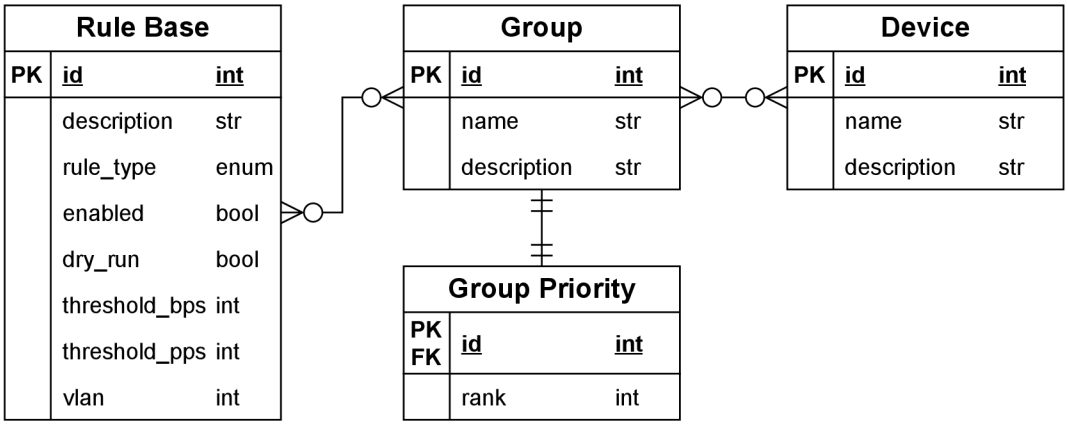


Figure 3.7: Rule and device groups schema

As shown in the diagrams, the schemas are interconnected, and in several tables, there is a foreign key referencing a column in a table from a different schema. This is especially evident in the table that contains the mitigation statistics. Mitigation statistics are collected in the Mitigator Processor stage and exported to the database by every mitigation window. The record stored in the database includes the identifier for the mitigation rule, as well as the Mitigator instance on which it was collected. This means that it is possible to see the effectiveness of a specific mitigation rule, as well as how well it functions on a given device, since some Mitigator instances may be deployed on high-speed links on the perimeter of a network, while others may be deployed farther inside to protect specific objects or subnets. Additionally, every Mitigator instance exports device statistics data, which show the total sum of incoming and outgoing data for the instance, regardless of the mitigation rule.

Although device statistics contain generally useful data for the network administrator, to see how much traffic is traveling through the mitigation device, it provides little useful data for the configuration of the Mitigator devices, except as a way to estimate the volume of traffic and the possible values for useful thresholds. However, mitigation rule statistics provide an integral insight for the configuration of Mitigator devices and the DCPro DDoS Protector infrastructure as a whole. Using mitigation rules, it is also possible to see the volume of traffic from a given source to a given destination, since the statistics only contain the packets and bytes matched by the rule. The fields contained in the statistical records collected by the Mitigator are shown in Figure 3.8. A record is always identified with the device on which it was collected and the mitigation rule, since a packet is evaluated by every matching rule. The dry-run and rule status fields only record the state of the rule at the moment, whether it was active or not. Each processed packet increments the `packets_in` and `bytes_in` by the respective amounts. If the packet in question is not dropped by the rule, it also increments the `packets_out` and `bytes_out` in the same way. The fields `packets_in_activation` and `bytes_in_activation` are used primarily by those rules, which are automatically activated after encountering a preset number of packets with certain characteristics, such as some TCP flags being set. Finally, `packets_out_gen` and `bytes_out_gen` contain the number of packets and bytes generated by the rule. Currently, there is only one interactive rule type, TCP Authenticator, which can autonomously generate a TCP packet with an RST flag to reset the connection and verify the liveness of the client trying to connect to a device in the protected network.

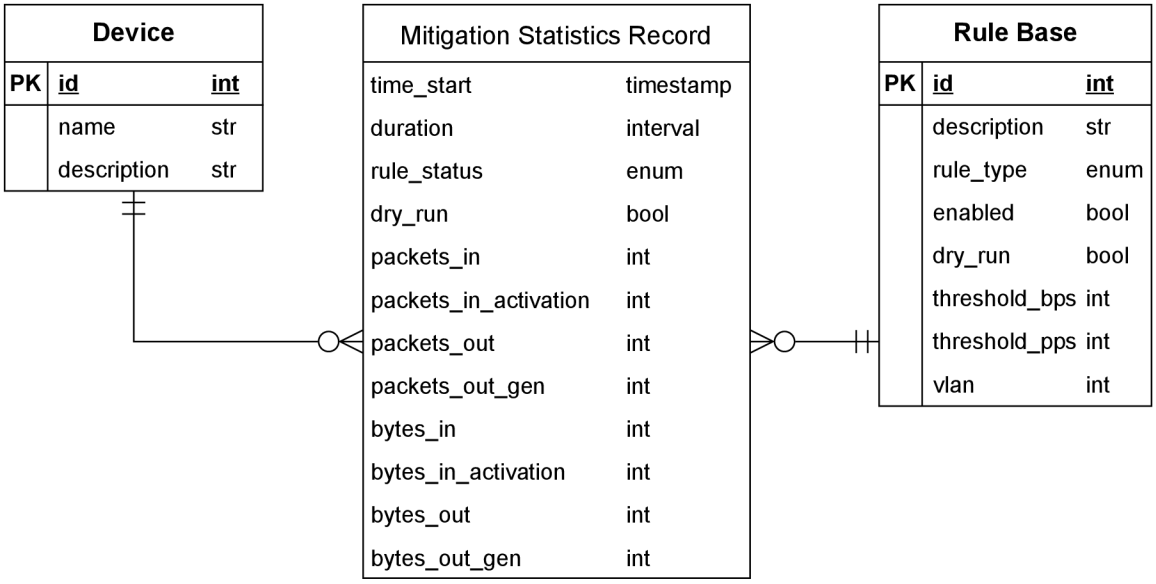


Figure 3.8: Diagram of the mitigation statistics record table

3.2.10 Complementary Tools

Finally, it is helpful to mention the tools that act as a user interface for a deployed DCPro DDoS Protector instance. The tools include the CLI[6] tool `dcproctl` and the web GUI ExaFS. ExaFS is a web GUI originally developed as a network management system over ExaBGP, with a complex user authentication system and more features. As part of the DCPro DDoS Protector project, an extension was developed, described in [31], which is used for the configuration of a DCPro DDoS Protector instance. ExaFS makes it possible to display graphs, showing user-important information extracted from the mitigation statistics. The graphs can be shown for statistics collected by any rule, on any Mitigator device. Both ExaFS and `dcproctl` use the `dcproapi` library to connect to the database that stores mitigation rules and statistics. The `dcproapi` library serves not only as a base for any tool that provides a user interface but is also used by all daemons and other utility tools and scripts written in Python, such as the ADT and KitNET daemons, and the database management tool `dcpro_db`, which manages database initialization and migrations.

[6] command-line interface

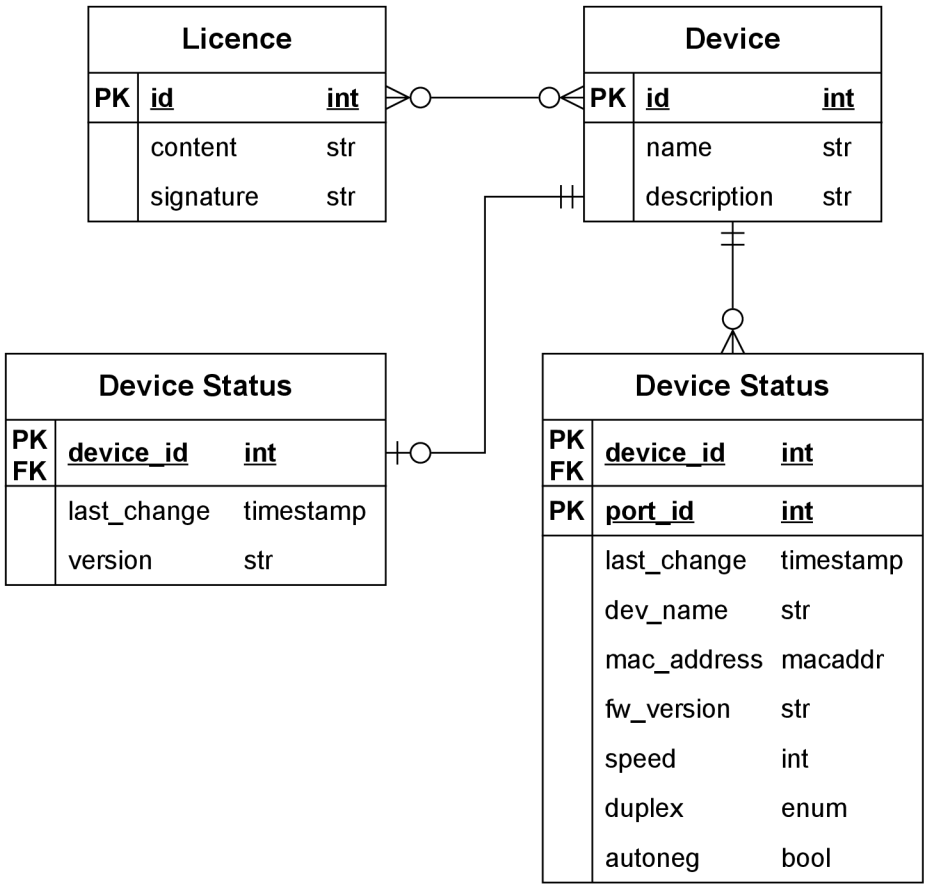


Figure 3.9: Device schema

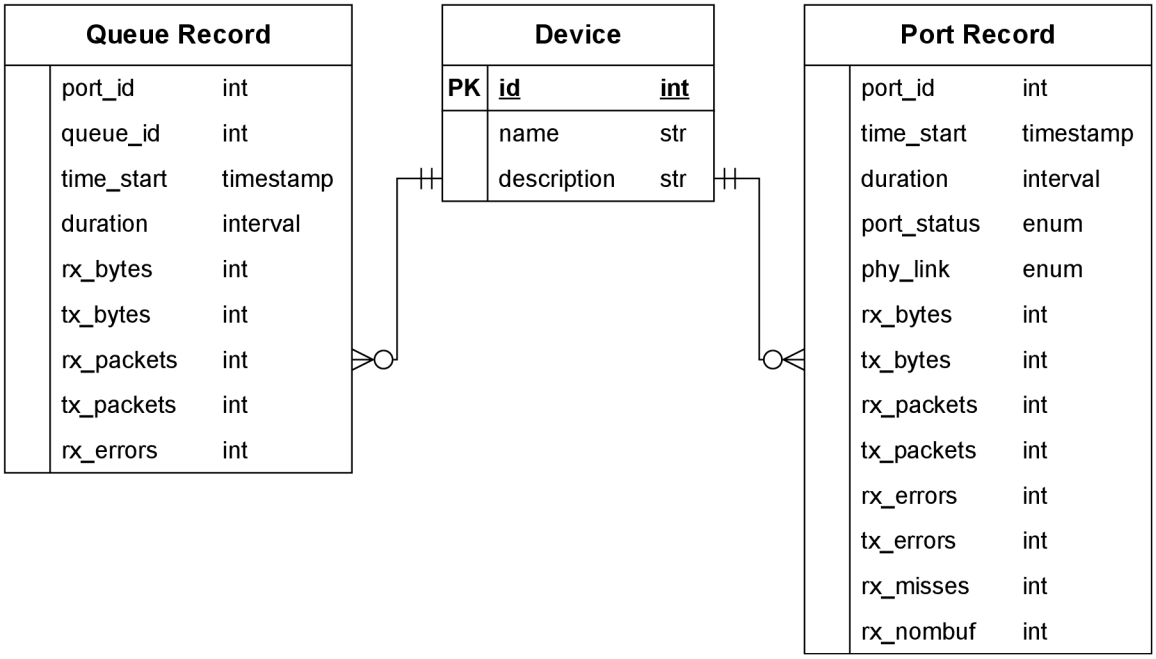


Figure 3.10: Device Statistics schema

Chapter 4

Solution Design

The task of this work, as set out in Chapter 1, is to integrate the sFlow exporters, IPFIXcol, the NEMEA framework and the DCPro DDoS Protector to form a comprehensive and distributed system for the mitigation of DDoS attacks. Currently, the DCPro DDoS Protector architecture does not have access to any network traffic that was not redirected to it. It is not desirable to direct all traffic to the DCPro DDoS Protector, as it would increase the latency and possibly cause legitimate traffic to be classified as a false negative and dropped. This problem can be solved by integration with the NEMEA system, so that data processed by any existing running NEMEA instance can be used in the DCPro DDoS Protector context. Secondly, the solution should also contain integration with sFlow exporters, as a variety of basic network devices, such as routers, can export sFlow.

The integration of the wider DCPro DDoS Protector architecture with the NEMEA and sFlow exporters should produce several results. Firstly, the administrator of the DCPro DDoS Protector should be able to broadly access the network traffic data processed by NEMEA, ideally in a format similar to how the DCPro DDoS Protector statistics are shown. In case of sFlow, the format question is simpler as sFlow messages contain individual sampled packets. Secondly, there should be a way to distinguish data coming from different sources, probes, routers, etc. What is especially important is to separate the inner network nodes from the gateways that connect the CESNET network to the outer web. Third, it should make it possible to use the traffic data from NEMEA to train the Machine Learning models to be used in the DCPro DDoS Protector mitigation methods which use Machine Learning.

4.1 Current status

Currently, all individual parts of the proposed architecture are deployed in the CESNET network. The IPFIXcol2 flow collector is deployed in multiple instances, most of which collect real-world traffic. Virtually all of those instances offer a TRAP interface, which can be utilized by any NEMEA module as a source of data in UniRec format. Although there is currently no deployed collector for sFlow messages, many of the routers used in the network can be configured to export sFlow.

NEMEA Framework is also deployed in various configurations in many instances. The highly modular nature of the NEMEA framework makes it very easy to add new modules to already running instances. In addition, there already exists a single link to the DCPro DDoS Protector architecture, which is the ADT Daemon briefly discussed in Subsection 3.2.6. The

ADT Daemon launches two NEMEA modules for every ADT Flow; `unirecfilter`, which is used to filter incoming UniRec data, so that it matches the IP addresses and ports specified in the ADT Flow rule, and a custom module, which gathers the filtered UniRec data and saves them into a `pandas dataframe` object, which it then uses to generate a BPF and a corresponding filter rule for the DCPro DDoS Protector.

The DCPro DDoS Protector is also deployed, processing live data for testing purposes. However, studying the current deployment of the DCPro DDoS Protector may provide the least insight of all the systems discussed here, simply because it is located at the end of the proposed chain of integration, and the only feature missing from a DDoS Protector device run in a testing environment is the ability to filter live data. Although it might be desirable to test the eventual solution in filtering live data, for the purposes of this thesis, it will not be discussed.

4.2 General Design Considerations

The code produced as part of the thesis will be used by the DCPro DDoS Protector development team to further develop the tools and take inspiration from. Thus, the most basic consideration in devising the design of the solution was to explore the options of the implementation of the current architecture, so that the solution requires as few changes as possible and that the concepts in the solution implementation are easily understood by the members of the DCPro DDoS Protector development team.

Prototypes

It was decided that the development carried out as part of this thesis should focus on producing working prototypes. While it is desirable to create fully working systems and tools, it is expected that some modules and systems may need a lot of additional work to cover all the possible use-cases, such as covering all the possible sFlow packet types. Instead, the solution proposed and implemented as part of this thesis should focus on finding the optimal way to connect the systems, so that it becomes possible to study its drawbacks and venues of future development.

Reuse of Already Existing Tools

A special focus has been placed upon the reuse of already existing resources, be they libraries or already used algorithms, such as the one used in the ADT daemon. The reasoning behind this idea is tied to the fact that the designed and implemented prototypes should be mainly comprised of the implementation of the actual new algorithm. Any other code produced should either come from the use of libraries already utilized in the development of the DCPro DDoS Protector and the various supporting tools, or it at least should reflect already existing patterns in the system, so that the structure is familiar to the developers who might be interacting with it in the future.

Reusability and Modularity

The last consideration connects to the previous two. As most of the implementation will be made up of prototypes mostly unsuitable for production deployment, the high modularity of the solution makes it possible for the various modules to be reused as part of the future

development of the production implementation. To achieve modularity, the various proposed parts should be standalone programs, with little dependencies on each other. While the design involves a specific use-case for the whole architecture, the modules should be general enough to allow for various future modifications to the architecture. Some parts or whole prototypes might be reused as they are, while some modules might be connected in a different way than what is proposed in this chapter or replaced altogether. The changes to the existing code and the modules created as part of this thesis should therefore aim to be as nonspecific as possible, especially the interfaces and other parts, which will interact with the surrounding systems and modules.

4.3 Data Format and Source Identification

Since the aim of this thesis is to integrate the NEMEA system, IPFIXcol, sFlow exporters and DCPro DDoS Protector, it is necessary to decide the formats for the exchange of data between the different systems. We can generalize the various used formats into two categories based on their content. The first category is packet data. This includes actual packets processed by DCPro DDoS Protector, as well as PCAP files produced by Packet Capture mitigation rules, and the packed samples captured using sFlow. The other category is flow data, which includes IPFIX, NetFlow, and UniRec data formats. We can assume the formats inside each category are fully compatible, and it is possible to transform the data from one format to another without a significant loss of information, which would be relevant for the purposes of this thesis, i.e. the display of statistical data to the administrator of DCPro DDoS Protector, and the use in the training of the ADT machine learning models.

Finally, the format in which the statistics will be stored and displayed to the administrator had to be decided. Since the tools used for the visualization of statistics in the form of graphs, such as ExaFS, already use the format of statistics used by the DCPro DDoS Protector, it was decided to use this format. To do this, the various data formats needed to be converted. For sFlow, the conversion is relatively simple, as the same mechanism which is used for live packets in the DCPro DDoS Protector can be used for the sampled packets as well. However, flow data can generally be converted to the format, as the mitigation statistics also aggregate the total numbers of bytes and packets per certain interval. However, such a conversion inevitably leads to a loss of information or an imprecise reconstruction of information lost by the initial aggregation into flows.

Because different protocols and probes produce data of various quality and granularity, it is important to pass the information about the specific source of data to the administrator. A source of data in this context is an exporter that uses any of the aforementioned protocols. Because it is expected that the same traffic may pass through multiple exporters, it is also important to distinguish between them, so that there is as little data duplication as possible. The considerations are explained in more detail in the following subsections.

4.3.1 Data Sources

In the proposed architecture for a distributed DDoS protection system, there are multiple sources of network data. Primarily, it is the DCPro DDoS Protector Mitigators, which process a stream of packets which are redirected to it, and periodically produce statistics, which contain the information on how many packets passed through and how many were dropped by the Mitigator. However, the deployment of a Mitigator instance is costly, and the monitoring data produced by the widely used protocols IPFIX and NetFlow contains

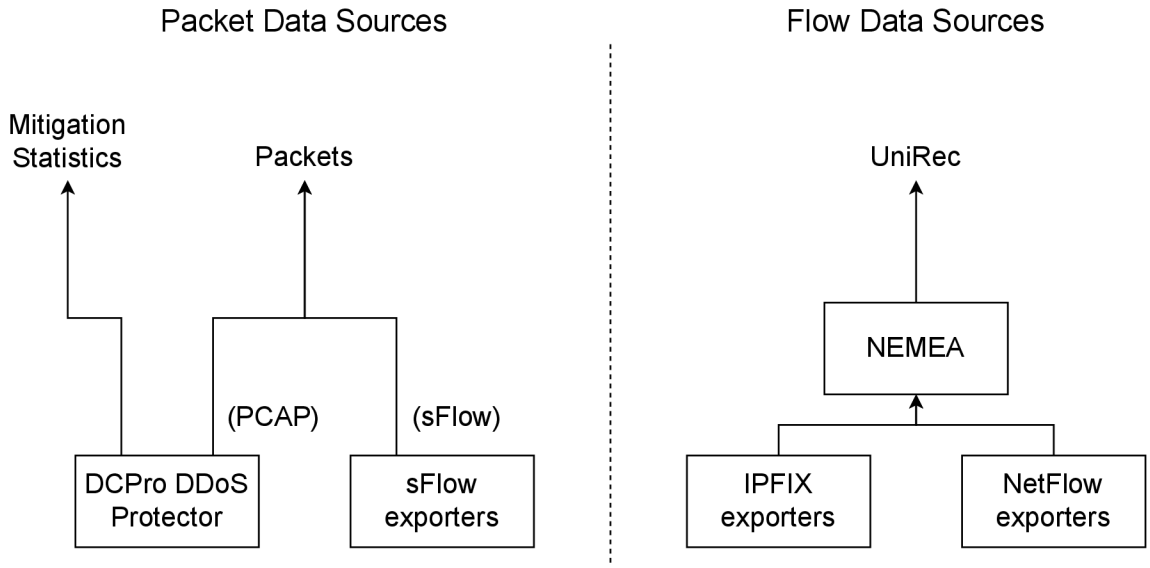


Figure 4.1: Overview of the considered data sources

information valuable to the administrator. In the CESNET network, the flow data in both formats is collected using the IPFIXcol2 collector. The IPFIXcol2 collector offers multiple ways to store and/or export data. One of those ways includes writing the data converted to the UniRec format to a TRAP interface, which is used to transmit data between modules in the NEMEA framework. Finally, the last source of data is the sFlow protocol, which enables ordinary network devices, such as routers, to periodically export sampled packets. According to the initial source of the network monitoring data, we may broadly divide it into two categories, flow data and packet data. The data sources and the formats they use are shown in Figure 4.1, which lists the flow data sources and packet data sources separately.

It is necessary to represent the individual network monitoring data exporters in the DCPro DDoS Protector database so that it is possible to distinguish between them. This should make it possible to display statistics collected on specific devices. The issue is that when a packet passes through several routers, it is registered and aggregated on each of them. The administrator should be able to show statistics exported only from particular routers, such as those on the perimeter of the network, to eliminate data duplication, or to monitor the traffic directed to a specific protected object inside the network.

Packet Data Sources

Packet data in the context of this thesis is any data collected by network devices which contain individual packets. From the technologies discussed in this section, DCPro DDoS Protector Mitigator and sFlow-compatible devices function as sources of packet data.

DCPro DDoS Protector Mitigator devices might be launched with variable configuration of internal modules. This allows the devices to be optimized for the collection of statistics. By dropping all packets just after statistics are collected, the total throughput might increase by reducing the total computing power required for the system to work. Due to this, a Mitigator device can be effectively used as a network traffic probe. DCPro DDoS Protector devices also support packet capture at the same time as statistics collec-

tion. This means that the processed packets not only contain useful network monitoring information, but may be used for the training of complex machine learning models, which require additional data present in the packets but removed in flows.

The other source of packets is the sFlow protocol, or more accurately, the various devices producing sFlow messages. The sFlow protocol does not have the same restrictions for deployment as the DCPro DDoS Protector, since it does not cause the exporter device to significantly slow down the incoming traffic and is much less resource intensive to run, since the DCPro DDoS Protector requires a dedicated machine, while sFlow can be configured on many existing routers. Compared to the DCPro DDoS Protector is that the sampling rate is significantly lower. While a DCPro DDoS Protector probe will process all of packets from an entire network subset if traffic coming to or from a certain subnet is redirected to it, an sFlow exporter will always have access to the entire network traffic but will only export every n-th packet.

From the description provided here, it should be rather obvious that both approaches for the collection of network monitoring data have their uses. It is fully expected that both types of data sources will co-exist in a single distributed system. However, the difference in their function makes it important to distinguish between the two sources when communicating the collected monitoring data to the user. Especially the sampling rate for the sFlow exporter and the portion of traffic redirected to a DCPro DDoS Protector probe.

Flow Data Sources

The other type of network monitoring data source is represented by the IPFIX and NetFlow protocols. Like sFlow, those two protocols are natively supported by a myriad of ordinary network devices. It is important to distinguish these data from the monitoring data produced using sFlow or DCPro DDoS Protector devices, since the conversion of packets into flow records neccessitates the loss of important information. This makes it impossible to deduce information, such as the number of packets or bytes that would be dropped by a DCPro DDoS Protector device, which is not an issue for sFlow. Generally, the relationship between bytes and individual packets is completely lost in the compression to flows, as well as the information about the time when the specific packets were recorded.

Both IPFIX and NetFlow typically aggregate packets into flows in a 5-minute window. This is generally enough to detect a DDoS attack and effectively mitigate it while it is still ongoing, as most DDoS attacks typically last for hours [54]. However, it is still useful to notify the administrator as soon as possible. What helps is the fact that IPFIX and NetFlow export the flows earlier once certain thresholds on packets are crossed, so a basic volumetric attack becomes apparent sooner than after five minutes. It is impossible to reconstruct the original information in the same resolution as is offered by DCPro DDoS Protector or sFlow, so the only solution is to at least inform the user about this fact. Additionally, we need to consider that the collection of the flows and their conversion to the common UniRec format causes additional variable delay.

Since both IPFIX and NetFlow messages are converted to UniRec format to be later processed by NEMEA modules, and since our main focus is on the DCPro DDoS Protector environment, we can consider the IPFIX and NetFlow messages covered as part of a single data source, which is the NEMEA system, or more precisely, a TRAP interface. TRAP interfaces were described in Subsection 2.6.1.

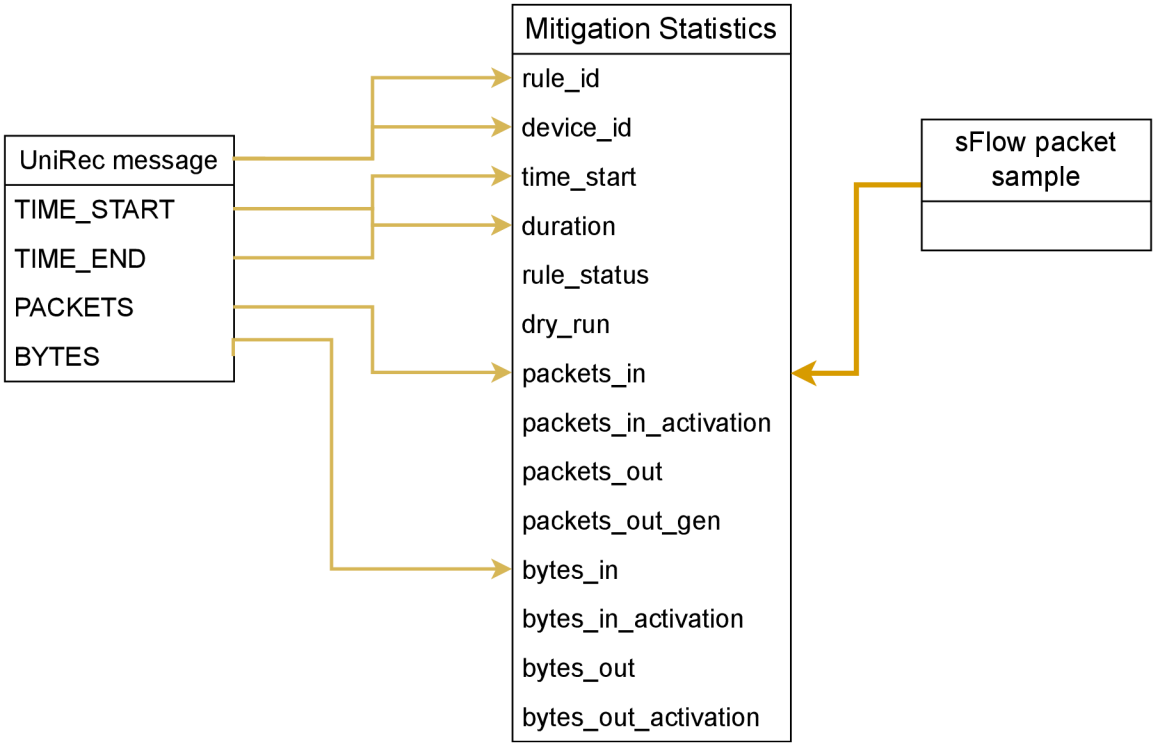


Figure 4.2: Relationship between the UniRec, sFlow and DCPro DDoS Protector mitigation statistics formats

4.3.2 Data Format

Once the sources of the network monitoring data were identified and classified, it was necessary to analyze the formats of data exchange and data storage. The data formats currently in use by the tools mentioned in the previous subsection were already described in the sections on the various tools, in Chapter 2 and Chapter 3. To achieve the aim of this thesis, there are two considerations. First, what format should serve as the final data storage format, which will be used to display the information to the end user, and secondly, how do the formats correspond to each other, to see how much information may be lost in transmission and what are the considerations for displaying data from different types of sources.

Because the DCPro DDoS Protector already utilizes a web GUI called ExaFS, it makes sense to convert all the network monitoring data to the format used in the DCPro DDoS Protector database. ExaFS provides graphs for every rule that show how many packets and bytes were dropped by the rule or passed through. The DCPro DDoS Protector database contains more information, such as the number of packets generated by the rule, in case it is a rule type that uses an interactive method to validate incoming traffic, and the number of packets which count towards the activation of the rule, which is important in rules with more complex methods.

As visualized in Figure 4.2, UniRec and sFlow contain information, which can be translated into the DCPro DDoS Protector mitigation statistics format. In case of sFlow, since sFlow contains sampled packets, and the DCPro DDoS Protector mainly interacts with the packet headers, which are left untouched by the sFlow sampling mechanism, the same statistical data which are produced by DCPro DDoS Protector from live packets can be

extracted from packets sampled and collected using the sFlow protocol. On the other hand, flows aggregate information from multiple packets which share some attributes. Because of this, it is impossible to extract the same amount of information from them. This affects the counters for the number of packets that were not dropped and the number of packets that counted toward the activation of a more complex rule. However, the most important values, such as the starting timestamp and duration, can be calculated from the values contained in a UniRec message. In both cases, the device identifier of the converted data represents the probe which collected the UniRec or sFlow data and exported the converted statistics in the DCPro DDoS Protector format.

The sFlow data format requires a simpler conversion, as the protocol transmits packet data. We can therefore reuse the methods already used by the DCPro DDoS Protector for the classification of packets by the mitigation rules, as well as the mechanism for the mitigation window, which aggregates the relevant values taken from multiple packets into a single statistical record.

For UniRec, the method is more complex. There are two central issues. One is that a flow can be collected inside up to a five-minute window. The DCPro DDoS Protector usually exports the statistics to the database in one- or two-second intervals. Therefore, while it is possible to convert the flow directly to the DCPro DDoS Protector format as is, it would result in a much lower resolution, and would produce potentially misleading data, since the DCPro DDoS Protector statistics displayed in ExaFS do not indicate in any way the duration of the collection window, since it is implied the duration of all of them is identical. The solution may be in splitting the collected flow into segments of identical duration, such as 5 seconds. In this way, we can improve the granularity of the data, which will better correspond to the graphs produced from the traffic processed by sFlow or DCPro DDoS Protector. Since not all the fields of a mitigation statistics record can be reconstructed from the fields of a UniRec message, it is a matter of consideration whether to use the same database table for both mitigation statistics and the statistics extracted from UniRec data, or to define a new database table with a reduced number of columns, and possibly a different configuration of the TimescaleDB `hypertable` structure, which is used by the mitigation statistics table to efficiently store time series data.

4.3.3 Source Identification

In the context of integrating the DCPro DDoS Protector with both the NEMEA framework and the sFlow data from routers, it becomes crucial to establish a mechanism for accurately identifying the sources of network traffic data. This identification is essential because routers within the network may exist at various locations, some at the edge of the network, while others are positioned internally. Consequently, the same traffic can be captured by multiple routers, leading to possible duplication of data.

To address this problem, we need to find a source identification mechanism within the integrated system. This mechanism should differentiate between data originating from different sources, such as probes, routers, and other network nodes. Fortunately, both NetFlow and sFlow packets contain the source identificator.

The source identification process should ideally align with the existing structure of the DCPro DDoS Protector statistics. This ensures that administrators can easily interpret and analyze the network traffic data processed by NEMEA in a manner consistent with the overall monitoring and mitigation framework of the DCPro DDoS Protector. Since the DCPro DDoS Protector controller already supports having several instances, each with an

identification in the devices table, we can extend this way of identification to the network traffic data sources.

All of the discussed network traffic export protocols, sFlow, NetFlow, and IPFIX, support the identification of the exporter. The UniRec format used in NEMEA uses dynamic templates for the transmission of data between modules and may therefore be very easily extended to include the identifier. IPFIXcol2 has access to the NetFlow and IPFIX identifiers and actively uses them, but does not export them to NEMEA at the moment. Lastly, what needs to be done is find a way to represent the various exporters in the device table inside the DCPro DDoS Protector database. Since mitigation statistics received from the various Mitigator instances are already identified by the device on which they were collected, this is a natural extension of the concept of devices in the DCPro DDoS Protector architecture.

4.4 sFlow Processor

To process sFlow packets to generate statistics in the DCPro DDoS Protector format, and to use it to train the models of the machine learning rules, a component was devised, conceptually called sFlow processor. Since sFlow is functionally similar to the capture of live packets, the solution should aim to include as many components of the DCPro DDoS Protector, which already implement the necessary methods.

There was some consideration on how to implement the sFlow processor. To fully reuse most of the already existing systems, it would be best to introduce a sFlow parser module, which would act as an additional stage in the DCPro DDoS Protector modular pipeline. The sFlow parser would precede the packet parser stage, as after the sampled packet is parsed out of the sFlow message, it is essentially a normal packet.

However, this approach posed several difficulties. First, the code of the internal parts of the DCPro DDoS Protector is licensed as proprietary and cannot be modified as part of this thesis. Secondly, even if we utilized the modular nature of the DDoS Protector, where the individual modules and stages of the pipeline can be linked together from libraries, which only reveal the public interface, the internal structure is heavily optimized toward packet processing. In case an incoming sFlow packet contains multiple packet samples, there is no way to effectively split them and direct them to the next stage of the pipeline as several individual packets.

Due to this, it was decided to implement an sFlow parser which would function as a server listening for sFlow packets. The sFlow parser should then parse out any sampled packets and write them to a virtual TAP/TUN interface. This is because the DCPro DDoS Protector as a DPDK application is expected to be bound to an interface. The sampled packets would then arrive at the Mitigator instance as ordinary packets. As a further improvement, the modular nature of the Mitigator pipeline can be leveraged to improve the efficiency, as several stages can be considered irrelevant when used on packets sampled as monitoring data. The only additional consideration is the sFlow exporter identification mechanism. However, this is largely a secondary issue.

4.4.1 TUN/TAP

TUN/TAP, described in [26], is a virtual network interface for user space programs provided by the Linux kernel. It can be dynamically created and is closed once the file descriptor is closed. TUN/TAP refers to a single concept; the difference is that TUN interfaces are used

to write or read IP packets, while TAP interfaces are used to write or read Ethernet frames. Since the DCPro DDoS Protector expects to encounter an incoming Ethernet frame, the solution needs to use a TAP interface.

TAP interface outwardly appears as a normal interface. It contains all the functionality which could be used by the DCPro DDoS Protector, such as allowing for a BPF to be bound to it. Additionally, TAP interfaces can be configured to be multi-queue. In this way, the DCPro DDoS Protector can be bound to a single TAP interface, while there may exist multiple sFlow parsers, each writing the parsed out sample packet to a single queue. Thanks to this, it is possible to achieve production rate throughput even with a relatively unoptimized implementation.

4.4.2 sFlow packet parser

The sFlow parser component itself should act as a simple server, listening on UDP port 6343, which is the port dedicated to receiving sFlow messages. The parser should then parse out any sampled packets from the message. In case the message does not contain a packet sample, but is instead of any other sFlow message type, it should be ignored. While messages containing the counter values of network devices can be useful in the future, the aim of this work is more directed towards giving the DCPro DDoS Protector administrator essential information about the traffic which is not being redirected towards the DCPro DDoS Protector at the moment.

The prototype can also focus on the implementation of the parsing of sFlow messages contained within IP version 4 packets. The reason is that most of the network devices currently used in the CESNET network, which could export sFlow messages, use an IPv4 address.

Additionally, focusing on IPv4 only makes it possible to relay the address of the server, since the 32-bit IPv4 address can be fully contained within a 48-bit MAC address. Every DCPro DDoS Protector stage has full access to a packet that is being processed, so the module for exporting statistics can be modified to parse out this information and find the respective database record for the sFlow exporter identified by the IP address. However, this functionality will require additional changes on part of the DCPro DDoS Protector, possibly even a new statistics exporter module, as the DCPro DDoS Protector only attaches its own device ID. This new DCPro DDoS Protector instance can be called the DCPro sFlow Probe, as it will only be used for the export of statistics from collected sFlow-sampled packets.

4.4.3 DCPro sFlow Probe

The DCPro DDoS Protector Mitigator pipeline configuration, shown in Figure 3.2, contains many stages, which are concerned almost exclusively with the purpose of filtering live network traffic. Therefore, for the purposes of sFlow processing, it was decided to create a variant of the Mitigator pipeline configuration, which would omit certain stages that only make sense when the Mitigator runs on live data and not past sampled traffic. This reduced variant is called DCPro sFlow Probe. However, it should theoretically be able to process any kind of packet data, such as replayed PCAP files, if the need arises in the future.

As described in the previous subsection, the DCPro DDoS Protector instance used to process sFlow will differ from an ordinary DCPro DDoS Protector in that it will have to keep the counter values not just for every rule, but for every sFlow exporter and rule combination. This will significantly increase the memory requirements for the statistics

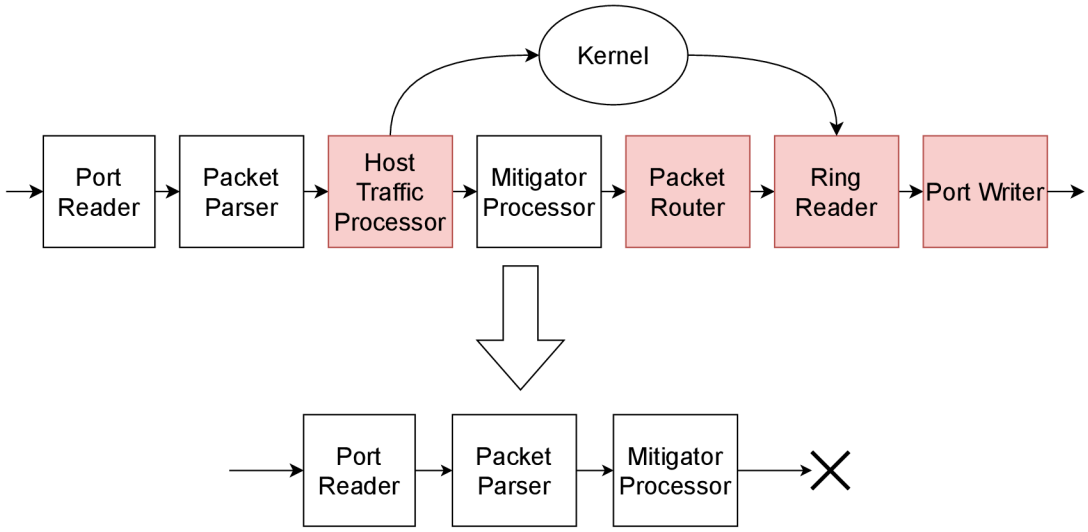


Figure 4.3: DCPro sFlow Probe pipeline configuration

exporter module. To make sure that it does not exceed an acceptable level, we can perform a simple calculation to determine the possible future memory size.

Currently, the counters take up 32 bytes of memory, namely:

- 4 B for incoming packets
- 4 B for incoming packets, which count towards the activation of the mitigation method
- 4 B for outgoing packets
- 4 B for packets generated by the mitigation method
- 4 B for incoming bytes
- 4 B for incoming bytes, which count towards the activation of the mitigation method
- 4 B for outgoing bytes
- 4 B for bytes generated by the mitigation method

If we assume the number of supported sFlow exporters will be in the lower tens, the maximum a hundred, since the most of the network traffic information needed to successfully mitigate an incoming DDoS attack will be collected on the peripheral network devices, then we can calculate the following equation:

$$M = C * E \tag{4.1}$$

Where M represents the new total memory requirement, C is the memory taken up by the counters in bytes, and E is the expected number of sFlow exporters. When we assume the number 100 as the upper limit on the expected number of sFlow exporters, and the total size of the counters 32 B, the equation yields the value of 3.2 kB, which is fully acceptable.

However, to counteract the additional overhead introduced by this mechanism, we can utilize the modular structure of the DCPro DDoS Protector to remove unused pipeline

stages, since the DCPro sFlow Probe will not need to deal with any host traffic, nor is it expected to send the packets which pass the mitigation modules back. The stages are shown in Table 4.1, together with their inclusion status and the reason for the decision.

Table 4.1: Pipeline Stages and Their Inclusion Status

Stage Name	**Status**	**Description**
Port Reader	Included	Port Reader stage is required for the sFlow Probe to receive packets from the interface.
Packet Parser	Included	Similarly, the Packet Parser stage is included, as it contains the module for the classification of packets by matching rules.
Host Traffic Processor	Excluded	This stage directs traffic intended for the host machine to it. Since sFlow does not contain live traffic, it serves no purpose.
Mitigator Processor	Included	The Mitigator Processor stage is the core of the Mitigator pipeline, as it is the stage where it is decided whether the packet should be dropped or allowed to pass. The stage is left in the sFlow Probe, as it also generates the statistics.
Packet Router	Excluded	The Packet Router serves to determine and update the MAC address for the packets that successfully passed the Mitigator stage. Since no packets leave the sFlow Probe, this stage is also removed.
Ring Reader	Excluded	The Ring Reader reads packets generated by the kernel from a packet ring and sends them to the final Port Writer stage. Since Host Traffic is not supported, this stage is also removed.
Port Writer	Excluded	The final stage writes the packets back to the network interface. Since the sFlow Probe receives packet samples, there are not packets to write back.

The removal of certain stages can significantly help the throughput of the instance, as the number of removed stages is around half of the total. The new pipeline configuration is visualized in Figure 4.3, together with a comparison with the complete Mitigator pipeline, where the removed stages are marked in red.

Additionally, we can further reduce the memory capacity and computational power demands of the DCPro sFlow Probe by using only the Filter mitigation module. We can do this since the expected use case now is to see how many packets and bytes were matched by the rule and whether the packet or byte thresholds were exceeded at any time. Packet matching is performed collectively for all rule types, regardless of the rule type, and only requires the configuration data stored in the `rule_base` table. As the Filter rule type has the simplest implementation and the least memory requirements, it is the perfect candidate.

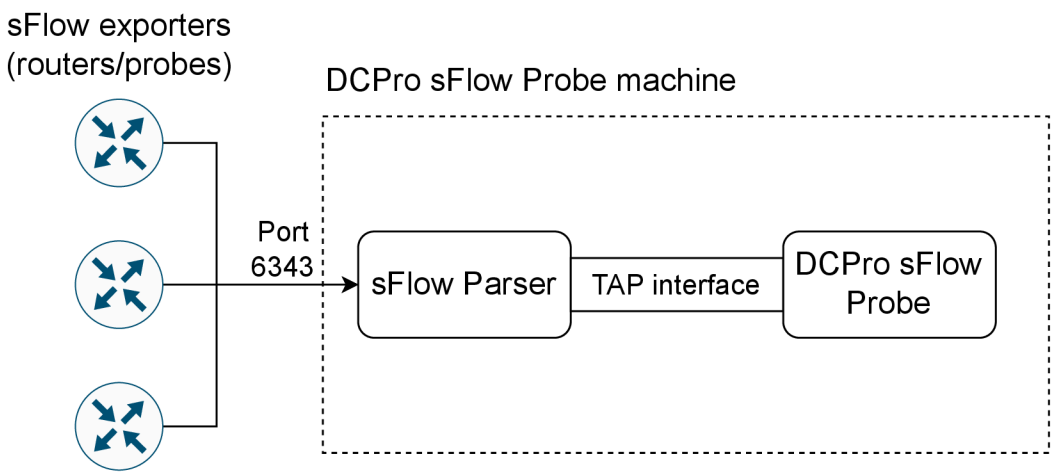


Figure 4.4: sFlow Processor architecture

Thanks to this, even rules which would need to allocate additional memory for their internal supporting data structures, such as a table containing IP addresses for Amplification and SYN drop rule types. The queries used by Mitigator to load the mitigation rules are contained in a SQL file. We can use the Filter mitigation module on each rule by altering the SQL queries used by the DCPro DDoS Protector to load every rule as a Filter rule, and to include a default BPF value of `null`, since it would otherwise be missing. Because mitigation rules are applied in sequence and the packets dropped by one rule are not visible to the next one, all the rules are to be in a dry-run mode, to ensure the administrator can see the effectiveness of every single rule alone.

As an exception, Packet Capture and machine learning rules can function without any modification. Machine learning rules are not directly loaded into the DCPro DDoS Protector anyway, since they are only utilized by the machine learning daemons. The only issue may be the inclusion of a manufactured MAC address in captured PCAPs. However, it is highly unlikely that the MAC address fields will ever be used for DDoS mitigation, so we can safely ignore this drawback.

4.4.4 sFlow Processor Conclusions

Thanks to the proposed measures, the DCPro sFlow Probe configuration for the DCPro DDoS Protector, used in conjunction with the sFlow parser, should fulfill all the needs and considerations set out in the introduction of this chapter. It is designed to be a relatively lightweight system compared to a DCPro DDoS Protector Mitigator, and to reuse as much existing technology as possible. The proposed design should be simple to understand for any DCPro DDoS Protector developer. It is expected that the throughput of the solution may be sufficient enough for this system to be deployed in production setting, since even if the sFlow packet parser is not implemented very efficiently, if we use the multiple queues of a TAP interface, we can solve any throughput disparity between it and the DCPro sFlow Probe.

The final design of the sFlow Processor architecture is shown in Figure 4.4, where multiple sFlow exporters, such as routers, supply the machine on which the DCPro sFlow Probe is running through the sFlow Parser. The sFlow Parser listens for incoming sFlow

messages on the port assigned for the collection of sFlow, which is port 6343, and sends the sampled packets to the DCPro sFlow Probe using the TAP interface. If the volume of sFlow is too high, multiple NICs[1] can be used on the machine, and a sFlow Parser instance can run on each of them. The sFlow Parsers can then share a multi-queue TAP interface, so that all extracted packets arrive on the same DCPro sFlow Probe.

If the issue with the identification of the sFlow exporter is not resolved during the implementation of the prototype, we can assume that the DCPro sFlow Probe also acts as a singular exporter. Additionally, every sFlow exporter has a set sampling rate, which is useful information for the administrator. Since the sampling rate is a relatively static attribute of an sFlow exporter, it can be stored in the database together with a record for the exporter itself.

4.5 DCPro NEMEA Daemon

The other conceptual module is the DCPro NEMEA Daemon. DCPro NEMEA Daemon should function as an ordinary NEMEA module. However, due to the expected amount of traffic, it should implement a parallel mechanism similar to the mitigation windows of the DCPro DDoS Protector. In this way, it should be possible not to export statistics for every UniRec record but instead one or more aggregated records once a given interval.

Because of this, it cannot function as a server like the sFlow parser, which purely reacts to any incoming packets, but must also implement a complex main loop, which includes periodic execution of the statistics export function. Thankfully, we can utilize the existing `daemon` module of `dcproapi` Python library. Because of this, the prototype can be implemented quickly and function as a useful demonstrator, since only the modules that actually process UniRec data will need to be implemented, as the basic daemon structure is already included in the `daemon` module. However, unlike the sFlow Processor, it is unlikely that the DCPro NEMEA Daemon will achieve the throughput necessary for its effective production deployment.

To achieve higher levels of throughput, a NEMEA module may be implemented in C, which removes much of the additional overhead caused by the use of Python and its runtime environment. However, the implementation of a fully working daemon in C, which would continuously aggregate the statistics, as well as periodically export them and train the machine learning models, would be exceedingly difficult and unwarranted. There would be many additional design considerations, which would require a lot of additional input from the members of the development team, since the project is still going to continue. To be implemented effectively, the daemon would need to reuse much of the components of the DCPro DDoS Protector, especially the packet classifier, which produces a list of rules which match the incoming packet. The packet classifier would need to be modified to accept UniRec data. Although this should not be a conceptual problem, as the packet classifier only uses the so-called packet features to classify the packet. The required values of the packet features data structure can be mostly constructed from data found in a UniRec record. However, the code structure of the current packet classifier implementation would need significant changes, which are beyond the scope of this thesis. The implementation of the accelerated packet classifier has been quite complex and has been the main aim of another thesis, as described in [51]. Therefore, the solution proposed here will focus only on a prototype that is most likely implemented using `dcproapi` and written in Python to

[1] Network Interface Card

reduce the development time. This also means that instead of an accelerated rule classifier, the prototype will aim to implement a slower but functionally identical version. In case it is later shown that the prototype may achieve sufficient throughput, the classifier can be modified and extracted to be used as a standalone C module with a Python interface.

4.5.1 Daemon Structure

The `daemon` module of the `dcproapi` offers a main loop component and an environment, which can be used to launch asynchronous routines just once, periodically or as a reaction to an external event. The routines may be launched as part of the main application thread, or in a separate process, since Python does not support genuine multithreading, due to the Global Interpreter Lock, as explained in [33] and [37].

The Daemon should be highly modular, with the main part being the event loop itself. The event loop registers events asynchronously and executes the respective servicing routines, one per iteration. This means that while the events are conceptually asynchronous, there is no parallelism, as everything happens in a single thread and is executed in the order in which it was requested.

4.5.2 Main Loop

The main loop should be used for two main functions. First, it should continuously collect the UniRec data in a list by repeating an execution of a simple task, whose only job is to run the `read` method or an equivalent on the TRAP interface. The TRAP interface, the list for the storage of UniRec records, and any other supporting data structures should be initialized during the initialization of the loop module. Second, it should periodically launch more complex routines for handling the collected data, such as matching the records with the mitigation rules, aggregation and subsequent export of the extracted statistics, and training of the machine learning models of the ADT Flow rules.

4.5.3 Statistics Export Module

The module for the export of statistics should be periodically launched from the main loop in a separate process. The only data required for the module to function are the UniRec data itself and the database connection parameters. Every time the method provided by the module is called, the module should load all the rules which are currently present in the database. To do this, it would be advisable to again use the methods for the retrieval of DCPro DDoS Protector database data offered as part of the `dcproapi` library.

Once the rules are loaded, the UniRec data is to be processed. Because a single UniRec record may contain data collected in an up to five-minute long interval, and the starting time of the interval may vary and may be delayed by several minutes, there needs to be an algorithm which would make it possible to aggregate the various records into a limited number of statistical records. This will inevitably also increase the granularity, while there will be no change to the accuracy of the data, since flows in general contain only a summary of a network traffic profile and not detailed information, like the time of the peak in the number of incoming packets. Just like the mitigation window mechanism of the DCPro DDoS Protector, the statistics from all the various UniRec records can all be aggregated into one or more intervals. It should be efficient to split the whole aggregation window into multiple intervals. In case a record is split over more than one interval, it's values also have to be divided by the number of intervals and distributed equally among them. Preferably,

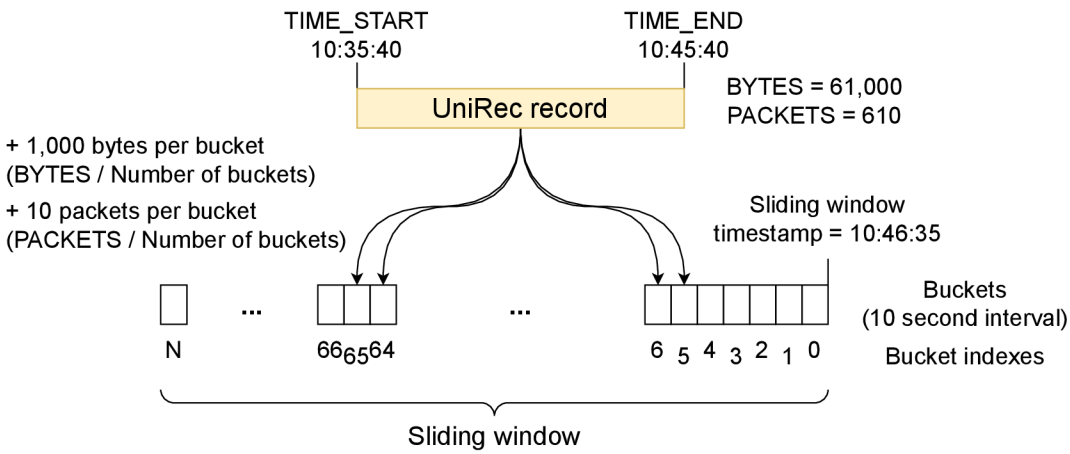


Figure 4.5: Nemea Daemon sliding window for the aggregation of statistics

those intervals may have length closer to a mitigation window, so that the administrator is presented with data of similar granularity to that of the DCPro DDoS Protector.

Therefore, we need to maintain a data structure for every rule, which contains byte and packet counters for every interval of the same duration in the last few minutes. The exact number of those intervals required must together add up to the equivalent of five minutes, since that is the maximum duration of a single UniRec record. However, UniRec records may be introduced with some delay. Therefore, if we assume that the delay for most records will not exceed two minutes, we need enough counters to cover a seven-minute window. It is possible then to calculate the number of counters needed for the data structure, by dividing the total length of the window, e.g. 7 minutes, with the length of a single interval, e.g. 10 seconds.

The intervals which are represented by the counters should ideally also be used in the main loop as the interval between the periodic execution of the statistics export routine. The structure then forms a sliding window that shifts every 10 seconds. While it might be required in the future to restrict the window to a fixed length, the Python implementation of the prototype should make it very easy to dynamically alter the size of the window structure in case a record, which is earlier than the lower bound of the window, is encountered. The sliding window is visualized in Figure 4.5, together with the distribution of an incoming UniRec record among the counters at the various intervals. The figure shows the UniRec record being delayed by 55 seconds compared to the timestamp of the end of the sliding window. The UniRec record, which contains 5 minutes of aggregated data, is then distributed into buckets equally, by dividing the aggregated numbers of bytes and packets by the number of buckets, and adding the result to the byte and packet counters inside every bucket. In this illustrative case, the increment value is 1,000 for every byte counter and 10 for every packet counter.

4.5.4 Machine Learning Module

The other module which has to provide a routine to be executed periodically by the main loop is the machine learning module. The proposed machine learning module mostly implements the same functionality as the NEMEA ADT Flow Daemon, effectively replacing it. The reason is that it is simpler to have a single modular system, which will read from a single

NEMEA interface and distribute an identical set of collected data to all different processing modules, rather than launching several separate NEMEA modules, each preceded with a `unirecfilter` module. Like the Statistics Export Module, the Machine Learning Module takes the data collected in the main loop and processes them in a separate process. Again, what is also needed are the database connection parameters. The only difference is that the only rules that are loaded should be those with type ADT Flow.

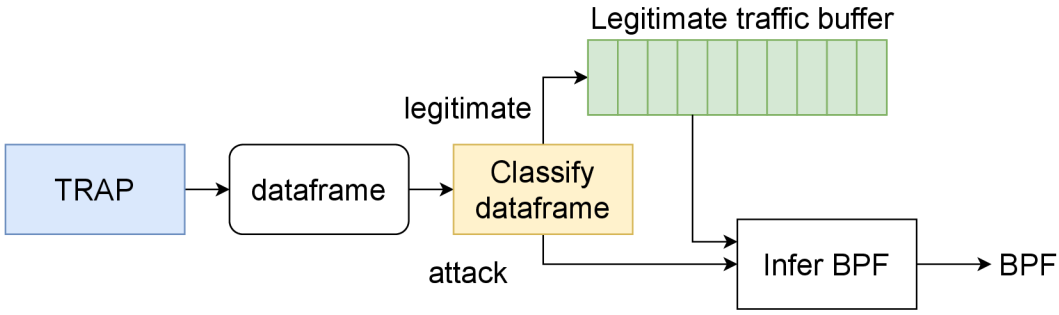


Figure 4.6: ADT Flow Daemon legitimate traffic buffer mechanism

Like the ADT Flow Daemon, the Machine Leaning module aggregates UniRec records into instances of the `dataframe` class provided by the `pandas` library. Since the creation of a BPF filter requires two dataframes as input, one containing legitimate traffic, the other malicious one, the Machine Learning Module needs to keep past dataframes. As shown in Figure 4.6, ADT Flow Daemon increases the accuracy of the generated BPF by storing up to ten dataframes with legitimate traffic. These dataframes can then be combined to increase the accuracy of the machine learning model. The proposed solution in the DCPro NEMEA Daemon Machine Learning Module should imitate this approach, as it has already been proven in deployment. The only difference in the proposed mechanism is that the NEMEA Daemon needs to keep the buffers for every ADT Flow rule, whereas the ADT Flow Daemon runs a separate instance for every rule.

Once new dataframes are constructed from fresh data and any are classified as containing attack traffic, they should be passed to the function implementing the actual machine learning technique, which is part of the internal `ml_module` library, and returns a BPF. Finally, for every generated BPF, a new Filter rule should be created and inserted into the database.

4.6 Overview of the Proposed Changes

Once the proposed changes are implemented, the DCPro DDoS Protector architecture should have full access to flow and packet traffic data from the entire CESNET network. Integration of real-time network traffic with machine learning mitigation methods could produce better results than is possible with only the traffic that is redirected directly to the DCPro DDoS Protector. The DCPro DDoS Protector administrator should be able to see all the collected network monitoring data in the same format as is produced by the DCPro DDoS Protector, including how much traffic was matched by every mitigation rule.

Chapter 5

Solution Implementation

The implementation of the proposed solution had a simple objective: to create one or more working prototypes of the designed modules and to test the feasibility of the algorithms. The basic task was to implement the algorithm to show its function and to analyze potential weaknesses and possible ways to mitigate them. The additional task was to measure the throughput of the prototypes and provide some demonstration of the implementation. Since both modules proposed in Chapter 4 work with continuous streams of data, the throughput of the applications is essential information, as it is not feasible to delay the execution or process the data in batches at a later time. The throughput data may be later used to determine which technologies should be used to develop a live version of the modules.

It is highly unlikely that a working efficient high-throughput solution could be implemented without serious effort from multiple team members. Therefore, the process of implementation was continuously discussed with the development teams of DCPro DDoS Protector, NEMEA Framework, and IPFIXcol. The immediate findings were shared with the respective team to ease any later developments and to leverage the knowledge and take into account the specific considerations of the various parties.

Most of the work was performed under the DCPro DDoS Protector development team, as it is the most affected by the output of this thesis. Thanks to this, the implementation of the prototypes received a regular stream of input and provided instant feedback for any implementation decisions. The prototypes were developed in several iterations. Every iteration, the code of the prototypes, and the output they produced was also reviewed by several of the team members.

5.1 Source Identifiction

As explained in Subsection 4.3.1, it is greatly beneficial for the end user to be able to clearly differentiate data coming from various sources, be they dedicated probes or ordinary routers configured for the export of statistical data.

For sFlow messages, this information is included only in the L3 packet header, as sFlow exporters are identified only by their IP address, which is also set as the source IP address of the packet containing the sFlow message. However, identification of sFlow sources is a task which does not need any modification of existing systems and thus can be fully implemented in the sFlow processor.

NetFlow and IPFIX messages contain the exporter IP address and ID. Currently, deployed NEMEA systems do not have access to this information within the UniRec records,

as IPFIXcol does not pass this information to the UniRec output template. Most NEMEA modules do not expect a static template structure for the incoming UniRec records. Especially utility NEMEA modules, which only modify the flow of records, such as `replay`, `multiplexor` or `flowcounter` modules. Other NEMEA modules tend to be implemented in a way where the module only operates with several required fields in the UniRec record and any other additional fields are ignored. Thanks to this, it was only necessary to modify the code of IPFIXcol UniRec exporter module.

Furthermore, as mentioned in Subsection 4.3.3, the database schema was modified, to include the information about the source of the statistical data. Since the DCPro DDoS Protector database already distinguishes statistics coming from different Protector instances, also called devices, most of the work was primarily on extending the functionality of the devices table.

5.1.1 Changes to IPIXcol

The changes required to be implemented for IPFIXcol were straightforward, as IPFIXcol has access to the whole contents of IPFIX and NetFlow messages, including the source IP address and the source identifier. Since UniRec supports dynamic templates, where a UniRec record can contain many different fields, and those fields are specified by both the sender and the recipient of the record, the only thing that needed to be done was to include the IP address and the ID of the source in the template for the output TRAP interface.

The specified changes were made exclusively in the code of the `unirec` output plugin. Since IPFIXcol converts both IPFIX and NetFlow to an internal representation modeled after IPFIX, the added fields are called `ODID`, which stands for Observation Domain ID of the exporter, and just like the respective field in an IPFIX message, and `EXPORTER_IP`, which is the source IP address of the packet that contained the IPFIX message. Firstly, both `ODID` and `EXPORTER_IP` required a mapping and a translation function in the `translator` component, which translates values from the internal representation to the UniRec format. As the field `EXPORTER_IP` is not included in the IPFIX message, but is retrieved from the L3 header, there also had to be a change in the message context given to the translator to also include the L4 and L3 headers that wrapped the message.

This change was minor and does not significantly affect the NEMEA and IPFIXcol instances currently deployed at CESNET, since the vast majority of NEMEA modules specify their own template format and do not interact with any additional UniRec fields. The changes were consulted and implemented with the help of the IPFIXcol development team.

5.1.2 Changes to DCPro Protector Database

The DCPro DDoS Protector controller database schema was then modified to provide a way for the administrator to see an overview of the devices which act as sources for statistics. The changes are shown in Figure 5.1, where the new tables and columns are shown in green. The devices table was generalized, and the additional information specific to the device type is included in respective tables. The types include `mitigator`, which represents all devices which were present in the table before the change, `sflow_exporter`, which represents any network device which exports sFlow messages, and `flow_exporter`, which represents a device exporting any kind of flow data, be it NetFlow or IPFIX. The fact that the data arrive through the use of NEMEA in UniRec format is omitted, since it has little

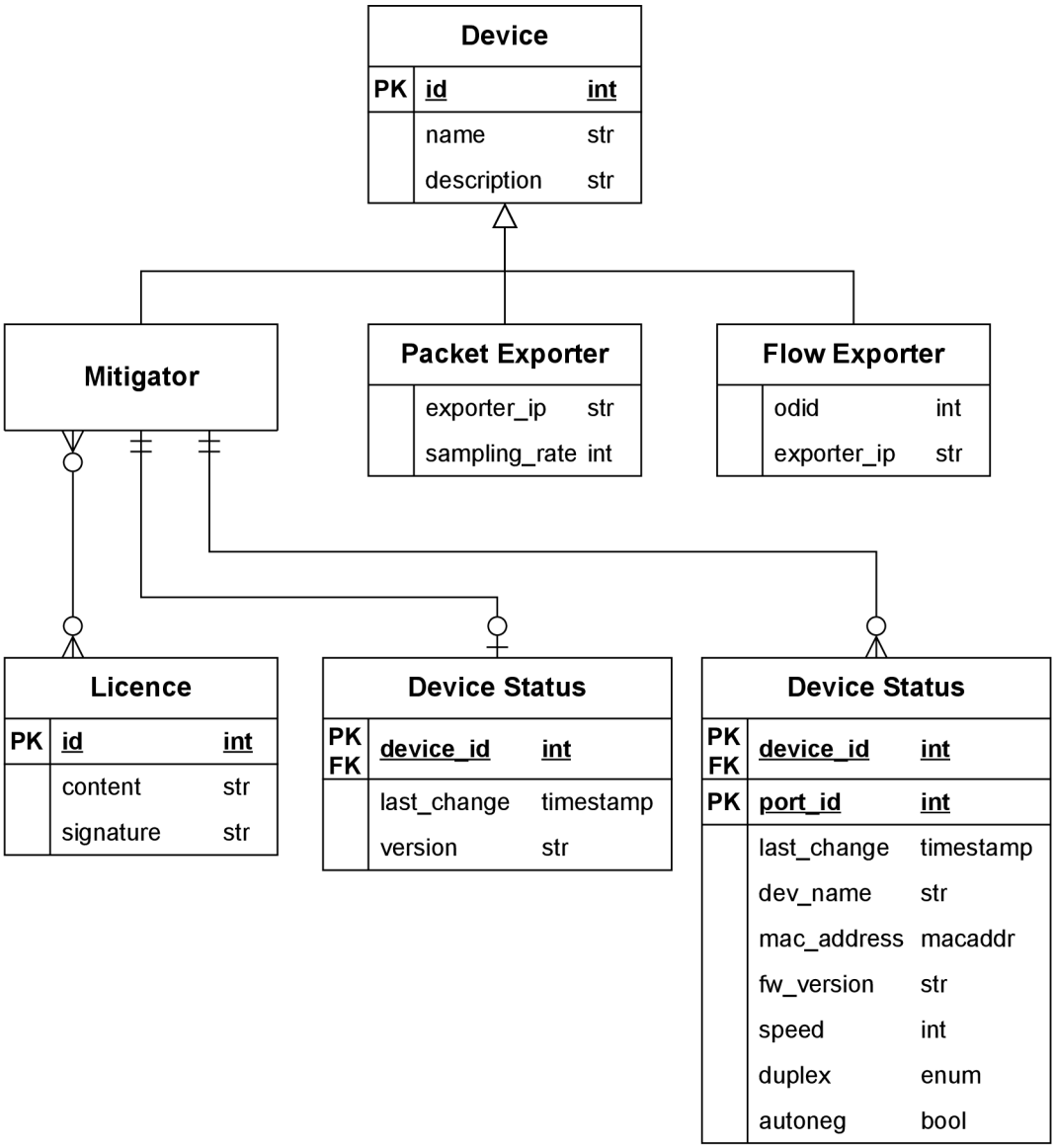


Figure 5.1: Updated database schema for devices

to do with the probe itself, and the implementation of the connection from the probe to the database may change in the future.

The DCPro DDoS Protector Database is defined using `sqlalchemy` ORM Python library. Therefore, the implementation of the changes consisted of introducing a new inheritance relationship between a generalized devices table and specialized tables representing the sFlow exporters, flow exporters and the entities in the original devices table, instances of DCPro DDoS Protector.

5.2 DCPro sFlow Probe

As discussed in Subsection 4.4.3, the DCPro sFlow Probe is a specific configuration of the DCPro DDoS Protector pipeline. The mechanism of DCPro DDoS Protector pipelines is explained in detail in Subsection 3.2.2. As designed, the implemented pipeline configuration is a reduced one, as we do not need the modules listed in Subsection 4.4.3 for the DCPro sFlow Probe to work.

The components representing unnecessary stages have been removed from the YAML files of the Mitigator pipeline configuration. The DCPro sFlow Probe is therefore functionally similar to the Mitigator, the only difference lies in the reduced functionality, such as the management of host traffic or writing any packets back to the port, which precludes it from serving as any kind of network filter device.

The changes to the YAML configuration file also required some minor modifications to the queries used to load the mitigation rules from the rule database.

The identification mechanism for the sFlow exporter was not implemented as part of this thesis. The reason is that it would require the introduction of a whole new component to the DCPro DDoS Protector. The code of the DCPro DDoS Protector is licensed as proprietary, so such deep modifications could not be made and published as part of this thesis.

5.2.1 Changes in YAML Configuration

The YAML configuration is organized into multiple parts, where bottom-level files contain the configuration of self-contained modules, and there is a hierarchy where upper-level configuration files can import the lower-level ones. The YAML files support variables, which can be specified on launch of the DCPro DDoS Protector using command-line arguments.

The pipeline configuration for Mitigator is contained in the file `dcpro_pipeline_mitigator.yaml`, which import two other configuration files, `dcpro_pipeline_base.yaml` and `dcpro_pipeline_mitigator_${mitigation.controller}.yaml`. The second imported configuration can be specified using the variable `mitigation.controller`. The two options are `file` for a simple Mitigator configuration, which parses the mitigation rules from a YAML file during the start of the application. This configuration is used mainly for testing. The other option is `postgresql`, which specifies the components to retrieve the rules from the PostgreSQL database, as well as to write the mitigation statistics back to it.

For the DCPro sFlow Probe, the `mitigation.controller` variable is set by default to the value `postgresql`, as using the `file` variant is virtually useless for a probe whose primary objective is to export statistics to the database. However, it is still possible to use this mode for debugging without a running database, so an explicit override by the user using a command-line argument is not disabled.

The configuration for most of the basic pipeline modules, such as the modules for writing to and reading from the port, processing host traffic intended for the kernel of the machine on which the DCPro DDoS Protector is running, a packet parser and a packet router module, is present in the `dcpro_pipeline_base.yaml`. Due to the fact that whenever a module is unused as an attribute of another module, in a way that would connect it to the hierarchical tree that arises from the root application module, it is ignored by the dependency injection framework of the DCPro DDoS Protector, we can largely leave the file `dcpro_pipeline_base.yaml` untouched. The only consideration for the DCPro sFlow Probe is to not use the pre-prepared list of modules `dcpro_pipeline_common_rx`, which

contains the modules used for reading and processing traffic coming from the port, as the list contains the host traffic module `dcpro_pipeline_host_traffic`, which would be unused in case of the DCPro sFlow Probe intended use, as it is not expected the probe will receive any data intended for the kernel, as it will be bound to the virtual TAP interface created by the sFlow Packet Parser.

The actual changes to the configuration of the pipeline were therefore made in the file `dcpro_pipeline_mitigator.yaml`. There are two main root modules that needed to be changed. Firstly, it was the root module `dcpro_pipeline_mitigator`, which changed from:

```
&dcpro_pipeline_mitigator:
    - *dcpro_pipeline_common_rx
    - *dcpro_mitigator_proc
    - *dcpro_pipeline_common_tx
```

to:

```
&dcpro_pipeline_mitigator:
    - *dcpro_pipeline_rx_reader
    - *dcpro_pipeline_pkt_parser
    - *dcpro_mitigator_proc
```

This change essentially implements the pipeline structure proposed in Figure 4.3. See that the pipeline ends at the Mitigator Processor stage, after which all packets are simply dropped. This means that both statistics and packet capture work in full.

The other change of the configuration from the original Mitigator to the probe one is the reduction in the number of Mitigator Processor modules, where there is one module per rule type. The list changed from the original values:

```
&dcpro_mitigator_modules:
    - *mitigation_module_packet_capture
    - *mitigation_module_exact_match
    - *mitigation_module_malformed_traffic
    - *mitigation_module_filter
    - *mitigation_module_syn_drop
    - *mitigation_module_amplification
    - *mitigation_module_tcp_auth
```

to the new reduced one:

```
&dcpro_mitigator_modules:
    - *mitigation_module_packet_capture
    - *mitigation_module_filter
```

As was proposed, the configuration only supports the Packet Capture and Filter rules, and removes the other mitigation modules. However, to force the Filter mitigation module to accept rules of all the different rule types, except for Packet Capture, it was necessary to change the database queries used to load the rules. This was done so that the actual type of the other rules may be hidden and be virtually re-typed to the Filter rule type for the internal use in the DCPro sFlow Probe. The changes to the queries are described in Subsection 5.2.2.

Additionally, it was necessary to modify certain configurations of the Packet Parser stage. By default, the Packet Parser checks whether the declared Ethernet frame length and IP packet length match the real length. If not, the packet is immediately dropped.

Afterwards, the packet is checked more thoroughly, such as whether the packet checksum is valid. If any of the safety features are invalid, the packet is blackholed. The blackholing can be safely turned off, as we do not expect the DCPro sFlow Probe to be receiving invalid packets. Thanks to this, the sFlow Parser only needs to parse out the sampled packet and add padding to it, as the sampled packets contained in the sFlow message might only contain a header and the first few bytes of the payload.

5.2.2 Changes in the Configuration of Database Queries

As discussed in the previous subsection, to make it possible for all the rules to be used by the Filter mitigation module, it was necessary to load them from the database as filter rules. The DCPro DDoS Protector includes a component called `file_sql_query_manager`, which is configured in the configuration file `dcpro_pipeline_mitigator_postgresql.yaml` to use queries specified in the file `mitigation_queries.sql`.

When the DCPro DDoS Protector needs to load the mitigation rules, the database access module first creates a list of all the rule bases and then reads the information from the additional type-specific tables based upon the type given in the base table. This means that there was no need to change any of the many queries used to load the additional attributes of the other rule types, and it was only needed to modify the query, which retrives the rule bases, so that it would replace the original type with the value `filter` for every rule other than Packet Capture. The resulting modified query only needed a simple addition of a `CASE` conditional expression compared to the original query.

Additionally, it was necessary to set all rules as being in the dry_run mode. Otherwise, it would not be possible to see how much of the traffic was matched by every rule. Even though the packets would be dropped anyway, this change makes it possible for the rules at the end of the priority order to receive every packet, since the packets are only dropped after passing through all the rules. Thus, the `dry_run` flag is set to `true`.

The modified query is shown in Figure 5.2. Notice that `file_sql_query_manager` requires a format where every query needs to be named and delimited with the markers `-- BEGIN (query name)` and `-- END`.

The only other query that was needed to be updated was the query used to load the table with the attributes specific to the Filter type, which includes the BPF. Since BPF does not need to be specified, it was decided to use the default value of `null` in case no record for the given rule existed in the `mitigation_rules.filter` table, which happens when a non-Filter rule is loaded with the forged type. This has been achieved by using a `UNION` clause on two `SELECT` queries. The first subquery retrieves all the filter rules, including the mode and BPF, the other retrieves the same data structure, with values `'block'` and `null` set for mode and for BPF respectively, but for all the other rule types.

Once the changes to the database and the YAML configuration of the DCPro DDoS Protector Mitigator pipeline were made, the result was a fully working pipeline configuration for the export of traffic statistics and possible capture of incoming packets. The implementation heavily borrows from the current configuration, which was also the goal initially set during the design phase, as the maintenance of the code is greatly simplified by producing changes in as few files as possible. The two new files that were created are the `dcpro_pipeline_sflow_probe.yaml`, which is a modified version of the original file `dcpro_pipeline_mitigator.yaml`, and `sflow_queries.sql`, which is based on the `mitigation_queries.sql` file with the changes mentioned in this section.

```
-- BEGIN mitigation_rules.base.select
SELECT
    id,
    CASE
        WHEN rule_type = 'filter' THEN rule_type
        WHEN rule_type = 'packet_capture' THEN rule_type
        ELSE 'filter'
    END AS rule_type,
    enabled,
    'true' AS dry_run,
    threshold_bps,
    threshold_pps,
    vlan
FROM mitigation_rules.device_rule
WHERE
    device_id = $1::int
-- END
```

Figure 5.2: Modified rule base retrieval query

5.3 sFlow Packet Parser

The sFlow Packet Parser was implemented as a basic server application written in C to achieve reasonable throughput even for a prototype. Due to the ability of using multiple write queues in a single TAP interface, there is a real possibility the sFlow Packet Parser will be used in a production environment, albeit with heavy changes. The implementation was focused on the creation of a simple prototype with a basic structure which could be later expanded upon. However, there are also limitations to the prototype that need to be solved later. Most importantly, it is the fact that the sFlow protocol supports many message types other than that containing a sampled packet. Support for the other message types may be included later. However, in that case, the sFlow parser would need to fulfill more roles than only parsing out sampled packets and passing them to the virtual TAP interface.

The sFlow parser has three basic parts. The first is the initialization of the TAP interface and the port binding. The other two parts are procedures, which are executed as part of the main loop. The first procedure parses the incoming packet, and the other writes the extracted packet samples to the TAP interface. The sequence of function calls is visualized in Figure 5.3.

5.3.1 Initialization

The initialization of the sFlow parser has three steps. Firstly, we need to process the command line arguments, secondly we need to initialize the socket and bind it to a port, and finally the TAP interface needs to be created and brought up. Each of these steps is executed in a separate function.

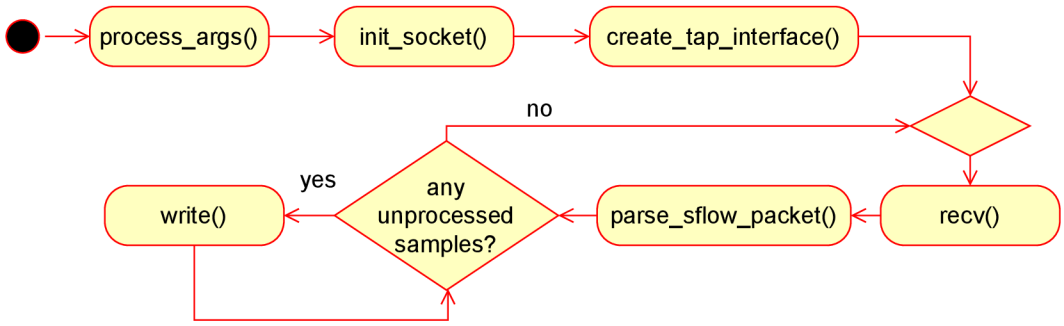


Figure 5.3: sFlow Packet Parser activity diagram

Parsing the Arguments

The first function is called `process_args`. It has the following function header:

```
void process_args(int argc, char *argv[], int *port, char *interface)
```

Where the parameters `argc` and `argv` are taken directly from the parameters of the `main` function. Both port and interface are allocated on the stack in the `main` function just before `process_args` is called. The size of the interface is limited by the value of the `IFNAMSIZ` constant defined in the `if.h` header file. The port and interface name are passed by pointers.

The arguments are parsed using the `getopt` library. The command line arguments are specified by the flags `-p` for the port, `-i` for the TAP interface name and `-d` for an interface name, if the socket needs to be bound to a specific interface. All three arguments are optional. The default value for port is 6343, which is the dedicated port number for sFlow collectors, as mentioned in the preceding chapter. The default value for the interface name is `tap0`. It is expected that the option to change these will be used for experiments and testing only.

Socket Creation and Binding

Once the port identifier and the interface name are parsed and stored in the variables, the next step is to create and initialize the socket to listen on the given port. This is done in the `init_socket` function, which has the following function header:

```
void init_socket(int *sFlowSocket, int port)
```

The variable `sFlowSocket`, which contains the file descriptor, is declared in the `main` function. Firstly, the function `socket` is called with the values `AF_INET` for the parameter `domain` and `SOCK_DGRAM` for the parameter `type`. This means that the socket is created in the IPv4 domain and uses UDP datagrams.

Once the socket is created, it is bound to the given port. For the IP address, the value is set to `INADDR_ANY`, meaning the socket accepts packets from all IP addresses. The socket is then configured with options `SO_BINDTODEVICE` if a specific network interface is requested in the respective argument, and `SO_NO_CHECK`, which makes the socket accept packets with invalid checksum. The last option should be particularly helpful during testing and experimentation if manufactured packets are used.

Creation and Initialization of TAP Interface

Finally, the TAP interface is created and initialized. This is done by the function `create_tap_interface`, which has the following function header:

```
int create_tap_interface(char *interface)
```

The function takes the interface name as an parameter and returns the file descriptor for the created TAP interface. There is a sequence of shell commands that are executed to create the interface and bring it up. Firstly, it is checked whether an interface with the given name already exists. If it does, the program assumes that the interface was created previously by the user. The management of any existing interfaces is left strictly to the user, while the creation of a TAP interface is done by the sFlow Parser, as it is a more complicated process. The shell commands necessary for the creation of a TAP interface are executed using the `system` function. The verification of the existence of the TAP interface is performed using the command `ip link show <interface>`. Then, if the interface does not exist, it is created using the command `sudo ip tuntap add <interface> mode tap`. After the interface is created, it needs to be brought up using the shell command `sudo ip link set <interface> up`.

Once the interface is up and running, a file descriptor is opened for a write-only operation. Then a `ifreq` structure is created, which represents a request for an I/O operation on a network interface. The `ifr_flags` attribute is set with the flags `IFF_TAP`, which means the request is to open a TAP interface, and `IFF_NO_PI`, which stops the interface from attaching any additional information to the raw packets. Once the `ifreq` structure is initialized with the flags, it is used for the request to attach the file descriptor to the created TAP interface, with the `ioctl` function, with the flag `TUNSETIFF`. After `ioctl` runs successfully, the file descriptor is now fully initialized and can be used to write any packets directly to the TAP virtual network interface.

5.3.2 Packet Parsing

Once the socket for receiving sFlow messages and the TAP interface and the file descriptor for writing into it are initialized, the primary loop is started. The loop first reads the incoming packet from the socket using the following function by calling the function `recv`, as shown in the following code snippet:

```
ssize_t bytesRead = recv(sFlowSocket, buffer, sizeof(buffer), 0);
```

The function `recv` is a blocking function, which waits for the packet to arrive on the socket, writes its payload to the buffer and returns the number of bytes which were written into the buffer. Afterwards, the function `parse_sflow_packet` is called. This function has the following header:

```
uint32_t parse_sflow_packet(
    char *buffer,
    ssize_t bytesRead,
    char ***sFlowSamples,
    uint32_t **sFlowSamplesSizes
)
```

The function returns the number of unpacked samples. Firstly, it reads the type of the sFlow Agent IP address. Then it uses it to calculate the offset and stores the Agent's IP

address. Then it parses the number of samples contained within the message. From every sFlow sample header, the type and enterprise fields are parsed. These contain information on the sample content. If the sample does not contain a sampled IPv4 or IPv6 packet, the size and data of the sample is set to 0 and NULL, respectively. What is also parsed is the size of the sampled data, to set the offset for the next sample.

For every valid packet sample, the type is extracted. This is done to calculate the possible discrepancy between the actual size of the sample and the declared size of the packet, since the DCPro DDoS Protector packet parser automatically drops misformed packets, where the declared and actual sizes do not match. The supported packet types are IPv4 and IPv6, although other types will simply be sent as is without any additional padding. The memory for the sample is then allocated and in case of IPv4 and IPv6, the sample data is then padded with zeros. The actual resulting size of the packet is then stored in the `sFlowSamplesSizes` variable and the pointer to the allocated memory containing the padded sample is added to the `sFlowSamples` array. The number of samples is then returned once all the samples have been processed.

5.3.3 Packet Writing

Finally, the parsed packet samples are written to the TAP interface. This is relatively simplest and most straightforward part of the implementation, since the packet samples contain the whole raw packets. This means that it is only needed to loop through the samples and call the function `write` in the following way:

```
ssize_t bytesWritten = write(tap_fd, sFlowSamples[i], sFlowSamplesSizes[i])
```

Once every sample is written, the memory allocated for the variables containing the samples is freed, and the main loop starts all over again with a new sFlow message being read.

5.4 DCPro NEMEA Daemon

The implementation of the DCPro NEMEA Daemon largely followed the design specified in Section 4.5. The DCPro NEMEA Daemon is an application written in Python and implemented using the daemon module of the `dcproapi` library, which is part of the DCPro DDoS Protector development tools. The DCPro NEMEA Daemon is implemented as a prototype, as it is fully expected that an application suitable for deployment in production would require a higher focus on the throughput of the system and much better optimalization. Furthermore, it would be helpful to reuse multiple modules of the DCPro DDoS Protector, especially the accelerated packet classifier module described in [51]. However, as already mentioned in the chapter on the design of the tool, the DCPro DDoS Protector code is protected under a proprietary license and cannot be reused as such. It should be possible to extract the packet classifier into a standalone component with a public interface. However, the amount of work required to decouple the module from the modules connected to it is too high for the scope of this thesis. This is mostly due to the fact that its internal structure is based heavily on the processing of packets. Instead, the implementation serves to showcase the required functionality.

5.4.1 Basic Daemon Structure

The `dcproapi` library used in the implementation provides a basic structure for the daemon, with its `daemon` module. The module provides two abstract classes, which are used to implement a basic daemon structure. The first one is `AsyncRunner`, which is implemented by the `UvloopRunner`. The runner class should exist as a single instance, and it handles the main loop of the daemon application. The other is the `AsyncReader` abstract class, which serves as a basic interface for any module, which implements handlers for scheduled tasks and any asynchronous events happening on the main event loop, such as incoming database notifications.

The core of the DCPro NEMEA Daemon is mainly contained in the implementation of a single reader class called `NemeaStatsReader`. The reader class contains several methods, which are executed by the runner as asynchronous tasks. Firstly, the command-line arguments are parsed. The daemon supports arguments for the specification of the path to a configuration file and the identifier of the TRAP interface to connect to.

Configuration files provides form of long-term configuration, which is an approach used elsewhere in the DCPro DDoS Protector supporting tools and daemons. The configuration file is called `nemea_stats.ini` by default, and it is used to define variables that are unlikely to change often. In the current state of the DCPro NEMEA Daemon, these are the database connection parameters, the logger settings, and the setting for the number of workers, which are used in the `daemon` module of the `dcproapi` library.

The daemon application uses a dependency injection framework provided by the `dependency_injector`[1] Python library. The library provides a simple interface for the configuration of dependency injection through a `Container` class. This class contains the classes which should be instantiated, and the dependencies between them. It can also handle command-line arguments and pass them as parameters during instantiation. Thanks to the use of this technique, the daemon is highly modular and can be easily expanded with additional modules. The `NemeaStatsReader` instance is injected with all the configuration, as well as the entity manager objects provided by `dcproapi`.

The initialization of the reader instance involves the creation and initialization of a TRAP interface to receive the UniRec data. The reader then continually executes a data gathering task, which is explained in detail in Subsection 5.4.2. The reader also creates a periodic task that occurs once per an interval provided in the configuration. The default length of the interval is ten seconds. As will be explained in detail later in Subsection 5.4.3, this task then launches computation-heavy tasks in separate processes. These include the aggregation and export of the statistics and the training of the ADT Flow module. Thanks to the use of separate processes, the limit put on parallel execution by Python's Global Interpreter Lock, explained in Subsection 4.5.1, is overcome.

5.4.2 Data Gathering

Data gathering is the only task that is continually being processed at all times, except in the relatively small time window when the processes for the execution of other tasks are started. The data gathering task uses the TRAP interface, which is created on initialization of the runner. Afterwards, the `recvBulk` method of the TRAP interface object is called. The method returns a list containing UniRec objects, which are essentially represented as Python dictionaries. The `recvBulk` method required a timeout, which is set to 1 second,

[1] https://python-dependency-injector.ets-labs.org

and it can automatically handle a change in the template of the incoming UniRec data. The number of returned records is currently set to 1, but can be changed for experimentation.

Once the method executes, the contents of the returned list are checked first. If the list is empty, the task terminates and is staged for a new execution in the next iteration of the event loop. Several values of the UniRec record are serialized, as more complex objects, such as the IP address class used by the `pytrap` library, cannot be passed to newly created processes. Once the conversion is complete, the record is appended to a list of records. The list of records is flushed once every interval of the sliding window, after which the extracted statistics are exported, and the ADT Flow models are trained.

5.4.3 Window Handling

When the interval given for the sliding window expires, the `handle_window` method is executed as a periodic task. Inside the `handle_window` method, several process tasks are created using the `create_process_task` method provided by the runner instance. The tasks created in this way run in a different process, so they do not affect the primary event loop, since even though the primary even loop allows for scheduling tasks asynchronously, only one task can be actively executed at a time. Using different processes, the limitation put on multithreading in Python can be overcome. The mechanism for insertion of extracted statistics and training of ADT Flow machine learning models is shown in Figures 5.4 and 5.5, and is further described in the following subsections. The diagrams are a customized version of the UML sequence diagram, and are used to visualize both the sequence of executed operations, as well as indicate the different processes by different colors of the sequence box. The periodic task is visualized as an endless loop in the main event loop. The task that reads the UniRec messages from the TRAP interface is not shown, as it is not parallel and is always scheduled on the main event loop. The sequence diagram does not use only objects, but also the coroutines which are scheduled as separate process tasks, to show the parallel nature of the mechanism. Any method invoked as part of the coroutine is then executed within that process.

5.4.4 Export of Statistics

The export of statistics extracted from UniRec into the DCPro DDoS Protector database is handled by what is called the Statistics Export Module. During experimentation, it was decided to also implement several useful changes to the DCPro DDoS Protector database, specifically to the way statistics are stored, so that the space is effectively utilized. Optimization of storage space use proved useful even in testing, as the mechanism used to extract and aggregate the values contained in the UniRec data resulted in the creation of multiple statistical records per single collection window.

Statistics Export Module

The sequence of execution of the Statistics Export Module is visualized in Figure 5.4. Export of statistics is handled by the `_export_statistics` static method, which is passed as the coroutine to the `create_process_task` method. The parameters passed to the export method include the identifier of the DCPro NEMEA Daemon device representation in the DCPro DDoS Protector database, the database connection parameters and various other configuration variables, the list of UniRec records and the timestamp of the end of the data collection window.

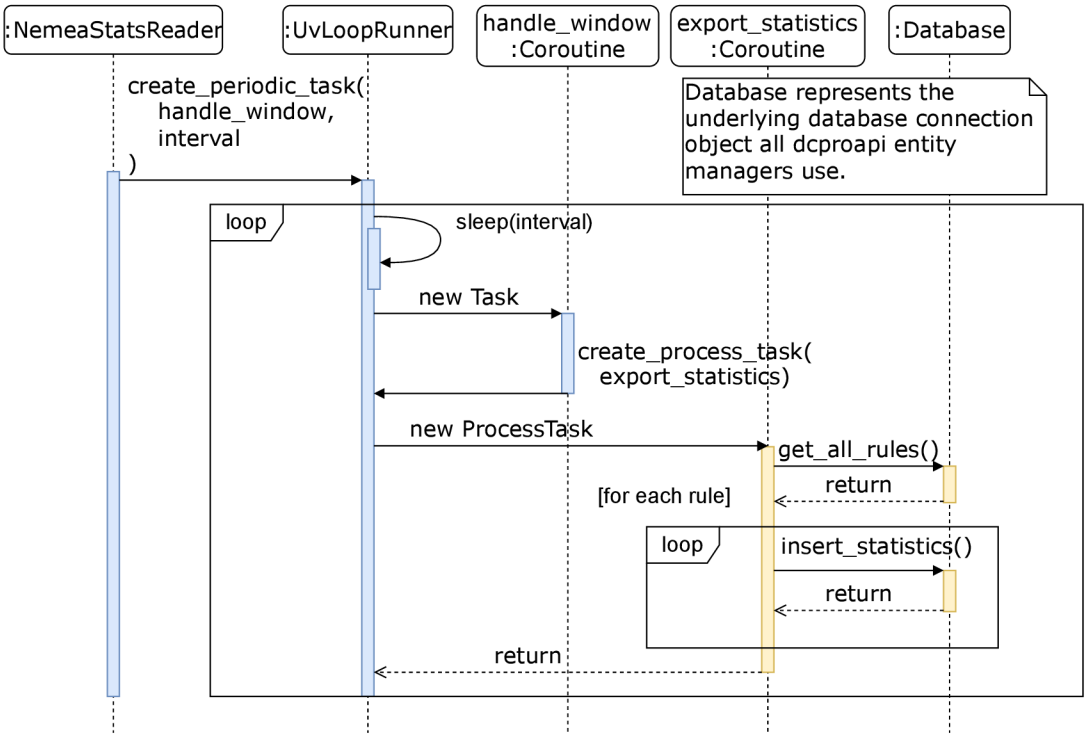


Figure 5.4: Sequence diagram of the export of statistics

The first part of the initialization is retrieving the mitigation rules from the database. Since the mitigation rules are loaded only once during the export of the statistics, in case any new rules are added in the meantime, they are only taken into account in the next collection window.

Secondly, a data structure is created to contain the aggregated data. As proposed in Chapter 4 and visualized in Figure 4.5, the implementation splits the byte and packet counters contained within the UniRec records into several buckets of the same length and updates the counters in them. The structure containing the buckets is instantiated for every rule and for every registered flow exporter device. Every bucket contains a packet and a byte counter, which is incremented by the value I if the UniRec matches the given rule. The value I is calculated using the following equation:

$$I = Total/Bucket_count \tag{5.1}$$

In this equation, I represent the value to be incremented, $Total$ is the total number of packets or bytes contained in the UniRec record and $Bucket_count$ is the number of bucket calculated from the UniRec duration and the interval represented by each bucket. The number of buckets in the structure is dynamic, as new buckets can be added if a record with an earlier `TIME_START` timestamp is encountered. This method of splitting each bucket into several buckets produces skewed data, as we do not know the exact temporal distribution of packets and bytes in each UniRec record. However, this is offset if the number of UniRec records is large enough.

The method `classify_flow` determines which rules match the incoming record and returns a list of the IDs of the matching rules. The classification is implemented in the

same way as in the DCPro DDoS Protector, which uses source and destination IP addresses and ports.

Finally, once all records are processed, a statistical record is inserted into the database for every bucket of every rule. This means that if the bucket size is ten seconds and the longest a record was collected is five minutes, up to 30 records are inserted per rule per collection window. While this approach generates a relatively large number of records compared to the DCPro DDoS Protector, it scales much better for any kind of number of UniRec records than producing one statistical record for every UniRec record, and it produces data in much more useful granularity than if only five minute buckets were considered, especially since the flow might be much shorter.

Database Modifications

Because a relatively large amount of data is inserted into the DCPro DDoS Protector database and because the statistical records extracted from UniRec overlap, a more complex storage solution was devised. Since UniRec contains only the `BYTES` and `PACKETS` fields, a new table was created, named `flow_records`. The table replaced all byte and packet columns in the mitigation statistics table with only one column for bytes and another for packets. When the statistics are read, the values are converted to the mitigation statistics format in a join. Additionally, the TimescaleDB extension uses a `hypertable` structure, which can have a specific retention time for the data. The retention policy for `flow_records` is set to only 1 day compared to the 14 days for mitigation statistics. After that time elapses, the records are stored in a materialized view which uses continuous aggregation. All the data that would be dropped are aggregated using 10 second time buckets, so that the overlap is removed, and is saved into the materialized view for longer persistence.

5.4.5 ADT Flow Module

The ADT Flow module is the other task which is started every time a collection window finishes. The ADT Flow module is represented by the `_load_ml_rules` and `_train_ml_models` private static methods, which are again passed to the `create_process_task` runner method to run as a separate process. The sequence of method calls and the creation of separate processes is visualized in Figure 5.5. The implementation is heavily inspired by the previous implementation of the ADT Flow Daemon, which was described in Subsection 4.5.4. During the initialization of the DCPro NEMEA Daemon, a special structure is created, which contains a circular buffer of `dataframe` objects containing legitimate traffic data for every ADT Flow rule. At first, the structure is an empty Python dictionary which is later filled with the buffers as values and the rule identifiers as keys.

At first, the `_load_ml_rules` method is scheduled as a process task. This is done so that the main event loop is not slowed down by awaiting the result of costly time-consuming database operations, as the current implementation of the `dcproapi` supports only synchronous database connections. The process task is modified by adding a callback method called `_on_ml_rules_loaded`, which then schedules a process task for every single ADT Flow rule for the derivation of the BPF from the collected UniRec messages. The `_on_ml_rules_loaded` callback is passed to the `add_done_method` of the process task object as a partial function, so that it already contains the collected UniRec messages.

The `_on_ml_rules_loaded` callback method then creates a separate process task for every rule from the method `_train_ml_model`. The method expects the rule object, the data collected inside of the window, and the circular buffer kept in the reader object as

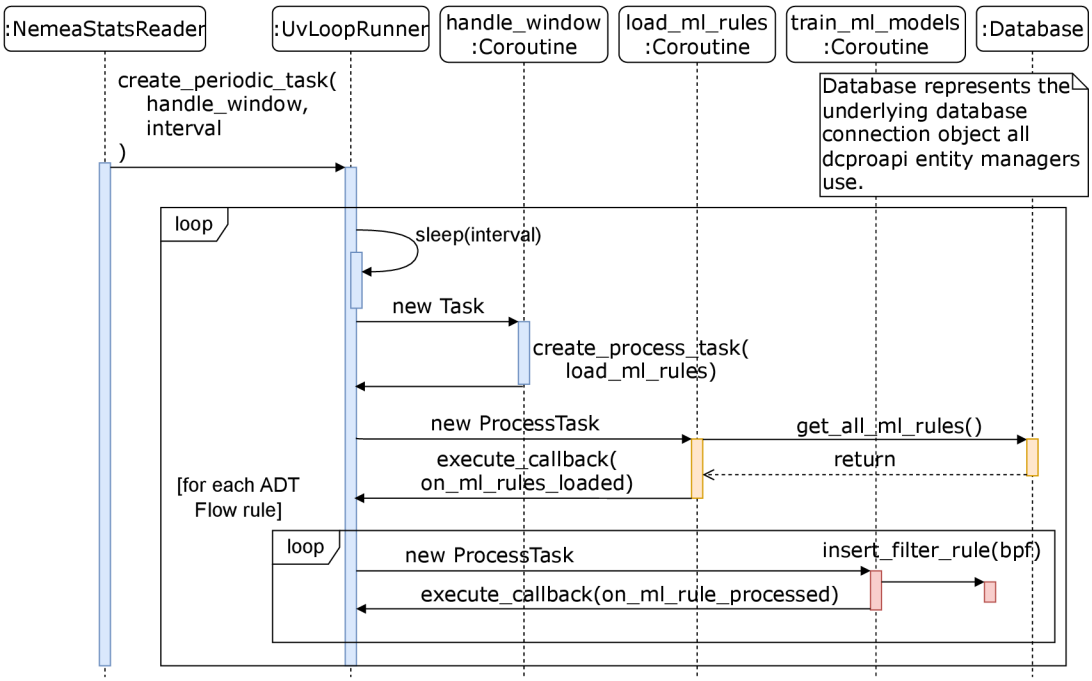


Figure 5.5: Sequence diagram of the ADT Flow module

the parameters. Inside the method, the data are first classified, and a list of UniRec messages that are matched by the given ADT Flow rule is created. In case the number of matched records exceeds the configured limit, the other records are simply ignored. The classification function is reused from the statistics exporter module. Afterwards, the matched data is analyzed using a function provided by the `ml_netflow` module, which takes the byte and packet thresholds of the rule. If the traffic contained in the list is analyzed as attack traffic, the circular buffer is linearized into a list containing legitimate traffic data. Both lists are transformed into `dataframe` objects, which are then passed to the `create_rules_from_netflow` method provided by `ml_module` library, which returns a generated BPF. The generated BPF is then used to create a new filter rule using the `create_and_upload_rule` method provided again by the `ml_netflow` module. In case the data are classified as legitimate, it is immediately returned as a tuple, where the first element is the rule identifier and the second one is the list containing the UniRec messages.

The process tasks created for `_train_ml_model` have a callback method added to them, called `_on_ml_rules_processed`. In case the value returned from the process task is a tuple with the dataframe and rule ID, the existing circular buffer contained in the reader object is appended with the list containing messages with legitimate traffic, to be used for a more precise BPF inference in a later collection window.

5.5 Experimentation and Evaluation

During the implementation of the prototypes, they were already experimented upon. The first set of experiments focused on finding potentially problematic areas of future development. Especially the sFlow Parser was extensively investigated, as there was a competing

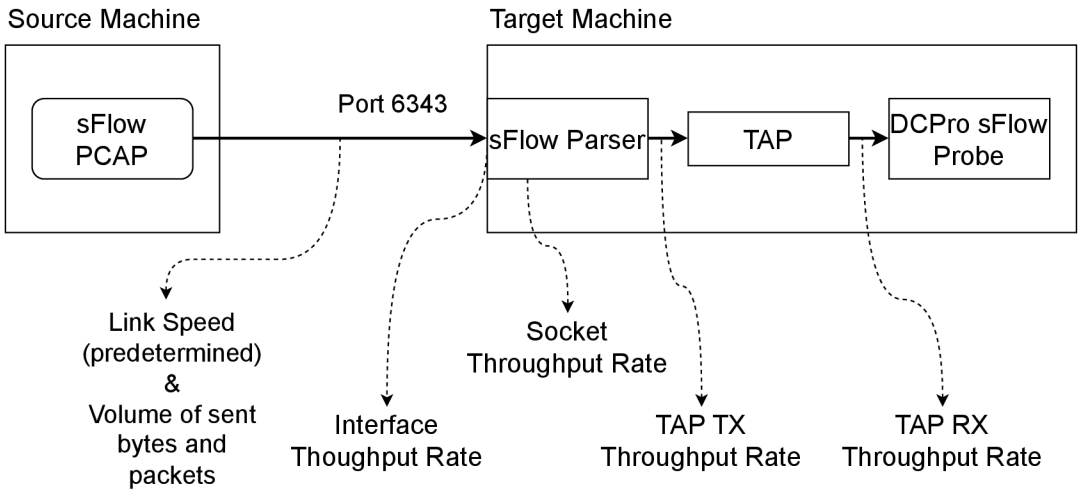


Figure 5.6: sFlow testing architecture

design possibility in a DCPro DDoS Protector pipeline with an additional sFlow parser stage. Both the NEMEA Daemon and the sFlow Processor were tested on real traffic. After the general functionality of the tools was affirmed, the other tests focused primarily on the applications' throughput, which is absolutely crucial when processing large quantities of data produced by high-speed network links in real time.

5.5.1 sFlow Parser and DCPro sFlow Probe Testing

The testing of the throughput sFlow Parser and the DCPro sFlow Probe was complicated by the fact the sFlow Parser is a socket application listening on a UDP port and is connected to the DCPro sFlow Probe using a virtual TAP interface. The problem lies in the fact that both the UDP socket and the TAP interface have their own limitations on the throughput, and some packets might be dropped before even arriving at the sFlow Parser.

For the first round of experimentation, the sFlow Parser and DCPro sFlow Probe were supplied with live sFlow messages from a router deployed by the DCPro DDoS Protector development team. Once it was confirmed that both applications process the data correctly, the sFlow messages were captured into a PCAP file. Although tools like `tcpreplay`[2] make it possible to replay packets from a PCAP file on the same machine machine, such replayed packets are inserted into the network stack so that they are inaccessible to a socket. Therefore, sFlow messages had to be replayed from another machine, using `tcprewrite`[3] to rewrite IP and MAC addresses and to fix the checksum. The final architecture used to test the sFlow Parser and DCPro sFlow Probe is visualized in Figure 4.4, together with the sources of valuable data. To correctly identify a potential bottleneck, both the physical and the TAP interfaces were monitored. The throughput of the socket was evaluated using a simple log. Thanks to the use of captured sFlow traffic, we can determine the ratio of bytes to packets. Since the number of Ethernet frames in the PCAP is 18, and the total sum of bytes is 18076, we can expect the average number of bytes per packet to be round

[2] https://tcpreplay.appneta.com
[3] https://tcpreplay.appneta.com/wiki/tcprewrite

1004. The sFlow messages contain 95 sflow samples, which have a total declared length of 82559 bytes.

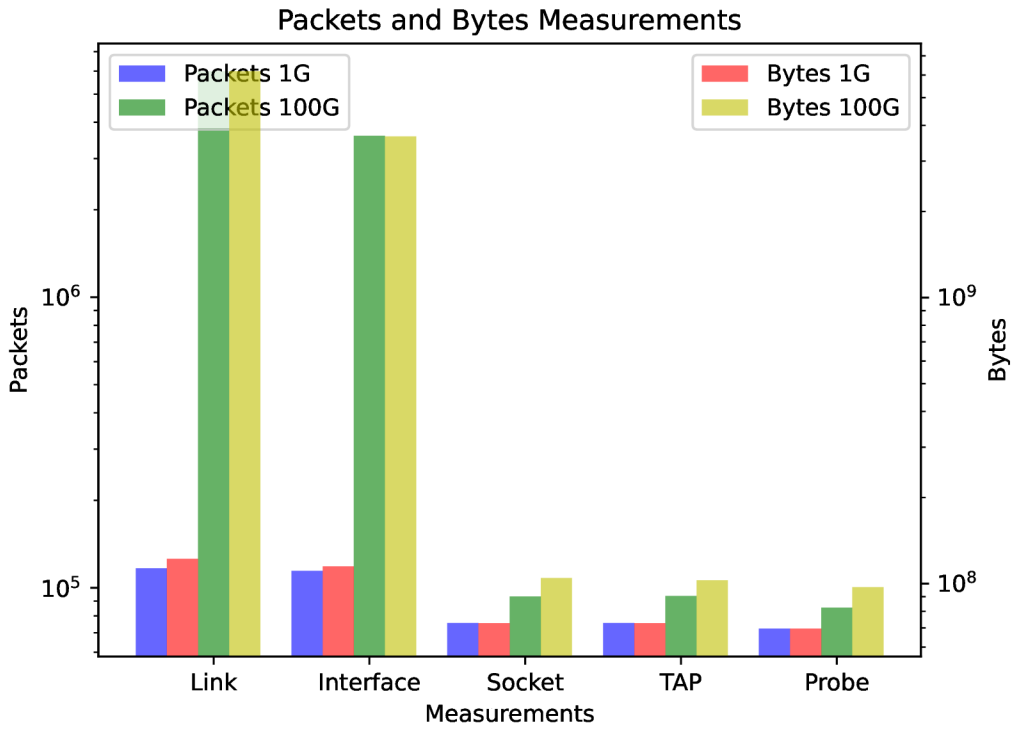


Figure 5.7: sFlow architecture throughput

The throughput of the whole system was tested first on a 1 Gbps link and then on a 100 Gbps link. Using the `tcpdump` and `iftop` tools to retrieve the number of packets and bytes that passed through the network interfaces. The number of packets and bytes which the sFlow parser read from the socket and which were written to the TAP interface was kept in counters and displayed after the application was terminated, as the printing operation can be relatively time consuming. Finally, the number of packets and bytes received by the sFlow Probe was retrieved from the statistics. The result of the throughput tests is visualized in Figure 5.7, where the test cases are shown for the 1G and 100G link. The graphs show the actual volume of data that travel through the link, which was only around 55.83 Gbps for the 100G link. There was an issue with generating such large amounts of traffic, which was resolved by using the TRex[4] packet generator. Note that the numbers of bytes and packets shown for TAP and the sFlow Probe are divided by the ratios of sFlow messages to samples retrieved from the PCAP file, so that the values can be directly compared. The scale of the values is logarithmic, so that the results of the 1G and 100G test cases can be visualized together.

From the results compared side by side, we can see that the throughput of the UDP socket used by the sFlow Parser plateaus around 100,000 packets per second. The value differs slightly in both tested cases, most likely due to the different NIC which were used, and their level of optimalizations and compability with the kernel. In case of the 1G test, it

[4] https://trex-tgn.cisco.com

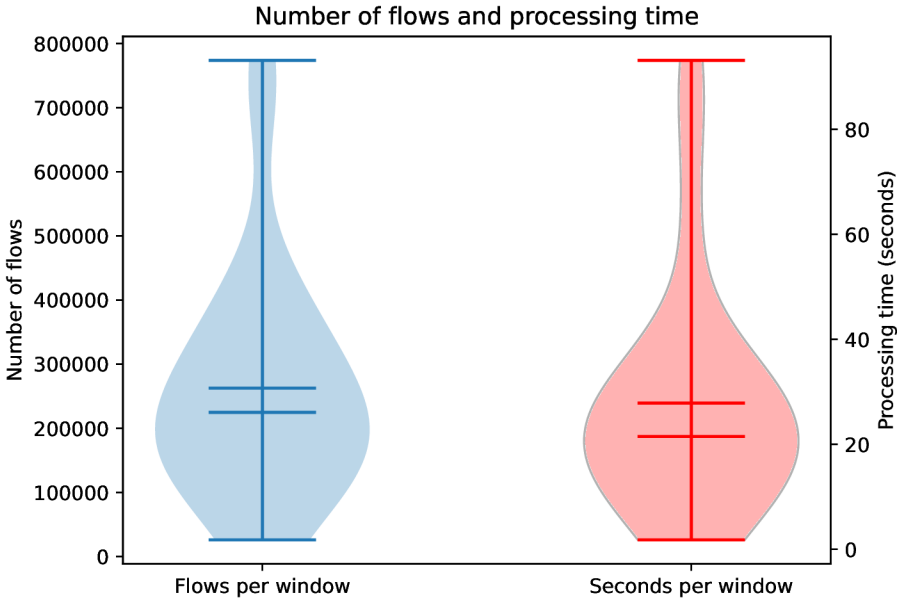


Figure 5.8: Distribution of the throughput of statistics collection window by the number of flows and processing time per window

was the *Intel(R) Ethernet Controller X710 for 10GbE SFP+*[5], and the *NVIDIA Mellanox ConnectX-5*[6] 100G NIC. In both test cases, we can see that the throughput of the TAP interface and the DCPro sFlow Probe match the throughput of the sFlow Parser, and there are no more significant packet losses. The limits that kernels place on the throughput of sockets are one of the reasons why the DCPro DDoS Protector and other such high-volume network applications are implemented using the DPDK framework, which bypassed the kernel. The performance analysis of DPDK-based applications in [4] also mentions this fact.

5.5.2 DCPro NEMEA Daemon Testing

The NEMEA Daemon was experimented with using traffic collected by a testing IPFIXcol instance and a NEMEA instance, which acts as a UniRec preprocessor. The NEMEA Daemon was appended to the last module of the NEMEA instance. Using the `flowcounter` NEMEA module for measurement, the average detected number of UniRec records that passed through the instance every second was close to 100,000. However, individual measured values ranged from less than 75,000 to more than 125,000 records per second.

Like the sFlow Parser and the sFlow Probe, the NEMEA Daemon was also tested primarily for throughput. However, it soon became apparent that the throughput could vary significantly due to the tools used for the implementation, and the time needed to process the data also varied. Figure 5.8 shows the distribution of the number of flows collected within a 10 second aggregation window and the amount of time it took to aggregate them and upload them to the database as statistical records. Both graphs contain 25 measurements, each of which corresponds to the same window. The values of the 25 measurements

[5] https://www.intel.com/content/www/us/en/products/details/Ethernet/700-controllers/x710-controllers/products.html

[6] https://www.nvidia.com/en-us/networking/Ethernet/connectx-5/

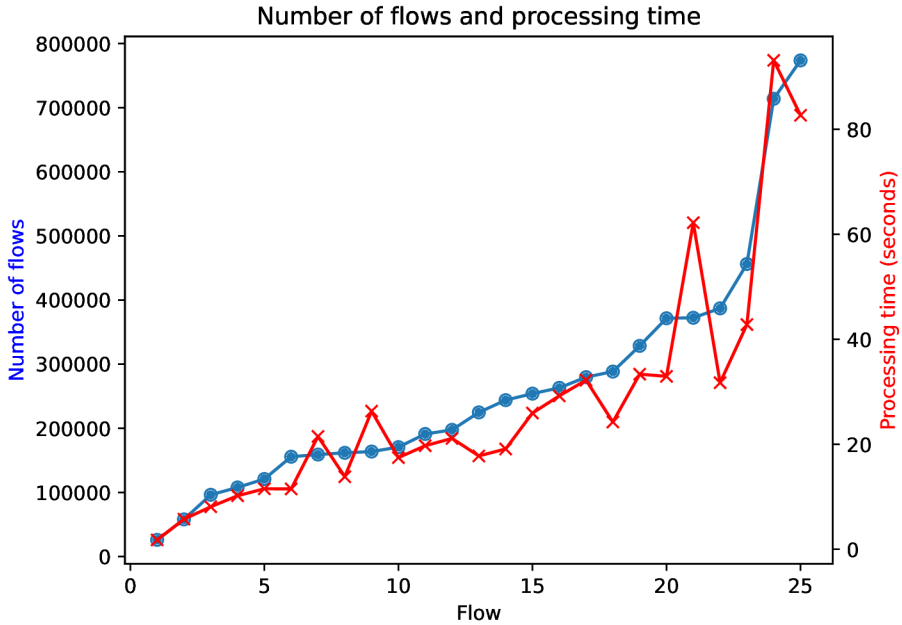


Figure 5.9: Relationship between the number of flows and the processing time of each window

are shown in Figure 5.9, where the degree to which the flow count and the spent time correspond to each other is also shown. The values in both figures were measured when two mitigation rules were active. On average, the number of flows per second handled by a statistics export window is about 9432.7665. This means that the NEMEA Daemon can on average handle the same volume of data as is currently being processed on the testing instance of NEMEA and IPFIXcol. However, in case a DDoS attack is ongoing, we can assume the number of incoming UniRec records to increase significantly, as NetFlow and IPFIX exporters will export flows prematurely if a certain threshold of traffic is reached.

Furthermore, the number of active mitigation rules severely affects the speed at which the flows are processed, since every rule is checked to see whether it matches the given rule. As can be seen in Figure 5.10, there is a steady decrease in the throughput of the statistics module with each additional rule. Every number in the figure is an average calculated from a five-minute run with the given number of active rules. The duration of the statistics export can affect the execution of the whole daemon if it becomes too long, as the daemon can only support a handful of active processes.

The machine learning module was tested for general functionality. However, since a separate process is executed for every ADT Flow rule, it makes little sense to test it for throughput. As the ADT rules were already tested for the optimal input size, all that needed to be verified is that the module is fully functional.

5.5.3 Summary and Evaluation

From the data gathered, we can conclude that the sFlow parser is suitable for a limited real-life volume. Depending on the configuration of the sFlow exporters, especially the sampling rate, it may be fully suited for the task of gathering at least some data. The main bottleneck currently seems to be the kernel handing over packets to the socket. This

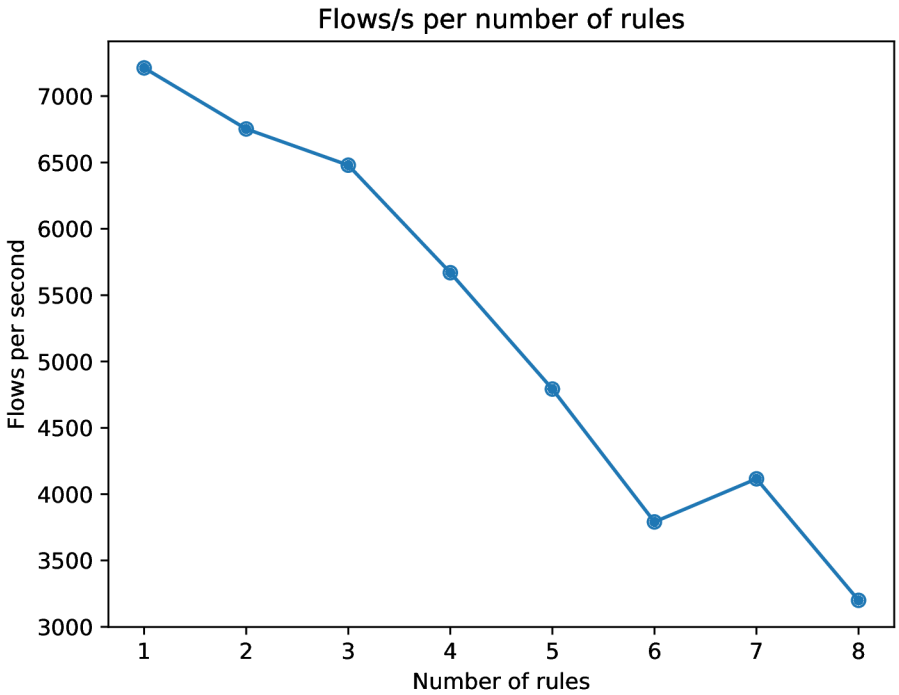


Figure 5.10: Relation between the throughput of NEMEA Daemon statistics export module and the number of active mitigation rules

should be easily solved by deploying the sFlow Parser on multiple network interfaces, and using a multi-queue TAP interface with a single DCPro sFlow Probe. Together with additional optimizations, it should make the sFlow Parser ready for high-volume production deployment.

The NEMEA Daemon, on the other hand, is less efficient with the data and, therefore, unsuitable for a production deployment, outside of low-volume uses. It seems to suffer from its implementation decisions, such as the use of asynchronous processing in Python, which severely limits its potential for high-speed execution. With certain crucial optimizations, such as the use of the ACL accelerated packet classifier extracted from DCPro DDoS Protector, it could become much better suited for real-life deployment. Additionally, it is also possible to deploy an array of NEMEA Daemon instances using a `demux` NEMEA module. However, first a discussion of whether the chosen approach is suitable or whether a new implementation should be developed using more efficient tools should be held. The optimized implementation could use modules written in C or C++, or it could be written in those languages wholesale, as the NEMEA Framework provides an API in those languages.

Chapter 6

Conclusion

The aim of this thesis was to propose and subsequently implement a solution for the integration of the DCPro DDoS Protector and the tools used for network monitoring and analysis at CESNET, so that they can form a single distributed architecture for the mitigation of DDoS attacks. First, the protocols used for network monitoring in the CESNET network were discussed. These include NetFlow and IPFIX for flow-based data, and sFlow for packet samples. The IPFIXcol2 flow data collection system and the NEMEA data processing system were then analyzed. Special attention was paid to the format of the data and the information that needs to be extracted. Finally, the implementation of the DCPro DDoS Protector was studied in detail to understand the needs of the system and its administrator.

Based on the knowledge gained, a solution to the problem was devised. The proposal was to convert data collected from NEMEA and sFlow to the format used for DCPro DDoS Protector statistics and to devise a way to train machine learning models used for DDoS mitigation with these data. The solution was implemented as a fully functioning prototype for demonstration purposes. The DCPro NEMEA Daemon was created to handle the flow-based data collected by IPFIXcol2 and processed by NEMEA, so that they can be used for DDoS mitigation by the DCPro DDoS Protector. Packet samples created using the sFlow protocol are handled by two main applications. The first is the sFlow Packet Parser, which parses the samples out of a sFlow message and writes them to a virtual network interface. The other is the DCPro sFlow Probe, which is a reduced configuration of the DCPro DDoS Protector mitigation pipeline.

Both tools were thoroughly tested on captured real-world traffic. The results show that the deployment of the sFlow Parser and DCPro sFlow Probe is viable. In the case of the DCPro NEMEA Daemon, the implementation serves well to demonstrate the functionality. However, it needs further changes so that it can be deployed effectively in a high-volume traffic environment. Together, the two implemented tools proved the feasibility of the solution and set the future goals of increasing the throughput of both applications. The aim of this thesis was thus fulfilled, as the creation of a distributed architecture for the mitigation of DDoS attacks using DCPro DDoS Protector, with network monitoring data received through sFlow and NEMEA, was achieved.

Bibliography

[1] Aitken, P., Claise, B. and Trammell, B. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* [RFC 7011]. RFC Editor, september 2013. DOI: 10.17487/RFC7011. Available at: https://www.rfc-editor.org/info/rfc7011.

[2] Alias, S. B., Manickam, S. and Kadhum, M. M. A Study on Packet Capture Mechanisms in Real Time Network Traffic. In: *2013 International Conference on Advanced Computer Science Applications and Technologies*. 2013, p. 456–460. DOI: 10.1109/ACSAT.2013.95.

[3] Beitollahi, H. and Deconinck, G. Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Communications*. 2012, vol. 35, no. 11, p. 1312–1332. DOI: https://doi.org/10.1016/j.comcom.2012.04.008. ISSN 0140-3664. Available at: https://www.sciencedirect.com/science/article/pii/S0140366412001211.

[4] Belkhiri, A., Pepin, M., Bly, M. and Dagenais, M. Performance analysis of DPDK-based applications through tracing. *Journal of Parallel and Distributed Computing*. 2023, vol. 173, p. 1–19. DOI: https://doi.org/10.1016/j.jpdc.2022.10.012. ISSN 0743-7315. Available at: https://www.sciencedirect.com/science/article/pii/S0743731522002271.

[5] Beneš, D. *DDoS Mitigation Configuration Tool*. 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Šišmiš, I. L. Available at: https://www.vut.cz/studenti/zav-prace/detail/145210.

[6] Carasec, E. *Optimization of DDoS Mitigation Rule Inference*. 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Martin Žádník, P. Available at: https://www.vut.cz/studenti/zav-prace/detail/145160.

[7] Cejka, T., Bartos, V., Svepes, M., Rosa, Z. and Kubatova, H. NEMEA: A framework for network traffic analysis. In: *2016 12th International Conference on Network and Service Management (CNSM)*. 2016, p. 195–201. DOI: 10.1109/CNSM.2016.7818417.

[8] Claise, B. *Cisco Systems NetFlow Services Export Version 9* [RFC 3954]. RFC Editor, october 2004. DOI: 10.17487/RFC3954. Available at: https://www.rfc-editor.org/info/rfc3954.

[9] Daněk, J. *Generátor záznamů o síťových útocích.* Brno, CZ, 2014. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vut.cz/studenti/zav-prace/detail/79723.

[10] *DDoS Protector* [online]. Liberouter [cit. 2024-01-25]. Available at: https://www.liberouter.org/technologies/ddos-protector/.

[11] Dulik, M. Network attack using TCP protocol for performing DoS and DDoS attacks. In: *2019 Communication and Information Technologies (KIT).* 2019, p. 1–6. DOI: 10.23919/KIT.2019.8883481.

[12] Elsayed, M. S. and Azer, M. A. Detection and Countermeasures of DDoS Attacks in Cloud Computing. In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN).* 2018, p. 708–713. DOI: 10.1109/ICUFN.2018.8436989.

[13] Falowo, O. I., Ozer, M., Li, C. and Abdo, J. B. Evolving Malware and DDoS Attacks: Decadal Longitudinal Study. *IEEE Access.* 2024, vol. 12, p. 39221–39237. DOI: 10.1109/ACCESS.2024.3376682.

[14] Goldschmidt, P. *Mitigation of DoS Attacks Using Machine Learning.* Brno, CZ, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23613/.

[15] Goldschmidt, P. and Kučera, J. Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques. In: *Proceedings of the IM 2021 - 2021 IFIP/IEEE International Symposium on Integrated Network Management.* International Federation for Information Processing, 2021, p. 772–777. ISBN 978-3-903176-32-4. Available at: https://www.fit.vut.cz/research/publication/12359.

[16] Habib, A. Edge-to-Edge Network Monitoring to Detect Service Violations and DoS Attacks. In: *Handbook of Research on Information Security and Assurance.* IGI Global, 2009, p. 179–192.

[17] Halenár, M. *Prohlížeč netflow záznamů pro řešení bezpečnostních incidentů.* 2011 [cit. 2024-05-11]. SUPERVISOR: RNDr. Daniel Kouřil, Ph.D. Available at: https://theses.cz/id/lg4thi/.

[18] 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. *IEEE Communications Surveys & Tutorials.* 2014, vol. 16, no. 4, p. 2037–2064. DOI: 10.1109/COMST.2014.2321898.

[19] Hoque, N., Bhattacharyya, D. K. and Kalita, J. K. Botnet in DDoS Attacks: Trends and Challenges. *IEEE Communications Surveys & Tutorials.* 2015, vol. 17, no. 4, p. 2242–2270. DOI: 10.1109/COMST.2015.2457491.

[20] Hoque, N., Bhattacharyya, D. K. and Kalita, J. K. Botnet in DDoS Attacks: Trends and Challenges. *IEEE Communications Surveys & Tutorials.* 2015, vol. 17, no. 4, p. 2242–2270. DOI: 10.1109/COMST.2015.2457491.

[21] Huták, L. *Nová generace IPFIX kolektoru*. 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/studenti/zav-prace/detail/114769.

[22] Huták, L. *IPFIXcol2*. February 2022 [cit. 2024-01-23]. Available at: https://github.com/CESNET/ipfixcol2/blob/master/README.rst.

[23] John, W., Tafvelin, S. and Olovsson, T. Passive internet measurement: Overview and guidelines based on experiences. *Computer Communications*. Elsevier. 2010, vol. 33, no. 5, p. 533–550.

[24] Karnwal, T., Sivakumar, T. and Aghila, G. A comber approach to protect cloud computing against XML DDoS and HTTP DDoS attack. In: *2012 IEEE Students' Conference on Electrical, Electronics and Computer Science*. 2012, p. 1–5. DOI: 10.1109/SCEECS.2012.6184829.

[25] Kenney, M. *Ping of Death* [online]. Insecure.org, October 1996 [cit. 2024-01-23]. Available at: https://insecure.org/sploits/ping-o-death.html.

[26] Krasnyansky, M. *Universal TUN/TAP device driver* [online]. 2000 [cit. 2024-05-12]. Available at: https://docs.kernel.org/networking/tuntap.html.

[27] Kumar, D. and Girdhar, A. Network monitoring & analysis along with comparative study of honeypots. In: *2017 International Conference on Intelligent Sustainable Systems (ICISS)*. 2017, p. 736–739. DOI: 10.1109/ISS1.2017.8389270.

[28] Kumar, S. Smurf-based distributed denial of service (ddos) attack amplification in internet. In: IEEE. *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. 2007, p. 25–25.

[29] Mahjabin, T., Xiao, Y., Sun, G. and Jiang, W. A survey of distributed denial-of-service attack, prevention, and mitigation techniques. *International Journal of Distributed Sensor Networks*. 2017, vol. 13, no. 12, p. 1550147717741463. DOI: 10.1177/1550147717741463. Available at: https://doi.org/10.1177/1550147717741463.

[30] Mammeri, Z. Z. Introduction to Computer Security. In: *Cryptography: Algorithms, Protocols, and Standards for Computer Security*. 2024, p. 1–32. DOI: 10.1002/9781394207510.ch1.

[31] Man, J. *User Interface For DDoS Mitigation Configuration*. 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/studenti/zav-prace/detail/145448.

[32] Marques, P. R., Mauch, J., Sheth, N., Greene, B., Raszuk, R. et al. *Dissemination of Flow Specification Rules* [RFC 5575]. RFC Editor, august 2009. DOI: 10.17487/RFC5575. Available at: https://www.rfc-editor.org/info/rfc5575.

[33] Mattson, T. G., Anderson, T. A. and Georgakoudis, G. PyOMP: Multithreaded Parallel Programming in Python. *Computing in Science & Engineering*. 2021, vol. 23, no. 6, p. 77–80. DOI: 10.1109/MCSE.2021.3128806.

[34] Mills, C., Hirsh, D. and Ruth, G. *Internet Accounting: Background* [RFC 1272]. RFC Editor, november 1991. DOI: 10.17487/RFC1272. Available at: https://www.rfc-editor.org/info/rfc1272.

[35] Ming, L., Leau, Y.-B. and Xie, Y. Distributed Denial of Service Attack in HTTP/2: Review on Security Issues and Future Challenges. *IEEE Access*. 2024, vol. 12, p. 33296–33308. DOI: 10.1109/ACCESS.2024.3371013.

[36] Mirsky, Y., Doitshman, T., Elovici, Y. and Shabtai, A. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In:. January 2018. DOI: 10.14722/ndss.2018.23211.

[37] Nguyen, Q. *Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multithreading, and parallel programming*. Packt Publishing, 2018. ISBN 9781789341362. Available at: https://books.google.cz/books?id=Tn18DwAAQBAJ.

[38] Parliament, C. Directive 2006/24/ec of the European parliament and of the council of 15 March 2006 on the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communications networks and amending directive 2002/58/ec. *Official J. Eur. Union*. 2006, vol. 105, p. 54–63.

[39] Phaal, P., Panchen, S. and McKee, N. *InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks*. 2001.

[40] Ramzy Shaaban, A., Abdelwaness, E. and Hussein, M. TCP and HTTP Flood DDOS Attack Analysis and Detection for space ground Network. In: *2019 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. 2019, p. 1–6. DOI: 10.1109/ICVES.2019.8906302.

[41] Rossow, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In: *NDSS*. 2014, p. 1–15.

[42] Saleh, M. A. and Abdul Manaf, A. Optimal specifications for a protective framework against HTTP-based DoS and DDoS attacks. In: *2014 International Symposium on Biometrics and Security Technologies (ISBAST)*. 2014, p. 263–267. DOI: 10.1109/ISBAST.2014.7013132.

[43] *SecurityCloud GUI* [online]. CESNET [cit. 2024-05-11]. Available at: https://github.com/CESNET/SecurityCloudGUI/blob/master/README.md.

[44] *SecurityCloud Wiki* [online]. CESNET [cit. 2024-05-11]. Available at: https://github.com/CESNET/SecurityCloud/wiki.

[45] Sharma, A. K. and Kumar, R. A Comprehensive survey of DDoS Attacks: Evolution, Mitigation and Emerging trend. In: *2024 3rd International conference on Power Electronics and IoT Applications in Renewable Energy and its Control (PARC)*. 2024, p. 185–188. DOI: 10.1109/PARC59193.2024.10486696.

[46] Škápik, A. *Detekce volumetrických útoků DoS a DDoS v reálném čase na L3 síťové vrstvě*. Brno, CZ, 2017. Bachelor's thesis. Vysoké učení technické v Brně, Fakulta

informačních technologií. Available at: https://www.vut.cz/studenti/zav-prace/detail/114862.

[47] Steinberger, J., Schehlmann, L., Abt, S. and Baier, H. Anomaly Detection and Mitigation at Internet Scale: A Survey. In: Doyen, G., Waldburger, M., Čeleda, P., Sperotto, A. and Stiller, B., ed. *Emerging Management Mechanisms for the Future Internet*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 49–60. ISBN 978-3-642-38998-6.

[48] Štofa, I. *Filtr IP toků*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vut.cz/studenti/zav-prace/detail/106422.

[49] Sultana, S., Nasrin, S., Lipi, F. K., Hossain, M. A., Sultana, Z. et al. Detecting and Preventing IP Spoofing and Local Area Network Denial (LAND) Attack for Cloud Computing with the Modification of Hop Count Filtering (HCF) Mechanism. In: *2019 International Conference on Computer, Communication, Chemical, Materials and Electronic Engineering (IC4ME2)*. 2019, p. 1–6. DOI: 10.1109/IC4ME247184.2019.9036507.

[50] Vanitha, K., UMA, S. V. and Mahidhar, S. Distributed denial of service: Attack techniques and mitigation. In: *2017 International Conference on Circuits, Controls, and Communications (CCUBE)*. 2017, p. 226–231. DOI: 10.1109/CCUBE.2017.8394146.

[51] Vojanec, K. *Akcelerace aplikace pro potlačení DDoS útoků*. Brno, CZ, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vut.cz/studenti/zav-prace/detail/145209.

[52] Všetečka, J. *Použití BGP FlowSpec v síti CESNET pro ochranu před účinkem DDoS útoků*. 2018 [cit. 2024-05-11]. Available at: https://theses.cz/id/velaf9/.

[53] Wang, A., Chang, W., Chen, S. and Mohaisen, A. Delving Into Internet DDoS Attacks by Botnets: Characterization and Analysis. *IEEE/ACM Transactions on Networking*. 2018, vol. 26, no. 6, p. 2843–2855. DOI: 10.1109/TNET.2018.2874896.

[54] Wueest, C. The continued rise of DDoS attacks. *White Paper: Security Response, Symantec Corporation*. 2014.

[55] Yu, S. An overview of DDoS attacks. *Distributed Denial of Service Attack and Defense*. Springer. 2014, p. 1–14.

Appendix A

Contents of the included storage media

The root directory of the included storage media is structured in the following way:

```
root directory
├── source/                 Source code
│   ├── nemea_daemon/       DCPro NEMEA Daemon
│   ├── sflow_parser/       sFlow Parser
│   └── conf/               DCPro sFlow Probe configuration
├── bin/                    Executable file
├── scripts/                Scripts used for testing
├── doc/                    Thesis source code
└── thesis.pdf              Thesis text
```