BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DDOS MITIGATION CONFIGURATION TOOL

NÁSTROJ PRO KONFIGURACI POTLAČENÍ DDOS ÚTOKŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DALIBOR BENEŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. LUKÁŠ ŠIŠMIŠ**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2021/2022

Bachelor's Thesis Specification

Student:	**Beneš Dalibor**
Programme:	Information Technology
Title:	**DDoS Mitigation Configuration Tool**
Category:	Networking

Assignment:

1. Study techniques and approaches of Distributed Denial of Service (DDoS) attacks. Learn and understand an application developed by CESNET that serves to protect hosts against this type of attacks.
2. Explore currently available configuration possibilities that the application offers with a specialized set of rules.
3. Design and propose a user interface and a tool that allows to configure the application and obtain statistical information that describes the process of blocking the attacks.
4. Implement the proposed tool and evaluate its functionality.
5. Discuss the achieved results and further possibilities of the work.

Recommended literature:

- According to the instructions provided by the supervisor.

Requirements for the first semester:

- Completion of items 1 to 3.

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

Supervisor:	**Šišmiš Lukáš, Ing.**
Consultant:	Huták Lukáš, CESNET
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	October 29, 2021

Abstract

Distributed Denial of Service (DDoS) attacks are a common security concern for computer networks and the services using them. One of the forms of defence is to use the DCPro DDoS Protector device developed by the CESNET association. The DDoS Protector actively mitigates ongoing attacks aimed at the protected network. The mitigation device is configurable using so-called mitigation rules. The goal of this work was to design and implement a command-line configuration tool for the DDoS Protector. A part of this work involved the creation of a reusable configuration API in Python. The configuration tool was implemented in Python using the API. It has been tested and successfully deployed as a part of the DCPro DDoS Protector package.

Abstrakt

Útoky odepření služby (DDoS) jsou v současné době častým bezpečnostním rizikem pro počítačové sítě a služby, které je využívají. Jednou z možných forem ochrany je využití zařízení pro potlačení DDoS útoků DCPro DDoS Protector vyvíjeného sdružením CESNET. Zažízení DDoS Protector aktivně potlačuje probíhající útoky, které cílí na chráněnou síť. Zařízení je možné konfigurovat pomocí takzvaných mitigačních pravidel. Cílem práce bylo navrhnout a implementovat konfigurační nástroj pro zařízení DDoS Protector s uživatelským rozhraním na příkazové řádce. Část práce zahrnovala vytvoření znovupoužitelného databázového API v jazyce Python. Konfigurační nástroj byl pomocí výše zmíněného API implementován v jazyce Python. Nástroj byl otestován a následně úspěšně nasazen jako součást balíčku DCPro DDoS Protector.

Keywords

DoS, DDoS, DCPro Protector, DDoS Protector, DCPro DDoS Protector, CESNET, configuration tool, Python, API design

Klíčová slova

DoS, DDoS, DCPro Protector, DDoS Protector, DCPro DDOS Protector, CESNET, konfigurační nástroj, Python, návrh API

Reference

BENEŠ, Dalibor. *DDoS Mitigation Configuration Tool*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Šišmiš

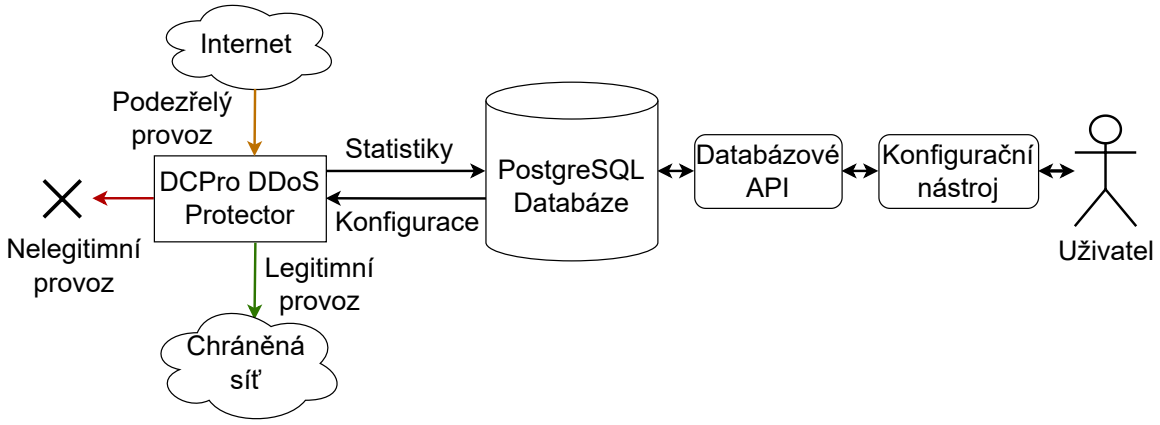
Rozšířený abstrakt

Mezi nezbytné součástí návrhu a nasazení informačních systémů patří řešení problematiky bezpečnosti. S rozšířením aplikací a systémů, které spoléhají na přenos dat po síti se tato otázka stává o to více důležitou. Z hlediska bezpečnosti existuje snaha, aby vyvíjený systém splňoval tři základní kritéria, kterými jsou integrita, důvěrnost a dostupnost. Největší nebezpečí pro zajištění stálé dostupnosti v současnosti představují útoky využívající odepření služby (DoS) a distribuované odepření služby (DDoS). Cílem těchto útoků bývá zahlcení cílového stroje nebo linky tak, aby se poskytovaná služba stala částečně nebo zcela nedostupnou pro legitimní uživatele. Tento druh útoku může být obzvlášť nebezpečný, pokud cílí na integrální služby, jako jsou například uzly páteřní sítě, nebo portály a datová úložiště státních institucí.

Jednou z možných forem ochrany je potlačení probíhajícího útoku v reálném čase. Tento přístup je oproti ostatním způsobům náročný na zdroje, ovšem přináší možnost použití pokročilých technik pro detekci a záchyt podezřelého provozu. Tato práce se úzce týká zařízení pro potlačení DDoS útoků DCPro DDoS Protector vyvíjeného sdružením CESNET v rámci projektu Ministerstva Vnitra České Republiky *VI20192022137* a na něj navazujícího *VB01000015*.

Předmětem práce byly návrh a implementace konfiguračního nástroje pro zařízení DCPro DDoS Protector s rozhraním na příkazové řádce. Konfigurační nástroj poskytuje uživateli pohodlný způsob pro manipulaci s databází, ze které zařízení DCPro DDoS Protector za běhu načítá svou konfiguraci ve formě takzvaných mitigačních pravidel. Nástroj neumožňuje pouze změnu nastavení zařízení, ale také zobrazování statistik sbíraných zařízením DDoS Protector, které jsou ukládány do téže databáze jako konfigurace.

Realizace se skládá ze dvou částí, znázorněných v Obrázku 1. První z nich je znovupoužitelné aplikační rozhraní (API) nad databází, která obsahuje konfiguraci a statistiky zařízení DCPro DDoS Protector. API je implementované v programovacím jazyce Python 3, který byl vybrán zejména pro co nejrychlejší vývoj. Databázové API poskytuje funkce, pomocí kterých lze vytvářet, upravovat či načítat mitigační pravidla z databáze. Podobně také zjednodušuje načítání statistik, které jsou periodicky sbírány zařízením DDoS Protector. API usnadnilo vývoj konfiguračního nástroje a nadále umožňuje tvorbu dalších aplikací, které ho mohou využívat pro komunikaci s databází.



Obrázek 1: Zapojení konfiguračního nástroje v rámci zařízení DCPro DDoS Protector.

Druhou částí je samotný konfigurační nástroj, který je postaven nad výše zmíněným API. Díky tomu je konfigurační nástroj taktéž implementován v programovacím jazyce Python. Návrh nástroje se zaměřil především na rozšiřitelnost o podporu nových typů pravidel a přívětivé uživatelské rozhraní. Nástroj uživateli umožňuje provádět základní operace s pravidly a jednoduché zobrazování statistik, které byly nasbírány za posledních pět minut.

Konfigurační nástroj je také připraven k použití ve skriptech a v rámci integrace s ostatními nástroji, které jsou vyvíjené v rámci projektu DCPro DDoS Protector. Integrace je především ulehčena možností výpisu dat ve formátu JSON namísto tabulky, která je sice lépe čitelná pro uživatele, ale hůře zpracovatelná programem. Další funkce nástroje využitelná pro skriptování je takzvaný *transakční mód*. Ten umožňuje načítat jednotlivé příkazy ze standardního vstupu, a tedy i ze souboru. Příkazy jsou následně provedeny v rámci jediné databázové transakce.

Konfigurační nástroj byl spolu s databázovým API řádně otestován, a to jak z hlediska funkcionality, tak i uživatelské přívětivosti. Nástroj je úspěšně nasazen jako součást balíčku DCPro DDoS Protector, a v současnosti je využíván například sdružením NIX.CZ, které zajišťuje jeden z páteřních síťových uzlů v ČR. Přesto práce na dalším rozvoji konfiguračního nástroje neustává. Příštím plánovaným krokem vývoje je například přidání podpory pro konfiguraci více instancí zařízení DCPro DDoS Protector z jediné centrální databáze, za pomoci shlukování mitigačních pravidel do skupin.

DDoS Mitigation Configuration Tool

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. The supplementary information was provided by Ing. Lukáš Huták. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dalibor Beneš
May 11, 2022

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 who introduced me to CESNET's security research projects, namely *Ing. Jan Kučera* and *Ing. Lukáš Huták*. I must also thank my family for the emotional support, without which I would not be able to complete this project.

Contents

Chapter 1

Introduction

Ever since the birth and widespread expansion of the Internet, there have been attempts to misuse the technology with malicious intent. Computer systems are being used by most private companies and public institutions to store and process information and data. While the connection of these systems to the outside world has been enormously beneficial for the users, it also leaves them open to attack. The most devastating type of attack, which involves the theft of sensitive information or outright taking of control of the system, is rare, as it can only be done against systems with very poor security measures. Today, DDoS or Distributed Denial of Service is a much more common kind of attack. A Denial of Service attack does not target the internal structure of an Internet based system or service. Instead, it tries to sever the connection the system has with the outside world, making it impossible for the external users to access it. Although the aims of such attacks vary greatly, the techniques are mostly identical. What makes them possible is the fact that any computer system has its technical limits which can be overwhelmed, such as the number of requests it can serve in a reasonable amount of time. Therefore, such an attack traditionally involves flooding the victim with requests. Distributed attacks are more difficult to defend against, as they use a large number of devices to launch an attack. This makes it more difficult to identify the source of the attack, and to separate the legitimate user from the attacker.

This thesis is concerned with active mitigation of an incoming DDoS attack with the use of the DCPro DDoS Protector mitigation device described in Chapter 2, which is developed as a project of the Czech operator of the national electronic infrastructure CESNET. Unlike other, slightly more passive forms of defence, active mitigation involves real-time analysis of incoming network traffic in search of suspicious patterns, which may indicate an ongoing DDoS attack. The DDoS Protector can be dynamically configured at runtime, using so-called *mitigation rules*. These rules contain the configuration for the various mitigation modules, each of which is specifically designed to counter a single type of DDoS attack. Currently, it is possible for the mitigation device to load those rules from a PostgreSQL database that stores them. The database also collects statistical data produced by the DDoS Protector.

The aim of this thesis is to provide a way for the administrator to configure the DDoS Protector through the database. As of now, there is no application that would allow comfortable access to the configuration data. A configuration tool with a simple command-line interface was proposed to solve this problem. Chapter 3 details the creation of an API for access to the configuration database, followed by Chapters 4 and 5, which describe the design and implementation of the configuration tool, respectively.

Chapter 2

DCPro DDoS Protector

DCPro DDoS Protector is a software device developed by CESNET[1], with the goal of protecting a network from external DDoS attacks [4]. 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 [9].

While the previous iteration of the DDoS Protector utilised hardware acceleration using a programmable FPGA[2] board (more information in [19]), the current version uses common network cards. For performance reasons, it is inefficient for the DDoS Protector to access the network cards through the operating system, like any ordinary network application would do. Instead, the DPDK framework[3] is used. In this way, the processing power of the network card can be fully utilised without any delays caused by the overhead of the different layers between the card and the application.

The work on the current version is still ongoing. Although the aim of the project is not likely to change, it is important to take into account the possibility of changes to already existing components. Therefore, one of the crucial tasks of designing supporting applications is to ensure ease of expansion or modification.

Figure 2.1 illustrates the way the mitigation device protects a network. The router which acts as the access point to the protected network redirects unverified traffic to the 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.

2.1 DDoS Attacks

Denial of service (DoS), in the context of computer networks, has various definitions and many dimensions. An example of a straightforward definition would be the one provided by the International Telecommunications Union (ITU-I) recommendation X.800: [13]

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

[1] https://www.cesnet.cz
[2] Field-programmable gate array
[3] https://www.dpdk.org

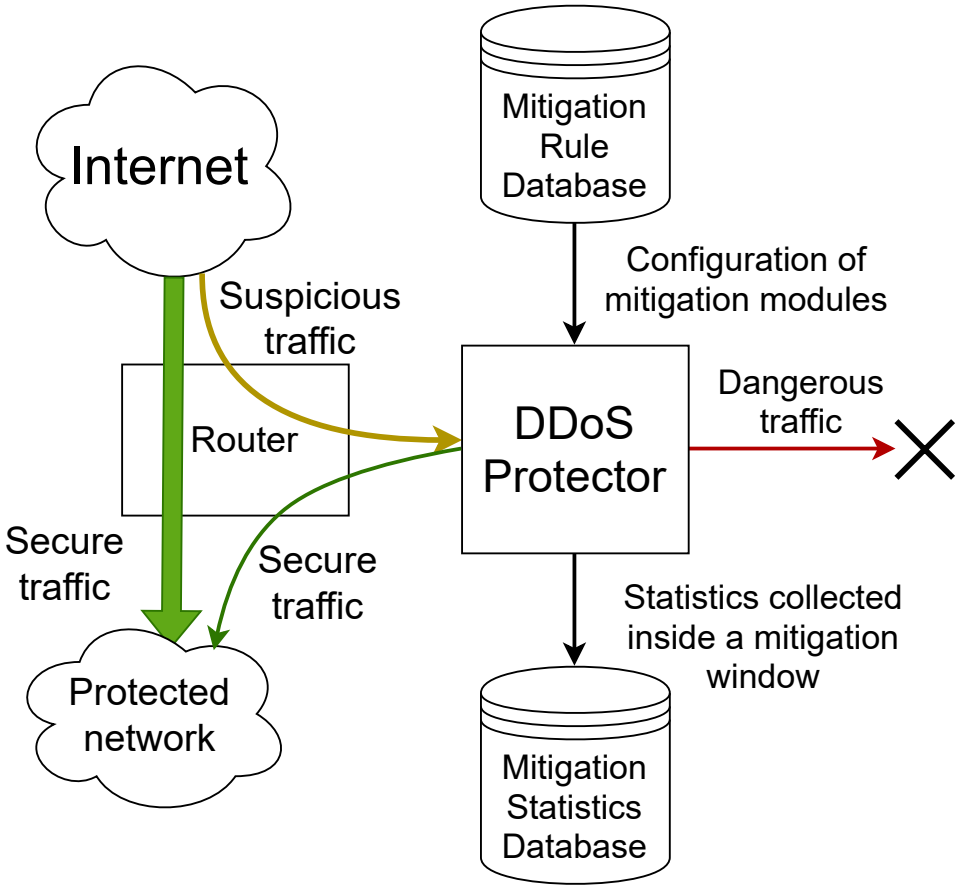


Figure 2.1: Schema of the link DDoS Protector has with the rest of the network.

The actions that cause DoS can be malicious or accidental and may originate remotely or locally, from the user or the server. The damage is caused by some form of resource exhaustion at the target point, which has a negative effect on the availability of the provided service. Such events pose a serious threat to security and dependability of integral services (e.g. search engines).

DoS Attacks are any of these actions that are carried out with malicious intent. The motivation for such attacks is not universal, and there is not even a single dominant factor. [24] describes DoS attacks and the possible ways to defend against them in general.

Distributed Denial of Service (DDoS) attacks are a subgroup of DoS attacks. The only difference is that they are launched using a large number of hosts. Most often, these hosts are remotely controlled computers that are used without the knowledge of their owners. Such computers are also called bots, as they only launch the attack when a command is issued by a master entity, called a botnet. The distributed nature of botnets makes it difficult to separate legitimate users from attackers.

[11] lists at least three basic categories of botnets, which include the Agent-handler architecture, IRC-based architecture [31] and Peer-to-peer based architecture.

DDoS attacks target and abuse different resources at various layers of the TCP/IP stack, which are listed in [32]. The targets, as shown in Figure 2.2, include networks and the implementation of the various protocols, the operating systems and the data structures and algorithms they use, and also the network-based applications. The TCP Internet protocol

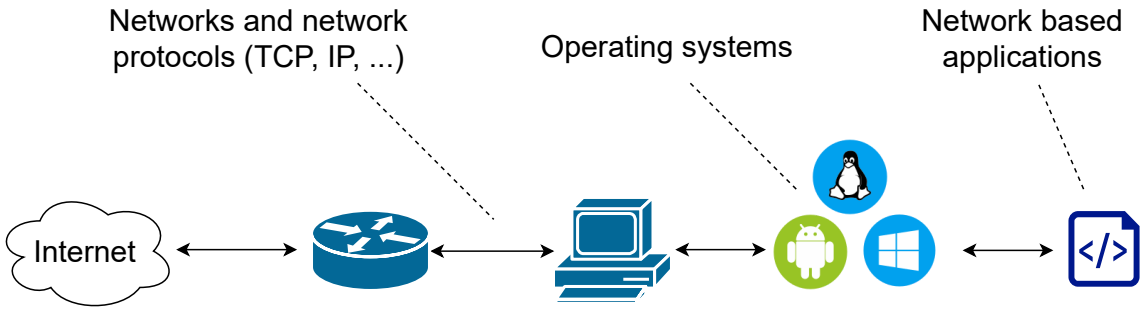


Figure 2.2: Targets of DDoS attacks

contains a number of vulnerabilities that are abused. Similarly, the HTTP network protocol uses XML structures to transport messages. XML can be exploited for an attack through coercive XML parsing, oversize XML payload, and others described in [17] and [3].

There are multiple types of DDoS attacks, which come in two categories. The first are semantic attacks, which exploit specific flaws in the implementation of certain protocols. The resources required for carrying out such an attack are asymmetrical to the resources of the victim, as the attacker may use a relatively weaker machine. An example of a semantic attack is the *land attack* [28], which makes the victim machine communicate with itself in an endless loop using a modified TCP SYN packet, until it runs out of resources to allocate new connections. The other type of DDoS attack is high-rate flooding, also called *brute-force*. High-rate flooding attacks consume a victim's critical resource to deny the legitimate user access to the provided service. These attacks require substantial resources to be successful. Therefore, they are futher divided based on the technique they use to gather more computing power.

Unlike semantic attacks, which can be stopped by fixing the flaws in the design and implementation of certain protocols and network applications, the massive amount of resources required for a successful execution of a flooding attack makes them much easier to detect. As a result, the DCPro DDoS Protector can provide an effective active defence against them. As stated in [15], the greatest problem lies in separating malicious traffic from the legitimate one. The other ways to prevent or mitigate DDoS attacks are described in [7]. They include making it harder for a host to become a part of a botnet, actively disrupting the creation of a botnet, or providing the service with an abundance of resources, so that the number of bots required for a successful attack becomes untenable. These actions can be very effective in stopping small to mid-range DDoS attacks [23]. Even though DDoS attacks are considered to be a form of crime, using legal methods for deterrence has been proven mostly ineffective [12]. The methods for the mitigation of attacks implemented by the various DDoS Protector modules are discussed in Section 2.3.

2.2 Internal structure

The main corpus of the DDoS Protector code is written in C. One distinguishing feature is the implementation of the dependency injection technique. This means that the code is composed of modules which offer an interface and multiple possible implementations of the interface. For example, collecting statistics can be done either by exporting the data to a database or by writing them into a simple file. This is done by using different implementations of the same interface to export the data. Configuration is done through

a YAML structured file, where the user can specify which implementations of different modules are to be used. This effectively makes the entire application modular and easily expandable. Additionally, the implementations of different modules can be swapped during runtime.

The core of the device is a packet processing pipeline. The device continuously receives packets from a router that acts as a gateway to the protected network. These are first disassembled, then analysed, and finally either passed back to the router or dumped. The goal is to identify packets that show signs of an ongoing DDoS attack aimed at the protected network and mitigate the threat. There are several modules that implement mitigation methods, which correspond to specific types of DDoS attacks. These methods are configured by mitigation rules, which provide crucial user-given data (filtering thresholds, etc.).

For performance reasons, the device can run multiple pipelines in parallel. Because some rules depend on past collected data, there has to be a way to transfer these data to all the pipelines. This is done using mitigation windows. A mitigation window is a time interval during which the mitigation data are collected. The device operates with two alternating mitigation windows. One is actively mitigating incoming attacks, while the other collects and aggregates values from different counters across all the parallel pipelines. Every time the windows switch status, all pipelines receive the same aggregated data. This approach is also useful for the periodic transmission of statistical data. Figure 2.3 explains how the two mitigation windows work and also what their role is in the internal structure of the DCPro DDoS Protector.

2.3 Mitigation Rules

As mentioned in Section 2.2, the various mitigation modules are configured using mitigation rules. A mitigation rule is internally a data structure that contains multiple fields with configuration data. Every rule has a type corresponding to a specific mitigation method. The rule type also specifies the data fields or attributes of the rule. Individual rule types are described in [22]. The fields may contain threshold values, subnet IP addresses etc. specified by the user. The user can create or modify rules based on observations of past behaviour. However, Machine Learning techniques can be applied to effectively devise new rules. This is described in detail in [14] and [8].

Mitigation rules have a common set of attributes. These include source and destination ports and IP addresses the rule affects, the VLAN identifier, etc. Then there are type-specific attributes, e.g. filter rules have a byte and packet threshold. A special attribute is the priority of a rule. Rules of a certain type are all processed at the same time by the respective mitigation module in a given order. More general rules should be given priority, as they are more likely to cause the whole packet to be dumped, and therefore cut the time spent processing it.

2.3.1 Rule Structure

As mentioned above, every mitigation rule is a data structure. The fields of the data structure contain the specific configuration for the mitigation module. Each rule has a type, which corresponds to a single DDoS Protector mitigation module it configures. The rule types vary in some fields, which are used to mitigate an attack of a certain kind. Therefore, there is a direct relationship between the types of attack, mitigation modules and the rules used to configure them. As shown in Figure 2.4, a certain type of attack, SYN flood in

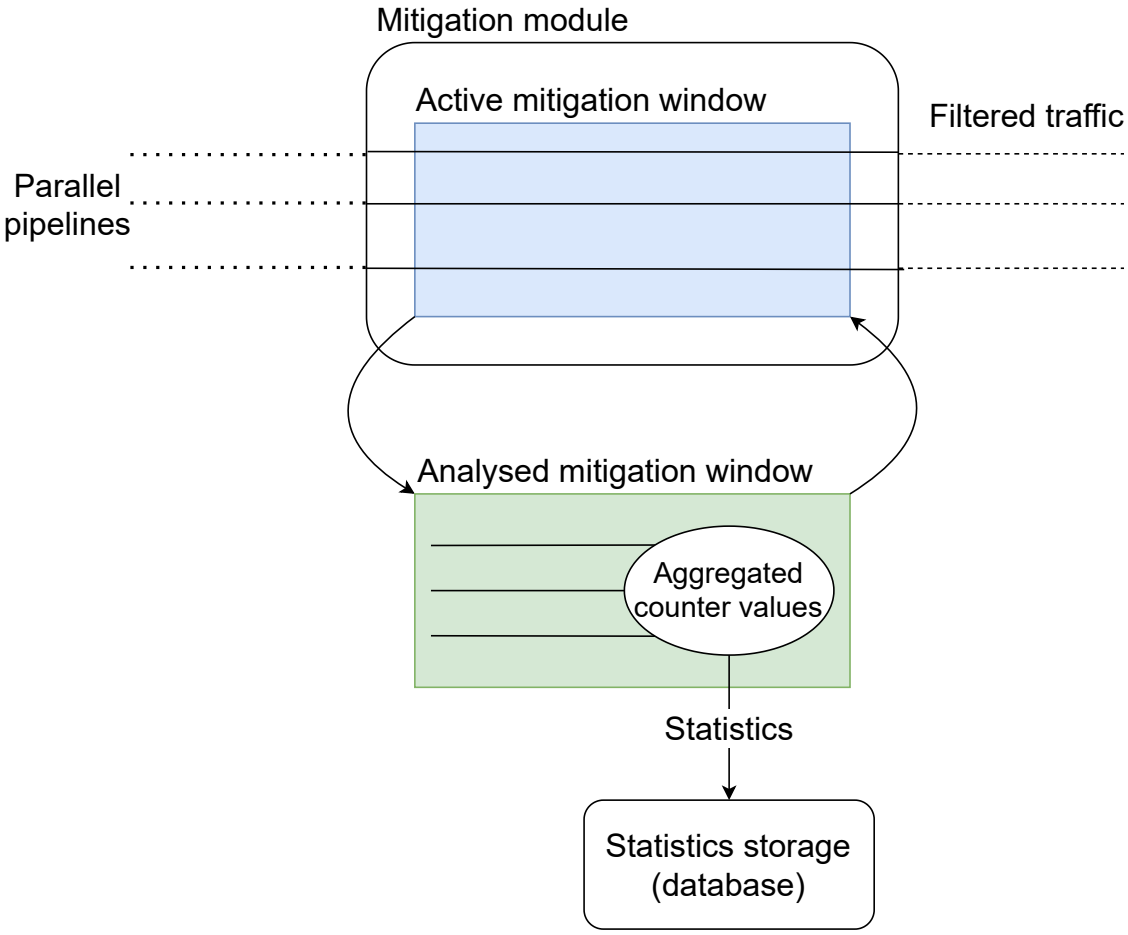


Figure 2.3: Schema of the use of the two mitigation windows.

this case, is mitigated by a specific mitigation module, which was implemented precisely to protect against it. The rules used for the configuration of this module are specific only to it, as they differ in some fields from the other types of mitigation rules.

All types share common attributes. These include the following:

- Rule identifier
- Rule status
- Dry-run mode status
- VLAN identifier
- Thresholds in packets and bytes per second
- Source and destination subnet IP adresses
- Source and destination L4 ports

The rule identifier is a numerical value which is unique for all rules of any type. It is used to track changes to the rule and as a key in the mitigation rule database described in Subsection 2.3.6.

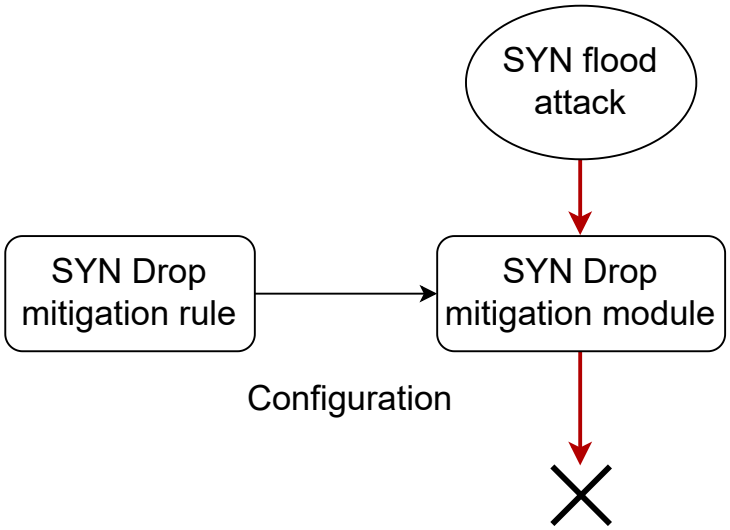


Figure 2.4: Relationship between the type of attack, mitigation module and mitigation rule.

Rule status means that the rule can be either enabled or disabled. Disabled rules are not actively used by the mitigation device, but may still be present, so that they can be turned on later.

The rule can also be in a dry-run mode. The dry-run mode stops the mitigation device from dropping or generating packets. However, statistics are still produced, as if some packets had been dropped. This allows the administrator of the DDoS Protector to test rule configuration on actual traffic before it is put to use. The downside is that dry-run mode cannot be effectively used on active methods, which involve the creation of response packets to verify the incoming traffic as legitimate.

The VLAN identifier field makes the rule match packets with a specific VLAN ID. If the value of the field is zero, the rule matches packets with any VLAN ID.

Thresholds activate the rule after reaching a given value of packets or bytes per second. The packets are calculated on L2 without the Ethernet FCS field.

The rule may match only packets with destination and source IP addresses in given subnets. Every rule may contain multiple IP addresses for different source and destination subnets.

Similarly to subnets, the rule may also only match packets with a certain destination or source port. If no ports are given, the rule applies to all packets.

The specific mitigation rule types are discussed in the following subsections.

2.3.2 Filter

The filter rule type provides configuration for a simple passive filter mitigation method. Any packet matched by a filter rule is dropped.

In addition to the common fields mentioned in 2.3.1, filter rules may also include a list of L4 protocols. Only packets transported using any of the given protocols can be matched by the rule. In case the field is left empty, all protocols are matched.

Figure 2.5 shows a filter rule represented in YAML. The rule filters traffic with VLAN 200 coming from IP 10.66.0.15 to any destination IP and communicating with destination port 80 (TCP)

```
id: 1
enabled: True
dry_run: False
vlan: 200
ip_src: 10.66.0.15
ip_dst: []
port_src: []
port_dst: 80
protocol: TCP
```

Figure 2.5: Example of a filter rule in YAML.

2.3.3 SYN Drop

SYN Drop is a passive mitigation method for protection against *TCP SYN flood* attacks, which are described in [5]. Detailed information about the method implemented by the DDoS Protector can be found in [29]. SYN flood attack is a *brute-force* attack, which exploits the flaws in the implementation of the three-way handshake used by the TCP protocol, described in Figure 2.6. In a normal TCP connection, the client first sends a TCP SYN packet; the server then allocates resources for the new connection and replies with a SYN/ACK packet, to which the client responds with a final establishing ACK packet. As visualised by Figure 2.7, the attacker exploits this algorithm by sending a large number of TCP SYN packets, which eventually causes the target machine to run out of resources, due to all the half-open connections it keeps.

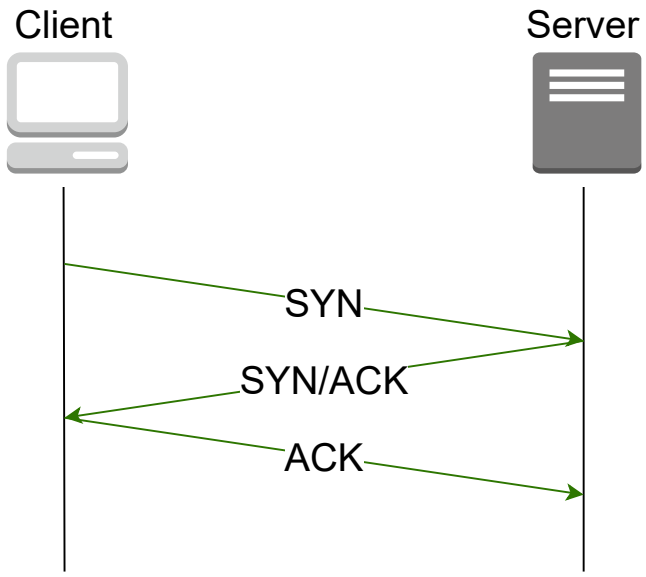


Figure 2.6: Graph showing a TCP three-way handshake.

The mitigation module implementing this method is able to inspect and block only the initial SYN-packets of a TCP three-way handshake. Therefore, it is best suited to moderate traffic, when the number of SYN packets directed to devices in the protected network exceeds the limit they can safely handle.

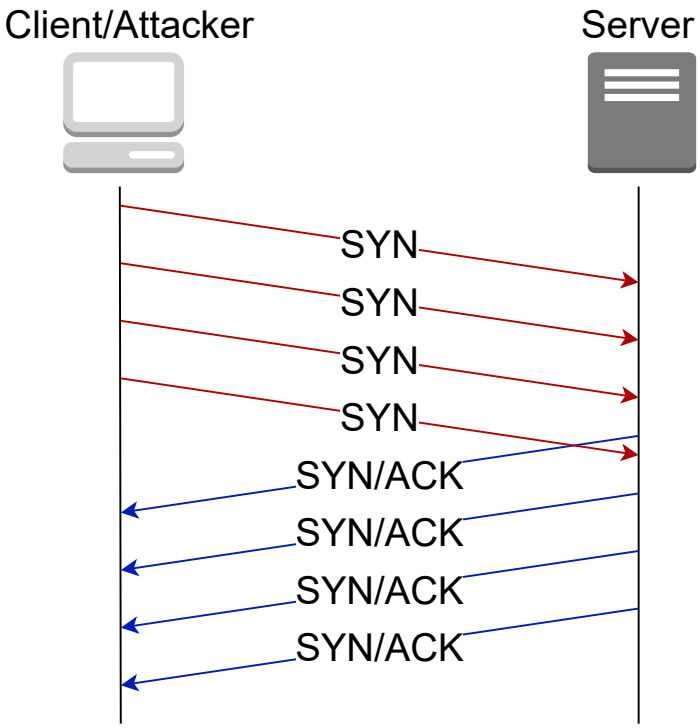


Figure 2.7: Graph showing a SYN flood attack.

SYN drop rules include two thresholds; a hard and a soft one. Soft threshold represents the number of SYN packets that are allowed without receiving any ACK packets. Hard threshold represents the total number of allowed SYN packets. If the hard threshold is reached, all subsequent SYN packets are dropped, regardless of received ACK packets.

Whenever there is an outside attempt at establishing a TCP connection to a device in the protected network, the first incoming SYN packet is dropped. This mechanism provides protection in the situation, when an attacker is randomly spoofing a large number of connections from different IP addresses. More information about IP spoofing in SYN flood attacks can be found in [18]. The only downside is that it causes a negligible delay for a legitimate user.

The following SYN packets are allowed, until the soft threshold is reached or an ACK packet is sent by the the client. In other words, the soft threshold is the number of simultaneous connections, which may be pending before at least a single one is established. All subsequent SYN packets are allowed until the hard threshold. The decision process is summed up by Table 2.1, which shows whether an incoming SYN packet is allowed or dropped by the rule, based on the number of previously received SYN and ACK packets. The SYN and ACK counters for each host are reset every 4 and 20 seconds respectively, so that active legitimate users are not blocked long-term. The counters for each host are stored within a record table. The capacity of the record table is configurable for each rule, through the third and last field specific to SYN drop rules – table exponent. Table exponent is a numerical value between 10 and 30, which sets the size of the table using the formula: $size = 2^x$, where x is the table exponent and $size$ is the resulting size of the record table. If the table runs out of capacity, some counters older records may be replaced with newer ones, thus resetting those counters prematurely.

		SYN			
		1	<2, soft>	(soft, hard>	(hard, inf)
ACK	0	Drop	Allow	Drop	Drop
	<1, inf)	Allow	Allow	Allow	Drop

Table 2.1: SYN drop threshold decision table

2.3.4 Amplification

Amplification mitigation method protects from attacks of the same name. Amplification attacks function by using an amplifier network as a middle man between the attacker and the victim machine. The amplifier network is any newtork of computers that allows for broadcast messages. This way a single broadcast packet can generate a much greater response. This is shown in Figure 2.8 An example of an amplification attack would be the *Smurf attack* [21], which abuses ICMP echo requests.

The mitigation method, which is described in [2], is passive. Its goal is to block traffic from a number of greatest contributors whenever there is an unexpected traffic increase. Amplification rules should be used to moderate traffic, when the amount of incoming packets is larger than what the devices can safely handle or when the capacity of the network connection is reached.

The method operates on a user-defined portion of traffic specified by the rule. In addition to the packet and byte thresholds, which are common to all rule types, Amplification rules also feature corresponding limits in bytes and packets per second. If a threshold is reached, the method will block traffic coming from greatest contributors, until the total traffic drops to the limit value. To determine the biggest contributors, the method starts collecting packet or byte statistics for individual hosts (source IP addresses), when 70% of the respective threshold is reached. This is called *standby mode*.

In addition to the limits, Amplification rule also contain a number of fields, which can further reduce its scope. These fields include fragmentation, L4 protocols, packet lengths, TCP flags and table exponent.

Fragmentation field allows to specify which packets are to be matched by the rule based on the fragmentation point of view. The options include:

- ANY: any packet, fragmented or non-fragmented,
- NO: non-fragmented packets only,
- YES: fragmented packets only,
- FIRST: only first fragments of fragmented packets,
- LAST: only last fragments of fragmented packets,
- MID / MIDDLE: all fragments except FIRST and LAST,
- NOFIRST: MIDDLE and LAST fragments (i.e. all fragments except FIRST).

The rule can also be used to only match packets with specific L4 protocols. There is support for matching only TCP, UDP, ICMP or SCTP packets. Multiple protocols are allowed. In case no protocols are given, all supported L4 protocols are considered.

Packet lengths field makes it possible to match packets with certain size or length. The length of a packet is considered to be the L2 packet length without the FCS field. Packet

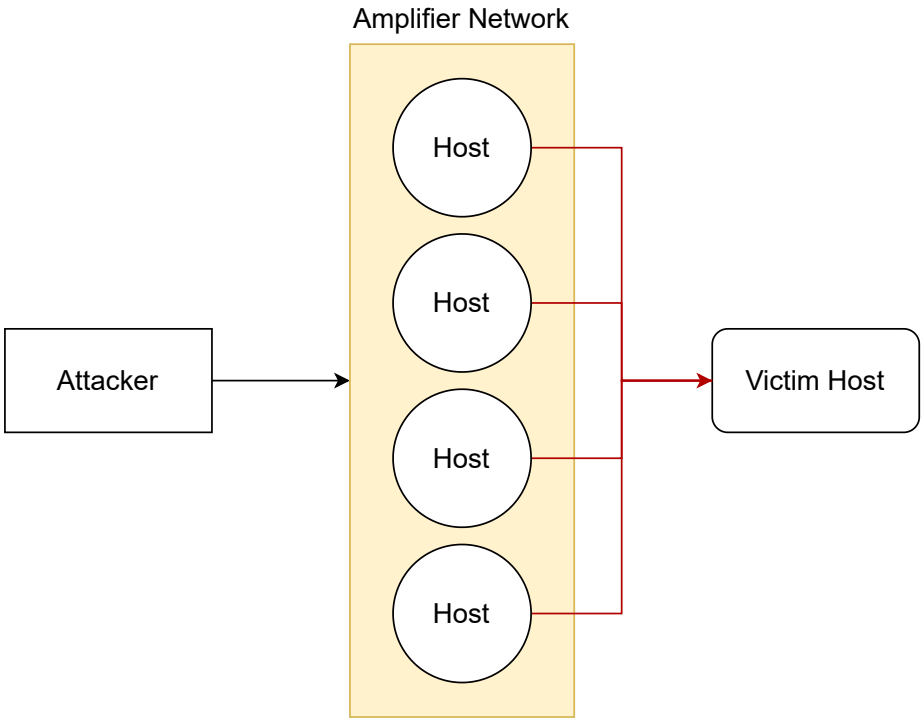


Figure 2.8: Schema of an attack involving an amplifier network.

lengths can be given as ranges. If the length of an incoming packet fits within the range, it may be matched by the rule. If no lengths are given, all packets are considered.

TCP flags field allows to restrict the matched TCP packets to those, which either contain or not contain certain flags. The function of the different TCP flags is explained in [6]. The flags include:

- C: Congestion window reduced,
- E: ECN-Echo,
- U: Urgent,
- A: Acknowledge,
- P: Push,
- R: Reset,
- S: Synchronize,
- F: Finalize.

Since a TCP packet header may contain any of these flags at the same time, these flags may be combined together, so that only the packet with the specific header is matched. An Amplification rule may contain multiple combinations of flags. Using these, a packet is accepted only if the corresponding flag is set. It is possible to negate this, by using an exclamation mark '!', which makes it so that a packet is accepted as long as the corresponding

flag is not set. For example, the combination `[!C!E!U!P!RS!F]`, will match only SYN and SYN+ACK packets.

Table exponent functions in an almost identical way to the one used in SYN drop rules. As previously stated, Amplification rules keep a hash table of IP addresses together with the byte and packet statistics for each of them. The table exponent field configures the size of hash table. In case the capacity is insufficient, the older records may be replaced by newer ones. This may lead to the blocking of wrong hosts, as the statistics will not be complete.

2.3.5 TCP Authenticator

Unlike the other methods, TCP Authenticator is interactive. This means that it can generate its own network traffic as part of the DDoS mitigation algorithm. The method can be used to counter SYN flood attacks, if passive methods, such as SYN drop prove to be ineffective. However, it is much more computationally expensive and has some other limitations.

When a TCP Authenticator rule is active, the protector acts on behalf of the protected network or device by responding to SYN packets of TCP three-way handshake. The module implements two different mitigation algorithms, `SYN_AUTH` (SYN Authentication) and `RST_COOKIES` (Reset Cookies). Every TCP Authenticator rule may use only one algorithm.

Both algorithms are based on intentionally crafting a response to a SYN request and analysis of the client's reaction. Every SYN packet is first checked, whether it comes from a previously authenticated host. Otherwise, the rule drops the request and generates a response based on the algorithm it uses.

Reset Cookies

As shown in Figure 2.9, the `RST_COOKIES` algorithm responds to a SYN request with an invalid SYN+ACK packet. The response contains an authentication token with an acknowledgement number. According to RFC 793 [1], the host is expected to respond with a RST packet that contains the authentication token. If the token is the same, the host is authenticated.

From the host's perspective, the first attempt at establishing a connection always fails. However, modern operating systems try to reestablish the connection after sending the RST packet. Therefore, the only drawback is a negligible delay.

SYN Authentication

The `SYN_AUTH` algorithm responds to a SYN request with a valid SYN+ACK packet. Figure 2.10 describes the authentication of a legitimate user. The response contains an authentication token in form of a sequence number. According to RFC 793 [1], the host is expected to respond with an ACK packet containing the token incremented by one. In case this is done, the connection is authenticated. If it is so, the algorithm generates a RST response to terminate the estalished TCP connection. Therefore, from the perspective of the host, the first attempt at establishing connection always results in being reset by the server.

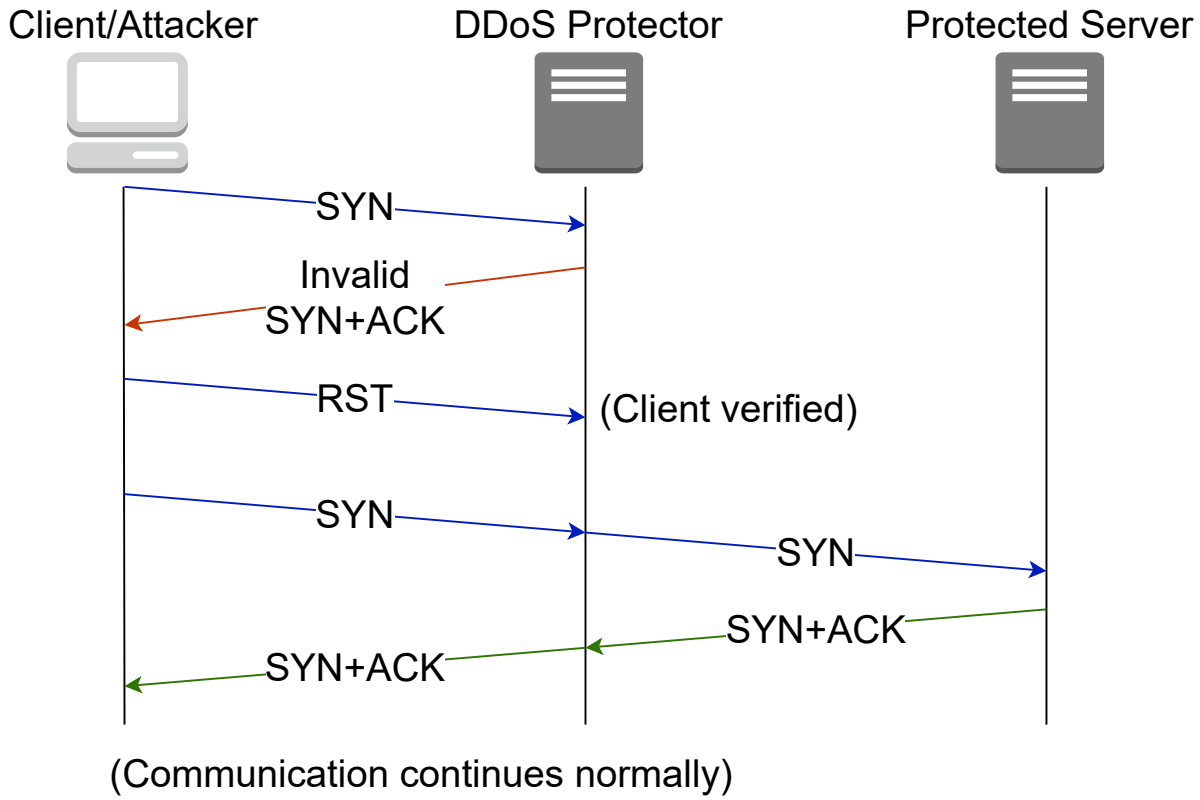


Figure 2.9: Reset Cookies algorithm

Algorithm type is one of the fields of a TCP Authenticator rule. The other fields include validity timeout, hard threshold and table exponent.

Validity timeout field contains a time interval, which has to be at least 1000 ms. This interval sets the time, after which a previously authenticated host has to be re-validated, so that there is no possible risk of an attacker using a spoofed authenticated IP address.

The records for each host are once again stored in a hash table. The size of the hash table can be configured using the table exponent field. The table exponent follows the same pattern as in SYN drop and Amplification rules. The possible values range from 10 to 30. The size of the table is then determined using the formula $size = 2^x$, where $size$ is the resulting table size and x is the table exponent.

In case there are still too many packets coming from already authenticated hosts, there is the hard threshold, which sets the maximum number of possible connections for each host. Any new attempts at establishing new connections will fail, as the SYN packets will be dropped by the TCP Authenticator mitigation module.

2.3.6 Mitigation Rules Storage

Mitigation rules can be loaded from a file or from a database. During the initial phases of development, loading rules from a JSON file was the preferred option because of the ease of implementation. However, this approach has two issues. Firstly, it is not very well scalable, as the files containing the rules have to be present on the same machine, and JSON files are not optimised for storing large amounts of data. The other drawback is that there is no

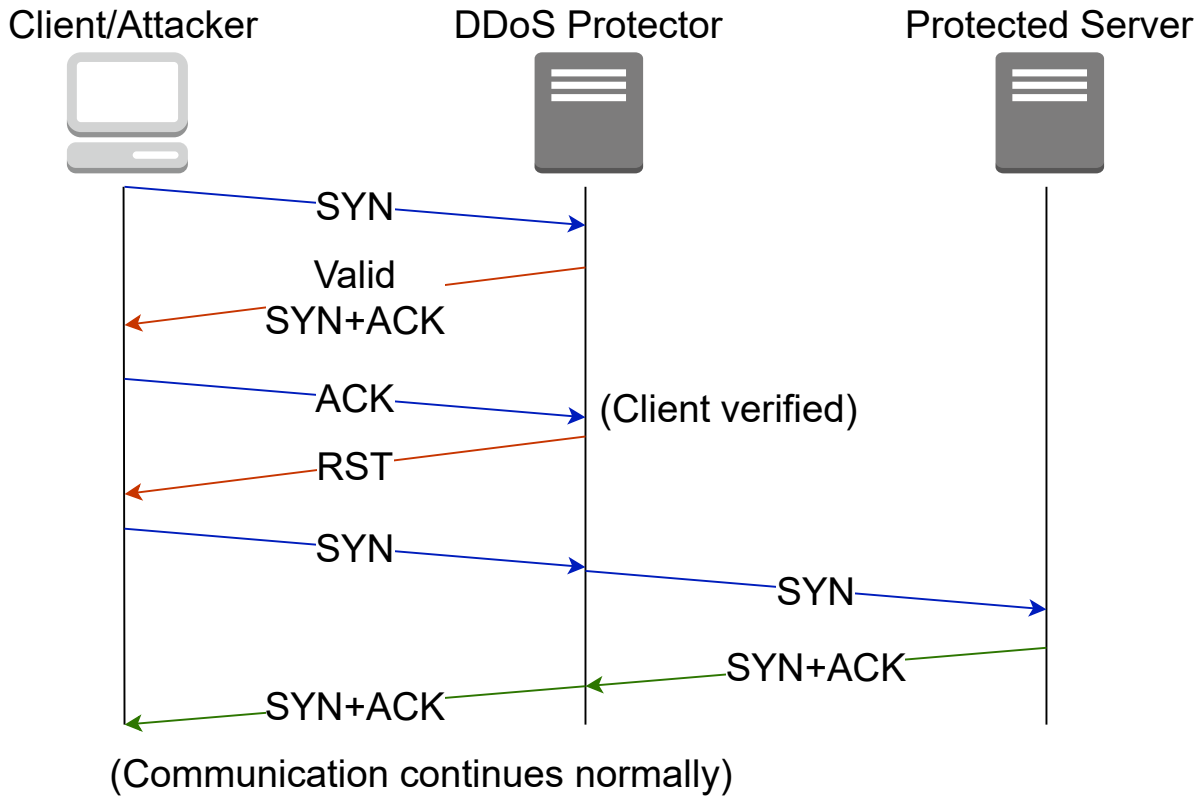


Figure 2.10: SYN Authentication algorithm

easy way to consistently and immediately change the rules on multiple running instances of the DDoS Protector. A database system solves both of these problems. On the other hand, it is more difficult to implement.

The database management system used in this project is PostgreSQL[4]. The types of mitigation rules are transformed into a schema of a relational database. This schema is described by an ER diagram in Figure 2.11. The database can run on its own standalone machine, and multiple instances of the DDoS Protector can load their mitigation configuration from the same source. Additionally, the DDoS Protector rule manager module continuously listens for database notifications. Whenever a change is made, the user can notify the device that the rules need to be reloaded. The rules are reloaded as a whole by the mitigation device. The changes take effect in the first mitigation window after they are processed.

[4] https://www.postgresql.org

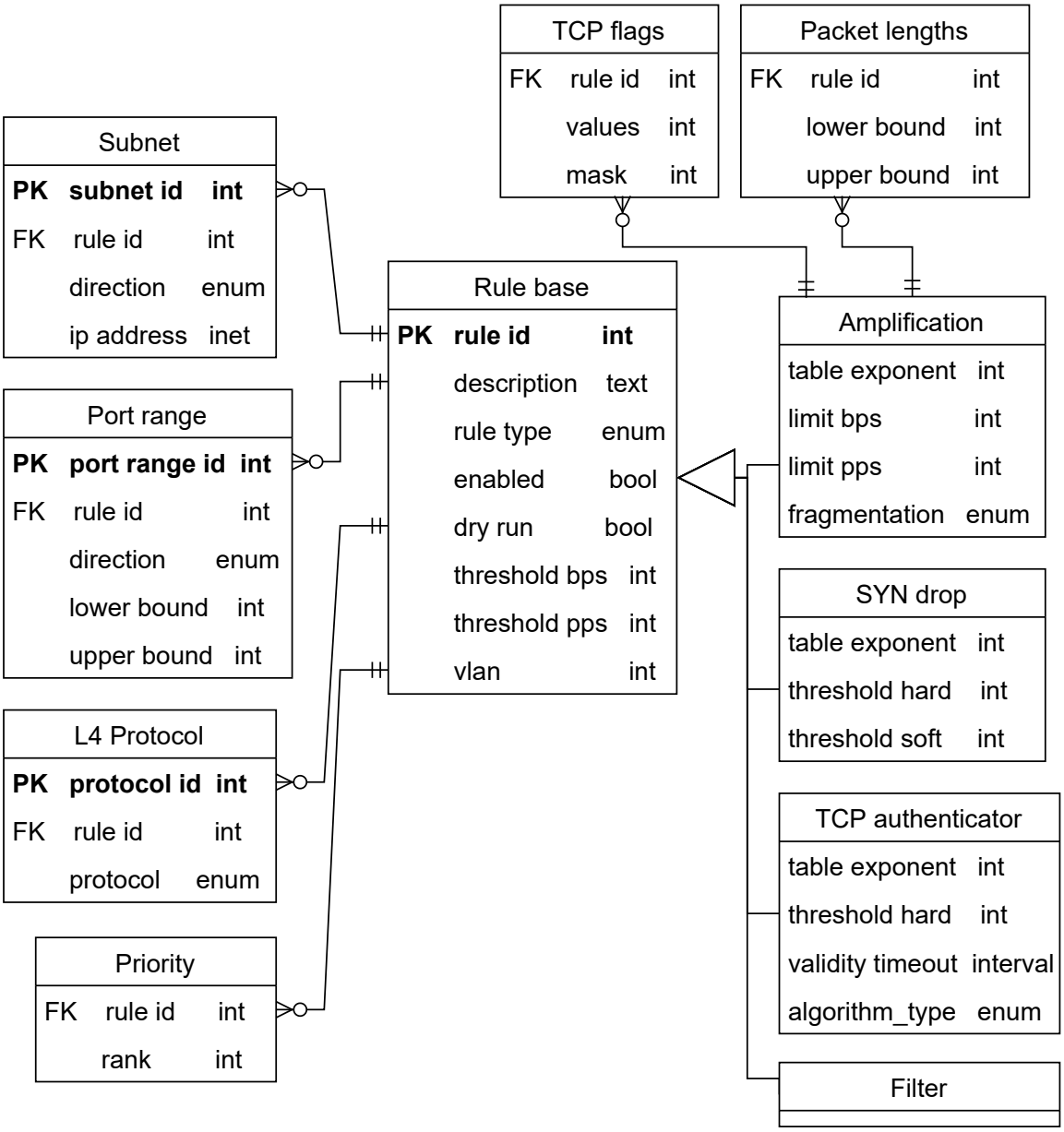


Figure 2.11: ER diagram describing the database schema used for mitigation rule storage.

2.4 Statistics

An important output produced by the DDoS Protector is the statistical data of its functioning. These data are crucial in analysing past traffic and devising changes to the configuration of the device. As stated in Section 2.2, the export of statistics occurs every time a mitigation window becomes active, which is described in Figure 2.3. Statistics are exported for all mitigation rules loaded in the device.

2.4.1 Identified Statistical Parameters

Every statistical record is first headed by a timestamp and the duration of the monitored interval. The primary statistical data are composed of the aggregated counter values the mitigation window collects, together with the current rule status. The structure of a mitigation statistics record is explained in detail in Figure 2.12, which shows the values of the different counters in both bytes and packets. These are actually separate fields but are displayed here in one place for simplification.

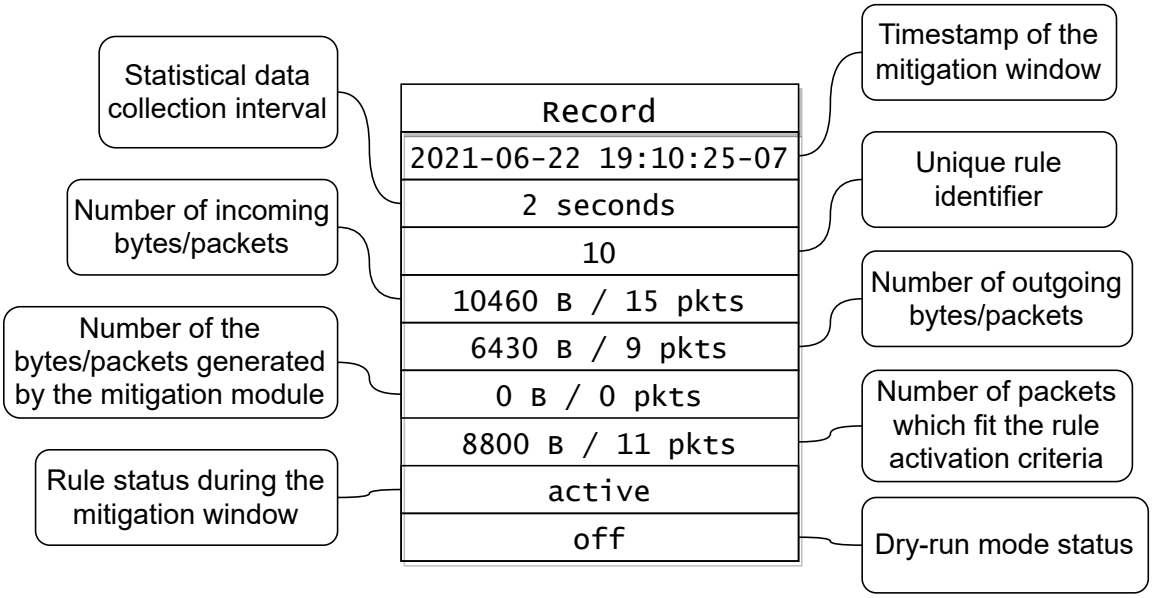


Figure 2.12: Diagram detailing the structure of a mitigation statistics record.

Every statistical value is exported in both bits and packets. Some mitigation methods become active only after receiving a certain number of packets that fit specific criteria. Some modules of the DDoS Protector can also generate packets in response to suspicious traffic. The rule can be in three different modes – idle, active, or on standby. These depend on the internal functioning of the specific mitigation module. The dry-run mode allows testing the functionality of a rule on actual incoming traffic without producing any actual mitigating effect.

One more thing to consider is that there might be multiple DDoS Protector instances running at the same time. Although multiple independent databases are possible, it is definitely more convenient and efficient to use a single one instead. To allow for this, every record can contain additional information about the specific device from which it comes from.

2.4.2 Mitigation Statistics Storage

The modular structure of the DDoS Protector makes it possible to use multiple concurent export modules. The basic way to display collected statistics is by using a log file. This approach does not scale well and is generally not usable for a more thorough automated analysis of the statistical data. As with the storage of mitigation rules, the better way is to use a database management system. Because the data are collected as a time series, it is most efficient to use a specialised system aimed at storing time series data. Several were

considered, such as InfluxDB[5] or OpenTSDB[6]. Finally, TimescaleDB[7] was chosen as the database management system to be used. Although it offers basically identical functionality to the other systems, the main advantage is that it is an extension of the PostgreSQL database management system, which is already used for the storage of mitigation rules (see Subsection 2.3.6). The result is that both the mitigation rules and the statistics can be stored in a single database.

[5] https://www.influxdata.com
[6] http://opentsdb.net
[7] https://www.timescale.com

Chapter 3

Database API

3.1 Motivation

The first step in mitigator configuration was the creation of a way to manage the database that stores the mitigation rules and statistics. As already stated in 2.4.2, both can be stored in a single instance of the PostgreSQL database management system. It is not desirable to access the data directly from the final configuration tool. This is because the database schema is still in development and, thus, subject to future changes. Instead, it is a good practise to insert an additional abstraction layer that would handle database manipulation. This layer should provide a way to connect the database query language with the programming language of the configuration tool. The aim is to further encapsulate the relatively complex series of required database operations and transform them into a simple programming interface. This API should then enable the developer to write code centered around mitigation rules instead of having to deal with SQL[1] queries to access the various database tables and the relationships between them.

Since the DCPro DDoS Protector stores collected mitigation statistics inside the same PostgreSQL database, the database API is also used for their retrieval.

Another significant factor for the creation of an API for mitigator database is that there are multiple planned ways for the configuration of the mitigation device, not only using the configuration tool discussed in this thesis but also using a REST[2] API or a similar service. The database API therefore prevents the possible duplication of code, which manages the same functionality and, in turn, makes the whole project easier to maintain and expand.

The role of the DCPro DDoS Protector database configuration API, or `dcpro_cfg_api` is perhaps more clearly described in Figure 3.1, which shows the different layers, from the mitigation rule and statistics database up to the final configuration tool. The API is here to serve as a middle man, which translates abstract operations centered around mitigation rules into actual database queries.

3.2 Design

The following sections describe in greater detail the requirements for the different use cases of the database API, as well as a basic outline of their specific functionality.

[1] Structured Query Language
[2] Representational state transfer

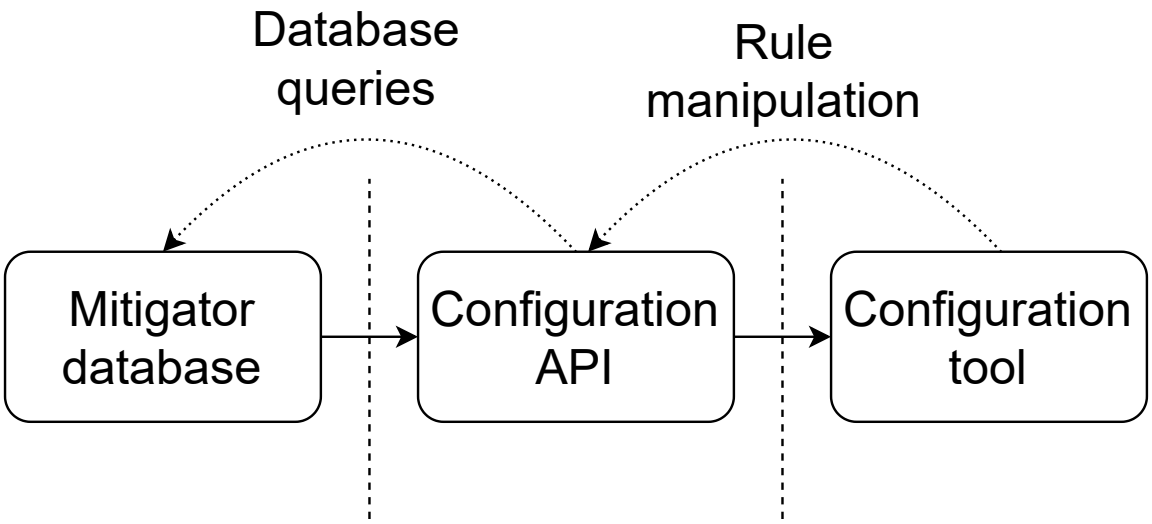


Figure 3.1: Conceptual diagram of the inner layers of mitigator device configuration.

3.2.1 Rule Representation

Generally, it is reasonable to expect the individual rules to be represented by some kind of a specialised data structure. The exact implementation depends on the programming language chosen for the API. An object oriented language would enable the usage of ORM [3], which is a technique for converting data between relational database and object oriented programming languages. This idea is further supported by the fact that the different rule types can directly correspond to classes, if a class oriented language is chosen. The benefits of ORM mainly include improved code maintainability and readability, as stated in [30]. However, this results in slightly slower execution speed of the code, which may be detrimental to time-critical applications. Since the configuration of the mitigation device does not require precise timing, the positives of the ORM approach were taken into great consideration when choosing the language for the implementation of the mitigator database API.

3.2.2 Transactions

Even though the main task of the database API is to encapsulate individual database operations, there exists a special requirement that it should still enable the developer to execute the operations in a way that resembles a database transaction or session. A database transaction is primarily defined as a sequence of database operations that satisfies the ACID[4] properties. These basic concepts are discussed at length in [10].

There is also another consideration which needs to be taken into account. While the configuration tool is not likely to feature running multiple threads, other planned services, such as a REST API might want to reserve a thread for every received request. Because it is unlikely that a single transaction would be used in handling more than one such request, the implementation of the API should reflect that the transactions are not thread-safe and are instead created specifically for every thread.

[3] Object Relational Mapping
[4] Atomicity, consistency, isolation, durability

3.2.3 Rule manipulation

As stated above, the API should offer a number of basic operations for rule manipulation. These can follow the established CRUD[5] pattern. CRUD defines operations that provide complete control over the data in persistent storage while maintaining a minimal number of them. This makes the API effective and yet simple to use by developers. The additional positive feature of this pattern is the fact that it is commonly used in application frameworks [27]. This fact primarily benefits the sustainability of both the API's code and the code using it.
To fit CRUD, the API must provide the following operations for rule manipulation:

- Create rule – Inserts a new rule into the database based on the rule attribute values provided.
- Read rule – Retrieves a rule from the database based on a value specific to the rule, e.g. its identifier.
- Update rule – Save updated values from a provided rule structure to the database.
- Delete rule – Deletes a rule based on a specific value of its attribute.

Because there are multiple types of rules, which contain different attributes, it is important to design the API code in a modular way, so that it allows for easy extension with the support for an additional rule type.

3.2.4 Retrieval of statistics

The other main use of the API is to retrieve statistical data collected by the mitigation device, which document the effectiveness of individual mitigation rules. This functionality should not become complicated. The only task here is to hide the actual database queries from the developer and return the raw data in a form native to the API programming language.

3.3 Implementation

3.3.1 Programming Language Selection

Once the general outline of the database API was established, the most important task was to choose the programming language for the implementation. The primary requirement was the existence of a well-maintained, or in better case, official, library for connecting to a PostgreSQL database. This is achieved by a number of modern programming languages, including Java, C#, Python, and C. The C language provides the official PostgreSQL library called *libpq*. The other need is that the language should offer high development speed and readability for later expansion of functionality. This often comes at some expense by reducing the execution speed of the code. However, performance is not a critical issue for the configuration of the mitigation device. The only requirement is that the execution of simple operations should not take more than two seconds, which is the proposed interval for a mitigation window.

[5] Create, Read, Update and Delete

As was already discussed in Subsection 3.2.1, using an object-oriented class language may result in a more compact and comprehensive code with the use of ORM. Because of this, the Python programming language was eventually chosen as the language for the implementation. The other qualities of Python were the speed of development, which is higher than C# and Java, and also the fact that it is already being used in the DCPro DDoS Protector project for automated testing. Due to this, it was also possible to decide to use the Python 3.6 version, so that the whole project does not require the installation of multiple Python interpreters.

3.3.2 psycopg2

Python features a PostgreSQL connection library called `psycopg2`[6], which implements a low-level API for basic database operations. Most of its features consist of sending SQL queries given by the developer to the database, fetching the results, and converting the PostgreSQL data type to those native to Python.

The SQL queries are constructed from Python strings, which are passed to the `psycopg2` API. The library escapes any special symbols and parses the queries in such a way as to prevent SQL injection, be it an attack or a bug. The API allows for the creation of multiple independent database connections. Any queries run inside them have to be committed before they take a permanent effect. This behaviour is directly related to database transactions and was used to achieve the same functionality for mitigation rule manipulation, as discussed in Subsection 3.2.2.

Since the API implemented by `psycopg2` handles the transmission of data between the database and the application using it, the DDoS Protector database API can use it as a lower layer. As illustrated in Figure 3.2, `psycopg2` manages the communication between the database and the database API, while the database API acts as the upper layer, which translates mitigation rule manipulation into appropriate SQL queries.

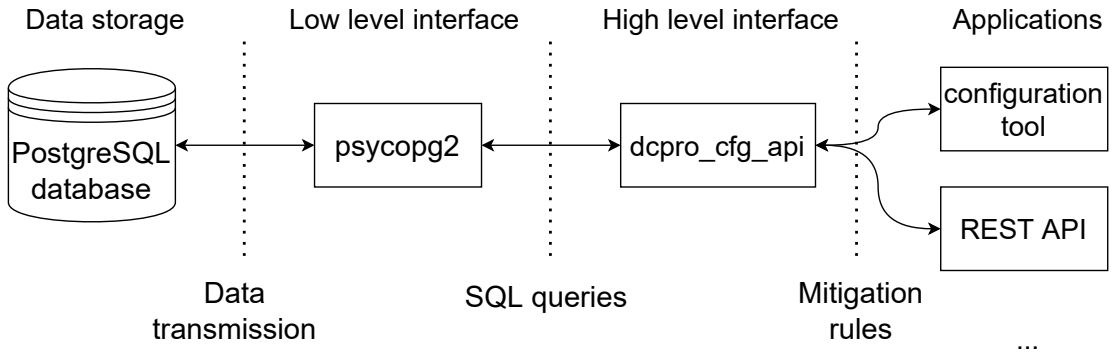


Figure 3.2: Diagram showing the relationship between `psycopg2`, `dcpro_cfg_api` and the other layers.

3.3.3 Rule Representation and Object Relation Mapping

Once the issue of data access layers has been resolved, the next step was to decide the format of the mitigation rules. As was previously discussed in the design Subsection 3.2.1, it is preferable to utilise the object oriented nature of Python, and use some sort of ORM.

[6] https://pypi.org/project/psycopg2/

Firstly, several ORM frameworks for Python were considered. The most promising were those mentioned in [26], *Django* and *SQLAlchemy*. The paper also provided useful information on the execution speed of different ORM frameworks and libraries across the programming languages discussed during the design phase. The important fact is that the speed of all the frameworks is around the same magnitude. This further justifies the use of Python as the language for the implementation. However, the discussed ORM libraries could not be utilised in the end, as one of the features of both Django and SQLAlchemy is that they define their own database schemas and tables. This was unwanted, as database schemas for storing mitigation rules and mitigation statistics were already created and in use, as descibed in Subsections 2.3.6 and 2.4.2. Instead, a custom way for basic ORM had to be created.

Although ordinary Python classes could be used to represent the various types of mitigation rules, after some considerations, it was decided that the `pydantic`[7] library would be used instead. The main reason for this was the concurrent development of a REST API, which would eventually use the database API as its base. The REST API was designed to use the FastAPI[8] framework, which can utilise `pydantic` classes for the representation of the data it works with. The primary use of `pydantic` is for type checking and conversions, as well as additional value constraints for the different attributes. This behaviour was also deemed useful for the configuration tool.

The `pydantic` library allows for the creation of supporting classes, which can provide useful functionality, such as value enumeration. To illustrate this, the code for the custom `RuleType`, `Uint16Range`, and `L4Protocol` classes is included below:

```
from enum import Enum
from typing import NamedTuple
from pydantic.types import conint, conlist

class Uint16Range(NamedTuple):
    low: conint(ge=0, le=65535)
    high: conint(ge=0, le=65535)

class RuleType(str, Enum):
    amplification = "amplification"
    syn_drop = "syn_drop"
    filter = "filter"
    tcp_authenticator = "tcp_authenticator"

class L4Protocol(str, Enum):
    UDP = 'UDP'
    TCP = 'TCP'
    SCTP = 'SCTP'
    IGMP = 'IGMP'
    ICMP = 'ICMP'
    OTHER = 'OTHER'
```

[7] https://pydantic-docs.helpmanual.io
[8] https://fastapi.tiangolo.com

The `Uint16Range` contains values of the special data type introduced by the `pydantic` library called `conint`. Its name is an abbreviation of *constrained integer*. As the name suggests, it allows for adding additional constraints to what values the attribute can contain.

The primary `pydantic` classes are constructed by inheritance from the `BaseModel` class. There is support for more layers of inheritance, which is beneficial in avoiding code duplication, since the rule types share a common base. This base can be constructed into its own class, which can be shared by the classes which represent the different rule types.

As an example, the following is the code for the class which represents the mitigation rule type filter:

```
from pydantic import BaseModel
from pydantic.networks import IPvAnyInterface

class RuleModel(BaseModel, validate_assignment=True):
    id: Optional[int] = None
    rule_type: RuleType
    description: Optional[str]
    enabled: bool = True
    dry_run: bool = False
    threshold_bps: conint(ge=0) = 0
    threshold_pps: conint(ge=0) = 0
    vlan: conint(ge=0, le=4095) = 0
    ip_src: List[IPvAnyInterface] = []
    ip_dst: conlist(IPvAnyInterface, min_items=1) = []
    port_src: List[Uint16Range] = []
    port_dst: List[Uint16Range] = []
    priority: Optional[int]

class FilterRule(RuleModel):
    protocol: List[L4Protocol] = []
    rule_type = RuleType.filter
```

All of the classes representing rule types, including the `FilterRule` class, directly correspond to database tables described in Subsection 2.3.6.

3.3.4 Data Access Layer

With the mitigation rule representation defined, it was necessary to create an inner data access layer of the database API, which would transform the data stored in the mitigation rule object into SQL queries for the `psycopg2` library and back. This layer should not be directly accessed by the developer using the API. Instead, this should be done by a different module, which would provide the public interface.

Since the database schemas for storing the mitigation rules and the statistics are separate, the data access layer is split into two parts. One part contains the code for accessing the mitigation rules, the other for accessing statistics. Both use the `psycopg2` library in a similar way, but statistics are retrieved from it in a raw form and do not go through any additional conversions, unlike mitigation rules.

Figure 3.3 provides an overview of all components of the data access layer and their dependencies. The `RepositoryExtensionInterface` is implemented by as many extensions as there are supported mitigation rule types, so only two are explicitly listed for illustration.

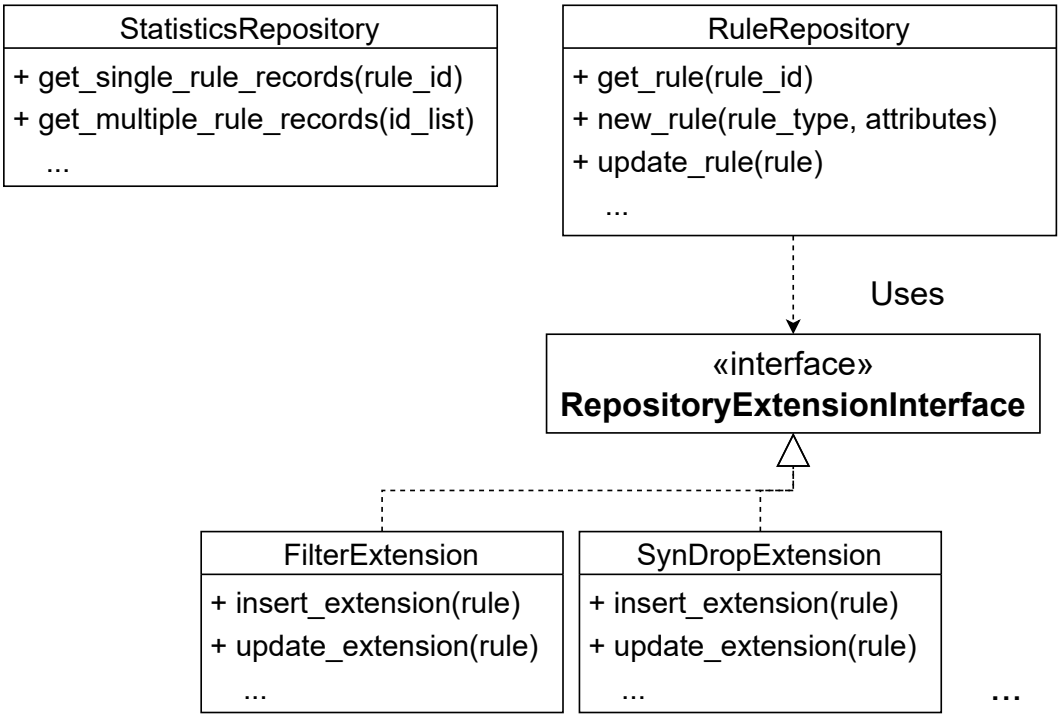


Figure 3.3: Code structure of the API data access layer

Future extension of support to additional rule types was taken into consideration. Because of this, the structure of the mitigation rule data access layer is much more modular. The core of the functionality is contained in the `RuleRepository` class. This class implements database operations common to all rule types, such as inserting the base of the rules or retrieving it. The queries specific for the various rule types are implemented by classes inheriting from the `RepositoryExtensionInterface` abstract class, which acts as the interface for rule repository extensions, e.g., `FilterExtension` for the *Filter* rule type. The extension classes are registered, and whenever there is a need for rule type specific operations, such as retrieving the rest of the rule that is not contained in the common rule base, the appropriate extension for the rule type is used. This ensures that adding support new rule types is easily done by implementing a simple interface. The methods provided by the `RuleRepository` class roughly correspond to those described in Subsection 3.2.3.

Changes to rule priority affect all rules, as the priority ranks are an uninterrupted series of numbers, starting with 0 and ending with the value of $C - 1$, where C is the rule count. Because of this, priority modification does not fit the CRUD pattern. Instead, it is done through different methods than ordinary rule modification.

`StatisticsRepository` class implements methods for retrieving raw statistical records. It is possible to retrieve records belonging to a single rule, or to all the rules. Because the total number of records can be very high, there are also optional parameters to define the bounds of an interval in which the records were collected. Records that were collected outside of this interval are not included in the data returned by the method, as the total number of records can quickly grow to a significant figure and may take a long time to fully load. For most use cases, the user only needs to see the statistics for the last few minutes. Therefore, it is more efficient to allow for the retrieval of only those statistics, which fit into a specific time window.

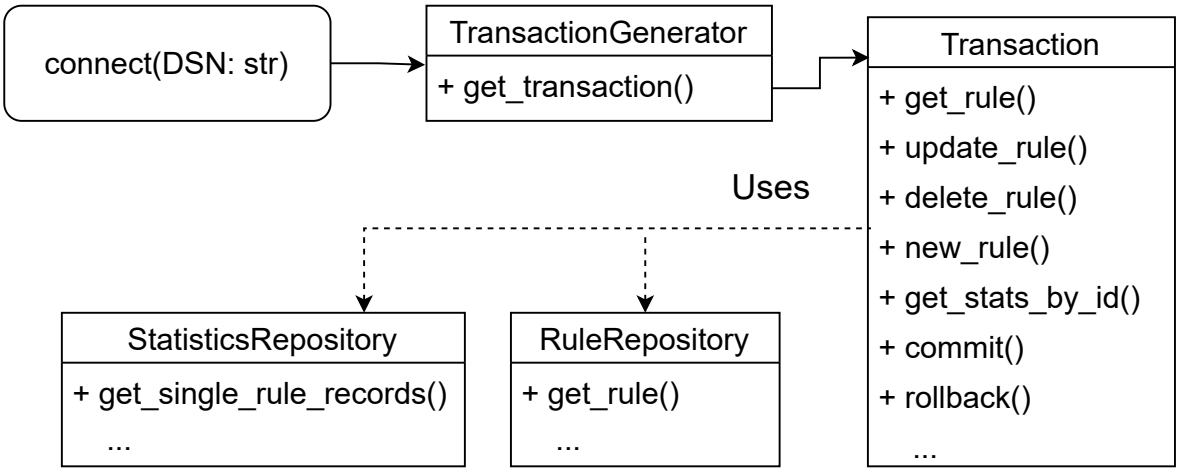


Figure 3.4: Database API code structure

3.3.5 Public Interface

As shown in Figure 3.4, the starting point of the whole API is the `connect()` function. It takes a DSN[9] string as its only parameter. This string is used by the `psycopg2` library to connect to a single specific running database instance. The function creates and returns an instance of the `TransactionGenerator` class, which, as the name suggests, generates new instances of the `Transaction` class and may be used across multiple threads, as its state is immutable.

Transactions

The `Transaction` class has two primary tasks. Firstly, it contains the public interface of the API implementation, which is accessed through its methods. The other one is that it mimics database transaction functionality. All the database operations executed by the data access layer are done using a `psycopg2` database connection created when the `Transaction` class is instantiated. The connection object is passed to the data access layer repositories at their instantiation. Thanks to this, the `Transaction` class simply propagates the `psycopg2` `commit()` and `rollback()` methods into identically named methods of the `Transaction` class. Simiarly to an actual database transaction, any changes done to the data stored in the database using the methods provided by the `Transaction` class must be confirmed by `commit()` to take permanent effect. If the application is interrupted or the `rollback()` method is called, all changes are lost. For better optimisation, `Transaction` can be instantiated with the optional argument `read_only`. If it is set to `true`, the database connection does not allow for any data modification, but it can run much faster, as it does not have to ensure there are no conflicts caused by concurrent transactions.

Transaction concurrency is allowed. The `psycopg2` database connection can work under several different isolation levels. These include the possibility of reading data modified by other transactions, may it be already commited or even uncommited. However, due to the use case of the database API, the `Serialization` isolation level was chosen, which makes the database emulate serial transaction execution, as if transactions were executed one after another. This makes it impossible to read changes done from other transaction after the

[9] Data Source Name

transaction begun. The downside is that there is a possibility of a serialization error being raised whenever the same data is modified by two concurrent transactions. In that case, one of the users simply has to wait and try again.

Methods for Rule Modification

The methods for mitigation rule modification, which are exposed by the API through the `Transaction`, primarily offer the operations listed in Subsection 3.2.3. The only difference is that there are separate methods for modifying rule priority. The reasoning for this was discussed in Subsection 3.3.4.

The different methods and their inner implementations are shown in Figure 3.5. As was stated in Subsection 3.1, the database operations are wrapped up into the API. The `set_priority()` method is an abstraction of the various methods which deal with rule priority. The methods provided by the API include:

- `new_rule()` – Used for the creation of a new rule. The rule attributes are passed using keyword arguments. The only universally required argument is the rule type. The other arguments may or may not be required as well, depending on the given rule type. Additionally, different rule types may have one or more attributes, which are specific only to them or are shared with only some other rule types. In case of any conflict, an exception is raised. On successful rule creation, an appropriate instance of a mitigation rule class is returned, based on the specified rule type. This instance contains the rule identifier and priority, which cannot be explicitly given and are instead automatically assigned at rule creation.

- `update_rule()` – Used for saving modified attribute values of an already existing rule. To retrieve a rule, it is advised to only use the `new_rule()` and `get_rule()` methods provided by the API. Using a rule object explicitly instantiated by the user is possible, but there is the possibility of raising an error if no rule with such identifier currently exists in the database. The rules are distinguished by their unique identifiers. If the identifier of a rule object is directly changed by the user, the API has no way of registering it. Any modification done by passing such a rule to the method will result in the change of data belonging to the rule with the newly given identifier. Such an action is up to the discretion of the user, and therefore not handled by the API in any other way. Changing the rule type is not allowed, and it results in an exception. This is due to the rule types having different attributes.

- `delete_rule()` – Used for removing a certain rule from the database. The implementation of this method is quite straightforward. The only argument it requires is the identifier of the rule to be deleted. There is no need for a whole object to be passed, if the user stores the identifiers externally from the application. An exception is raised if no such rule exists. For optimisation, there is a special method `delete_many()`, which removes either all rules or those which have a certain attribute value specified by a keyword argument.

- `get_rule()` – Used for retrieving rules stored in the database. Similarly to the `delete_rule()` method, the rule identifier is the only required argument. Another identical feature is the existence of a special `get_many()` method, which functions in the same way as `delete_many()`, only the rules are retrieved instead of deleted. A significant difference from other methods is that there is no exception raised if the

rule with the given identifier does not exist in the database. In that case, `None` is returned instead. Because of this, the `get_rule()` method should be used for checking if a given rule exists. The method returns an instantiated rule with all the attribute values retrieved from the database.

- `set_priority()` – Used for modifying the priority of a single rule. While it is listed here as a single method, there are actually four methods with very similar function. Functions `set_rule_priority_first()` and `set_rule_priority_last()` take a single rule identifier as the argument and place the rule at the first or the last place in the priority ranking respectively. Functions `set_rule_priority_before()` and `set_rule_priority_after()` take two arguments. The first is the identifier of the rule to be moved and the other is the identifier of the target rule. The moved rule is inserted either before or after the target rule, based on the method.

The rule objects instantiated by these methods can be used across different transactions. The only danger is that the data may have been concurrently modified, which could lead to a serialization error, as was already stated.

Methods for Retrieving Statistics

The retrieval of statistical records is served by two methods, `get_stats_by_id()` and `get_stats_many()`. As stated in Subsection 3.3.4, the age of the returned records can be restricted by specifying a lower and/or upper bound. The bounds are represented by `datetime` or `timedelta` objects. The difference is that `datetime` contains an absolute timestamp, while `timedelta` is an interval translated into relative time before now.

3.3.6 Installation

A special `setup.py` file was included to to install the database API as a Python module using the `setuptools` library with `pip` or a similar Python installation tool. Because of this, it was necessary to add the special `__init.py__` Python directory files, to include the subdirectories in the installed module. The API is used as an internal library in the DCPro DDoS Protector project, for the creation of supporting Python applications and tools.

The `setup.py` file adds useful information about the Python module, such as description, version number and used libraries. Thanks to this, Python installation tools can implicitly download the required packages, without the need for the user to specify them directly.

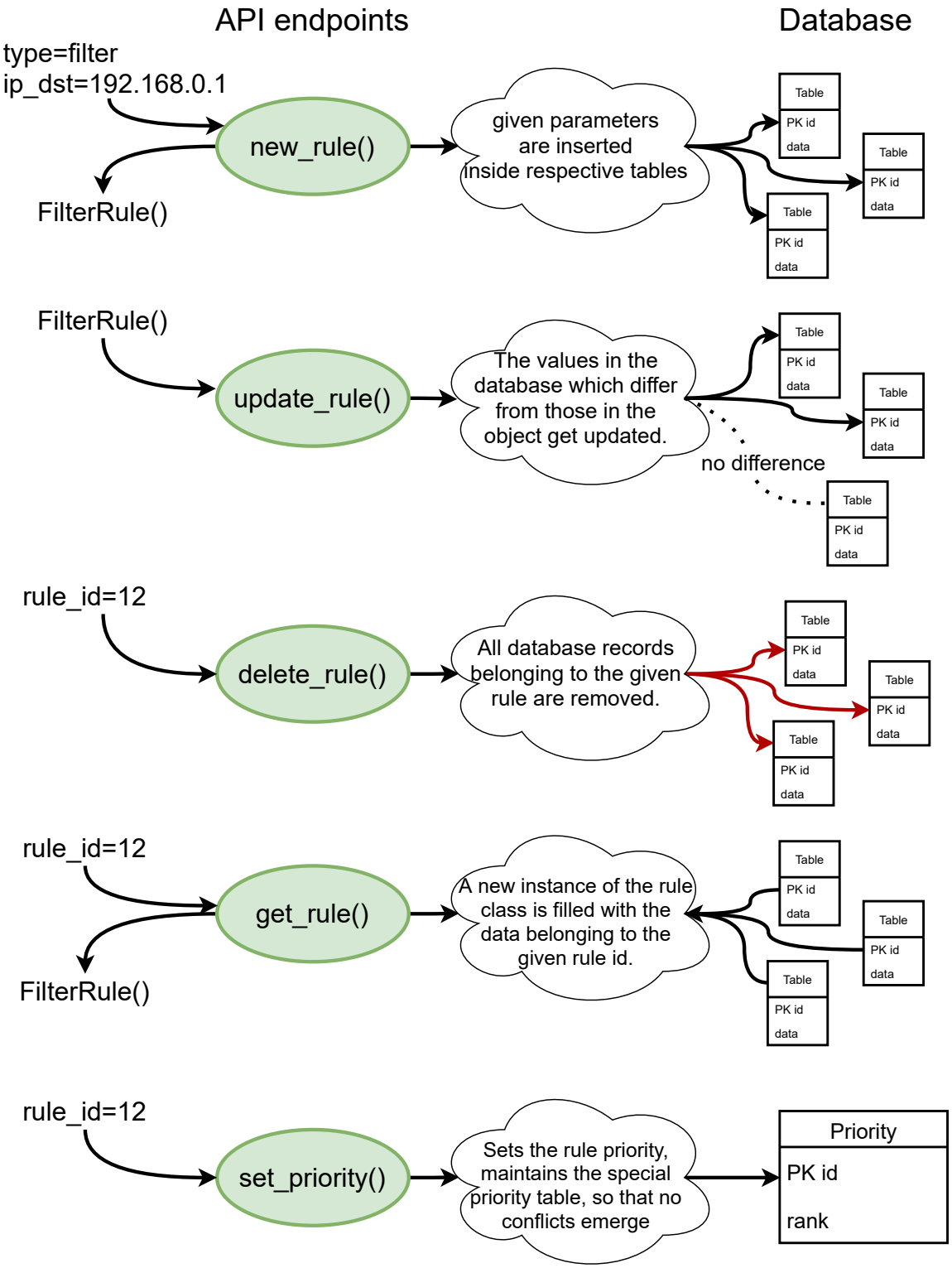


Figure 3.5: Diagram explaining the function of the different API methods.

3.3.7 Directory Structure

The directory structure of the `dcpro_cfg_api` implementation is shown in Figure 3.6.

```
dcpro_cfg_api/
├── setup.py
└── dcpro_cfg_api/
    ├── __init__.py
    ├── transaction.py
    ├── exceptions.py
    ├── rule_models.py
    └── DAL/
        ├── __init__.py
        ├── stats_repository.py
        ├── rule_repository.py
        └── rule_repository_extensions/
            ├── __init__.py
            ├── rule_repository_extension_interface.py
            ├── filter_extension.py
            ├── amplification_extension.py
            ├── tcp_authenticator_extension.py
            └── syn_drop_extension.py
```

Figure 3.6: Directory structure of the database API.

As described in Subsection 3.3.6, the `__init__.py` files are special files, which are used by the Python module installation tools to designate the directories, which are part of the module or library. The `setup.py` packaging file must be at the top of the directory hierarchy. Because of this, the `dcpro_cfg_api/` directory contains a subdirectory with an identical name. This subdirectory is installed as the Python library by the installation tools, therefore, it needs to bear the name of the API.

The topmost module contains the `transaction.py` file, which reveals the public API described in Subsection 3.3.5 through the `Transaction` class. Custom exceptions are defined externally in the `exceptions.py` file. The rule classes defined using the `pydantic` library described in Subsection 3.3.3, and all utility classes, such as port ranges and TCP flags, are contained inside `rule_models.py`. The data access layer, which was described in Subsection 3.3.4 is contained in the `DAL/` directory. This is because `StatsRepository` and `RuleRepository` classes are defined inside individual files named `stats_repository.py` and `rule_repository.py`, respectively. The practise to place the definition of different classes into separate files is used throughout the project. The reason for this is that it improves the readability and modularity of the code.

The `RuleRepository` class utilises extension classes, which implement the `RepositoryExtensionInterface`. The interface and the classes that implement it are located in separate files in the `rule_repository_extensions/` subdirectory. These files again follow the same naming convention recommended by [25], using the same name as the class, but written in the `lower_case_with_underscores` style instead of `CamelCase`.

Chapter 4

Configuration Tool Design

One of the most needed features of the DCPro DDoS Protector is finding a suitable and user-friendly way of manipulating the data stored in the configuration database. Using raw SQL queries may be sufficient for a developer working on a project. However, it may prove too difficult for an end user who needs to periodically access the data in a small number of defined use cases. The solution is to create a configuration tool for the DCPro DDoS Protector, or `dcproctl`. The `dcproctl` configuration tool has two basic functions. First, it acts as an additional abstraction layer over the data. Second, it presents a user-friendly encapsulation of data manipulation.

4.1 Current Status

Currently, the DDoS Protector project (described in Chapter 2) does not offer a way to properly modify the database data. The previous iteration of the project used to include a configuration tool with a command line user interface. Unfortunately, the present version is entirely different in its internal structure. The former tool also suffered from design shortcomings that posed an obstacle to its extensibility.

As of this day, the project developers have to use raw SQL to access and modify the data. The situation has been partially alleviated with the introduction of the API discussed in Section 3. However, this is still an inadequate solution for the end user, as there must exist an application which would interface with the user. Therefore, the use of the API helps with the development of a new iteration of the configuration tool, which has already existed in previous versions.

4.2 Specifications

To better understand the task of designing the configuration tool, it is useful to review the required specifications. The previous iteration of the configuration tool provided only a basic command-line interface with a custom argument parser and minimal functionality. The new `dcproctl` configuration tool provides an expanded number of functions, such as displaying statistics, displaying rules by their attributes, working in transaction mode etc. The `dcproctl` is still based on a command-line interface, which enables its use in scripting and on machines that do not support complex graphical interfaces.

The general goal of the application is to allow easy and straightforward configuration of the DDoS Protector using mitigation rules and a way to display the produced statistics,

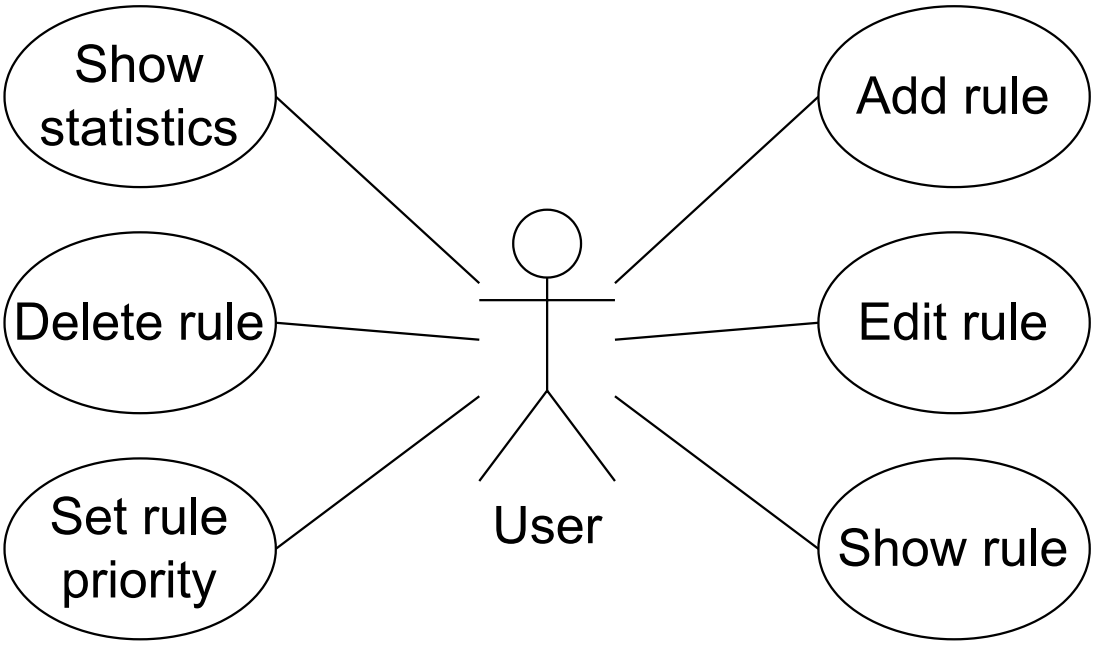


Figure 4.1: Simplified use-case diagram for the configuration tool.

to see if the configuration is correct. These use cases are shown in a use case diagram in Figure 4.1. The aforementioned API helps to provide an easy way to access the data saved in a PostgreSQL database. The API is further described in Subsections 2.3.6 and 2.4.2. To use the API efficiently, the code for the tool should preferably be written in Python, as this is the language of the API.

In the following subsections, several focal points of the design are discussed in detail.

4.2.1 Extensibility

The previous iteration of the tool was designed in such a way that made it relatively difficult to modify it. This is in part the reason why developing a completely new configuration tool is needed, instead of only improving the previous one.

Extensibility should be one of the primary concerns of this project. The reason is that there is a great possibility that additional mitigation modules will be added to the DDoS Protector. These new mitigation modules would then be configured using rule types specific to them. The code of the tool should be structured in a way that makes it easy to add support for any new rule type.

4.2.2 Basic Rule Manipulation

The core functionality of the tool should revolve around the manipulation of mitigation rules. The user should be able to comfortably add new rules, edit or delete existing rules, and change their priority. Special care should be taken to find an effective way to display the rules. Rules can contain attributes which consist of lists of variable length (e.g. list of source IP addresses to be matched). The length of those lists should not be much greater than ten items at most. This can also be an issue when editing rules. There should exist a way to modify the attributes inside a list directly instead of overwriting the whole list.

4.2.3 Displaying Statistics

The other main task is to provide a way to display the mitigation statistics. The primary use of the configuration tools is to be making minor changes to the DDoS Protector configuration or preparing scripts for later execution. Therefore, the administrator is expected to want to check if a rule functions correctly after modifications. It makes little sense to show tables of raw data, or even data collected in large intervals. Instead, there only needs to be a summary of statistics collected in the recent few minutes.

4.2.4 Transaction Mode

Lastly, there exists a requirement for a transaction mode. It should allow for multiple commands to be done as part of a single transaction. That means that either all are successfully completed or none are. The slower step-by-step configuration of the DDoS Protector device would result in a period of unwanted behaviour, within which incoming attacks might not be effectively mitigated. To counter this, a transaction could be prepared inside a file or written to the standard input and then executed at once. If the execution of a transaction from a file was supported, it would allow the user to prepare the configuration for the DDoS Protector in advance. Different sets of configurations could then easily be switched on and off. This, in turn, would make the job of the administrator much easier and open the possibility of simple automation.

4.3 User Interface and Arguments

The most important task of the `dcproctl` configuration tool is to include all the main required features identified in Section 4.2 and provide them through a simple but effective command-line user interface. There should be some separation of functionality using different subcommands for mitigation rule manipulation and the display of statistics. The design of the user interface also greatly influenced the choice of an argument parser library for the eventual implementation.

4.3.1 Design Process

The design process of the user interface went through several iterations before any work on the actual implementation began. The first issue was to cover all the use cases of the configuration tool, which were already partially identified thanks to the development of the database API (`dcpro_cfg_api`). These included basic operations over mitigation rules and the display of statistics.

A proposal on the possible format of different commands and arguments was discussed with the DCPro DDoS Protector development team. use the configuration tool in their future work. The provided feedback proved to be invaluable in identifying the basic command structure that would be comfortable to use.

4.3.2 Basic Principles

The first principle was to divide the commands into those used for operations with mitigation rules and those for displaying statistics at a sufficiently high level. This eventually allowed for relatively separate development of both functionalities and also for the different

parts of the implementation to be fully isolated, which made it easier for code review and debugging.

It was also decided to use as little of the dash '-' character as possible, so that the commands are visually clean.

Attributes of the mitigation rule, which allowed multiple values, such as various L4 protocols, were given special consideration. Naturally, it was concluded that the respective arguments need a way for the insertion of multiple values. There was also the case for attributes that cover ranges, such as the destination and source ports. These are defined by their bounds. Therefore, it was decided that there should be a way to add not only single ports, but also whole ranges.

4.3.3 Mitigation Rule Manipulation

The proposed rule commands for the configuration tool follow the same pattern as the methods provided by the `dcpro_cfg_api`. It was also decided that there should be the option to display a detailed help message at every level of the subcommand hierarchy. This is because the number of commands is quite large, and there is the possibility of expanding the configuration tool with additional functionality. To achieve this, it was concluded that the argument `help` should exist for each command, e.g.:

```
# dcproctl help
```

or

```
# dcproctl rule help
```

for mitigation rule related commands. Then this pattern should follow for all other commands.

Rule Access and Identification

Before any proposed commands provided by the tool's command-line interface are listed, it is important to consider the way mitigation rules are accessed. It was decided that the mitigation rules should be accessed using their unique identifier number. Using priority ranks was also considered, as it may be more natural for the users. However, the priority of rules changes often, especially as setting the priority of a single rule as the first may move every other rule by one place. Therefore, it makes more sense to access the rules using the identifier which is not only unique, but also constant.

Rule Creation

For the creation of a new rule, there is the `rule add` command:

```
# dcproctl rule add <type> <arguments>
```

This command requires that the first argument is the rule type. This is because the additional arguments correspond to the rule attributes, which are different for each rule type. Due to this, the command:

```
# dcproctl rule add <type> help
```

should list the specific attributes of the given rule type.

The command should execute a simple `dcpro_cfg_api` transaction, during which a single rule is created from the arguments passed to the tool. The tool should then display the identifier of the newly created rule, which is dynamically assigned by the database. The rule is also placed at the lowest priority.

A concrete example of its use would be the following:

```
# dcproctl rule add filter dstip 192.168.0.0 protocol TCP,UDP dport 40-83, 9
```

When this command is executed, a new filter type rule should be created. The rule should filter packets with destination IP addresses in the 192.168.0.0 subnet, with L4 protocols TCP or UDP and the destination port 9, and any ports between 40 and 83. Ports can be given either as single port numbers, or as whole ranges.

Rule Modification

The `rule edit` command should exist for the modification of an already existing rule. As discussed previously, the specific rule is accessed using its identifier. The commands should allow for setting a new value to any of the rule attributes using identical arguments as rule creation:

```
# dcproctl rule edit <id>
```

Because all rules contain at least one attribute, which may contain more than one value, there have to be subcommands, which would allow for additional values to be added. These are `rule edit add` and `rule edit delete`, which add and delete any attribute values from the given arguments, respectively.

Therefore, all possible commands for rule modification include the following:

```
# dcproctl rule edit <id> set
# dcproctl rule edit <id> add
# dcproctl rule edit <id> delete
```

Examples of their actual use would include:

```
# dcproctl rule edit 1 add dport 200-600, 5432 sport 0-4400

# dcproctl rule edit 1 delete dstip 127.0.0.0

# dcproctl rule edit 1 set threshold-bps 120 disabled
# dcproctl rule edit 1 set dstip 127.0.0.0, 192.168.0.0
```

The use cases for the `add` and `delete` subcommands are relatively straightforward. The given values are added or deleted from the specific rule attributes. The `set` subcommand is a bit different. It allows for setting single-value attributes, such as the value of the bytes per second threshold, turning the rule enabled or disabled, etc. However, it also makes it possible to overwrite the already existing values of a multivalue attribute with brand new values.

Displaying Rules

Displaying the mitigation rules and the values of their attributes is one of the most important use cases of the configuration tool. It is perhaps more useful than the other commands, since rule management is probably going to be mostly automated using machine learning or other such techniques, which would determine the most optimal mitigation rules. Instead, the administrator of the DCPro DDoS Protector will only need to check on the correct functioning of these mechanisms.

The rule values should be displayed inside formatted tables. It would be preferable to have two types of tables: an overview and a detailed one. The overview table can list only certain rule attributes that are common to all the rule types. This is so that it can display a large number of rules clearly and efficiently. The detailed table should function in the exact opposite manner. Its aim is to show all the attributes and their values of a specific rule.

The `rule show` command can be used to display the rules. The different variants of the command should include the following:

```
# dcproctl rule show id <id>
# dcproctl rule show all
# dcproctl rule show <arguments>
```

The first is the `rule show id` command, which is used to display a detailed table containing the attributes of a single rule.

The `rule show all` command lists all the rules in an overview table. There is the possibility of adding the `detailed` argument, which would make the tool show the rules in individual detailed tables. This functionality is not specifically required as it is not going to be used very often, especially once the number of rules grows larger.

Because the number of mitigation rules stored in the database may grow to hundreds, even a simplistic overview table would quickly fill up the space of the terminal window. One solution would be to provide support for paging. However, the sheer number of records might still be overwhelming to the user. Instead, it was decided to provide a way to list only certain rules based on the values of their attributes. This is done by using the `rule show` command with additional arguments, which directly correspond to the different rule attributes, for example:

```
# dcproctl rule show dstip 127.0.0.0 threshold-bps 150
# dcproctl rule show protocol IGMP, ICMP
```

The rules are then to be filtered, based on whether their attributes contain the given values. The number of these rules may be very high, so by default they are displayed in an overview table. However, this case would benefit the most from the option of using detailed tables instead, as some specific configurations will be used only by a handful of rules.

Rule Deletion

Rule deletion is relatively simpler compared to the other use cases of the configuration tool. The proposed commands are similar to the ones used to display the rules:

```
# dcproctl rule delete id <id>
```

```
# dcproctl rule delete all
```

The possibility of deleting rules based on the values of their attributes was also considered. The only issue is that a new rule with a certain attribute value may be created in the meantime between displaying the rules and deleting them. This would then result in the unintended deletion of an unknown rule. Instead, it was proposed to delete the rules based on their immutable identifiers only. The other use case is to remove all the mitigation rules from the database. In actual deployment, this command would be rarely used. However, it may be beneficial during the development of the mitigation device and in some other specific situations.

Rule Priority

As was discussed in Subsection 3.3.5, changes to the priority order of the mitigation rules are different from simple rule modification in that they also affect other rules. Because of this, the `dcpro_cfg_api` provides methods for changing the priority of a rule separately from updating the other rule attributes. The command-line interface of the rule directly mirrors this.

The following commands were proposed:

```
# dcproctl rule priority set <id> first
# dcproctl rule priority set <id> last
# dcproctl rule priority set <id1> before <id2>
# dcproctl rule priority set <id1> after <id2>
```

Setting the priority rank of a rule to the first or last place, which signify highest and lowest priority, respectively, is a simple operation which should be possible by using the `priority set first` and `priority set last` commands.

The `priority set before` and `priority set after` commands are a little more complicated. In either case, the other rule accessed using the identifier `<id2>` is not directly modified. Instead, it acts as a sort of reference point to set the priority rank of the rule with the identifier `<id1>`. Setting the priority before makes the `<id1>` rule directly precede the rule `<id2>`, thus it becomes exactly one rank higher in priority. The `priority set after` command works the other way, setting the priority of rule `<id1>` to be exactly one rank below rule `<id2>`.

Transaction Mode

As discussed in Subsection 4.2.4, the configuration tool should also present a transaction mode, in which the commands mentioned above could be linked together and executed at a single moment. This functionality is already present in the `dcpro_cfg_api`, which should result in a rather simple implementation.

The main question was how to input the individual commands in transaction mode. In the end, it was decided that instead of passing the commands as command-line arguments, it would be more effective to use standard input instead. Thanks to this, it would be possible to either input the commands dynamically at the time the configuration tool is executed or prepare them beforehand in a file and pass the file in place of the standard input.

As an example:

```
# cat transaction_file
    add filter dstip 127.0.0.1
    edit 5 set threshold-bps 120

# dcproctl rule transaction < file
```

The contents of `transaction_file` could also have been given after simply executing the command by itself:

```
# dcproctl rule transaction
```

Note that the commands given in transaction mode no longer need to specify the `rule` subcommand, as there are currently no plans to use this mode in the display of statistical data. Moreover, it is also not allowed to display mitigation rules in the transaction mode, as the operations are yet to be successfully committed.

Additionally, it was proposed to include a special token, which would allow the modification of a previously given rule. Since the identifier is dynamically assigned to a newly created rule by the database, it cannot be passed as an argument to the `rule edit` or `rule priority` command. Instead, the identifier can be replaced by a `PREVIOUS_RULE` token, which is assigned the value of the previously created rule, for example:

```
# cat transaction_file
    add filter dstip 127.0.0.1
    edit PREVIOUS_RULE set threshold-bps 120
    priority set PREVIOUS_RULE first

# dcproctl rule transaction < transaction_file
```

This is especially useful as the priority of a rule is also assigned during rule creation, and thus this is the only way to set the priority of a rule created inside a transaction.

4.3.4 Statistics Commands

The commands concerning the display of statistical data are comparably simpler than those dealing with mitigation rules. As the user only needs to check if a rule is currently properly functioning, a single command was suggested:

```
# dcproctl stats show <id>
```

This command shows the statistics of a single rule with the given identifier `<id>`, which were collected within a five minute window. There should also be information on the most recent record, which describes the ongoing mitigation done by the rule. Therefore, the complex functionality provided by the `dcpro_cfg_api` remains largely unused by the configuration tool and will only be used by the REST API and any more sophisticated application developed in the future.

Chapter 5

Configuration Tool Implementation

5.1 Development Process

The development of the DCPro DDoS Protector configuration tool or `dcproctl` was carried out in several iterations. Although these iterations were not complete in the sense of complex testing, they served as a way to explore the options for the implementation and identify the shortcomings of different approaches. After the initial design phase, a fully functional application was created for demonstration and feedback purposes. This application indicated the need for a separate database manipulation layer, which eventually led to the creation of the `dcpro_cfg_api` described in Chapter 3.

Already during the first design iteration, Python was chosen as the language for the implementation. There were several reasons for this. Even before the `dcpro_cfg_api` database API was created, it was already decided that several utility and automated testing tools in the DCPro DDoS Protector project would use Python 3.6 as the implementation language because of its relatively simple integration with the PostgreSQL database, and also because of the speed of development it provides over C or other commonly used languages. The arguments made for choosing Python as the implementation language for the `dcpro_cfg_api`, which were made in Subsection 3.3.1 are also valid for the `dcproctl` configuration tool.

The first is, as already stated, the speed of development. Since Python code contains fewer lines of code than many other programming languages, as stated in [20], it is suggested that it has somewhat better development speed and also maintainability than languages, which need more lines of code to express the same functionality. On the other hand, because Python is an interpreted scripting language, geared towards fast development, it does not utilise many of the features which make compiled static languages faster. Because of this, [16] describes ways to optimise performance of Python code. A prime example of such optimisation is that Python libraries are often wrapping C code, which is relatively much faster to execute. Since the configuration of the DDoS Protector is not time-constrained, execution times of up to 1 second can be tolerated, which is the minimal duration of a mitigation window, as described in Section 2.2. The new configuration is loaded by the DDoS Protector only when the mitigation windows switch status.

During development and user interface testing, performance issues were never encountered. Therefore, Python remains the optimal language for the configuration tool, and there are currently no plans for any large-scale optimisation effort, such as implementing whole modules in C.

5.2 Command Line Interface

Because the primary reason for the creation of the `dcproctl` configuration tool is to act as the user interface for the configuration database, the implementation of the command-line user interface was one of the crucial points of the implementation as a whole. At first, it was attempted to stay as close to the design proposal described in Section 4.3 as possible. To do this, a custom argument parser had to be created. However, the custom implementation of an argument parser soon proved to be untenable, especially since established libraries already exist, which handle this matter well.

The following libraries for parsing arguments were considered:

- Argparse[1] – Integrated as part of core Python libraries. Allows for complex commands, which can be constructed from multiple layers of subcommands. This is achieved by creating an argument parser object, which can produce additional subparsers, which are used to parse the different subcommands. This effectively creates a hierarchy. Arguments can be added to any parser at any level. Positional arguments differ from optional arguments only in the `-` or `--` prefix. Allows for type casting during the parsing of the argument string. Automatically adds a `-h/--help` argument to every parser, which outputs a formatted description of any subcommands and arguments that can be used. The arguments are then stored inside a `Namespace` object as key-value pairs. Well scalable, but a bit more complicated API than the other compared libraries.

- Docopt[2] – Python implementation of the *Docopt* command-line description language. Relatively simple, uses a string formatted in a similar way as a `argparse` help message. This makes it somewhat more complicated to maintain, especially with the large number of commands the `dcproctl` configuration tool is supposed to support. Because of this, the problem with the mandatory `--` prefix remains for optional arguments. The parsed arguments are then returned for processing in the form of an ordinary Python dictionary object.

- Click[3] – A third-party Python library for argument parsing. Instead of using objects as in `argparse` or strings as in `docopt`, the API of the `click` library provides decorators for functions. These decorators are used to define the command and any arguments which the functions can then access as ordinary local variables and, therefore, very easily process. Nested commands and subcommands must be defined in a way similar to `argparse`, by registering them with the functions that manage the parent command. This makes the code a little more complicated as there are two different concepts used, one for registering commands and the other for registering arguments.

After some consideration, the choice finally settled on the `argparse` library. The other two libraries proved to be much less scalable to the number of arguments and the complexity of commands required by the `dcproctl` configuration tool. Maintaining a very large string or of multiple decorated functions and the relationships between them would require significant effort. On the other hand, while the `argparse` API may be too complex for

[1] https://docs.python.org/3/library/argparse.html
[2] http://docopt.org
[3] https://click.palletsprojects.com/en/8.1.x/

simple applications, here multiple commands may be added in layered tree, where every node represents a subparser, which manages a single command, and its arguments. This hierarchy may be put into a number of compact functions, each managing a single subparser. This eventually proved to be reasonably scalable and maintainable, as new functionality was gradually introduced to the `dcproctl` configuration tool during the many iterations of development.

In the end, it also became obvious that if either library is chosen, the format for the optional arguments suggested in Section 3.3.5 is unattainable, as all the discussed libraries use the `--` prefix to distinguish between positional and optional arguments. Thus, this format was abandoned, and the `--` prefix was introduced for all the optional arguments. On the other hand, it was possible to use the proposed command structure in its entirety without any significant changes.

5.2.1 Final Interface Implementation

To reiterate, the commands proposed during the design phase, as described in Subsection 4.3, could be used by an `argparse` argument parser in almost identical format.

The only change made to the commands is that the `by-attributes` subcommand was added to the `rule show` command. Unlike the proposed variant, the `argparse` library does not provide an effective way to ensure mutual exclusivity of arguments and subcommands. Due to this, if the original proposal was implemented, the other variants of the `rule show` command would have access to the same arguments. Although this could be solved later during argument processing, the argument parser would produce misleading help messages. Therefore, it was simpler to assign mutually exclusive arguments under separate subcommands.

The implemented interface is shown in Appendix B.

The arguments, which correspond to the attributes of the different rules, are used in the `rule add`, `rule edit` and `rule show by-attributes` commands. These arguments are used to set the values of a newly created rule or a modified rule or, in the case of the `rule show by-attributes` command, to display rules based on the value of their attributes.

Because all types of mitigation rules differ slightly from each other in the attributes they have, the corresponding arguments are also different for every rule type. The rule types and the arguments, which can be used in the creation, modification, or display of the rules of those types, are listed below:

As was already stated in the introductory paragraphs, the primary difference from the proposed argument format is the addition of the `--` prefix in front of every argument. This is because the arguments are not positional and can be given in any order. Most of those arguments are also optional only, and in the case that they are not given, a default value is used instead. The `argparse` argument parser provides only one way to achieve the functionality needed for both of these cases, which is to specify these arguments as optional. However, optional arguments in `argparse` always require the `--` prefix, so that they can be distinguished from the positional arguments or their values.

5.3 Internal Structure

The internal structure of the `dcproctl` code follows the use of the `argparse` library used for argument parsing. The functionality concerning the configuration of the argument parser and their later processing is separated. The mitigation rule manipulation and display of

statistics is also fully separated. Due to this, it was possible to implement the functionality step-by-step in the individual iterations of development.

5.3.1 Directory Structure

There are separate directories for the manipulation of mitigation rules and for reading and displaying statistical data. Compared to the `dcpro_cfg_api` database API, there are also additional files used to build and install the `dcproctl` configuration tool.

The root directory contains files for the building and installation of the application. The `dcproctl/` subdirectory contains the files with the actual Python code. The `tool.py` contains the `main()` function, which acts as the entry point for the whole configuration tool. The other files are discussed in the following subsections, together with the function they fulfil. The functionality related to mitigation rules and statistics is separated into the `rule/` and `stats/` directories. The directory graph shown in Figure 5.1 serves primarily as a way to give a better sense of the internal structure of `dcproctl`, as the number of files is somewhat higher than in `dcpro_cfg_api`.

```
dcproctl/
├── Pipfile
├── Pipfile.lock
├── build.sh
├── setup.py
└── dcproctl/
    ├── __init__.py
    ├── __main__.py
    ├── tool.py
    ├── parser_types.py
    ├── rule/
    │   ├── __init__.py
    │   ├── parser.py
    │   ├── processing.py
    │   └── rule_type_handlers/
    │       ├── __init__.py
    │       ├── rule_type_handler_interface.py
    │       ├── filter_handler.py
    │       ├── amplification_handler.py
    │       ├── tcp_authenticator_handler.py
    │       └── syn_drop_handler.py
    └── stats/
        ├── __init__.py
        ├── parser.py
        └── processing.py
```

Figure 5.1: Directory structure of the configuration tool source code.

5.3.2 Argument Types

The argparse library allows for the type conversion of arguments during their parsing. The arguments are converted from raw string form using callback functions or class constructors. The arguments can be supplied with custom-made callback functions, which may convert the argument based on the value. This is useful in cases such as IP addresses, which use different classes for the representation of IPv4 and IPv6 addresses.

However, the most important feature is the formatted error message that the `argparse` library produces if the given argument could not be converted. This can be further utilised in checking the value of the argument, and not only the type. The callback typing functions print an error message every time a `TypeError` or `ValueError` exception is raised. For example, rule identifiers use the standard `int` class, which is the native way to represent any integer values in Python, and which is also used by the `dcpro_cfg_api` and `psycopg2` libraries. To make the rule identifier be represented using the `int` class, while also making sure the value is always positive, there has to be a typing callback function, which first checks that the argument does not contain a non-positive integer, and then casts the string as an `int` value.

Typing Functions

The `parser_types.py` file contains custom typing callback functions, which are specific to the arguments used in the `dcproctl` configuration tool.

Whenever a rule identifier is an expected argument, a special typing function is used, which may or may not include the special `PREVIOUS_RULE` string token, which was discussed in Subsection 4.3.3, as a valid value, in addition to any positive integer. While it would theoretically be possible to check if a rule with the given identifier already exists in the database, the `argparse` documentation recommends that the typing functions should remain simplistic and be concerned primarily with type conversion and basic value checking. Any complex functionality can be handled later when the parsed arguments are processed.

The Amplification rule attribute TCP flags is a special case. The task of the `dcpro_cfg_api` database API is to transfer data from Python to a PostgreSQL database and back. In both the database schema and the rule classes, the TCP flags are internally represented as two 8 bit fields, one containing the values of the flags and the other the mask, which specifies which flags are contained in the specific collection. This means that every TCP flags object can contain from one to eight flags, depending on the mask, which can be either allowed or forbidden, depending on whether the value bits corresponding to the mask are either 1 or 0. Although this format can be efficiently handled by the mitigation device, it can be confusing for a human user. Instead, the argument is given as a combination of capital letters which represent the abbreviations of the different flags. Multiple different flags can be given as a single argument value. Any flag can be prefixed with an exclamation mark `'!'`, negating the following flag. The whole conversion from the textual to the values-mask form is done by a typing function. For example the argument `--tcp-flags S!U` is converted into the database representation, with values `0b00000010` and mask `0b00100010`, which identifies the considered flags. As can be seen, the `U` or *Urgent* TCP flag is represented by the third highest bit in both the values and mask fields.

All rule types also have fields that contain destination and source ports. These are internally represented using ranges, which are defined by their lower and upper bounds. The respective argument values can be given as either a single port identifier, which is an unsigned 16 bit integer, or a range of ports, which has the `<lower_bound>-<upper_bound>` format,

where the bounds are port identifiers. The typing function then creates a `dcpro_cfg_api Uint16Range` object from the given bounds. If only one port identifier is given, it is given as both the lower and upper bound of the `Uint16Range` object.

Other fields of mitigation rules contain primarily integer values. However, these fields differ in the size of the integers they contain, with some values being unsigned 16 bit, 32 bit or even 64 bit. To manage this without the need to create multiple typing functions, a special function `constrained_integer()` was created. This function takes the constraints as arguments and produces a callback function, which can be used by the argument parser. This resulted in a more compact code and increased readability.

5.3.3 Argument Parsing

Due to the reasons discussed previously in Section 5.2, the `argparse` library was chosen for argument parsing. This subsection details the implementation itself, and not the individual arguments, which were described in the section explaining the choice of the library. Although the format of the arguments in general was clear even before the work on the implementation itself began, there still was the problem of deciding the proper way, which would allow for efficient code maintenance and the possibility of expanding the functionality of the configuration tool.

The `argparse` argument parsing library provides an API using `ArgumentParser` objects. A parser object has to be instantiated at the beginning. The object then contains methods for adding arguments or subparsers, which are also `ArgumentParser` objects. The subparsers are used to parse the different subcommands. These can provide the essentially same command structure as described in Subsection 5.2.1, as they can be nested indefinitely. Therefore, the subparsers create a hierarchy, which can also be represented by a tree. Most optional arguments were only added to the subparsers at the bottom of the hierarchy, since there is almost no overlap of functionality between the different commands. If any arguments were added to a subparser in the middle part of the parser hierarchy, the argument could only be given in between the commands on the command line, and the argument would be shared by all the commands which use the specific subcommand corresponding to the subparser. This approach was used only once, in the case of the `rule edit` command, where a positional argument `<id>` is given before the final subcommand. This is precisely because all of the subcommands for rule modification have rule identifier as a required argument.

The root `ArgumentParser` object is instantiated as part of the entry point inside the `tool.py`. Thanks to this, it is possible to create fully separated modules, which can then use subparsers created from the root argument parser. This makes the `dcproctl` configuration tool more extensible, as it is easy to separate the functionality.

Both mitigation rules and mitigation statistics have a `parser.py` module within their respective directories, which were mentioned in Subsection 5.3.1. These modules follow a simple pattern: There is a function for every added subparser, inside which new subparsers are called and their respective functions are called, or there are arguments added to the parser object. Figure 5.2 illustrates how every level of subcommands and their possible arguments is contained within a single function. If a command uses multiple subcommands, the nested functions branch out. It can therefore be said that the functions for adding subparsers and arguments are modelled after the same hierarchy as the commands and subcommands defined in Subsection 5.2.1 follow.

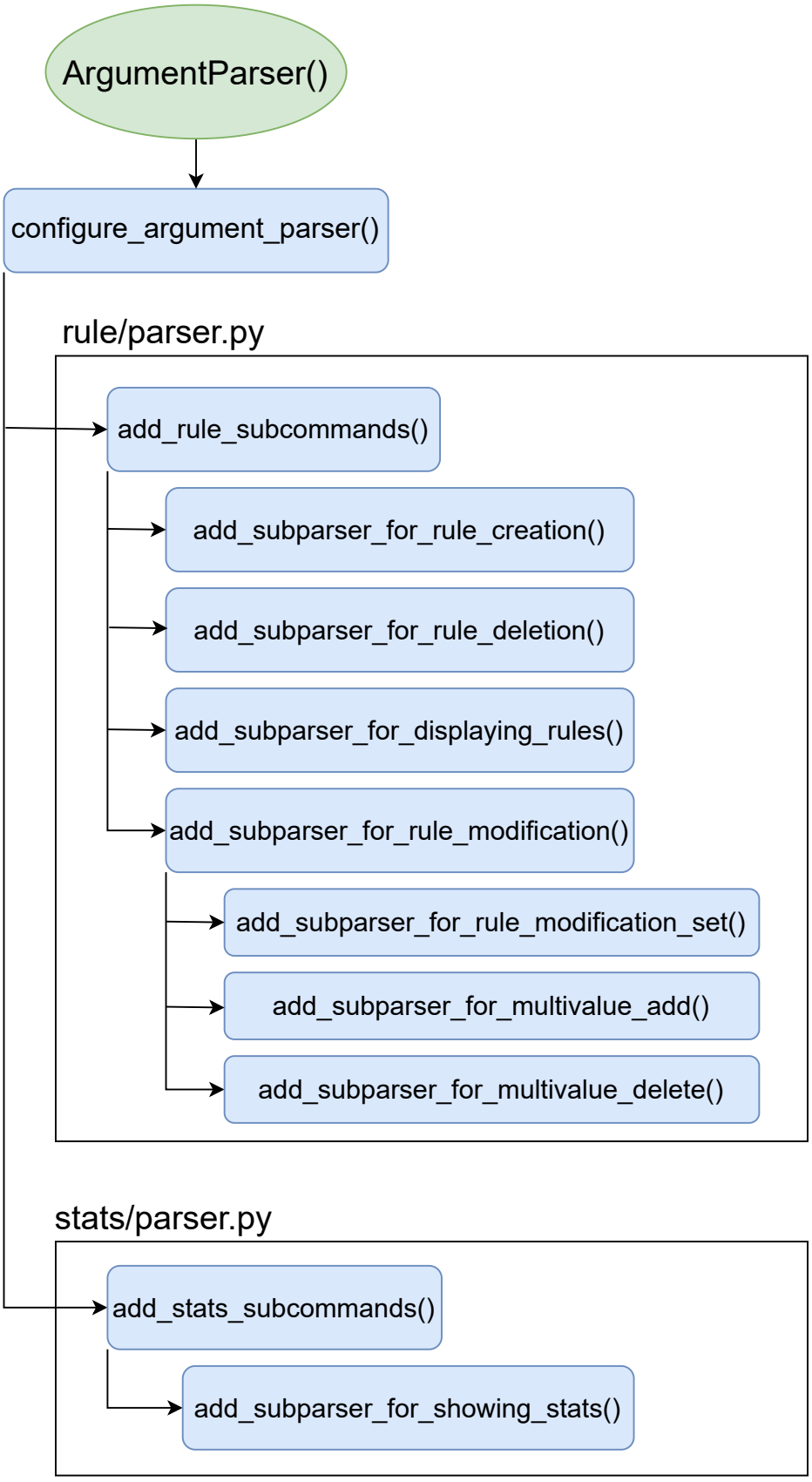


Figure 5.2: Hierarchy of functions for configuration of the argument parser.

5.3.4 Argument Processing

Once the argument parser is configured to include the arguments and commands, it can be used for parsing. The `argparse` argument parser either returns a `Namespace` object which contains the parsed arguments and commands, or terminates the application after printing out a formatted error message. The parsed arguments have already been converted to the required data types using the typing functions described in Subsection 5.3.2.

The returned `Namespace` object then needs to be processed, so that the action attributed to the command can be performed. Processing of parsed arguments is done in a similarly to how the subparsers and arguments were added to the argument parser. The commands form a tree structure, where the actual database operations involving `dcpro_cfg_api` are performed at the end nodes. Similarly to how Figure 5.2 describes the hierarchy of functions for adding arguments to the argument parse, the `processing.py` files provide functions that each process a single subcommand layer of a given command by calling another function for dealing with the lower level. Only at the bottom level are actual operations that involve mitigation rules or statistics executed. These include rule creation, modification, the display of statistics, or any of the use cases discussed in Section 4.2 and implemented in the command-line interface, which was described in Subsection 5.2.1. The `dcpro_cfg_api` `Transaction` object is created at a higher level, at the time it is clear that the used command will be able to modify any database data, or if the transaction can remain in the read-only mode. Thanks to this, the functions at the bottom, which execute the actual commands, can be used twice, once in the normal mode and also in the transaction mode, which is explained below.

The processing of commands, which deal with rule creation, modification or deletion is straightforward. The respective methods of the `dcpro_cfg_api` are called, with the processed arguments used as the parameters. The display of mitigation rules and statistics did require some more consideration, to find the most optimal way of showing the data to the user. The thought processes and the resulting implementation is discussed in the following subsections.

Display of Rules

As discussed in Section 5.2.1, which dealt with the implemented command-line interface, the rules can be displayed using three commands, in two table formats.

The `rule show by-id` displays an explicit number of rules based on the given identifiers. Therefore, the default action here is to present the user with a detailed table, such as the one shown in Figure 5.3, for every given rule. The detailed table lists every attribute of the rule and its value. In case of multivalue attributes, they are listed in ascending order of value.

The `rule show all` and `rule show by-attributes` commands can display a previously unknown number of rules, especially if multiple users have access to the same database simultaneously, or if there is some way for automated rule management. Therefore, the default for these commands is to display an overview table for every rule type. The rules in the overview table are listed according to their priority. This makes the overview tables useful for checking any rule priority changes.

All commands have the option to use the other type of table instead, which may be beneficial in certain cases.

In addition to that, there is also the option of printing a JSON representation of the rules. This is made simpler by the fact, that the `pydantic` library, which is used for

```
+-------------------+--------------------+
|         Id        |         1          |
+-------------------+--------------------+
|      Priority     |         0          |
|        Type       | tcp_authenticator  |
|    Description    |        None        |
|      Enabled      |        True        |
|      Dry-run      |         on         |
|        VLAN       |         0          |
|   Threshold bps   |         0          |
|   Threshold pps   |         0          |
|     Source ip     |                    |
|   Destination ip  |    10.55.0.0/16    |
|    Source ports   |                    |
| Destination ports |       0-1000       |
|  Validity timeout |      0:01:00       |
|   Threshold hard  |         0          |
|   Table exponent  |         18         |
|   Algorithm type  |    RST_COOKIES     |
+-------------------+--------------------+
```

Figure 5.3: Example of a detailed rule table.

rule representation by the `dcpro_cfg_api`, provides its objects with a built-in method, which returns their JSON representation inside a Python string, ready to be printed. The JSON representation is much easier for automated parsing, and it makes the `dcproctl` configuration tool much more useful for scripting and integration with any other tools, which may be devised as part of the larger DCPro DDoS Protector project.

Display of Statistics

As discussed in Subsection 4.2.3, it was decided that the tool should have a limited scope of showing statistics for a single rule at a time. As shown in Figure 2.12, the displayed table contains the statistical values of the last record, aggregates from the last minute, and aggregates from the last five minutes. The aggregate values should include the average per second, maximum per second, and total sum of the values collected inside the whole window.

5.3.5 Rule Type Handlers

One of the identified aspects that the newly created `dcproctl` configuration tool should feature was modularity and extensibility. The part of the tool, which is most likely subject to changes, is the one managing the needs of the different mitigation rule types, which primarily include adding and processing rule type specific arguments.

As illustrated by Figure 5.5, the operations that needed to be implemented separately for each rule type formed a single interface. This interface was eventually explicitly declared in a `RuleTypeHandlerInterface` abstract class. The classes inheriting from the abstract class,

```
Current rule statistics
-----------------------
time elapsed since last record: 0.94s
rule status: active
dry run: off
```

counter values	last record	1 m avg	1 m max	1 m total	5 m avg	5 m max	5 m total
input packets	1.15 Mpps	863.95 kpps	1.28 Mpps	50.97 Mpkts	629.30 kpps	1.28 Mpps	50.97 Mpkts
activation packets	1.15 Mpps	863.95 kpps	1.28 Mpps	50.97 Mpkts	629.30 kpps	1.28 Mpps	50.97 Mpkts
passed packets	65.95 pps	13.73 pps	69.00 pps	810.00 pkts	10.00 pps	69.00 pps	810.00 pkts
dropped packets	1.15 Mpps	863.94 kpps	1.28 Mpps	50.97 Mpkts	629.29 kpps	1.28 Mpps	50.97 Mpkts
generated packets	1.15 Mpps	863.94 kpps	1.28 Mpps	50.97 Mpkts	629.29 kpps	1.28 Mpps	50.97 Mpkts
input bits	554.29 Mbps	414.70 Mbps	612.11 Mbps	24.47 Gbits	302.07 Mbps	612.11 Mbps	24.47 Gbits
activation bits	554.29 Mbps	414.70 Mbps	612.11 Mbps	24.47 Gbits	302.07 Mbps	612.11 Mbps	24.47 Gbits
passed bits	39.04 kbps	8.13 kbps	40.85 kbps	479.52 kbits	5.92 kbps	40.85 kbps	479.52 kbits
dropped bits	554.25 Mbps	414.69 Mbps	612.11 Mbps	24.47 Gbits	302.06 Mbps	612.11 Mbps	24.47 Gbits
generated bits	498.83 Mbps	373.22 Mbps	550.90 Mbps	22.02 Gbits	271.85 Mbps	550.90 Mbps	22.02 Gbits

Figure 5.4: Example of a table containing recent statistical data.

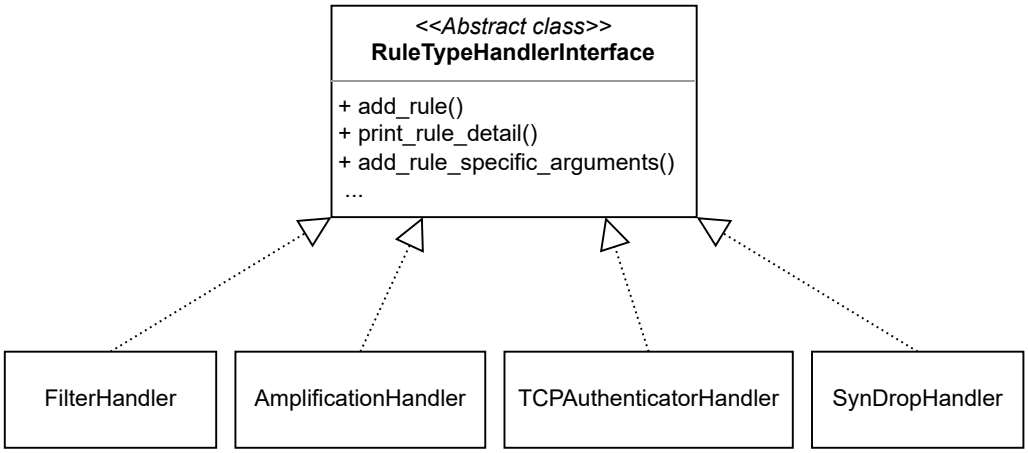


Figure 5.5: Rule Type Handler Interface class

or rule type handlers, implement the interface by providing methods for the manipulation with a single rule type.

There are two basic ways the functions can operate. The first is by extending the basic functionality. This is the case in adding arguments to the various subparsers that are needed for rule creation, modification, or display of rules by attributes. This is always done for all the rule types at once, so it only makes sense to include the rule type specific arguments in the methods implemented by the handler classes. The common base of arguments is added separately by a function located in the `parser.py` file, and then the respective methods of all the handlers are called to add the rule specific arguments to the argument parser.

The other case is processing the arguments for rule creation, modification or display. These also include a common base, so that there is no code duplication between the handler classes. However, this base is implemented by the `RuleTypeHandlerInterface` class. The handlers use the methods of the parent class as part of their implementation. This is due to the fact that whenever a new rule is created, or an already existing rule is modified, only one method, specific to the rule type, needs to be called.

The benefit of using this pattern became apparent already during development, when a new rule type was introduced to the mitigation device, the TCP Authenticator. Adding support for a new rule type proved to be swift and effective, which is very important for any future expansion of the set of mitigation rule types.

5.3.6 Transaction Mode

The transaction mode, which was already discussed several times, such as in Subsection 4.2.4, allows the execution of several commands inside a database transaction, so that they either all succeed or the whole transaction is aborted and no permanent changes are made.

Commands are given using the standard input instead of command-line arguments. The transaction is executed after the end of file special character is read. This makes it possible to use files to prepare transactions in advance or to make them reusable.

The commands themselves are separated from each other by a newline character. In case the line containing the command and the arguments is too long, it is possible to escape the line break by placing a slash `/` character at the end of the line.

Arguments are parsed in the same way as if they were ordinary command-line arguments. The built-in Python library `shlex` manages to produce a list of arguments similar to the

one passed to the application from the command line from an ordinary Python string. This approach solves any possible edge cases, such as escaping special characters.

Due to the fact that the `Transaction` object is instantiated at the time when the higher level of the command is processed, such as `rule edit` or `rule add`, the functions at the lower level, which process the arguments into new mitigation rules or modify already existing rules, can be reused in processing the transaction mode as well. This makes the implementation significantly simpler, as the functionality specific only to the transaction mode includes only the parsing of the arguments from the standard input stream and the processing of top-level commands.

5.4 Argument Autocompletion

Since the number of arguments used for the creation of a new rule or the modification of an already existing one is quite high, there was a need to improve the user experience. The `argparse` library provides a useful feature in the `-h/-help` argument, which is automatically added to any command and, if passed to the application, displays all the possible subcommands and arguments that can be used.

However, there is also an issue with the command and argument names. These are too long to be typed quickly, especially since the arguments use the rule attribute names in the original form without any use of shortening or abbreviations. Although this approach makes the identification of the corresponding argument-attribute pairs clear, the time needed to type those long names can become a matter of frustration for the user.

To remedy both problems at once, it was decided to use the package `bash-completion`, which provides the `bash` shell with the ability to autocomplete command line arguments with the press of the `<TAB>` key. There is a third-party Python library called `argcomplete`, which provides a wrapper for the `argparse` argument parser, so it is compatible with `bash-completion`. The `argcomplete` library also comes with commands, which can be used to create `bash-completion` scripts for specific CLI applications using the `argparse` library for argument parsing. The changes needed to make the `dcproctl` tool compatible were minimal; the `argcomplete` library needed to be added as a dependency and the `argparse` parser had to be wrapped with a `autocomplete()` function coming from the `argcomplete` library at the place in the code after it is configured, right before the arguments are parsed. This happens at the top level, in the `tool.py` file, so the change affects all the possible arguments and commands that the tool supports.

5.5 Installation

Similarly to the `dcpro_cfg_api`, it is possible to build and install the tool manually using `pip`. The `__main__.py` file makes the installed Python package executable through the interpreter, through the `-m dcproctl` argument. The `setup.py` file exists to manage the requirements for the dependencies, but unlike the `dcpro_cfg_api`, it also includes the option to install the `dcproctl` as a command-line application.

After the tool was implemented and tested in an isolated environment, some consideration was given to the way the tool is integrated within the larger DCPro DDoS Protector project. This included providing additional user support, such as writing a manual that could be used alone or as a part of the manual for the entire DDoS Protector mitigation device.

However, the more important issue was to find a way to build and install the `dcproctl` tool, so that it was easy to deploy and distribute. The DCPro mitigation device project uses CMake[4] to automatically generate a Makefile for the compilation of the code and the creation of the executables for the mitigation device and also any additional utility tools, which also include the `dcproctl`.

The project can then be distributed as an RPM[5] package. The most difficult task was the integration of the Python tool into the RPM package. Python applications and libraries are traditionally packaged using Python native tools, such as `pip`, which also manage any dependencies. However, there is also the possibility of installing the `dcproctl` configuration tool as a self-contained application using the shiv[6] utility. Thanks to this, it can be distributed as a standalone application without the need for the user to manually install any other dependencies.

The `build.sh` script builds the `dcproctl` configuration tool as a shiv application using a `pipenv` virtual environment. Pipenv uses the files `Pipfile` and `Pipfile.lock` to manage dependencies. The main advantage of using `pipenv` is that it allows for the stabilisation of versions of the various dependencies. Thus, it removes the possibility of conflicting versions of dependencies to be installed. The CMake configuration includes a custom command for building the `dcproctl` shiv executable using `build.sh`. This file is packed into an RPM package, together with a `bash-completion` script created using an `argcomplete` command.

The resulting RPM package can be unpacked using a package management tool such as yum. The self-contained `dcproctl` is installed as a command-line application. Autocompletion of arguments is also possible right away if the `bash-completion` package is installed, since the `argcomplete` script is installed into the `/etc/bash_complete.d/` directory.

[4] https://cmake.org
[5] https://rpm.org
[6] https://shiv.readthedocs.io/en/latest/

Chapter 6

Conclusion

The goal of the thesis was to design and implement an application with a command-line user interface, which would make it possible to configure the DDoS Protector mitigation device by creating and modifying a set of mitigation rules, which are contained within a PostgreSQL database. The mitigation rules contain specific configuration for the various DDoS Protector modules, which deal with the mitigation of different types of DDoS attacks. The database contains not only the mitigation rules, but also the statistical data created by the DDoS Protector, which are collected for every mitigation rule used.

Instead of creating a single monolithic application, the work aimed at providing a modular framework that could be easily expanded with additional functionality. This effort resulted in the creation of a separate database API written in Python, which encapsulates complex database queries into a simple programming interface. The API made it possible to create other applications for manipulation with the database containing the DDoS Protector configuration, other than the proposed CLI configuration tool. The API was tested using unit tests, which became a part of the automated continuous integration in the development of the DCPro DDoS Protector project.

The configuration tool was designed with the aim of providing a comprehensive user interface for the management of mitigation rules and a limited way to display certain statistical data. There was also a focus on making the inner structure of the code maintainable, in case support for additional mitigation rule types needs to be added. The configuration tool was then implemented using the Python programming language and the database API mentioned above. The user interface was continuously tested by potential users, who provided feedback on its improvements during the iterations of development. However, automated unit tests for the configuration tool are yet to be introduced. Finally, the tool was integrated within the DCPro DDoS Protector project, with solutions for packaging and deployment.

The configuration tool is currently being deployed as part of the DCPro DDoS Protector package. Nevertheless, the work does not cease, as there is still some room for small improvements, such as the addition of new ways to display the statistical data. There is also ongoing work on providing support for configuring multiple running instances of the DDoS Protector from the same database. Thanks to the option for JSON outputs and file input for the transaction mode, the configuration tool is also set to be smoothly integrated with other DCPro DDoS Protector utility tools, which are currently being developed.

Bibliography

[1] *Transmission Control Protocol* [RFC 793]. RFC Editor, september 1981. DOI: 10.17487/RFC0793. Available at: https://www.rfc-editor.org/info/rfc793.

[2] *Amplifikační modul* [online]. Liberouter [cit. 2022-05-07]. Available at: https://redmine.liberouter.org/projects/ddos-protector/wiki/Amplifikační_modul.

[3] Chonka, A., Zhou, W. and Xiang, Y. Defending Grid Web Services from XDoS attacks by SOTA. In:. April 2009, p. 1 – 6. DOI: 10.1109/PERCOM.2009.4912895.

[4] *DDoS Protector* [online]. Liberouter [cit. 2022-05-05]. Available at: https://www.liberouter.org/technologies/ddos-protector/.

[5] Eddy, W. *TCP SYN Flooding Attacks and Common Mitigations* [RFC 4987]. RFC Editor, august 2007. DOI: 10.17487/RFC4987. Available at: https://rfc-editor.org/rfc/rfc4987.txt.

[6] Forouzan, B. A. *TCP/IP protocol suite*. 4th ed.th ed. Boston: McGraw-Hill Higher Education, 2010. ISBN 978-0-07-337604-2.

[7] Geng, X. and Whinston, A. Defeating distributed denial of service attacks. *IT Professional*. 2000, vol. 2, no. 4, p. 36–42. DOI: 10.1109/6294.869381.

[8] 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/.

[9] 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.

[10] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. 1stth ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN 1558601902.

[11] 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.

[12] Hui, K.-L., Kim, S. H. and Wang, Q.-H. Cybercrime Deterrence and International Legislation: Evidence from Distributed Denial of Service Attacks. *MIS Quarterly: Management Information Systems*. june 2017, vol. 41, p. 497–523. DOI: 10.25300/MISQ/2017/41.2.08.

[13] International Telecommunication Union. *Data communication networks: Open systems interconnection (OSI); security, structure and applications-security architecture for open systems interconnection for CCIT applications: Recommendation X.800*. Geneva, Switzerland: Telecommunication Standardization Sector of ITU, 2008.

[14] Jacko, D. *Odvozování pravidel pro mitigaci DDoS útoků*. Brno, CZ, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/23920/.

[15] Jun, J.-H., Oh, H. and Kim, S.-H. DDoS flooding attack detection through a step-by-step investigation. In: *2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications*. 2011, p. 1–5. DOI: 10.1109/NESEA.2011.6144944.

[16] Jun, L. and Ling, L. Comparative research on Python speed optimization strategies. In: *2010 International Conference on Intelligent Computing and Integrated Systems*. 2010, p. 57–59. DOI: 10.1109/ICISS.2010.5655011.

[17] 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.

[18] Kavisankar, L. and Chellappan, C. A mitigation model for TCP SYN flooding with IP spoofing. In: *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*. 2011, p. 251–256. DOI: 10.1109/ICRTIT.2011.5972435.

[19] Kuka, M., Vojanec, K., Kučera, J. and Benáček, P. Accelerated DDoS Attacks Mitigation using Programmable Data Plane. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019*. Institute of Electrical and Electronics Engineers, 2019, p. 1–3. DOI: 10.1109/ANCS.2019.8901882. ISBN 978-1-7281-4387-3. Available at: https://www.fit.vut.cz/research/publication/12068.

[20] Kumar, A. and Panda, S. A Survey: How Python Pitches in IT-World. In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. 2019, p. 248–251. DOI: 10.1109/COMITCon.2019.8862251.

[21] Kumar, S. Smurf-based Distributed Denial of Service (DDoS) Attack Amplification in Internet. In: *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. 2007, p. 25–25. DOI: 10.1109/ICIMP.2007.42.

[22] Kučera, J., Viktorín, J., Hutták, L., Krobot, P. et al. *DCPro Protector Manual*. 2022 [cit. 2022-05-07].

[23] Liu, S. Surviving Distributed Denial-of-Service Attacks. *IT Professional*. 2009, vol. 11, no. 5, p. 51–53. DOI: 10.1109/MITP.2009.109.

[24] Raghavan, S. and Dawson, E. *An Investigation into the Detection and Mitigation of Denial of Service (DoS) Attacks: Critical Information Infrastructure Protection*. January 2011. ISBN 978-81-322-0276-9.

[25] Rossum, G. van, Warsaw, B. and Coghlan, N. *Style Guide for Python Code*. PEP 8. 2001. Available at: https://www.python.org/dev/peps/pep-0008/.

[26] Sivakumar, V., Balachander, T., Logu and Jannali, R. Object Relational Mapping Framework Performance Impact. *Turkish journal of computer and mathematics education*. Trabzon: Karadeniz Technical University Distance Education Research and Application Center. 2021, vol. 12, no. 7, p. 2516–2519. ISSN 1309-4653.

[27] Stanojević, V., Vlajić, S., Milić, M. and Ognjanović, M. Guidelines for framework development process. In: *2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR)*. 2011, p. 1–9. DOI: 10.1109/CEE-SECR.2011.6188465.

[28] 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.

[29] *SYN flood modul* [online]. Liberouter [cit. 2022-05-07]. Available at: https://redmine.liberouter.org/projects/ddos-protector/wiki/SYN_flood_modul.

[30] Vial, G. Lessons in Persisting Object Data Using Object-Relational Mapping. *IEEE software*. Los Alamitos: IEEE. 2019, vol. 36, no. 6, p. 43–52. ISSN 0740-7459.

[31] Zhu, W. and Lee, C. Internet security protection for IRC-based botnet. In: *2015 IEEE 5th International Conference on Electronics Information and Emergency Communication*. 2015, p. 63–66. DOI: 10.1109/ICEIEC.2015.7284488.

[32] Zlomislić, V., Fertalj, K. and Sruk, V. Denial of service attacks: An overview. In: *2014 9th Iberian Conference on Information Systems and Technologies (CISTI)*. 2014, p. 1–6. DOI: 10.1109/CISTI.2014.6876979.

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
`dcpro_cfg_api/`	Database API library
`dcproctl/`	Configuration tool source code
`bin/`	Executable files
`dcproctl`	Configuration tool shiv executable
`dcproctl-completion`	Bash autocompletion script
`man/`	Manual page source code
`doc/`	Thesis source code
`thesis.pdf`	Thesis text
`README.md`	README file

Appendix B

Configuration Tool Interface

B.1 Commands with Examples

Rule Creation

```
# dcproctl rule add <type> <rule_attributes>

# dcproctl rule add filter --ip-dst 192.168.0.0/16 --disabled
```

Rule Modification

```
# dcproctl rule edit <id> set <rule_attributes>
# dcproctl rule edit <id> add <multi_value_attributes>
# dcproctl rule edit <id> delete <multi_value_attributes>

# dcproctl rule edit 1 set --ip-dst 2001:db8::/32 --enabled
# dcproctl rule edit 2 add --ip-dst 127.0.0.0/24
# dcproctl rule edit 3 delete --protocol UDP IGMP
```

Displaying Rules

```
# dcproctl rule show by-id <id1> [<id2>, ...] [--overview | --json]
# dcproctl rule show all [--detail | --json]
# dcproctl rule show by-attributes <rule_attributes> [--detail | --json]

# dcproctl rule show by-id 1 2 3 --json
# dcproctl rule show all --detail
# dcproctl rule show by-attributes --type filter --ip-dst 192.168.0.0/16
```

Rule Deletion

```
# dcproctl rule delete by-id <id1> [<id2>, ...]
# dcproctl rule delete all

# dcproctl rule delete by-id 1 2 3
# dcproctl rule delete all
```

Rule Priority Modification

```
# dcproctl rule priority set <id> first
# dcproctl rule priority set <id> last
# dcproctl rule priority set <id1> before <id2>
# dcproctl rule priority set <id1> after <id2>

# dcproctl rule priority set 3 first
# dcproctl rule priority set 2 before 1
```

Rule Transaction

```
# dcproctl rule transaction

# dcproctl rule transaction < prepared_file
```

Displaying Statistics

```
# dcproctl stats show <id> [--json]

# dcproctl stats show 1
```

B.2 Rule Type Specific Arguments

Filter

Single value arguments/flags:

```
--enabled | --disabled
--description <description>
--dry-run {on,off}
--threshold-bps <threshold_bps>
--threshold-pps <threshold_pps>
--vlan <vlan>
```

Multi value arguments:

```
--port-dst <port_dst> [<port_dst> ...]
--port-src <port_src> [<port_src> ...]
--ip-src <ip_src> [<ip_src> ...]
--ip-dst <ip_dst> [<ip_dst> ...]
--protocol {TCP,UDP,SCTP,ICMP,IGMP} [{TCP,UDP,SCTP,ICMP,IGMP} ...]
```

Amplification

Single value arguments/flags:

```
--enabled | --disabled
--description <description>
--dry-run {on,off}
--threshold-bps <threshold_bps>
--threshold-pps <threshold_pps>
--vlan <vlan>
--fragmentation {ANY,YES,NO,FIRST,LAST,MIDDLE,NOFIRST}
--limit-bps <limit_bps>
--limit-pps <limit_pps>
--table-exponent <table_exponent>
```

Multi value arguments:

```
--port-dst <port_dst> [<port_dst> ...]
--port-src <port_src> [<port_src> ...]
--ip-src <ip_src> [<ip_src> ...]
--ip-dst <ip_dst> [<ip_dst> ...]
--protocol {TCP,UDP,SCTP,ICMP} [{TCP,UDP,SCTP,ICMP} ...]
--packet-lengths <packet_lengths> [<packet_lengths> ...]
--tcp-flags <tcp_flags> [<tcp_flags> ...]
```

Syn Drop

Single value arguments/flags:

```
--enabled | --disabled
--description <description>
--dry-run {on,off}
--threshold-bps <threshold_bps>
--threshold-pps <threshold_pps>
--vlan <vlan>
--threshold-syn-soft <threshold_syn_soft>
--threshold-syn-hard <threshold_syn_hard>
--table-exponent <table_exponent>
```

Multi value arguments:

```
--port-dst <port_dst> [<port_dst> ...]
--port-src <port_src> [<port_src> ...]
--ip-src <ip_src> [<ip_src> ...]
--ip-dst <ip_dst> [<ip_dst> ...]
```

TCP Authenticator

Single value arguments/flags:

```
--enabled | --disabled
--description <description>
--dry-run {on,off}
--threshold-bps <threshold_bps>
--threshold-pps <threshold_pps>
--vlan <vlan>
--validity-timeout <validity_timeout>
--threshold-syn-hard <threshold_syn_hard>
--table-exponent <table_exponent>
--algorithm-type {RST_COOKIES,SYN_AUTH}
```

Multi value arguments:

```
--port-dst <port_dst> [<port_dst> ...]
--port-src <port_src> [<port_src> ...]
--ip-src <ip_src> [<ip_src> ...]
--ip-dst <ip_dst> [<ip_dst> ...]
```