



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

USER INTERFACE FOR DDOS MITIGATION CONFIGURATION

UŽIVATELSKÉ ROZHRANÍ PRO KONFIGURACI POTLAČENÍ DDOS ÚTOKŮ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. JAKUB MAN

SUPERVISOR

VEDOUČÍ PRÁCE

Ing. JAN KUČERA

BRNO 2022

Master's Thesis Specification



Student: **Man Jakub, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **User Interface for DDoS Mitigation Configuration**
Category: Networking
Assignment:

1. Get acquainted with a device developed by CESNET that protects against DDoS attacks and with ExaFS application for network routing configuration.
2. Study the device's configuration options and analyze available frameworks to implement a so-called RESTful API.
3. Design an interface to configure this device and extend the ExaFS user application to control DDoS mitigation in the network.
4. Implement the proposed interface and the application extension.
5. Discuss the properties of the created solution, achieved results, and possibilities of further work.

Recommended literature:

- According to the instructions.

Requirements for the semestral defence:

- Items 1 to 3.

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

Supervisor: **Kučera Jan, Ing.**
Consultant: Huták Lukáš, CESNET
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: January 19, 2022

Abstract

Denial of Service (DoS) attacks is a common type of attack on internet networks and devices. A DDoS Protector device was developed to mitigate these attacks, but the configuration of this device can be complicated and requires users to be knowledgeable about the subject to configure the device to mitigate attacks effectively. This thesis focuses on providing a user interface that simplifies the configuration process of the Protector device to help less advanced users mitigate DoS attacks effectively. A web-based graphical user interface and a web application interface will be developed. The user interface will help users to configure the Protector device more straightforwardly. The application interface will allow other developers to integrate the Protector device configuration into other applications.

Abstrakt

Denial of service (DoS) (česky odepření služby) je častý typ útoku na internetové zařízení a síť. Za účelem mitigace těchto útoků bylo vyvinuto síťové zařízení „DDoS Protector“. Konfigurace tohoto zařízení však může být komplikovaná a pro efektivní funkci vyžaduje konfiguraci uživateli, kteří se dobře orientují v problematice DoS útoků. Tato práce se zaměřuje na zjednodušení konfigurace tohoto zařízení pomocí uživatelského rozhraní. Bude vyvinuto webové uživatelské rozhraní a aplikační rozhraní. Uživatelské rozhraní umožní i méně znalým uživatelům efektivně konfigurovat DDoS Protector. Aplikační rozhraní umožní dalším vývojářům integrovat konfiguraci tohoto zařízení do dalších aplikací.

Keywords

DDoS, DDoS protector, user interface, REST API, DDoS mitigation

Klíčová slova

DDoS, DDoS Protector, uživatelské rozhraní, REST API, mitigace DDoS

Reference

MAN, Jakub. *User Interface for DDoS Mitigation Configuration*. Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Kučera

Rozšířený abstrakt

Distribuované útoky typu odepření služby (DDoS) se rozšířily kolem roku 2000 a jsou stále hrozbou do dnešního dne. Cílem těchto útoků je odepření přístupu k cíli běžným uživatelům přetížením cílového zařízení. Útočník posílá na cílové zařízení velké množství požadavků za krátkou dobu, čímž dojde k přetížení cílového zařízení a jeho nedostupnosti pro legitimní uživatele.

Malé organizace většinou nemají rozpočet a expertízu pro zastavení DDoS útoků. Proto sdružení CESNET vyvinulo zařízení „DDoS Protector“. DDoS Protector je hardwarové zařízení schopné detekovat a mitigovat DDoS útoky v běžném síťovém provozu podle dané mitigační strategie a nastavených pravidel. Nastavení těchto pravidel pro efektivní mitigaci však může být složité a vyžaduje znalost principu různých typů DDoS útoků. Čím blíže je toto zařízení ke zdroji útoku, tím je efektivnější na jeho zastavení. DDoS Protector je proto připojen na výměnných bodech propojující organizace a poskytovatele internetu.

Cílem této práce je usnadnit konfiguraci zařízení DDoS Protector tak, aby i méně zkušení uživatelé mohli využívat toto zařízení pro mitigaci útoků přicházejících do jejich sítě. Za tímto účelem bude vytvořeno REST API pro konfiguraci pravidel a čtení statistik a grafické uživatelské rozhraní (GUI) napojené na toto REST API. REST API umožní dalším uživatelům automatizovat konfiguraci zařízení DDoS Protector a usnadní zapojení tohoto zařízení do jejich infrastruktury, GUI usnadní konfiguraci tohoto zařízení bez nutnosti programovat nová rozhraní.

Vyvinuté uživatelské rozhraní je rozšířením existujícího GUI používaného na páteřních uzlech pro konfiguraci přesměrování – ExaFS. Toto uživatelské rozhraní již podporuje přesměrování na DDoS Protector, ale nepodporuje jeho konfiguraci. Proto byla přidána možnost konfigurovat pravidla pomocí předpřipravených šablon pravidel, pojmenovaných podle typu útoku. Uživatelé mohou snadno pravidlo aplikovat výběrem jedné z šablon. Administrátoři mohou šablony libovolně přidávat a upravovat, podle detekovaných typů útoku.

Dále bylo vyvinuto REST API pomocí frameworku FastAPI. Toto REST API umožňuje čtení statistik ze systému AppFS, který DDoS Protector poskytuje. Dále umožňuje správu pravidel. Uživatelé se k REST API přihlašují pomocí API klíče, do budoucna je v plánu rozšíření o další možnosti autentizace.

User Interface for DDoS Mitigation Configuration

Declaration

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

.....
Jakub Man
May 16, 2022

Acknowledgements

I want to thank my supervisor Ing. Jan Kučera for his leadership and feedback on my work. I also want to show my gratitude to Ing. Lukáš Huták and Ing. Jan Viktorin from the CESNET organization for providing additional information about the Protector device and feedback on my implementation and Petr Adamec and his team for the consultation about the user interface and their feedback.

Contents

1	Introduction	2
2	Technologies	4
2.1	DDoS Protector	4
2.1.1	Mitigation rules	5
2.1.2	Mitigation methods	5
2.1.3	Rule configuration	11
2.1.4	AppFS statistics	12
2.2	Web API	13
2.3	REST API	13
2.4	FastAPI	17
2.5	Pydantic	17
2.6	OpenAPI	18
2.7	ExaFS	21
2.8	ZipApps	23
3	Design	25
3.1	Use cases	25
3.2	REST API design	27
3.3	GUI design	28
4	Implementation	32
4.1	REST API	32
4.2	User Interface	38
5	Testing	43
5.1	Test environment	43
5.2	Testing the REST API	43
5.3	Testing the UI	45
6	Conclusion	46
	Bibliography	48
A	Contents of the Attached CD	51

Chapter 1

Introduction

Distributed Denial of Service (**DDoS**) attacks became common in the early to mid-2000s and are still prominent today. The goal of these attacks is total prevention of the target's normal functioning – a complete „denial of service“. The attacker may request payment to stop the attacks. Network resources such as web servers have a finite limit to the number of requests they can handle simultaneously. **DDoS** attacks rely on this fact. Many requests are sent to the target, overwhelming the target's resources and causing the target to respond slowly or not at all. Attackers usually use a sizeable amount of computers infected with malware to send these requests to the target. These types of attacks are therefore difficult to stop [4].

The smaller organizations usually lack the budget and expertise to stop a **DDoS** attack. That is why the CESNET organization developed the **DDoS** Protector device. **DDoS** Protector is a hardware device capable of detecting and mitigating **DDoS** attack traffic. The Protector inspects the traffic, drops packets according to a given mitigation strategy and forwards the legitimate traffic to the target organization. The closer to the attack the device is, the more efficient it is at dealing with the attack. Therefore the primary goal of the **DDoS** Protector is to protect the infrastructure connecting the organizations to the internet backbone [5]. The **DDoS** Protector device is described in detail in chapter 2.

The goal of this thesis is to simplify the process of configuration of the **DDoS** Protector device. A Representational State Transfer (**REST**) Application Programming Interface (**API**) will be developed as well as a web user interface (**UI**). The web **UI** will help network administrators to respond quickly to incoming **DDoS** attacks by rerouting traffic to the **DDoS** Protector device. The **REST API** will allow other developers to integrate the Protector device configuration into other applications, further simplifying the configuration process.

The implementation of the graphical user interface (**GUI**) described in this thesis will extend the existing ExaFS **GUI**, which the administrators who will be working with the **DDoS** Protector device already use. ExaFS currently supports redirecting traffic to the **DDoS** Protector device but not configuring the device. Configuring the **DDoS** Protector device will be based on rule templates defined by administrators.

Chapter 2 describes technologies and libraries used to implement the **REST API** and the **GUI**. This chapter also describes why some of the libraries and software were selected. The **DDoS** Protector device is also described in detail in this chapter. Chapter 3 describes how the project was designed and shows the **REST API** design and the **GUI** mockups. Chapter 4 describes the implementation based on the design, the way the **REST API** application is structured and the challenges encountered during implementation. Chapter 5 focuses

on testing the implementation and describes testing methods and environment. In the final chapter, the whole thesis is concluded. The discrepancies between the design and the implementation are discussed. The final chapter also mentions plans for future work for the project described in this thesis.

Chapter 2

Technologies

This chapter describes technologies, libraries, and frameworks used to design and develop the **API** and the **UI**. It explains the reasons why certain technologies have been selected and how they are used in the project.

2.1 DDoS Protector

DDoS Protector is a network device developed by the CESNET organization. Protector is usually deployed in a pair with a router – the router forwards suspicious traffic to the Protector device, as shown in figure 2.1. Protector behaves based on rules set by an administrator during the configuration phase. The rules include conditions, limits, optimal traffic rate, and the protected network’s IP prefix.

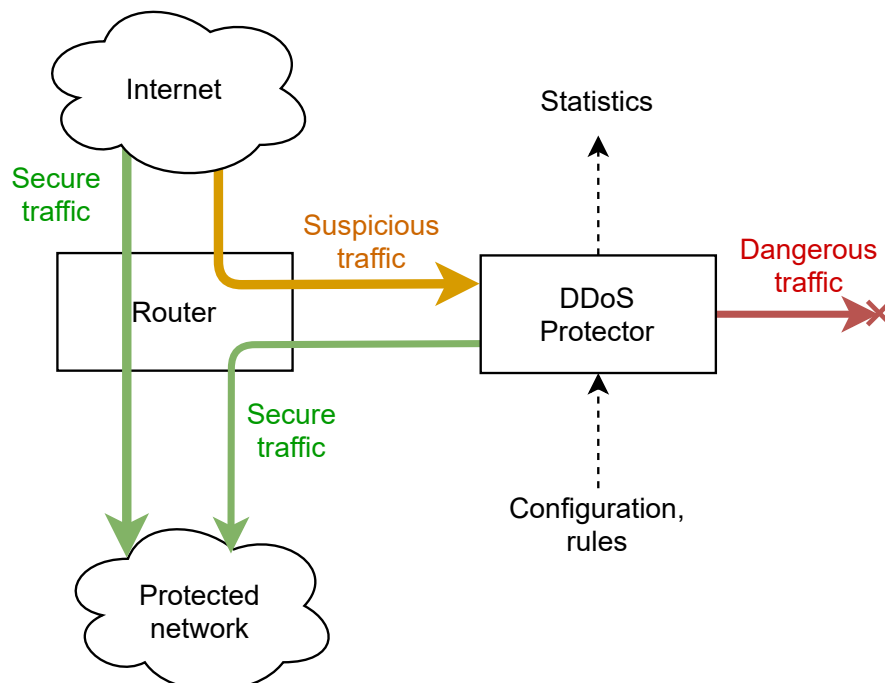


Figure 2.1: **DDoS** Protector connection [5]

Processing of the traffic is divided into stages in a pipeline. Each stage is responsible for a simple task, such as reading the traffic, parsing it or blocking certain packets. This approach improves throughput since the stages can work in parallel. It also simplifies problem-solving, as each stage provides its independent statistics.

The Protector can operate in two modes – blocker or mitigator. It serves as a simple firewall with either a deny list or an allow list in blocker mode. This thesis focuses on the mitigator mode – it serves as a semi-automatic system to prevent a **DDoS** attack on the wire.

2.1.1 Mitigation rules

Mitigation rules can be activated just as traffic reaches a set threshold in the form of bits or packets per second. The rule is left idle until the traffic reaches the set threshold using only minimal system resources.

The definition of each mitigation rule contains the following parameters [25], which are common for all mitigation methods:

- **ID**: A numeric identification number. It has to be unique amongst all mitigation rules.
- **enabled**: Specifies whether the rule is enabled or not. The **DDoS** Protector ignores disabled rules and reports no traffic statistics.
- **dry_run**: If enabled, the rule observes traffic and reports statistics but does not drop or generate packets. The purpose of this option is to test new configurations before deploying them.
- **threshold_bps**: Rule is activated when traffic exceeds this threshold. It is calculated on L2 without an Ethernet FCS field.
- **threshold_pps**: Rule is activated when the number of packets per second exceeds this threshold.
- **vlan**: ID of a VLAN. If set to zero, the rule matches only packets without a VLAN ID.
- **ip_src**: List of source IP addresses and networks. It can be empty to apply the rule to all source IP addresses.
- **ip_dst**: Same as **ip_src**, applies to destination IP addresses.

All parameters except **ID** are optional. If a parameter is not specified, a default value is used.

2.1.2 Mitigation methods

There are two categories of mitigation methods: passive and interactive. A passive method drops or passes packets based only on traffic observation, while an interactive method may generate traffic and act on behalf of a protected device or network in addition to traffic filtering.

```

filter:
  - id: 1
    enabled: True
    dry_run: False
    vlan: 200
    ip_src: 10.66.0.15
    ip_dst: []
    port_src: []
    port_dst: 80
    protocol: TCP
  - id: 2
    enabled: True
    dry_run: False
    vlan: 0
    ip_src: 192.168.0.0/24
    ip_dst: 192.168.0.1
    port_src: []
    port_dst: 53
    protocol: UDP

```

Listing 2.1: Filter rule example [25]

Filter

The *filter* method is a passive mitigation method that drops traffic based on flow metadata. A filter rule contains the source (`port_src`) and destination (`port_dst`) port and an L4 protocol (`protocol`), and the DDoS Protector drops any packet that matches the rule. All rule parameters are optional, and if they are empty, the rule matches any value for a given parameter. For example, a filter rule with source port set to 53 and other parameters empty will drop all traffic originating from a DNS server without checking the destination port or the protocol. Listing 2.1 shows an example of a filter rule. The first rule in the example filters traffic on VLAN 200 coming from IP 10.66.0.15 to any destination IP and destination port 80. The second rule matches packets without a VLAN number from IP 192.168.0.0/24 to host 192.168.0.1 and with destination port 53 – the DNS protocol [25].

SYN drop

SYN drop is a passive mitigation method against SYN flood attacks.

SYN flood attack is a Denial of Service (DoS) attack that misuses the TCP protocol to send many packets requesting to open a connection [1]. In normal circumstances, a client would send an SYN packet to the server, requesting to open a connection, and the server would respond with an SYN-ACK message to the client. The client then finishes establishing the connection with an ACK message, as shown in figure 2.2. This process is called a three-way handshake [11]. The SYN flood attack sends many SYN packets to the server but never responds to the SYN-ACK packet from the server, creating many half-open connections. The server has built-in memory for a data structure describing open connections, and this memory is finite. The memory can overflow by creating too many partially-open connections. Creating half-open connections is accomplished using

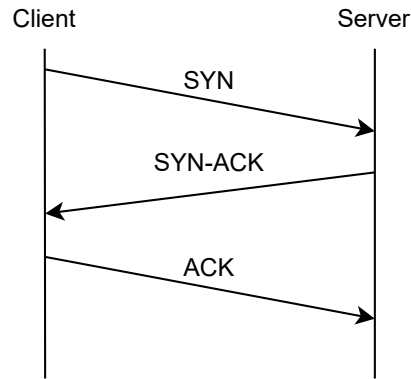


Figure 2.2: TCP three-way handshake [11]

IP spoofing – sending packets with many different source IP addresses. These packets appear legitimate, but the client never responds to the SYN-ACK message with an ACK message since the source IP address is spoofed. The target server’s memory will eventually fill, and the server will become unable to respond to any incoming legitimate connections. Usually, there is a timeout associated with incoming connections, but the attacking system will continue sending IP-spoofed packets faster than the victim server can timeout the old connections, so the server will never recover on its own [1].

When a rule of the SYN drop method is configured, the **DDoS Protector** inspects only the initial SYN packets of TCP three-way handshakes and ignores other types of packets. This rule should be activated when the amount of SYN packets directed to devices in a protected network exceeds the limit they can handle. Each SYN drop rule monitors SYN packets incoming to a protected network and maintains statistics summarizing the SYN-only packet counts from specific IP addresses [25].

The first SYN packet from a new client is always dropped to prevent attacks that use IP spoofing. Dropping the first SYN packet only introduces a short delay while establishing a connection for legitimate clients, after a specific timeout a new SYN packet is sent to the server, as shown in figure 2.3. However, since the attacker sends as many packets as possible with spoofed IP addresses, dropping the first packet slows down the attack significantly or stops it altogether if the attacker takes no countermeasures [25].

Each rule maintains two thresholds for the number of received SYN packets: soft and hard. The soft threshold represents the number of allowed SYN packets without the corresponding ACK packet, while the hard threshold defines the number of packets after which all consequent SYN packets are dropped, regardless of received ACK packets. Host counters store the number of SYN and ACK packets from each host. The **DDoS Protector** stores counters of each host in a record table. The capacity of the record table is defined in the rule definition in the table exponent field (`table_exponent`). Older counters may be replaced prematurely by new ones if the capacity is insufficient. Counters are periodically reset to avoid potential long-term blocking of legitimate hosts that are very active. SYN counters are reset every 4 seconds; ACK counters are reset every 20 seconds [25].

In addition to common rule parameters, the SYN drop rule may contain the following parameters [25]:

- `port_src`: A list of L4 source ports and port ranges.

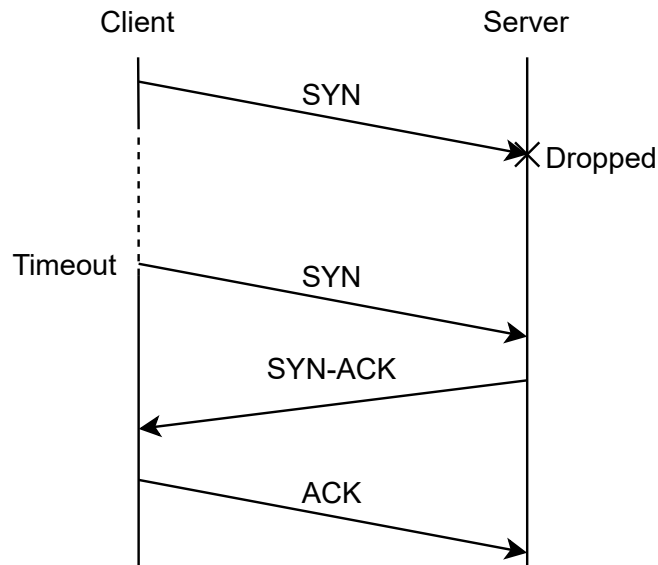


Figure 2.3: Legitimate client behaviour with first SYN packet dropped [25]

- `port_dst`: Specifies a list of L4 destination ports and port ranges.
- `threshold_syn_soft`: Defines the number of SYN-only packets allowed without receiving ACK packets.
- `threshold_syn_hard`: Represents the number of packets after which all incoming SYN-only packet is dropped regardless of received ACK packets.
- `table_exponent`: Determines the size of the record table. The size of the record table corresponds to the maximum number of unique source IP addresses in the table. The size of the record table is calculated as $2^{\text{table_exponent}}$. The default table exponent is 18, and the valid range is between 10 and 30.

All of the parameters, except soft and hard threshold, are optional. An example in listing 2.2 shows an SYN drop rule that will activate when matching network traffic reaches 100 thousand packets per second. The traffic threshold represents the number of SYN-only packets with VLAN number 200, directed to the protected network 10.0.0.0/16 and the destination port in range 0-1000. Each unique source IP address can make at most five simultaneous TCP handshakes until at least one connection is established. Then the IP address is allowed to start at most 55 new TCP handshakes every four seconds [25].

Amplification mitigation

Amplification mitigation is a passive mitigation method protecting against volumetric DDoS attacks. When a rule of this method is activated, it blocks traffic from a set number of most significant traffic contributors to reduce an unexpected traffic increase.

The method operates on a user-defined portion of the traffic specified by a rule. Each rule of this method specifies a traffic *threshold* and a traffic *limit* specified in packets or bits per second. When a total volume of specified traffic reaches the set *threshold*, the mitigation

```

syn_drop
- id: 100
  enabled: True
  dry_run: False
  threshold_pps: 100 k
  vlan: 200
  ip_src: []
  ip_dst: 10.0.0.0/16
  port_src: []
  port_dst: 0-1000
  threshold_syn_soft: 5
  threshold_syn_hard: 55

```

Listing 2.2: SYN drop rule example [25]

method blocks the most significant contributors until the traffic volume drops to the *limit* value or below.

When the traffic reaches the 70 % threshold, the amplification mitigation method enters a „standby mode“. In the standby mode, this method collects per-host statistics of received packets and bytes. The most significant traffic contributors are then identified based on these statistics. Host statistics are stored in a record table. The size of the record table is defined in the rule as a parameter.

In addition to common rule parameters, the amplification method rule may contain these parameters, all of which are optional [25]:

- **fragmentation**: Specifies behaviour when handling fragmented packets.
- **port_src**: List of source L4 ports and port ranges. If empty, the rule applies to all source ports.
- **port_dst**: List of destination L4 ports and port ranges. If empty, the rule applies to all destination ports.
- **protocol**: List of L4 protocols. Supported protocols are TCP, UDP, SCTP and ICMP. If empty, the rule considers all supported L4 protocols.
- **packet_lengths**: List of packet lengths and length ranges. Only packets of matching length are considered. The length of a packet is measured from an L2 packet without the FCS field.
- **tcp_flags**: List of TCP flag combinations. An exclamation mark can be added in front of the flag – the rule then only considers packets if the flag is not set. For example, to accept only SYN and SYN+ACK packets, set this value to `[!C!E!U!P!RS!F]`.
- **limit_bps**: Defines how much traffic is allowed to the protected network during an attack in bits per second. Traffic from the most significant contributors is blocked until traffic volume is limited to or below this target value. Bits per second are calculated on L2 without the Ethernet FCS field.
- **limit_pps**: Same as the `limit_bps`, but instead of bits per second, consider the number of packets per second.

```

amplification
- id: 200
  enabled: True
  dry_run: False
  threshold_bps: 20 G
  limit_bps: 10 G
  vlan: 200
  ip_src: []
  ip_dst: 10.0.0.0/16
  port_src: []
  port_dst: 0-1024
  protocol: [TCP, UDP]
  fragmentation: NO

```

Listing 2.3: Amplification mitigation rule example [25]

- **threshold_bps**: Specifies the amount of traffic considered an attack in bits per second. When this threshold is reached, the rule starts blocking traffic until the traffic is reduced to the set limit.
- **threshold_pps**: Same as the **threshold_bps**, but instead of bits per second, consider the number of packets per second.
- **table_exponent**: Determines the size of the record table, which stores the traffic statistics. The size of the record table corresponds to the maximum number of unique source IP addresses in the table. The size of the record table is calculated as $2^{table_exponent}$. The default table exponent is 18, and the valid range is between 10 and 30. Older records may be replaced with newer ones if the capacity is insufficient, leading to the blocking of incorrect hosts.

An example of an amplification mitigation rule in listing 2.3 shows a rule configuration that blocks all TCP and UDP traffic with the destination IP 10.0.0.0/16 and port in the range 0-1024 from the most significant contributors. When the traffic reaches 20 Gbps, the rule starts blocking the traffic, continuing until the traffic reaches 10 Gbps. The rule only considers traffic coming through VLAN ID 200.

TCP authenticator

TCP authenticator is an active mitigation method protecting against SYN flood attacks. This method should only be used if the SYN drop method does not protect against the attack since it protects against the same type of attack but is significantly more computationally expensive. However, since this method is more complex than the SYN drop method, it is also significantly more difficult for an attacker to overcome.

As an active method, the *TCP authenticator* method acts on behalf of the protected network. The **DDoS** Protector responds to the SYN packets from clients. Then an invalid SYN-ACK response is created and sent to the client, containing an authentication token and an unseen acknowledgement number. According to the RFC 793 standard, the client should respond with an RST packet that contains the authentication token [11]. Since

```
tcp_authenticator:
- id: 200
  enabled: True
  threshold_pps: 100 k
  ip_src: []
  ip_dst: 10.0.0.0/16
  port_src: []
  port_dst: [80, 443]
  validity_timeout: 60 s
```

Listing 2.4: TCP authenticator mitigation rule example [25]

an attacker is not trying to establish a full connection, they are unlikely to implement this behaviour; therefore, only legitimate clients are expected to respond with an RST response.

From the client's point of view, the first attempt to establish a connection always fails. However, that is not a problem in modern operating systems – most of them try to reestablish the session after sending the RST packet, introducing only a short delay.

Successfully authenticated clients are stored in a table with limited capacity. SYN packets from authenticated clients are allowed to pass without limitation. When the table is full, older records are replaced with newer ones. Removed clients are then required to authenticate again before connecting to the protected device. Each authentication record has a validity interval, after which the record is not valid anymore and removed from the table. The successfully authenticated clients might sometimes create too many connections. Therefore, limiting the maximum number of SYN requests sent from a single host is possible.

A TCP authenticator rule consists of these parameters in addition to common rule parameters:

- **port_src**: A list of source L4 ports and port ranges.
- **port_dst**: A list of destination L4 ports and port ranges.
- **validity_timeout**: Defines a maximum validity interval of host authentication. After this timeout, the client must authenticate again before connecting. The timeout must be at least 1000 ms.
- **threshold_syn_hard**: Specifies a maximum number of SYN requests each host can send after successfully authenticating the protected network.
- **table_explonent**: Determines the size of the authenticated host table.

All parameters except **validity_timeout** are optional. Listing 2.4 shows an example of a TCP authentication rule to authenticate clients sending packets with the destination address 10.0.0.0/16 and port 80 or 443. Clients are authenticated for 60 seconds, after which they have to authenticate again. The rule is activated when traffic to the destination reaches 100 thousand packets per second.

2.1.3 Rule configuration

The Protector rules can be saved on the Protector device's file system or in a SQL database.

If the rules are saved on the file system, they are defined in YAML format, and the file is referenced in the Protector’s configuration file. The only way to edit the rule YAML files is to connect to the Protector device and edit them manually. The rule configuration file can contain definitions of rules for various mitigation methods. Each mitigation method is introduced by its name and followed by one or more rule definitions. The execution order of these rules is primarily defined by the method type they belong to, secondarily by their order in the configuration file. The order of the methods in the file does not matter, only the rules within a method.

If the rules are saved in a SQL database, they can be edited using a command-line interface or a Python library. The Protector uses the PostgreSQL database to store the rules and the `psycopg2` library to connect to the database. The Protector Python library encapsulates communication with the database into *transactions*. In addition to accessing and editing the rules in the database, *transactions* validate the rules for correct syntax and fill the default values for optional rule parameters if the user did not fill them. The **REST API** developed as a part of this thesis will use the Python library to configure the Protector rules.

2.1.4 AppFS statistics

The Protector service exposes its internal state and statistics via the AppFS system. The AppFS acts as a mounted file system, the contents of the files in the file system are generated each time the file is accessed. Therefore, every time a file from the AppFS system is read, it contains up to date information. The file system is mounted in the `/var/run/dcpro_protector` directory by default, but the configuration file can change the mount directory.

The file content format varies depending on the file. Some files only contain a single value or several values divided by a new line character. Other files contain a key, and a value on a single line, divided by a colon. Usually, multiple key-value pairs are in a single file, divided by a new line character. Some files use a key-value pair format with multiple values per key divided by a space.

AppFS organizes the files into directories:

- The `ports` directory consists of subdirectories for each physical port on the Protector device. Each of these port subdirectories contains information about the physical port, such as its mac address, status (up/down), or traffic information.
- The `self` directory has information about the running instance of the Protector software – the version and an identifier of the system process.
- The `runners` directory consists of runners, which are processes for each port. Each runner directory includes a pipeline directory with information about the pipeline stages, overall pipeline statistics, and per-stage statistics. A given runner directory also holds information about the number of workers (threads) belonging to the runner and per-worker statistics.
- The `routing_manager` directory contains information about the internet routes since the Protector routes the traffic in addition to cleaning the traffic. The format of the routing information is the same as the output of the Linux `ip` utility.

2.2 Web API

Web APIs gained popularity with the adoption of web 2.0. Websites shifted from static content to user-generated, and websites became more interactive than before. Web APIs were necessary for this change since services began integrating other services, and the content of a page needed to update constantly [17].

A web API is an API for either a web server or a web browser. A server-side web API contains publicly exposed endpoints to a defined request-response message system, typically expressed in JSON or XML [13]. HTTP endpoints in web APIs use the HTTP methods POST, GET, PUT and DELETE, mapping to the four basic operations: create, read, update, delete (CRUD) [16]

The HTTP methods mentioned above operate on the collections and elements. POST creates a new element in a collection, the GET method retrieves an element or a collection of elements, PUT updates an element or multiple elements and DELETE removes collections or elements. There are more HTTP methods than mentioned, but they are usually not directly used in web APIs since their purpose differs from data manipulation [8].

A well-designed web API should only have two base URLs per resource: one for a collection, the second for a specific element in the collection. The URLs should not contain verbs to reduce the number of URLs and make the API easy to understand. For example, /dogs and /dogs/123 are good URLs, while /getDogOwner should be avoided. The verbs can be singular or plural but should be consistent in the whole API – either all verbs singular or all plural. The only exception is responses that do not involve a resource from a database, for example, calculating, translating or converting something. In these cases, a verb should represent the Uniform Resource Locator (URL), for example, /convert [16].

2.3 REST API

REST is a data-oriented architectural style for distributed environments. REST can structure data into XML, JSON, YAML, or any other machine-readable format, but usually, JSON is preferred. Any protocol can implement REST architecture, but REST was developed as part of the web standard and, as such, is often used with the HTTP protocol [9].

Guiding principles

Six principles guide RESTful architecture. If a web API follows these principles, it is called a RESTful API [9]:

1. **Client-Server architectural style:** By separating the UI concerns from the data storage concerns, the portability of the UI across multiple platforms and the scalability of the data storage is improved. The separation of components also allows them to evolve independently.
2. **Stateless:** Communication must be stateless, such that each request from the client to the server must contain all the information necessary to understand the request and cannot take advantage of any stored context on the server. Only the client keeps the session state. Statelessness improves visibility, reliability and scalability. Visibility is improved because the monitoring system does not have to look beyond a single request to determine the nature of the request. Reliability is improved due to the ease of recovering from partial failures. Scalability is improved by not storing

state between requests, allowing the server component to quickly free resources. A disadvantage of statelessness is decreased network performance due to redundant data in a series of requests since that data can not be left on the server.

3. **Cache:** The server should label data responding to a request as cacheable or non-cacheable. The label could be either explicit or implicit. If a response is cacheable, the client is given the right to reuse that data for later, equivalent requests. Caching eliminates some interactions, improving efficiency and performance.
4. **Uniform interface:** REST architecture emphasises a uniform interface between components. The overall architecture is simplified, and the visibility of interactions is improved. Implementations are independent of the services they provide, encouraging the independent evolution of components.
5. **Layered system:** The layered system style allows the architecture to be composed of hierarchical layers. Clients only see the layer with which they are interacting. The rest of the system is hidden from them. Layers can be used to encapsulate legacy services and protect new services from legacy clients. Layering can also simplify load balancing and the integration of new functionality to the server.
6. **Code on demand:** REST allows clients to extend functionality by downloading and executing applets or scripts. This principle is optional, so APIs might not implement it.

Architectural elements

The fundamental abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or an image, a temporal service, a collection of other resources or a non-virtual object. The state of a resource is known as the resource representation, which consists of the data, the metadata describing the data and the hypermedia links that can help the clients transition to the next desired state [9].

Frameworks

There are many REST API frameworks available. Since the DDoS Protector rule library is only available in the python programming language, only python frameworks are considered in this thesis.

Most popular frameworks provide features for building complete web applications and web APIs, which provides more flexibility in development, but the flexibility often introduces overhead, making the application slower. These frameworks are called full-stack frameworks. Unlike microframeworks, which provide only the essential functionality, full-stack frameworks often include MVC application patterns, database handling, form handling, security and more. Due to providing many features, full-stack frameworks are often large and complex, making them harder to learn. Speed is one of the considerations for choosing a framework to implement the REST API, so microframeworks are preferred for a project with just a REST API.

Another consideration is a set of features. The framework should provide as many of the features required to build a REST API as possible while not introducing unnecessary intricacies by introducing a complicated architecture or a large set of features not needed by a

REST API. Microframeworks often introduce a plugin system to extend their functionality as needed.

The popularity of a framework is another deciding factor. If a framework is not popular enough, it might be challenging to solve problems during development. More popular frameworks are often more secure since more people find possible vulnerabilities and other issues.

- At first, *Django* was considered since it is one of the most popular full-stack python frameworks. However, since it is a full-stack framework, it is complex and challenging to learn. It also provides many features that would not be useful in a **REST API** since Django was initially created to generate HTML in the backend, not to create APIs [6].
- *Flask* was another option. *Flask* is a popular python microframework that can be used to build both complete web applications and **REST APIs**. It is a modular framework, so developers can only enable the required features and not install unnecessary functionality, making the final application faster and smaller. Chapter 2.7 describes the Flask framework in detail [18].
- *BlackSheep* is an asynchronous full-stack framework for building web applications. It is based on the Model-View-Controller design pattern and, thus, more suitable for complete web applications than **REST APIs**. It also requires Python version 3.7 or newer; Frameworks using Python version 3.6 are preferred since it is the most popular version [20] [19].
- In the performance testing by TechEmpower, *Apidaora* was the fastest Python framework for building **REST APIs**. *Apidaora* is a microframework extensible using middlewares – small programs that modify the incoming or outgoing requests. However, it is not very popular, and according to the framework’s GitHub page¹, it was last updated fourteen months ago, as of January 2022. Therefore, there is a risk of the framework containing undiscovered vulnerabilities and errors [7] [24].
- *Sanic* is a microframework for building web applications. Since it is a web application framework, it is more complex and harder to learn than other microframeworks. Even though *Sanic* is a complex framework, it is still one of the fastest Python frameworks due to its heavy use of asynchronous functions. It has support for plug-ins using extensions. *Sanic* requires Python version 3.7, but *Sanic* extensions require Python version 3.8 [21] [24].
- In the end, *FastAPI* was selected as the framework for building a **REST API** for the **DDoS** Protector. It is one of the fastest python **REST API** frameworks and provides all the required features. It is a microframework that makes use of python features like type hinting, making it easier to learn. It is also regularly updated and has 100% test coverage, making it more reliable. Chapter 2.4 describes the FastAPI framework [15] [24].

Table 2.1 compares described frameworks. Performance was compared using the TechEmpower Web framework benchmarks. The frameworks were filtered to show Python frameworks only. The composite scores are based on multiple tests, including response time to a single query, multiple queries, JSON serialisation, database access times and data update times.

¹<https://github.com/dutradda/apidaora>

Framework name	Framework type	Minimum Python version	TechEmpower composite score (higher is better)	Advantages	Disadvantages
Django	Full-stack	3.6	280	- Excellent support - Great documentation	- Slower - Harder to learn
Flask	Micro	3.6	468	- Good support - Many available plugins	- Slower - No authentication support
Blacksheep	Full-stack	3.7	1486	- Very fast	- MVC design pattern - Less support
Apiadora	Micro	3.8	1529	- Very fast	- Infrequent updates - Unfinished documentation - Bad support
Sanic	Micro	3.7 (3.8 for extensions)	1331	- Fast and lightweight - Extensible by plugins	- Extensions require Python 3.8 or newer
FastAPI	Micro	3.6	1209	- Fast - Excellent documentation - Lightweight	- Relatively new framework

Table 2.1: REST API Python framework comparasion

2.4 FastAPI

FastAPI is a Python framework for building web APIs based on the implementation of the Uvicorn server and Starlette asynchronous server gateway interface (ASGI). FastAPI uses the Pydantic library and python type declarations for data validation, making it more convenient and faster to code. FastAPI also provides libraries for security and authentication, using API keys, OAuth2, JWT tokens and more.

FastAPI provides a dependency injection system. Dependency injection means that there is a way for code to declare required resources or functions – dependencies. FastAPI then reads these declarations and provides them in the functions that requested these dependencies – injects them. The dependency injection system minimises code repetition. For example, shared logic, shared path parameters, database connections, security, authentication or role requirements are handled through dependency injection [15].

FastAPI uses the dependency injection system to integrate plug-ins into the framework. Because the dependency injection system integrates dependencies directly into the functions that request them, plug-ins do not need to be written explicitly for FastAPI; any python library can be included with minimal or no extra work required [14].

Every API written using the FastAPI framework has automatically generated documentation. FastAPI generates the documentation on startup, so it does not add any overhead to running applications on the system. The documentation is available in the OpenAPI format, so it is possible to generate a client code using many available generators. Included user interfaces for the documentation are Swagger UI and ReDoc, but many more documentation generators exist for OpenAPI [15]. OpenAPI is described in chapter 2.6.

2.5 Pydantic

Pydantic is a data parsing library for the Python language. It utilises Python type hinting introduced in Python version 3.6 to enforce the hinted types at runtime. If the data is invalid, pydantic provides a user-friendly error message – these messages specify the location of the error in the data, a message detailing the error and the type of the error. Pydantic is a data parsing library, not a data validation library. That means that Pydantic guarantees the types and constraints of the output model but not the input data. However, Pydantic can be used for data validation, even if it is not the library’s primary purpose [3].

Pydantic uses Python classes to specify data formats. These classes are called „models“ and inherit a Pydantic class `BaseModel`. Untrusted data can be passed to a model, and after parsing and validation, Pydantic guarantees that the fields in the resulting model instance will conform to the field types defined in the model or raise an exception. Models can also inherit other models instead of the `BaseModel` class to extend the inherited models.

Model fields can be of any type, or even other models. Pydantic validates the data on runtime based on the specified types. If a value is filled in a model field when specifying the model, the field is considered optional, and the filled value is considered a default value. Optional values can also be specified using the `Optional` data type from the Python `typing` library. If a field is not optional and is missing in the input data, or the input data are of different types than specified, the model will raise an exception.

Pydantic allows programmers to specify additional validator functions if the built-in validations are insufficient. A decorator provided by Pydantic specifies validators. The decorator takes a list of field names as parameters, and the validator is then applied to the specified fields. If a star is specified as a field name, the validator is applied to all fields in

the model. Validator functions decorated by the decorator have to be members of a model class and take one parameter: a value of a field. Pydantic calls specified validators during the data parsing phase. Validators can access the input values, check or change them, and raise exceptions if the input value is incorrect [3].

2.6 OpenAPI

The OpenAPI specification defines a standard interface to RESTful APIs, which allows both humans and computers to discover and understand the capabilities of a service without access to the source code, documentation, or network traffic inspection. Documentation generation tools can use an OpenAPI definition to display the API, code generation tools to generate servers and clients in various programming languages, testing tools to test APIs and more [22].

An OpenAPI document is a JSON object, represented either in JSON or YAML format, defining or describing an API and conforming to the OpenAPI schema. OpenAPI documents may be made up of a single document or be divided into multiple connected parts. An OpenAPI document must contain the following fields [22]:

- **openapi**: Semantic version number of the OpenAPI specification version that the document uses.
- **info**: An object providing metadata about the API, such as the API's title, description, and version.
- **paths**: The available paths and operations for the API. This object may be empty to signify that the user has reached the correct destination but can not access any documentation. They still would have access to the info object, which might contain additional info regarding authentication.

In addition to the required fields, these fields are allowed in the OpenAPI document [22]:

- **servers**: A list of URLs of target servers and their descriptions.
- **components**: An element to hold various schemas for the specification.
- **security**: A declaration of which security mechanisms can be used across the API.
- **tags**: A list of tags with additional metadata used by the specification.
- **externalDocs**: **URL** and description of additional external documentation.

Listing 2.5 shows an example of an OpenAPI document in the YAML format, using the OpenAPI version 3.0.0 schema. The **API** described in the example document is titled „Sample API“ and contains one route – `/users`, with one available operation – `get`, which returns an array of user names. The **API** is available on two servers, one primary server and one staging server.

```

openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in
[CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9
servers:
- url: http://api.example.com/v1
  description: Optional server description, e.g. Main (production) server
- url: http://staging-api.example.com
  description: Optional server description, e.g. Internal staging server
for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200': # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string

```

Listing 2.5: Example of a minimal OpenAPI document [22]

SwaggerUI

SwaggerUI is an open-source **UI** for visualising and interacting with OpenAPI documents. It serves as documentation for **APIs** and a testing tool for **API** developers. The **UI** can be generated using just an OpenAPI document; it does not require an implementation of the **API**. SwaggerUI is a web-based **UI**, allowing developers to upload the **UI** for users to view it using just their web browser. The **UI** shows all available routes and methods usable on them and describes each path and object. It also lists the possible HTTP error codes each route can return.

As shown in figure 2.4, SwaggerUI allows users to test each route without implementing a client, using the „Try it out“ button. The **UI** can either connect to an implemented API hosted on a server or return example values from the OpenAPI file. If the **API** requires authentication or an **API** key, users can authorise themselves using the „Authorize“ button. SwaggerUI then adds the required HTTP headers when sending requests to authorise the requests to the API.

Swagger Petstore 1.0.6

[Base URL: petstore.swagger.io/v2]
<https://petstore.swagger.io/v2/swagger.json>

This is a sample server Petstore server.

Schemes HTTPS Authorize

pet Everything about your Pets Find out more: <http://swagger.io> ^

POST /pet/{petId}/uploadImage uploads an image ^

POST /pet Add a new pet to the store ^

PUT /pet Update an existing pet ^

Parameters Try it out

Name	Description
body * required object (body)	Pet object that needs to be added to the store Example Value Model <pre style="background-color: #2d3748; color: #a6c9ec; padding: 10px; border: 1px solid #2d3748; margin: 10px 0;"> { "id": 0, "category": { "id": 0, "name": "string" }, "name": "doggie", "photoUrls": ["string"], "tags": [{ "id": 0, "name": "string" }], "status": "available" } </pre>

Parameter content type application/json

Responses Response content type application/json

Code	Description
400	Invalid ID supplied

Figure 2.4: SwaggerUI example project [23]

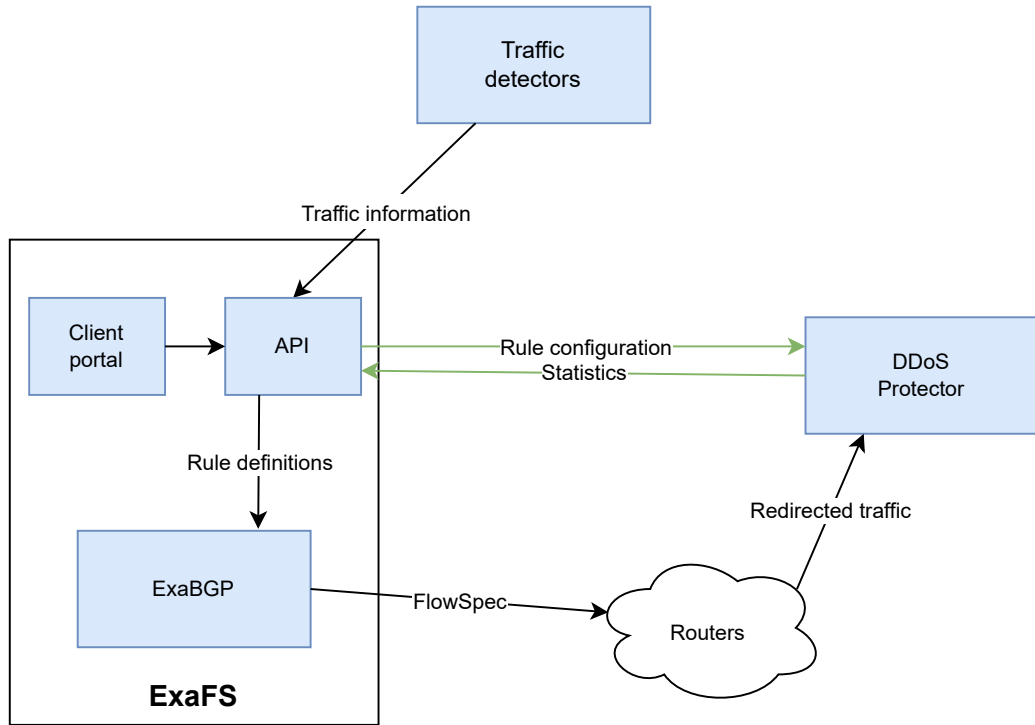


Figure 2.5: ExaFS integration into a network [26]

2.7 ExaFS

ExaFS is an open-source tool developed by CESNET to configure firewall rules using the ExaBGP protocol. It provides a **REST API** for web services and a web **UI**. The key features of ExaFS are user authorisation and a validation system for BGP commands [26].

Over time, the ExaFS system has been extended with additional features in addition to the ExaBGP protocol configuration. A Remotely Triggered Black Hole (**RTBH**) configuration has been added as well as the capability to route the specified traffic to the **DDoS Protector** device.

RTBH filtering is a simple and effective technique for mitigating **DoS** attacks. When **RTBH** is activated for a specific IP address, all packets with destination IP addresses matching the set IP address are dropped. **RTBH** is effective when the attacker’s IP address is known since most routers can forward traffic at a much higher rate than they can filter. However, with destination-based **RTBH** filtering, the affected device becomes offline since all traffic with its IP address as a destination is dropped [12].

ExaFS is a part of cybersecurity tools at CESNET, but any network that supports ExaBGP can use it. Figure 2.5 shows how is ExaFS integrated into a network. Traffic detectors collect traffic information and send it to ExaFS using its **API**. Users can interact with the API using a web application. The **API** validates the BGP rules; only error-free rules can pass into the ExaBGP **API**. Both syntax and access rights are validated before a rule can be stored in the database. The green arrows indicate the parts that will be implemented in this thesis. ExaFS will send rules to **DDoS Protector**, and **DDoS Protector** will provide mitigation statistics to the **UI**

The screenshot shows the 'New IPv4 rule' configuration page in the ExaFS UI. The page has a dark header with navigation links like 'Add IPv4', 'Add IPv6', and 'Add RTBH'. The main content area contains several form fields: 'Source address' (192.168.1.1), 'Source mask (bits)' (32), 'Destination address' (empty), 'Destination mask (bits)' (empty), 'Protocol' (TCP), and 'TCP flag(s)' (a list including SYN, ACK, FIN, URG, PSH, RST, ECE, CWR, NS). Below these are 'Source port(s) - ; separated' (25), 'Destination port(s) - ; separated' (empty), and 'Packet length' (>=1000). At the bottom, there is an 'Action' dropdown menu (open, showing options like QoS 0.1 Mbps, QoS 1 Mbps, QoS 10 Mbps, QoS 100 Mbps, QoS 500 Mbps, Discard, Accept, and Accept + community) and an 'Expiration date' field (2020/12/01 16:09).

Figure 2.6: Configuration of an IPv4 rule in the ExaFS UI [26]

The web application provides a **UI** and a **REST API** for ExaBGP rule **CRUD** operations. It uses the Flask framework. Administrators can create *communities* specified by IP prefixes, and users are then divided into these *communities* to access rule configuration for their specific networks. Each user can create IPv4 or IPv6 based rules or RTBH rules for the networks in their *community*. Figure 2.6 shows the **UI** for an IPv4 rule configuration. After specifying conditions for matching packets by a rule, an action to be performed on these packets can be selected. One of the available actions not shown in the figure is „Redirect to **DDoS** Protector“. In the current version of ExaFS, this action does not change the configuration of the **DDoS** Protector device, and it is impossible to configure **DDoS** Protector rules using the ExaFS **UI**. The ability to configure **DDoS** Protector rules using ExaFS **UI** will be implemented in this thesis.

ExaFS uses the Flask framework to provide the **GUI** and the **REST API**.

Flask

Flask is a microframework for building web applications, first released in 2010. It is based on the *Werkzeug* web server gateway interface and *Jinja* template engine. By default, *Flask* only contains a minimal web server implementation – without implementing a database layer, authentication, form validation and other features, where different libraries already exist to handle that functionality. Instead, *Flask* supports extensions to add that functionality to an application, such as if *Flask* itself implemented it. Since *Flask* is minimalistic, applications written using *Flask* only contain the functionality they need, making them faster and smaller.

```
#!/usr/bin/env python3
# Python application packed with zipapp module
(binary contents of archive)
```

Listing 2.6: Example contents of a `.pyz` file [10]

2.8 ZipApps

Python Zip Applications, ZipApps for short, are a Python standard for executing zip archives containing program files in Python format without manually unpacking the archive. These archives provide a way to distribute software consisting of a collection of modules in a single file. The implemented **REST API** will be distributed in the ZipApp format.

ZipApp files are identified by the `.pyz` file extension or the `.pyzw` file extension for windowed applications. As shown in listing 2.6, ZipApp files can be prefixed with an environment path, usually `/usr/bin/env` and either `python2` or `python3`, based on the major Python interpreter version used by the software, followed by an optional description and then binary contents of the archive. On Unix systems, the first line allows the OS to run the file with the correct interpreter via the standard „shebang“ support. On Windows, the Python launcher implements shebang support [10]. The binary contents of the archive follow shebang and description.

ZipApps require a suitable version of the Python interpreter to run. The Python interpreter provides a simple module for creating and manipulating ZipApp archives. Archives created by this simple tool only contain the modules defined by the software. If the software requires third-party packages to run, users need to install them before running the software.

Third-party applications are available, which provide advanced ZipApp features, such as dependency management and environment virtualization.

Pex

Pex, short for Python Executable, is a file format based on Python ZipApps and a tool to manage these files. It provides a general-purpose Python environment virtualization similar to Python `virtualenv` module. Unlike ZipApps, Pex files include a Python interpreter and all required third-party Python packages. Therefore, to run a Pex file, users do not need to have the Python interpreter installed on their computer. Due to including a Python interpreter, Pex files are not multi-platform – a different Pex file has to be created for different operating systems.

The Pex tool creates a virtual environment that includes the Python interpreter. By default, the Python interpreter is copied from the environment that the pex tool is running in. Then the tool uses the Pip package manager to download dependencies to its environment, builds and includes the modules of the software being packaged, includes a Pex-specific „main“ file to interact with the environment, and compresses the whole environment to a `.pex` file [27].

Shiv

Shiv is a tool for building ZipApp files with their dependencies included. Compared to PEX, Shiv is less complex. It uses the existing `site-packages` folder from a project to include dependencies in a ZipApp file and does not include a Python interpreter executable. Shiv

can also download required dependencies using the pip package manager if the dependencies are not present in the `site-packages` folder. In addition to including dependencies, Shiv adds a specialized `__main__.py` file that instructs the Python interpreter to unpack included dependencies to a known location. Then, the main file adds the unpacked dependencies to the Python interpreter's search path at runtime.

Shiv allows developers to specify an entry point in the application during packaging. An entry point is a specific function in the packaged application that the `__main__.py` file calls when a user starts the packed application instead of showing the Python interpreter interface. Specifying an entry point allows developers to simplify the usage of the packed applications since users can run the application and immediately get the results they expected instead of manually running the Python package [2].

Chapter 3

Design

The application is divided into the **REST API** for configuring the **DDoS** Protector device and the extension of the ExaFS **UI** to support the **DDoS** Protector rule configuration, which will use the **REST API**. To design the application, first, the use-cases had to be determined.

3.1 Use cases

The **DDoS** Protector device has two main use-cases: detecting and mitigating **DDoS** attacks. Two personas needed to be considered: user and administrator. Users can access the **UI** to browse statistics and manage rules, but the configuration process must be as simple as possible, so the users cannot accidentally cut the connection to their network when trying to respond to a **DDoS** attack. Therefore, administrators are responsible for creating rule templates, and users will then create rules based on the templates. Administrators will also handle adding new **DDoS** Protector devices to the **UI**. Figure 3.1 shows the use-case diagram.

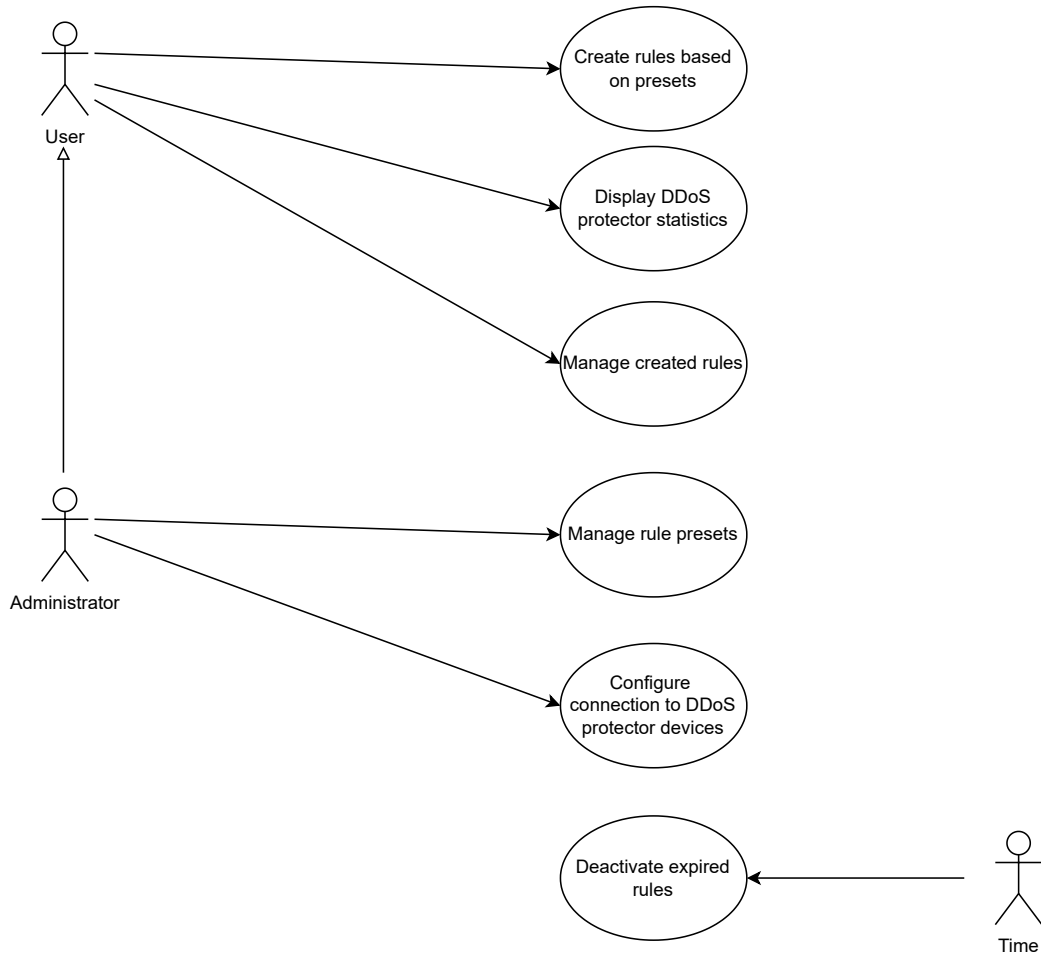


Figure 3.1: Application use-case diagram

Use Case 1 – Manage rule presets

An administrator detected a new type of attack or a current preset not mitigating attacks correctly. They create a new preset or modify or duplicate an existing one and fill the **DDoS** Protector rule values that correspond to mitigating the newly detected attack type. After saving the preset and giving it a descriptive name, the preset is available to users. Administrators may also remove presets if they deem them unsuccessful at stopping the **DDoS** attacks they are supposed to stop.

Use Case 2 – Configure the connection to DDoS Protector devices

When a new **DDoS** Protector device is added to the network or changes credentials, an administrator configures the device in the **UI** by filling in the hostname, the port and the **API** key of the new or modified device. The device is then available for administrators to configure as default to protect selected target network ranges or set as backup.

Use Case 3 – Create rules based on presets

A user detected a **DDoS** attack on their network. They create a new rule and fill in information about their network and targeted ports. Then they select the action to be

redirecting the traffic to the **DDoS** Protector device. They select a rule preset from the available preset based on the type of attack and either deploy them normally or in the dry-run mode to only collect statistics and check if the rule is configured correctly. After saving changes, the rule is available in the rule list.

Use Case 4 – Manage created rules

A user detected a repeated **DDoS** attack that they created a rule for in the past or made changes to their network. They go to the list of expired rules and activate the rule again by changing the expiration date to some time in the future. If changes were made to the user's network and some rules are no longer valid, the user removes the rules from the rule list.

Use Case 5 – Display DDoS Protector statistics

A user wants to check if a rule is appropriately configured and mitigates an incoming attack. The user selects the rule from the rule list or the whole device and selects the option to show statistics. Charts and statistics are displayed. The user checks these charts and statistics to see if the rule configuration mitigated the incoming attack.

Use Case 6 – Deactivate expired rules

Each rule in ExaFS has an expiration date and time. When the time is reached, the rule gets automatically deactivated. If the rule action is redirecting to **DDoS** Protector, the **DDoS** Protector rule is also deactivated. Deactivated rules stay in the rule list in the „expired“ category, so they can be reactivated when needed.

3.2 REST API design

First, a draft for the **REST API** structure was created in a document shared with other CESNET members, and the draft was then changed based on discussion with the people who had access to the document. All routes return data in JSON format. The `rules` route accepts HTTP methods GET, POST, PUT and DELETE; the other routes only accept the GET method.

- `/rules`: Routes for **CRUD** operations on rules and collections of rules. The `/rules` route serves for operations on collections of rules. The operation is based on the HTTP method: POST to create, GET to read, PUT to update, and DELETE to remove a rule.
 - `/rules/<rule ID>`: performs **CRUD** operations on a single rule specified by the rule identification number. Uses the HTTP method in the same way as the `/rules` endpoint.
- `/pipelines`: List of pipelines for each Protector worker and their status. Pipelines specify a sequence of operations on the incoming packets.
 - `/pipelines/<pipeline name>`: Status and the number of workers for a specific pipeline.

- `/pipelines/<pipeline name>/stages`: List operations in the specified pipeline and statistics for each operation, such as the number of errors and received and sent packets.
- `/pipelines/<pipeline name>/workers`: List Protector workers in a specified pipeline and their status (up or down) and statistics.
- `/ports`: List available port numbers on the Protector device and their status (up or down)
 - `/ports/<port number>`: Get a status, MAC address and firmware version for the specified port.
 - `/ports/<port number>/dev-stats`: Port statistics on the number of sent and received packets.
 - `/ports/<port number>/dev-xstats`: Extended port statistics. Includes the number of multicast, broadcast and discarded packets, the number of errors divided by error type and more, depending on the Protector device version.
- `/router/normal` or `/router/origin`: Reads statistics about either *normal* or *origin* routers, such as the number of next-hops, number of IP addresses assigned and number of VLAN destinations.
- `/version`: A single route for reading the `/self/version` file from the AppFS system. The route returns the version number of the running **DDoS** Protector software instance.

3.3 GUI design

After consultation with current users of the ExaFS system, it was decided that the configuration of the Protector device will be done using rule templates. Administrators will specify these rule templates based on their experience with previous attacks. Users will load a rule template, a pre-configured rule for blocking a specific type of attack, and edit values specific to their network. The **GUI** will simplify editing values like thresholds and limits using range sliders or pre-configured value selections. Administrators can choose to lock some of the rule parameters in a rule template to prevent users from changing values, that might potentially prevent the rule from blocking the attack or cause other unwanted behaviour.

Rule configuration

The first iteration of the design, shown in figure 3.2, was designed based on the current version of ExaFS. However, the original design is not very user friendly, and since inexperienced users will use the **GUI**, a new design was made. The problem with the current version of ExaFS is that there are too many values users can set, and the input fields are not organized. Some of the values are optional, which is not indicated in any way.

The new design, shown in figure 3.3, simplifies the configuration. Users will interact with the **GUI** starting from the top of the page and moving down without moving around the page. Values that can be left with a default value are hidden by default to reduce the amount of information on the page. Users can add these values by clicking the „Add field“ button at the bottom of the rules.

Exafs v 0.4.4 Add IPv4 Add IPv6 Add RTBH API Key Admin Logged in as

New IPv4 rule

Source address:

Source mask (bits):

Protocol:

TCP flag(s):

Destination address:

Destination mask (bits):

Source ports:

Destination ports:

Packet length:

Action:

Expiration:

DDoS protector preset:

Limit:

Figure 3.2: The first iteration of the GUI design based on the current version of ExaFS

New rule

Rule name:

Rule expiration date:

When incoming requests match...

Field	Operator	Value
<input type="text" value="Source IP address"/>	<input type="text" value="is in"/>	<input type="text" value="e.g. 192.168.0.0/24"/>
<input type="text" value="Destination IP address"/>	<input type="text" value="is in list"/>	<input type="text" value="Internal servers"/> Manage lists
<input type="text" value="Protocol"/>	<input type="text" value="equals"/>	<input type="text" value="TCP"/>
<input type="text" value="Source port"/>	<input type="text" value="is in"/>	<input type="text" value="e.g. 20-40,50"/>
<input type="text" value="Destination port"/>	<input type="text" value="is not in"/>	<input type="text" value="e.g. 20-40,50"/>

[+ Add field](#)

Then...

Choose an action:

DDoS protector rule preset:

Field	Value	
<input type="text" value="Rule type"/>	<input type="text" value="Amplification"/>	<input type="button" value="🔒"/>
<input type="text" value="Limit [bps]"/>	Restrictive <input type="text" value="10M"/> Relaxed	<input type="button" value="✕"/>
<input type="text" value="Limit [pps]"/>	Restrictive <input type="text" value="1k"/> Relaxed	<input type="button" value="✕"/>
<input type="text" value="Protocol"/>	<input type="text" value="TCP"/>	<input type="button" value="✕"/>
<input type="text" value="TCP flags"/>	<input type="text" value="SYN"/>	<input type="button" value="✕"/>

[+ Add field](#)

Figure 3.3: The second iteration of the ExaFS GUI design

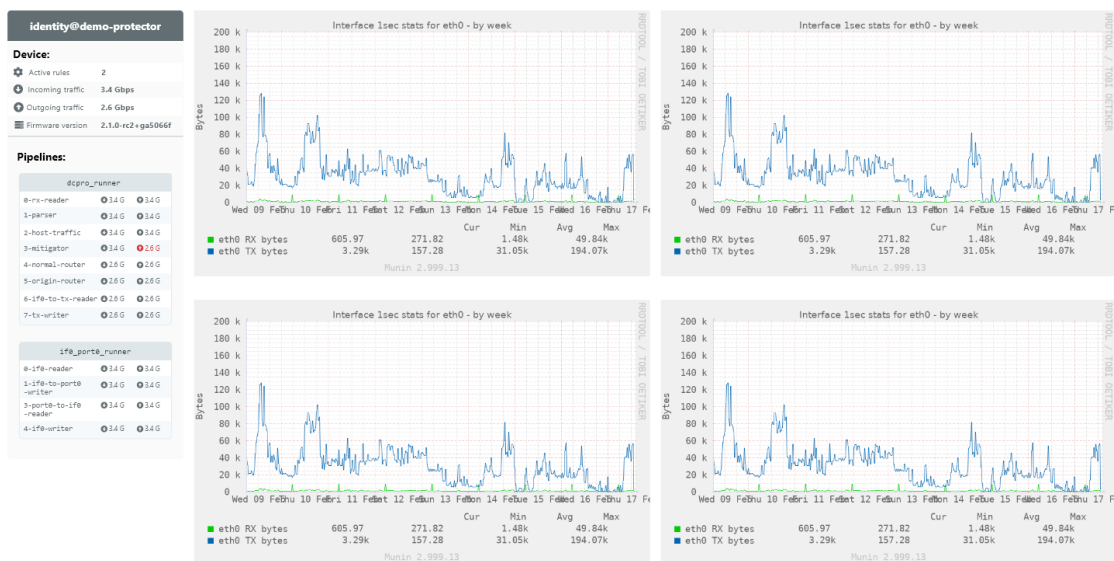


Figure 3.4: Design of the rule statistics page. Demo charts from Munin demo collector, available at <http://demo.munin-monitoring.org>. Charts might look different in the final version.

Rule template configuration

Administrators can create rule templates – a collection of predefined rule parameters. Some parameters can be set as user-editable, allowing users to change the value of a specific rule field when using a template. Administrators can duplicate templates, making it quick to create a similar template without filling in all the information again.

When users create a new rule and select „Redirect to DDoS Protector“ as the action, a dropdown with template names appears. When a user selects a template, the template appears as a list of inputs. Fields that are not set as user-editable appear as disabled, and fields that users can edit appear as different forms of inputs depending on the field. Fields like thresholds appear as a range select, fields which have defined sets of values appear as either a dropdown menu or a set of checkboxes, and numeric and text values appear as text inputs.

Reading statistics

The user interface will be capable of showing statistics from either DDoS Protector devices or specific rules.

When viewing device statistics, users can see the number of active rules on a device, firmware version, the mitigation pipeline, including current traffic for each step and overall traffic to and from the device, and charts with the recent traffic in and out of the device.

Rule statistics provide information about the rule, such as how long the rule has been active, the rule state – active, dry run or disabled, how many packets matched the rule, and how much traffic the rule dropped. The user interface also provides charts for the incoming and outgoing traffic that match the rule, if the rule is active or in the dry-run mode. Design of the rule statistics page is in Figure 3.4.

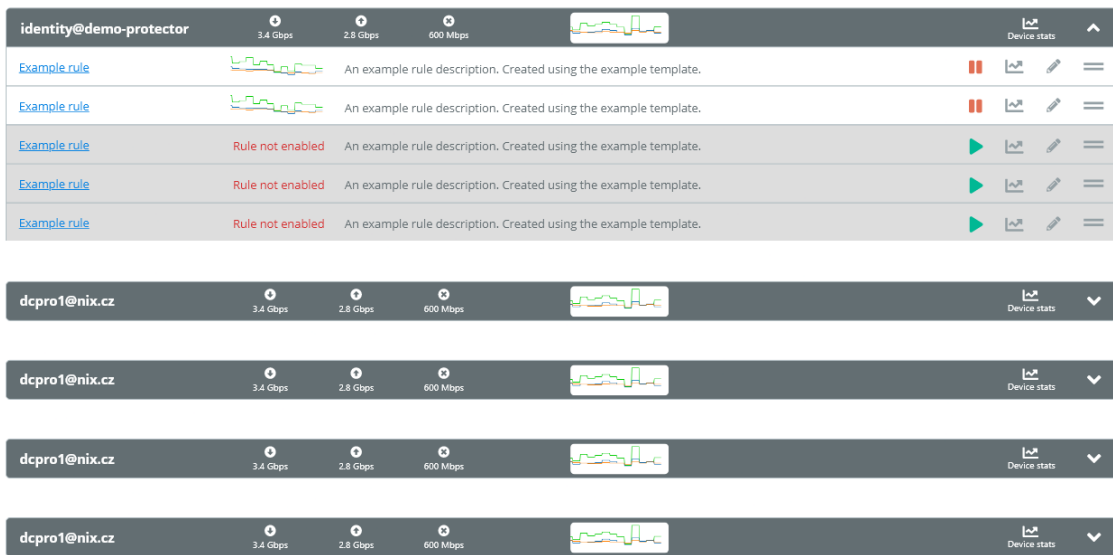


Figure 3.5: Design of the **DDoS** Protector rule list, divide by device instances.

List of DDoS Protector rules

The **DDoS** Protector rule listing page shows all configured rules, divided by the **DDoS** Protector device instances on which these rules are configured. The rule overview provides a quick way to disable and enable rules and view their description and incoming and outgoing traffic in the last hour. The rule table also allows users to reorder rules to change their execution order in the Protector device. As shown in Figure 3.5, disabled rules have different background colours, so users can quickly distinguish between enabled and disabled rules.

Chapter 4

Implementation

This chapter describes the implemented applications: The **REST API** and the inclusion of **DDoS** Protector configuration in the ExaFS application. The **REST API** provides access to the **DDoS** Protector device using the HTTP protocol; the ExaFS application uses the **REST API** to configure **DDoS** Protector rules and read mitigation statistics.

4.1 REST API

The **REST API** was implemented based on the design. It uses the FastAPI micro-framework and the **DDoS** Protector Python library to access rules saved in a database. The **REST API** also uses the AppFS system to access statistics reported by the Protector device.

The **REST API** uses an **API** key to authenticate incoming HTTP requests. The **REST API** expects the **API** key in the `x-api-key` HTTP header by default, but administrators can configure the header differently using the configuration file. If an incoming request does not include a key in its header, or the key is wrong, the **REST API** responds with an HTTP response with code 403 **Forbidden** and a message „Incorrect **API** key“.

A new requirement was added for the **REST API** between the design and the implementation – support for multiple instances of the **DDoS** Protector running on the same server. A new endpoint, `/instances`, was added to handle this change. The `/instances` endpoint returns a list of **DDoS** Protector instance identifiers from the AppFS system. These identifiers can then be passed as an optional parameter to the endpoints that read data from the AppFS system to filter data from specific instances.

At first, a prototype was developed to test the FastAPI framework and show the **REST API**'s functionality. The prototype was also used to test various authentication methods that FastAPI provides. Based on this testing, it was decided to use an **API** key for authentication, with more options possibly added in the future. Later, a new version was implemented to replace the prototype with an application with better code structure and better and more reliable code.

The **REST API** application is structured based on the backend part of the Full Stack FastAPI PostgreSQL project template¹. The template was simplified since the **REST API** uses the **DDoS** Protector Python library to access the database, so database structures from the template were not needed. The **REST API** application contains four directories: `api`, `core`, `schemas` and `tests`, and a `main.py` file. The directories contain an empty `__init__.py` file to make the Python interpreter recognize them as a package.

¹Available at <https://github.com/tiangolo/full-stack-fastapi-postgresql>

The main file

The `main` file serves as an entry point to the **REST API** application. It creates an instance of the FastAPI framework, configures basic information, such as the hostname and port, imports endpoints and registers exception handlers. Then it defines the `run` function that starts the webserver for the application. The `run` function is an entry point in the packed Shiv application.

API

The `api` folder contains a `dependencies.py` file and a single sub-folder called `v1`, containing a file for each API endpoint: `instances.py`, `pipelines.py`, `ports.py`, `router.py`, `rules.py` and `version.py`. Calling the folder `v1` helps with future changes to the **REST API**. When significant changes to the **API** are needed, a new version should be created in a folder called `v2`, `v3`, and so on. Legacy software that uses the old versions of the **API** can still use the endpoints from the older versions, allowing significant changes to the **API** without breaking software using the older versions of the **API**.

The `dependencies.py` file contains functions to be called using FastAPI's dependency injection system. These functions provide a connection to the database, check the **API** key in incoming requests, read **DDoS** Protector device instances from requests and provide an instance of the AppFS parser. The AppFS parser dependency also handles the optional „instance“ parameter to select a specific **DDoS** Protector instance when reading AppFS statistics.

The `rules.py` file handles all endpoints starting with the `/rules` **URL**. The functions check if incoming data are valid and call appropriate functions from the **DDoS** Protector Python API. It provides **CRUD** operations on the rules in the database to handle use-cases 3 and 4.

The `pipelines.py`, `ports.py`, `router.py` and `version.py` files provide endpoints for reading corresponding AppFS statistics. All endpoints in these files use the AppFS Parser class methods to get the information they provide in HTTP responses. The endpoints also use the AppFS parser dependency, adding an optional „instances“ parameter to each endpoint, allowing users to send requests with specified **DDoS** Protector instance identifiers to filter only the Protector instances which interest them.

Core

The `core` folder contains most of the logic of the **REST API** application. In addition to an empty `__init__.py` file, five files are present: `appfs_parser.py`, `cli_args.py`, `config.py`, `exceptions_handlers.py` and `helpers.py`.

The `appfs_parser.py` file contains the `AppfsParser` class, which provides functions for reading statistics from the AppFS system. It takes two arguments from the config file to find the AppFS instances and a list of instance names that the parser should consider when reading statistics. The two values from the config file are the base directory, where AppFS is mounted, and a pattern for the `fnmatch` library to specify the names of directories which contain AppFS filesystems. On initialization, the `AppfsParser` class reads all directories from the given base directory, filters them using the `fnmatch` pattern and looks for a `self/identity` file in the matched directories. If the identity file is present in a directory, the `AppfsParser` class adds the full path to the directory to the `self.identities` class attribute with the contents of the identity file as a key. If the `self/identity` file is not

present in a matched directory, the parser ignores the directory. The `AppfsParser` class provides methods for parsing generic AppFS files; Chapter 2.1.4 describes these files. The `AppfsParser` also provides methods to parse the `dev_stats` and `dev_xstats` files, combine pipeline information or worker information into one object, and combine multiple files with information about ports into one object. Support for multiple AppFS instances is solved using a decorator `_for_each_instance_wrapper`. This decorator uses the `self.instances` attribute to iterate any decorated function over detected instances of AppFS. The decorator adds the results of each iteration to a Python dictionary structure, where keys are the AppFS identities and values are the results of the decorated function. Using a decorator, the loop iterating over instances does not need to be repeated in the code, making future changes easier.

The `cli_args.py` file parses the Command-Line Interface (CLI) arguments passed to the REST API application and provides them in a variable that other parts of the application can import to get the CLI argument values. It uses the `argparse` Python library to read the arguments. The REST API application accepts two arguments: `--help` or shortened version `-h` or `--config` or shortened `-c`. The `help` argument prints the available arguments and a short description of the application, `config` sets the path to the configuration file. More arguments may be added in the future as the application grows.

The `config.py` file reads the configuration file based on the provided path to the file. If no path is provided, it reads a file called `restapi.ini` from the current working directory. In addition to reading the configuration file, `config.py` also provides the values as a `ConfigParser` object from the `configparser` Python library and checks the configuration file values for validity. An exception is raised if the configuration file does not specify an API key, contains invalid values for options with specified possible values or if the pattern to look for AppFS directories specifies sub-directories. If the config contains a combination of options that will start the REST API with no endpoints, the configuration parser prints a warning. If the configuration file is correct, the `config.py` file provides a variable with the configuration values that other modules can import to read the values.

The `exception_handlers.py` contains functions that handle exceptions raised by the application that are not handled elsewhere. In the REST API application, only one exception is handled this way – an exception raised when trying to connect to a database and failing. The handler function reads the error message and changes the exception into an HTTP response with the code 500 **Internal Server Error** and a message describing that the connection to the database failed.

Schemas

The `schemas` folder contains classes specifying models for FastAPI response models and data validation of incoming HTTP requests. The models can be either Python Enum classes or models made using the `Pydantic` library, described in chapter 2.5. FastAPI uses schemas to validate data in incoming requests and document response values. Format for incoming data is specified by using Python type hints in the parameters of a function that handles the request. For responses, the format is specified by setting the `response_model` parameter in the FastAPI route decorator.

Since the `DDoS Protector` Python library provides `Pydantic` models for most required structures, the `schemas` folder in the REST API only contains one file with an enum specifying the routing types – `origin` or `normal`. The rest of the response and input models are

specified by models provided by the **DDoS Protector** Python library or simple data types provided by Python or the `Pydantic` library.

Configuration

The **REST API** reads a configuration file in the INI format. The location and name of the configuration file can be set when starting the **REST API** using the `--config CLI` parameter. If no file is set as the configuration file, the **REST API** looks for a file called `restapi.ini` in the directory from which it was started.

As shown in listing 4.1, the configuration file is divided into five sections: `api`, `security`, `AppFS`, `rules` and `database`. The `api` section allows the configuration of the host IP address or name and port that the **REST API** will be available at the base **URL** of the **REST API** and the **REST API**'s name in the documentation. Administrators can specify the `API` key and the header in the security section; The header specifies which HTTP header contains the `API` key in incoming requests. The `AppFS` section specifies the mount point of the AppFS instances and a pattern of the directory names containing the instances. If this section is missing from the configuration, the AppFS-related endpoints are disabled in the **REST API**. The `rules` section contains only one option – `controller`. The `controller` option specifies where the rules are stored. In the current version of the **REST API**, the only valid options are `database` to store the rules in a PostgreSQL database and `none` to disable the rules endpoint. The `database` section is only required if the `controller` option from the rules section is set to `database`. The `database` section specifies the hostname and port of the database, the database name, and a username and password to access the database.

Tests

The tests folder contains files for unit testing the API. In order to make the tests independent from any running instance of the **DDoS Protector**, it contains a copy of the AppFS file structure and a connection to an in-memory database. It also contains an `api` folder that copies the structure of the **REST API** endpoints. Chapter 5.2 describes the tests in detail.

Documentation

As mentioned in chapter 2.4, FastAPI automatically generates documentation for all endpoints using the Swagger **UI**. Figure 4.1 shows using the documentation **UI** to create a new rule. It allows users to enter the `API` key to authorize themselves and, after authorization, send all available requests to the **REST API** endpoints. It shows all the required and optional information for the requests. If a field in a request has a limited set of allowed values, the documentation shows these values as a dropdown selection menu.

Figure 4.2 shows a result after sending a request without setting the `API` key first. The **UI** shows a command for the `CURL` utility to repeat the same request, request **URL**, the code and result of the request, and the response headers. Users can download the response body in JSON format.

The Swagger **UI** documentation was helpful during the early stages of development to test the endpoints before automatic tests were created.


```
[api]
host = localhost
port = 8000
base_url = /api/v1
project_name = DDoS protector

[security]
key = CHANGE_ME
header = x-api-key

[appfs]
path = /var/run/
dir_pattern = dcpro_*

[rules]
controller = database

[database]
host = localhost
port = 5432
dbname = dcpro
user = dcpro
password = dcpro
```

Listing 4.1: Default restapi.ini configuration values

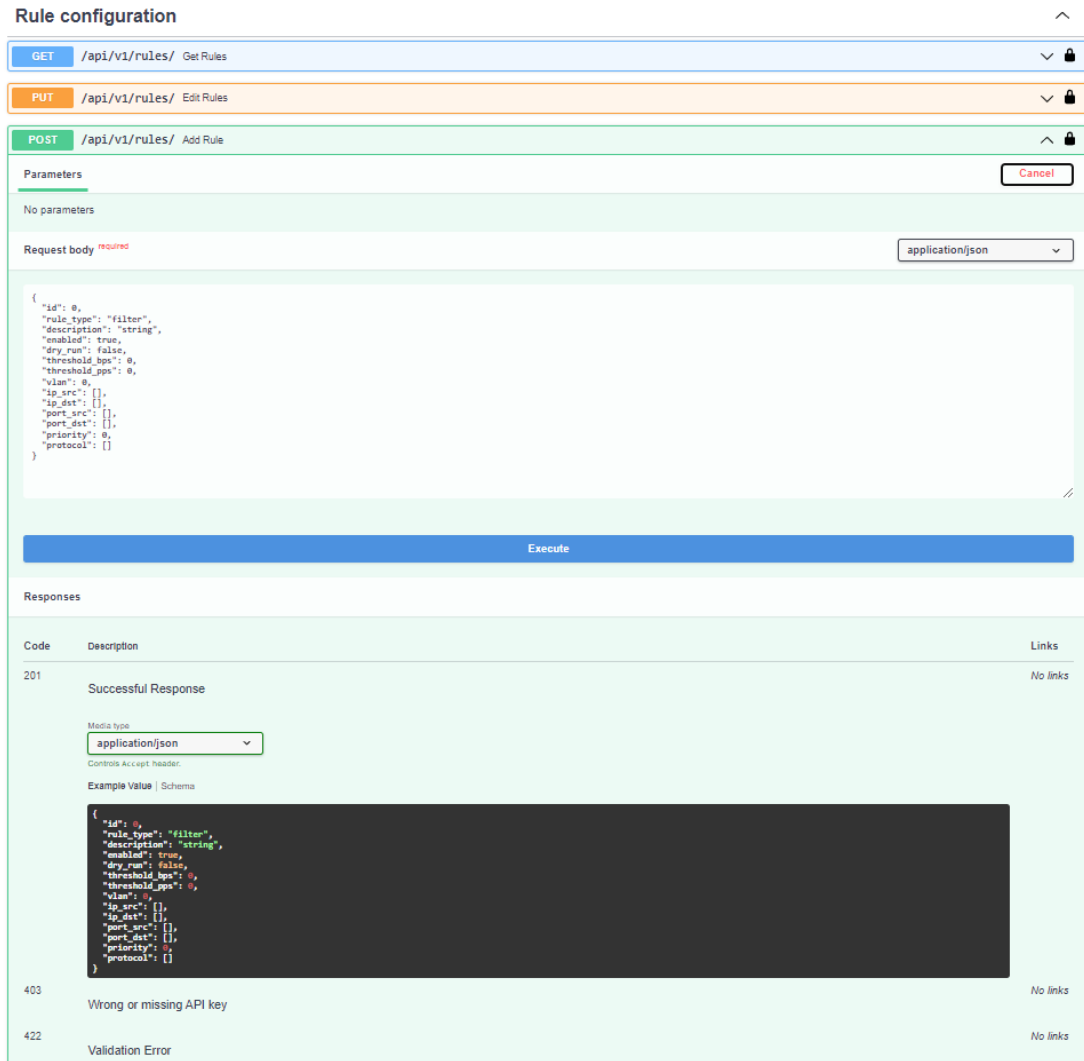


Figure 4.1: The documentation UI when sending a request to create a new rule.

Packaging

The REST API was included in the DDoS Protector RPM package. In order to include the REST API, it had to be packaged into an executable. Packaging into an executable was included in the build process for the DDoS Protector application, which includes running the CMake build software.

At first, PEX was selected to package the REST API since it includes the Python interpreter. However, PEX caused problems when including local dependencies that the REST API requires. The packaging is done using the CentOS operating system in versions 7 and 8. On the CentOS 7, the packaging with PEX worked with no problems, but on CentOS 8, the package failed to build correctly and usually was missing some or all packages. Even after many days of debugging, a solution was not found, so the packaging software was changed to Shiv. Since Shiv is more straightforward in principle, the packaging worked on the first try.

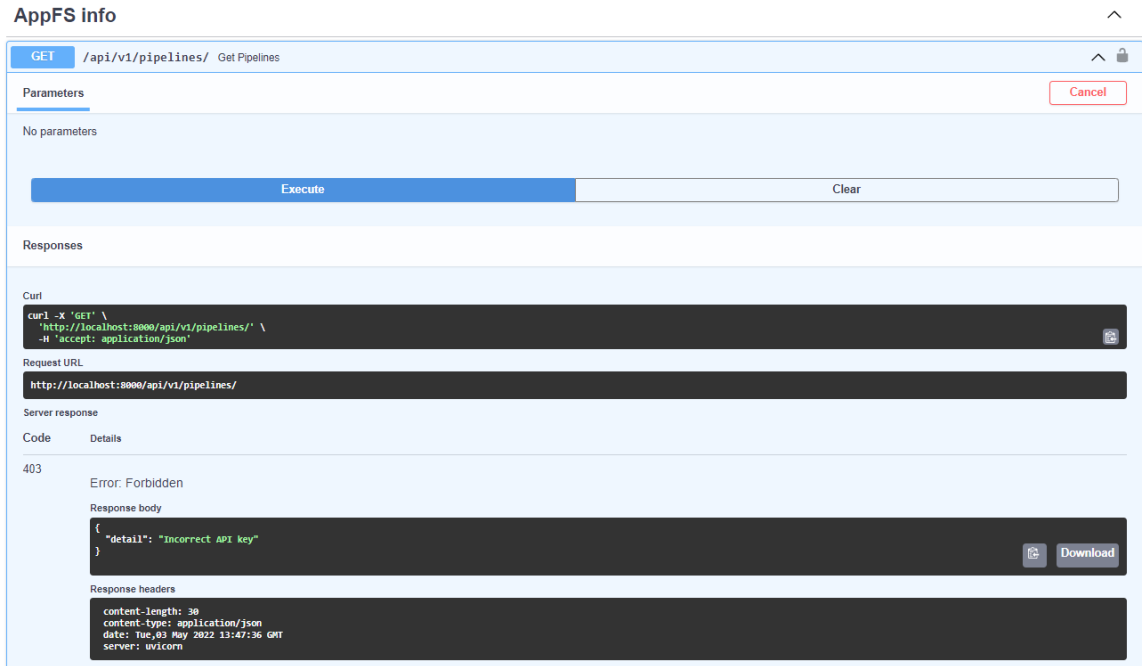


Figure 4.2: The documentation UI after sending a request without setting the API key.

RPM packages are created using a `.spec` file that specifies the commands and variables for the build process. DDoS Protector uses a file called `dcpro.spec`, which sets the version, specifies required programs and libraries and provides descriptions and installation instructions for various programs included in the DDoS Protector project. It includes a build and an installation process, which uses the CMake build software.

CMake contains instructions on building the various parts of the DDoS Protector. Among them is a build process for the REST API. CMake finds required libraries to build the REST API dependencies and calls a build script. The build script creates a Python virtual environment, installs Shiv into it, and calls Pipenv in the virtual environment to install the REST API dependencies. Finally, the build script finds the `site-packages` folder that contains the installed dependencies and calls Shiv to build the package. After running the script, CMake provides an installation command that copies the built REST API ZipApp and configuration file to the specified directories and sets the correct permissions of these files.

4.2 User Interface

The UI based on the ExaFS UI provides administrators with a rule template configuration and users with DDoS Protector device configuration and statistics. A prototype was implemented to show the flow of action to the future users of the UI, which was then shown to the current users of the ExaFS GUI to discuss the changes. After the discussion, the first version of the UI was implemented. It was agreed that the current developers of ExaFS would create a maintenance update before integrating the changes described in this thesis. Therefore, the support for DDoS Protector will be included in the open-source ExaFS repository after these maintenance updates are done.

Updating ExaFS dependencies

The first step of the **UI** implementation was updating some of the outdated user interface libraries. The ExaFS **UI** used the Bootstrap **UI** library in version 3 from 2018, which is no longer supported and does not receive any updates. The Bootstrap library was updated to the latest version, version 5.1.3, to apply the latest security and functionality updates. Version 5 no longer uses the jQuery library, dropped support for the Internet Explorer versions 10 and 11 and old versions of other browsers, changed some CSS class names and JavaScript `data-` HTML tag names and removed icons from the library.

ExaFS used the jQuery library in some scripts. These scripts were rewritten using plain JavaScript or replaced by functions from the new Bootstrap library version. Some CSS classes were replaced with new ones. The navigation menu and some input fields were remade in the new format to support the new Bootstrap version.

A new library for icons was added to replace the removed icons from Bootstrap version 3. A standalone Bootstrap-icons library was created to replace this functionality. It was selected to replace the removed icons due to its similarity to the removed icons and ease of use.

Rule configuration

As shown in Figure 4.3, the „New rule“ form was reorganised for better clarity. The source and destination address fields merged with their respective mask inputs, and this change allowed the source and destination port inputs to move under the respective addresses. The „comments“ input was shrunk to save screen space, and the action selection was moved to the end of the form since it now can reveal more options that appear below the action selection when selecting the „Redirect to **DDoS** Protector“ action.

Minor quality-of-life improvements were added: The ability to fill a company IP range by clicking a button and selecting the IP range from available ranges and automatically hiding the TCP Flags input field when a protocol other than TCP is selected. In future updates, a dynamic form validator will be added to check the input values as the user writes them instead of checking the values after the form is submitted, as is the case now.

A preset selection appears when users select the „Redirect to **DDoS** Protector“ action. Administrators define the presets, and users can only edit the values the administrators set as user-editable in the preset. In order to make the configuration process as simple as possible, the preset values are hidden by default. Users can view the values of the **DDoS** Protector rule by clicking the „Show advanced options“ button. An example of the advanced options is in figure 4.4. Users can only change the limits in predefined ranges. The edge values of these ranges are labelled as „relaxed“ and „restrictive“ to help users understand the consequences of changing the values of limits. Users can also modify the protocol in the example.

ExaFS / ExaFS_0.5.5 Active rules Add IPv4 Add IPv6 Add RTBH API Key Admin DDoS Protector <admin@example.com>, role: admin, org: Example Org.

New IPv4 rule

Source address / Source mask (bits) [Fill range](#) 192.160.0.1 / 24 Destination address / Destination mask (bits) [Fill range](#) 192.160.0.1 / 24

Source port(s) - ; separated Destination port(s) - ; separated

Packet length - ; separated Protocol UDP

Expiration date 05/12/2022 01:27 PM

Comments

Action Redirect to DDoS protector

DDoS protector rule preset Limit UDP traffic

[Show advanced options](#)

Deploy Dry-run deploy

Figure 4.3: The new ExaFS GUI when creating a new rule including a DDoS Protector rule

DDoS protector rule preset Limit UDP traffic

[Hide advanced options](#)

Rule type Amplification Restrictive (1M) Relaxed (10G)

Limit [bps] Restrictive (1k) Relaxed (1M)

Limit [pps] TCP UDP ICMP

Protocol

Deploy Dry-run deploy

Figure 4.4: Advanced options for the DDoS Protector rule preset

Rule preset configuration

Administrators can create rule presets by creating a blank preset or duplicating an existing preset. The UI for creating presets is in figure 4.5. Each preset has to have a name that will be presented to a user in the preset selection menu; Each name must be unique. After filling a name, an administrator can select a rule type and start adding rule fields by clicking the













ID	Name	# Rule fields	Edit
0	Limit all traffic (amplification)	3	   
1	Limit UDP traffic	4	   
2	SYN flood mitigation	5	   

Figure 4.6: List of rule presets

„Add field“ button. Each field has a dropdown menu with the field key selection, an input field which changes based on the field key, a checkbox signifying whether users can edit the value and a button to remove the field. The value input field currently supports text input, a dropdown menu and a group of checkboxes. Users will never be able to change the rule type due to its effects on the rule – some fields might become invalid, while others might become required, leading to invalid rules being created from presets. Therefore, the rule type can not be set as user-editable.

ExaFS / ExaFS_0.5.5 Active rules Add IPv4 Add IPv6 Add RTBH API Key Admin DDoS Protector <admin@example.com>, role: admin, org: Example Org.

Edit preset Limit UDP traffic - copy

Preset name

Rule type

Limit [bits/s]	<input type="text" value="1000000000"/>	<input checked="" type="checkbox"/> User can edit	<input type="button" value="x"/>
Limit [packets/s]	<input type="text" value="100000"/>	<input checked="" type="checkbox"/> User can edit	<input type="button" value="x"/>
Protocol	<input type="checkbox"/> TCP <input checked="" type="checkbox"/> UDP <input type="checkbox"/> ICMP <input type="checkbox"/> SCTP	<input type="checkbox"/> User can edit	<input type="button" value="x"/>

Figure 4.5: UI for editing rule presets

Administrators can see all configured rule presets in a table shown in figure 4.6. The table shows the numeric ID of the preset, the preset name, and the number of fields and provides four buttons to handle the presets. The first button displays the fields and values configured of the preset in a modal window, as shown in figure 4.7. The second button edits the rule, while the third creates a duplicate. These buttons redirect the administrator to the preset editing page and fill in the selected preset values and name. The only difference is the action after saving the preset – if the edit button was pressed, the form changes the rule; if the duplicate button was pressed, the form creates a new rule. The fourth button allows administrators to remove presets.

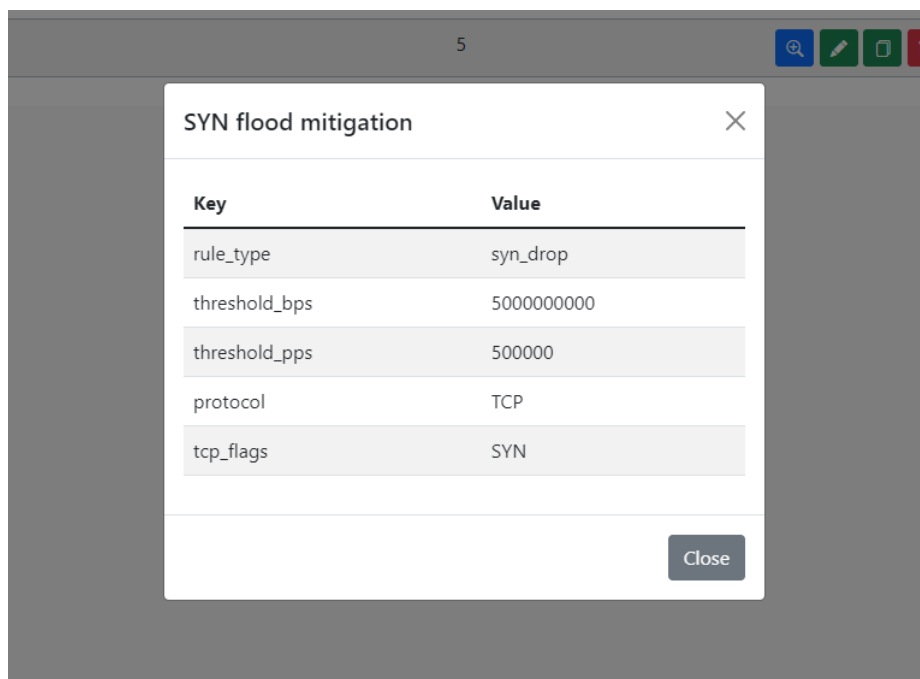


Figure 4.7: Modal window displaying the preset fields and values

Chapter 5

Testing

Since the project in this thesis is divided into two independent parts, two types of testing were used to test the system. Unit tests in the Pytest framework cover the **REST API** functionality. The **REST API**'s build process and code style were checked using Gitlab Pipelines. The **UI** was tested by running the **UI** in a test environment and manually going through the UI since it was often changing based on the user feedback. Automatic unit tests using Pytest will be included in future versions.

5.1 Test environment

The **DDoS** Protector software supports the CentOS Linux distribution in versions 7 and 8. Therefore, a virtual machine with CentOS 8 was selected to test the **REST API** and the ExaFS **UI**. CESNET also provided servers for testing with both supported versions of CentOS and all dependencies of the **DDoS** Protector software to make developing and testing the build process more painless. These servers were also used to test the compatibility of the **REST API** between the two versions of CentOS.

The **REST API** was tested in a virtual environment created using the Pipenv tool. The virtual environment contained all dependencies required to run the **REST API** without interfering with other libraries installed on the same computer. Since **DDoS** Protector uses PostgreSQL to store the rules, a PostgreSQL server was installed in the testing environment and configured with the proper SQL scripts.

The ExaFS **UI** uses the Shibboleth identity provider to authenticate users, which is challenging to set up locally. Luckily, ExaFS reads the **EXAFS_ENV** environment variable, and if the variable is set to „devel“, ExaFS bypasses the authentication process, removing the need for installing Shibboleth locally. The ExaFS application was run in a Python virtual environment created using the „virtualenv“ Python utility. ExaFS requires a connection to a MySQL server, so the MariaDB server was running on the testing virtual machine. The MariaDB server was configured using instructions from the ExaFS GitHub repository [26].

5.2 Testing the REST API

In the early stages of development, mainly in the prototype phase, the **REST API** was tested manually using the Swagger **UI** to send requests. Since the Swagger **UI** allows testing the endpoints with various input data, it, combined with the console output of the **REST API** application, was sufficient to test the application at first.

Later, as the number of endpoints grew, automatic unit tests were developed using the Pytest testing framework and the `TestClient` class provided by the FastAPI framework. The `TestClient` is based on the `Requests` library, which allows sending HTTP requests. The `Requests` library automates the HTTP connection, making writing the tests quicker. Only the `URL` and the HTTP method need to be specified. The `Requests` library handles the HTTP connection and provides a structure with the HTTP response's resulting data and HTTP status code. The FastAPI's `TestClient` class handles the redirection of these requests to the FastAPI application, so it does not need to be started manually before running the tests.

The `conftest.py` file is run before starting the tests. The file provides testing fixtures, such as the authentication header and the example rules, and sets up the connection to the temporary database. The temporary database is then initialized with the correct structure and example rules. Lastly, the `conftest.py` file overrides the database connection functions in the `REST API` to connect to the temporary database instead of the database specified in the config file. After the tests are finished, the `conftest.py` file removes the temporary database. The `prepare_config` function is run before the tests start to change the `REST API`'s configuration to known values, such as the `API` key and header and the path to the AppFS.

Each endpoint is tested multiple times to test all of the possible HTTP responses provided by the endpoint. The endpoints are tested with missing authentication, invalid and valid input data and requests for nonexistent information. There are two to six tests for each endpoint and HTTP method combination, depending on the endpoint.

A static copy of the AppFS structure is provided with the tests to test the endpoints that provide the AppFS statistics. The static copy was created by running the `DDoS Protector` software on a provided testing server and then copying the AppFS structure to the testing directory in the `REST API` application. Some of the values, such as IP and MAC addresses, were anonymized so that the testing AppFS could be shared. Testing the `REST API` on a live version of AppFS would make writing the tests challenging since most of the files in AppFS are constantly changing. The resulting values are known beforehand by having a static copy of AppFS to test on, so the `REST API`'s return values can be checked for correctness.

The testing framework sets up a temporary database each time the tests are started to test the rule configuration. This temporary database is removed when the tests are finished. The database is set up using the `pytest_postgresql` library. The library allows the creation of temporary PostgreSQL databases and their initialization with SQL scripts and data before the tests start. Testing the database manipulation itself is not the focus of the `REST API`'s tests since the database is handled by the `DDoS Protector Python API`, which has its unit tests to test that functionality. However, by using the temporary database in the unit tests, the tests are more in line with the actual usage of the `REST API`, which might help find more errors.

In addition to the unit tests, the `REST API`'s coding style and build process were tested using the Gitlab Pipelines continuous integration system. These pipelines are set to run each time new code is pushed to the `DDoS Protector` Gitlab repository. The pipelines check the code for the set code style, try to build, package and install the application on both supported versions of CentOS, and run the tests. If some of the pipelines did not pass, it blocks merging the code into the `master` branch and shows the error report.

5.3 Testing the UI

Before the implementation of the UI started, the UI was tested as a mockup in the Adobe XD mockup software. It allowed quick testing and modification of interaction between the pages and the new design.

Multiple users tested the UI and provided feedback, the UI was then modified based on the feedback. To test the UI, the ExaFS UI was set to a development mode by setting the `EXAFS_ENV` environment variable to `devel` before starting the UI. By setting this environment variable, the ExaFS UI skips the authentication process and sets the Flask framework into a debug mode, making it easier to test the UI and fix errors.

To test the UI, the DDoS Protector REST API was set to use a static snapshot of the AppFS structure and a standalone rule database. These settings removed the need to run the DDoS Protector device before testing the UI.

Chapter 6

Conclusion

This thesis provides an overview of the **DDoS** Protector device and compares **REST API** frameworks for the Python programming language.

After studying the Protector device and determining the use-cases for its configuration, a **REST API** and a **GUI** mockup were designed. A mockup of the **GUI** was first designed in the Adobe XD mocking software. The **REST API** was designed in a shared document. Both designs were consulted with other members of the CESNET organization and further modified based on their feedback. It was then decided that the **REST API** would be implemented first, and the **GUI** would be implemented as an extension of the ExaFS **UI** after the **REST API** was finished. Since the **DDoS** Protector Python API was being developed parallel to the **REST API**, changes were being made during development to reflect changes in the Python API.

Based on the framework comparison, the FastAPI framework was selected to implement the **REST API** for the Protector device. After the framework was selected, the implementation started. It was decided that the first version of the **REST API** will use an **API** key for authentication, with updates in the future adding support for more forms of authentication. Since the **DDoS** Protector Python API was still in the early stages of development when starting the development of the **REST API**, the endpoints for reading the AppFS statistics were implemented first. After that, support for rule configuration was added. Lastly, support for multiple instances of AppFS was implemented.

After implementing the **REST API**, work on the **GUI** was started based on the mockups. It was decided that the **GUI** would be implemented as an extension of the existing ExaFS **GUI**, so the ExaFS was extended by the **DDoS** Protector configuration. ExaFS uses the Flask framework, so the **GUI** uses Flask as well. ExaFS frontend dependencies were updated to the latest versions before implementing the **GUI**. The changes mentioned in this thesis will be a part of the open-source ExaFS repository after the current developers implement maintenance updates.

The „new rule“ form was updated with subtle quality-of-life updates and extended with a preset selection when the action „Redirect to **DDoS** Protector“ is selected. A list of **DDoS** Protector rules was implemented, divided by the Protector devices that the rules configure. A rule preset system was also implemented, allowing administrators to configure presets for various types of attacks that users can use to respond to an incoming **DDoS** attack quickly.

In the future, the **REST API** can be improved by adding support for multiple **API** keys or a more sophisticated authentication mechanism, such as JWT tokens. Another possible improvement to the **REST API** is adding support for reading mitigation statistics from the

DDoS Protector database. Database statistics were not considered in this thesis, as support for storing the statistics in a database was recently added to the DDoS Protector.

The GUI can be improved by adding more input types to the rule template configuration form and adding the ability to generate and display charts based on the statistics from the DDoS Protector database. Another possible improvement is form validation before the form is sent to the backend and the ability to test the rule templates before saving them. Support for multiple DDoS Protector devices should be added in the future, as it is not clear how the UI should work with multiple devices at this point.

Bibliography

- [1] CARNEGIE MELLON UNIVERSITY. *CERT Advisory CA-1996-21 TCP SYN Flooding and Spoofing Attacks*. MSU-CSE-06-2. Software Engineering Institute, Carnegie Mellon University, September 1996. 119-123 p. Available at: https://resources.sei.cmu.edu/asset_files/WhitePaper/1996_019_001_496172.pdf.
- [2] CARVALHO, L., WARSAW, B., LEWIS, J. and HERMANN, J. *Shiv Documentation*. Oct 2021 [cit. 2022-04-25]. Available at: <https://shiv.readthedocs.io/en/latest/>.
- [3] COLVIN, S. *Pydantic Documentation*. Mar 2022 [cit. 2022-05-01]. Available at: <https://pydantic-docs.helpmanual.io/>.
- [4] KASPERSKY LAB. *What is a DDoS Attack / DDoS Meaning* [online]. kaspersky.com [cit. 2021-11-20]. Available at: <https://usa.kaspersky.com/resource-center/threats/ddos-attacks>.
- [5] LIBERROUTER.ORG. *DDoS Protector / Liberrouter / Cesnet TMC group* [online]. liberrouter.org [cit. 2021-11-19]. Available at: <https://www.liberrouter.org/technologies/ddos-protector/>.
- [6] DJANGO SOFTWARE FOUNDATION. *Django Documentation*. Jan 2022 [cit. 2022-01-12]. Available at: <https://www.djangoproject.com/>.
- [7] DUTRA, D. *Apidaora*. Dec 2021 [cit. 2022-01-12]. Available at: <https://dutradda.github.io/apidaora/>.
- [8] FIELDING, R., IRVINE, U., GETTYS, J. and H. FRYSTYK, C. ad. *Hypertext Transfer Protocol – HTTP/1.1* [Draft standard]. RFC 2616. RFC Editor, June 1999. Available at: <https://www.rfc-editor.org/rfc/rfc2616.txt>.
- [9] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Dissertation. University of California, Irvine. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
- [10] HOLTH, D. and MOORE, P. *Improving Python ZIP Application Support*. PEP 441. 2013. Available at: <https://legacy.python.org/dev/peps/pep-0441/>.
- [11] INFORMATION SCIENCES INSTITUTE UNIVERSITY OF SOUTHERN CALIFORNIA. *Transmission Control Protocol* [Internet standard]. RFC 793. RFC Editor, September 1981. Available at: <https://www.rfc-editor.org/rfc/rfc793.txt>.
- [12] KUMARI, W., GOOGLE, MCPHERSON, D. and ARBOR NETWORKS. *Remote Triggered Black Hole Filtering with Unicast Reverse Path Forwarding (uRPF)* [Informational].

- RFC 5635. RFC Editor, August 2009. Available at:
<https://www.rfc-editor.org/rfc/rfc5635.txt>.
- [13] MAXIMILIEN, E. M., RANABAHU, A. and GOMADAM, K. An Online Platform for Web APIs and Service Mashups. *IEEE Internet Computing*. 2008, vol. 12.
 - [14] MONTAÑO, S. R. *Dependencies - First Steps - FastAPI*. Dec 2021 [cit. 2022-01-09]. Available at: <https://fastapi.tiangolo.com/tutorial/dependencies/>.
 - [15] MONTAÑO, S. R. *FastAPI documentation*. Dec 2021 [cit. 2022-01-06]. Available at: <https://fastapi.tiangolo.com/>.
 - [16] MULLOY, B. *Web API Design: Crafting Interfaces that Developers Love*. Apigee, 2012.
 - [17] O'REILLY, T. *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*. O'Reilly Media, 2005.
 - [18] PALLETS. *Flask Documentation*. Oct 2021 [cit. 2022-01-12]. Available at: <https://flask.palletsprojects.com/en/2.0.x/>.
 - [19] PREVATO, R. *BlackSheep*. Dec 2021 [cit. 2022-01-12]. Available at: <https://www.neoteroid.dev/blacksheep/>.
 - [20] Q SUCCESS DI. *Usage statistics and market share of Python 3 for websites*. Jan 2022 [cit. 2022-01-12]. Available at: <https://w3techs.com/technologies/details/pl-python/3>.
 - [21] SANIC COMMUNITY ORGANIZATION. *Sanic Framework*. Jan 2022 [cit. 2022-01-12]. Available at: <https://sanic.dev/en/guide/>.
 - [22] SMARTBEAR SOFTWARE. *OpenAPI Specification*. Feb 2020 [cit. 2022-01-09]. Available at: <https://swagger.io/specification/>.
 - [23] SMARTBEAR SOFTWARE. *Swagger UI*. Feb 2020 [cit. 2022-01-09]. Available at: <https://swagger.io/tools/swagger-ui/>.
 - [24] TECEMPOWER. *TechEmpower framework benchmarks*. Feb 2021 [cit. 2022-01-12]. Available at: <https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=composite&l=zijzen-sf>.
 - [25] VIKTORIN, J., KROBOT, P., KUČERA, J., HUTÁK, L., ŠURÁŇ, J. et al. *DCPro Protector manual*. Aug 2021. Internal DDoS protector documentation. Available at: <http://cisticka-devel.liberouter.org/gitlab-artifacter/build/man/html/index.html?branch=master&job=manual>.
 - [26] VRANÝ, J. *GitHub - CESNET/exafs*. Aug 2021 [cit. 2022-01-09]. Available at: <https://github.com/CESNET/exafs>.
 - [27] WICKMAN, B. and KHALID, Y. U. *Pex Documentation*. Jun 2015 [cit. 2022-04-04]. Available at: <https://pex.readthedocs.io/en/v2.1.75-public-signed/index.html>.

List of Acronyms

DDoS Distributed Denial of Service

DoS Denial of Service

UI user interface

GUI graphical user interface

REST Representational State Transfer

API Application Programming Interface

CRUD create, read, update, delete

ASGI asynchronous server gateway interface

RTBH Remotely Triggered Black Hole

CLI Command-Line Interface

URL Uniform Resource Locator

Appendix A

Contents of the Attached CD

Following directories and files can be found on the CD:

- Directory `src` – A directory containing all project source files.
- Directory `text` – A directory containing \LaTeX source files for this thesis.
- File `restapi.pyz` – A packaged version of the **REST API**.
- File `docker.tar` – A packed Docker image with running instances of the **REST API** and the ExaFS **GUI**.
- File `README.md` – A file describing the project and installation instructions.
- File `xmanja00.pdf` – An electronic version of this thesis.