

RADBOD UNIVERSITY NIJMEGEN



FACULTY OF SCIENCE

---

# Intrusion Detection System for 5G Core Systems

---

THESIS BSc COMPUTING SCIENCE

*Author:*  
Taha HAMMOUCHI

*Supervisor:*  
dr. D. RUPPRECHT

*Second reader:*  
dr. K.S. KOHLS

September 2023

## Abstract

The arrival of 5G networks revolutionized the way our society makes use of mobile communication by introducing various use cases. Like the use of 5G-powered robots to perform brain surgery at a distance, or 5G-powered military hardware on the battlefield. These use cases are made possible by changes in the architecture of the core network, such as the use of a service-based architecture. Meaning that the architecture of the 5G core network is based on entities, also called network functions, that exchange services with each other.

In the mentioned use cases, safety is of the utmost importance for the users of the network. For example, if a 5G-powered robot is used to perform brain surgery and the robot malfunctions, the possible damage to the patient's brain can be irreversible. As seen from the example, the use of 5G networks brings various security requirements and responsibilities, such as attack detection and prevention.

In this thesis, we identify unified data management, or UDM, as the most important asset of the 5G core network. The UDM is a network function in the 5G core network that is responsible for the registration of users in the network. Our contribution consists of performing attack detection against a denial-of-service attack on the UDM and discussing the deployment of an intrusion detection system in 5G core networks.

There are two ways to perform a denial-of-service attack against the UDM. The first way is an external attack by communicating with the UDM directly from outside the 5GC. The second way is to repeat the re-registration procedure from the UE to the UDM intensively over a short amount of time. We implemented and tested the first way in four different stages of development and theoretically discussed the second way.

Among other practical experiments, we suggested the practical execution of the second way of attack for future work. This thesis concludes that several practical experiments are needed to be able to determine the right deployment strategy for an IDS in the 5GC.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation . . . . .	4
1.2	Previous work . . . . .	4
1.3	Contribution . . . . .	6
1.4	Outline . . . . .	8
<b>2</b>	<b>Preliminaries</b>	<b>9</b>
2.1	5G Networks . . . . .	9
2.2	5G Core Network . . . . .	10
2.3	Intrusion Detection Systems . . . . .	11
2.4	STRIDE Methodology . . . . .	13
<b>3</b>	<b>Methodology</b>	<b>14</b>
3.1	Technical Setup . . . . .	14
3.2	Development . . . . .	15
3.2.1	Attack Framework . . . . .	15
3.2.2	Detection Logic . . . . .	18
3.2.3	Defense Framework . . . . .	19
3.3	Simulation . . . . .	21
<b>4</b>	<b>Testing and Results</b>	<b>22</b>
4.1	Visualization . . . . .	23
4.1.1	Stage 1: Experiments and results . . . . .	23
4.1.2	Discussion . . . . .	24
4.2	Attack detection by examining recorded traffic . . . . .	25
4.2.1	Stage 2: Experiments and results . . . . .	25
4.2.2	Discussion . . . . .	26
4.3	Attack detection by examining real-time traffic . . . . .	27
4.3.1	Stage 3: Experiments and results . . . . .	27
4.3.2	Discussion . . . . .	29
4.4	Attack detection and prevention in real-time . . . . .	30
4.4.1	Stage 4: Experiments and results . . . . .	30
4.4.2	Discussion . . . . .	31
4.5	Conclusion . . . . .	31
<b>5</b>	<b>Practical Discussion</b>	<b>32</b>
5.1	The First Stage . . . . .	32
5.2	The Second Stage . . . . .	34
5.3	The Third Stage . . . . .	34
5.4	The Final Stage . . . . .	35
<b>6</b>	<b>Theoretical Discussion</b>	<b>39</b>
6.1	Service Communication Proxy . . . . .	39
6.2	IDS deployment for the UDM . . . . .	41
6.2.1	IDS deployment for the UDM: First attack scenario . . . . .	42
6.2.2	IDS deployment for the UDM: Second attack scenario . . . . .	43
6.3	IDS deployment for the 5GC . . . . .	46
<b>7</b>	<b>Conclusion</b>	<b>49</b>
<b>8</b>	<b>Future Work</b>	<b>51</b>

<b>9</b>	<b>Appendix</b>	<b>53</b>
9.1	.env file open5Gs . . . . .	53
9.2	Source Code: attack_udm_original.py . . . . .	54
9.3	Source Code: attack_udm.py . . . . .	55
9.4	Source Code: not_attack_udm_original.py . . . . .	56
9.5	Source Code: not_attack_udm.py . . . . .	57
9.6	Source Code: request_udm.sh . . . . .	57
9.7	Flowchart for Defense Script . . . . .	58
9.8	Source Code: udm_testing.py . . . . .	59
9.9	Network Capture of traffic between the UDM and Malicious UE. . . . .	60
9.10	Technical specifications: virtual machine CoreNetwork. . . . .	61
9.11	Technical specifications: virtual machine AttackVM. . . . .	62
9.12	Network Settings for CoreNetwork and AttackVM. . . . .	63
9.13	SUCIs of UEs registered in the Web UI in open5Gs. . . . .	63
9.14	Source Code: netfilter_queue_intercept.py . . . . .	64
9.15	Source Code: plotting_udm_requests.ipynb . . . . .	68
9.16	Source Code: detecting_udm_attacks.ipynb . . . . .	70
9.17	Source Code: scapy_packet_handling.ipynb . . . . .	74
9.18	Source Code: udm_testing_corevm.py . . . . .	77
9.19	Source Code: realtime_capture.ipynb . . . . .	78

# 1 Introduction

In this chapter, we start by motivating our thesis. Next, we will present some previous work that has been done on 5G security, which will give us enough knowledge to discuss our contribution to the scientific community. In the third section, we will also state our main and sub-research questions. In the last section, we will outline the remainder of the thesis.

## 1.1 Motivation

With two-thirds of the world's population being mobile subscribers, we live in an era where reliable digital connectivity is the cornerstone of our society [1]. This many users of mobile networks causes a lot of security responsibilities. Such responsibilities include network security and endpoint security [2, 3]. Network security is preventing attacks, such as spoofing and eavesdropping attacks, on networks where mobile devices are connected [32, 33]. Endpoint security protects mobile devices, such as laptops and smartphones, from cyberattacks [2, 3].

The release of 5G brought some practical changes to the usage of mobile communication. An example of such a change is the use of a 5G-powered robot to perform remote brain surgery [4]. Such a revolutionary change is possible because of the ultra-high bandwidth, ultra-low latency and ultra-reliability that come with 5G technology [5]. Additionally, it has been mentioned that 5G technology enhances security and privacy as compared to its predecessors in mobile communication technology [6].

Considering the rising number of users of 5G and its various use cases, we can state that 5G brings enormous security concerns, which can cause catastrophic consequences if those concerns are not addressed properly [4].

In this thesis, we explore ways to deploy an intrusion detection system (IDS) in the 5G core network. We will also present a way to defend the UDM against an attempt at a denial-of-service attack. We will elaborate more on our contribution in section 1.3, but first, we will talk about the previous work that has been done on 5G security.

## 1.2 Previous work

To be able to understand the place that our contribution holds in the field of 5G security, we need to explain some previous work related to this topic. First, we will discuss a part of the 5G System Security Analysis done by Holtrup et al [6]. Second, we will focus on the Systematic Analysis of 5G Networks by Tang et al[4]. Last, we will give some additional pointers to other related work.

Holtrup et al., present a systematic risk analysis of standalone and non-standalone 5G networks following the STRIDE methodology. The authors define possible threat scenarios and derive a risk matrix that shows the likelihood and impact of those scenarios. At the end, the authors discuss possible mitigations and security controls [6].

The STRIDE methodology is a threat modeling methodology, and it stands for spoofing, tampering, repudiation, information disclosure, and elevation of privileges [7]. Standalone 5G networks are 5G networks that are deployed with their own core using a service-based architecture; non-standalone 5G networks are deployed based on the architecture of 4G core networks [8]. We would like to state that in this thesis, we will solely focus on standalone 5G networks (SA). In section 1.3, we will dive deeper into the work of Holtrup et al. regarding the defined threat scenarios that affect the service availability and network performance of 5G networks [6].

Tang et al., focus on the security of 5G core networks. The authors also give a review of existing 5G security results, which are divided into four categories: 5G general security, 5G signaling security, 5G authentication procedures security, and 5G network slicing and Software Defined Networks (SDN) usage security [4]. Further, the authors provide an overview of the main security-related features of 5G networks based on several 3GPP technical specifications [4].

Tian et al., provide an overview of the 5G security architecture. The authors also categorize the security features into four categories: Network access security, network domain security, user domain security and service-based architecture domain security [5]. Figure 1 provides an overall view of the systematic analysis of the 5G Core network security [4].

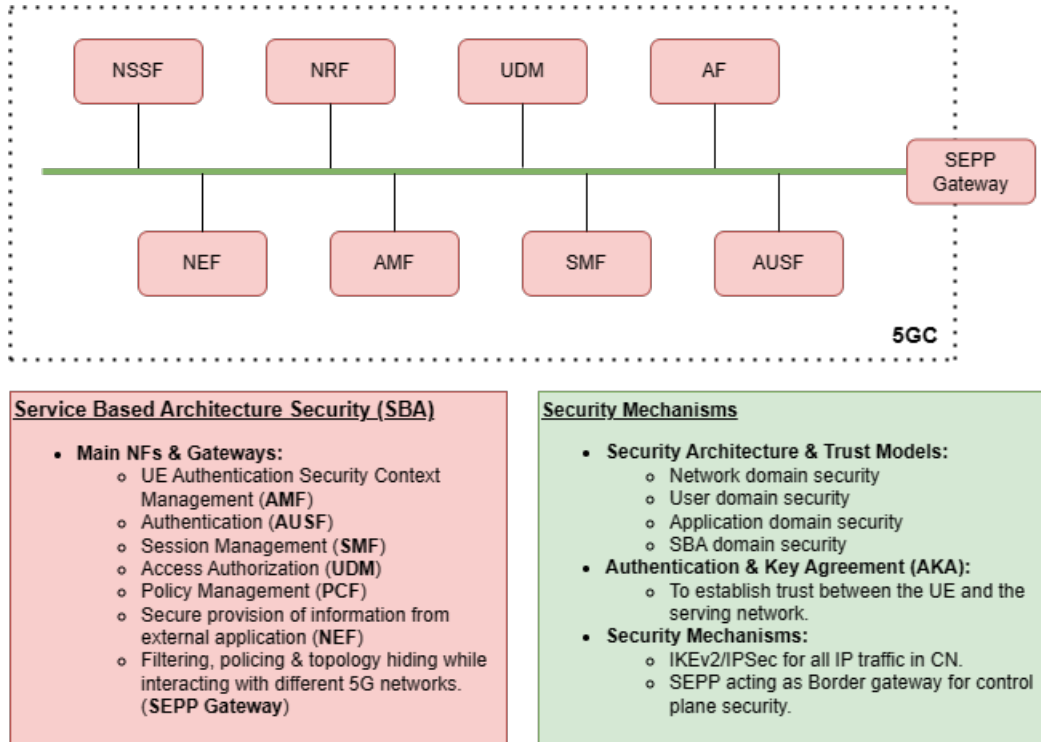


Figure 1: Overall view for the systematic analysis of the 5G core network security. (credit: Tang Q. [4])

Lastly, Tian et al., state that existing research on 5G security can be divided into two categories: physical layer security and logical layer security. Physical layer security provides security to the physical layer of the network in the form of encryption, cryptographic key distribution management, and authentication. Logical layer security provides security to the logical layer of the network in the form of securing Network Function Virtualization (NFV), security against virtualization threats, security against hypervisor hijacking, and so on [5].

### 1.3 Contribution

In this section, we will show how our work contributes to the scientific community. We will point out knowledge gaps that we derived from previous work and how we will close those gaps with our work. Then we will state the assumption on which our thesis is based. Then, we will discuss our contribution in the form of our research question and sub-questions. Last, we will show concrete steps on how we will answer our research question and sub-questions.

As stated in section 1.2, here we will dive deeper into the threat scenarios that affect the service availability and network performance of 5G networks. There are five assets to be protected in 5G networks: user identity and location, service availability, data integrity, data confidentiality and network performance [6]. In this thesis, we will focus on service availability and network performance. Holtrup et al., also state a few scenarios where the service availability of a 5G network is threatened. The first one is the physical or logical jamming of a legitimate or fake gNB. The second scenario is the exploitation of a software vulnerability in a gNB or network function. The third scenario is physical sabotage of the gNB or its antennas [6]. The gNB is the component that is responsible for the implementation of the radio interface with the user equipment (UE) of the 5G network [6].

Holtrup et al., states that the UDM is specifically exposed to certain threat scenarios: UDM database theft, software vulnerability in the UDM, and the lifting of keys that are used for link protection between network functions [6]. But, Holtrup et al., did not mention that the UDM is also vulnerable to denial-of-service attacks, which affect the service availability and network performance of the UDM and the 5G standalone network. This is exactly the assumption that we make in our thesis: to protect the service availability and network performance of a 5G network, we need to protect the service availability of the UDM. Especially because the UDM can be communicated with directly without the involvement of another network function, which we will show later in this thesis.

To further substantiate our contribution, it has been stated that there has been no scientific work done on the defensive aspects and opportunities of 5G networks [4]. Because of the complexity of the technology stacks in 5G networks, it is crucial to monitor and maintain visibility in the networks to be able to detect and mitigate attacks [4]. Therefore, well-known security mediums are needed to be proactively deployed by operators in 5G networks, and an intrusion detection system is such a tool [4]. It is also stated that the research on 5G security should be approached with the approach of developing a security solution as part of the 5G architecture instead of providing a patch to the architecture [5]. This way, the security of 5G networks is guaranteed, and the number of network attacks is greatly reduced [5].

That is why in this thesis we will contribute to the discussion about the deployment of an IDS in a 5G Core network by exploring different ways and scenarios to deploy an IDS in a 5G Core network. This way, we will provide a foundation for automated threat detection and response that can be deployed as part of the 5G architecture, which is lacking in 5G security. Most security solutions that exist depend on manual assistance, which can be difficult to provide due to the size and complexity of 5G networks [5].

Combining the work that has already been done and the suggestions for future work on 5G security, we are now able to derive our research questions and sub-questions.

**Research Question:**

- How can we upgrade the 5G core network with an intrusion detection system?

To be able to answer the main research question, we need to answer the following sub-research questions:

- **Sub-Research Question 1:**

- What is the most important asset in the 5G Core network?

- **Sub-Research Question 2:**

- What is the best way to defend this asset using an intrusion detection system?

- **Sub-Research Question 3:**

- What practical consideration do you need to keep in mind when deploying an intrusion detection system?

So, our contribution consists of the answers for our research question and sub-questions. To be able to answer our research question and sub-questions, we have to perform the following practical steps:

1. Attack detection against a denial-of-service attack on the UDM by analyzing recorded network traffic.
2. Attack detection against a denial-of-service attack of the UDM by analysing network traffic in real-time.
3. Attack detection against a denial-of-service attack of the UDM by analysing network traffic in real-time and blocking malicious traffic.
4. Discuss the deployment scenarios of an intrusion detection system in the UDM.
5. Discuss the deployment scenarios of an intrusion detection system in 5G core networks.

The first three steps are discussed in chapter 4. The remaining steps are discussed in chapter 7. In the next section, we will present the remainder of our thesis.

## 1.4 Outline

The next chapter is dedicated to providing background knowledge on the technical and theoretical aspects of this thesis. In chapter 3, we will present our setup and methodology. In chapter 4, we will show the testing and results of our implementation. In chapter 5, we will provide a practical discussion regarding our implementations, and in chapter 6, we will discuss the deployment scenarios of an IDS in the 5G Core network. Afterward, we will provide a conclusion and finalize our thesis with suggestions for future work.

## 2 Preliminaries

This chapter will provide a technical background for our thesis. In the next section, we will discuss 5G networks in general. In the second section, we will focus on the 5G core networks. In the third section, we will discuss intrusion detection systems, and in the last section, we will discuss the STRIDE methodology for threat classification.

### 2.1 5G Networks

In this section, we will introduce 5G networks in general. We will discuss some primary differences with 4G networks. Also, we will discuss the various use cases of 5G networks.

It is believed that 5G networks have a key role in enabling the infrastructure for digital transformation [4]. The introduced use cases of 5G networks include machine-to-machine (M2M) communications in all their variants, like device-to-device (D2D) communications [6]. Support for highly mobile devices, for example, vehicle-to-everything (V2X), and support for stationary devices on the Internet of Things (IoT) are also required [6]. Because of the various use cases that emerge from 5G networks, numerous security promises can be made regarding the functionality of mission-critical applications [4]. That is also why, in the 5G standards, security has a very high priority [4].

Another difference compared to the 4G architecture is that the 5G core architecture is a service-based architecture, which uses service-based interfaces between network functions [6]. Service-based interfaces are used for communication between network functions and are implemented over HTTP/2 over TLS over TCP/IP [6]. The adoption of the service-based architecture in 5G core networks allows great flexibility and expandability of the core network by being able to plug functionalities into the network without the need to change the existing architecture [4]. The service-based architecture also allows for the exploration of software-defined networks (SDN) and network function virtualization (NFV), which enable network slicing [4]. Network slicing is a technology that provides on-demand and dedicated network access and quality of service (QoS) to customers [4]. The service-based architecture and interface also make the 5G core network more accessible for the research community, which results in better research and development of 5G technologies. As shown in chapters 3 and 4, these technologies made it possible for us to conduct our practical research with the technical setup discussed in section 3.1. For further technical specifications for 5G networks, we refer to the 3rd Generation Partnership Project (3GPP) specifications [9].

In the next section, we will present 5G networks in more detail, with a focus on the 5G Core network. We will also go over some network functions and briefly explain their role in the 5G core network. We will leave the explanation of how network functions communicate with each other to chapter 6, as it sets the foundation for our discussion on the deployment approaches of an IDS in the 5G Core network.

## 2.2 5G Core Network

Before explaining how the 5G Core network is structured, we will first provide an abstract view of the end-to-end communication between the user equipment and an endpoint: another user equipment (UE) or an application server (AS) [10]. As shown in Figure 2, the communication goes via the access network to the core network and then via the data network to the endpoint [10]. Our focus in this thesis lies on the core network.

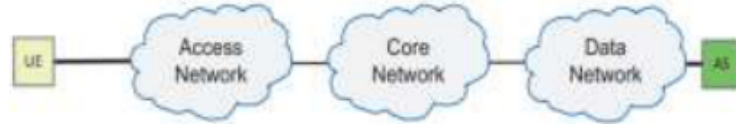


Figure 2: End-to-end 5G Architecture (credit: Gheyath Mustafa Zebari [10])

The functional components of the 5G Core network are organized into two components: The control plane (CP) and the user plane (UP) [18, 19]. In this thesis, when we mention the 5G Core network, we mean the Control Plane specifically in the 5G Core network. Figure 3 shows the standard architecture of the 5G Core network.

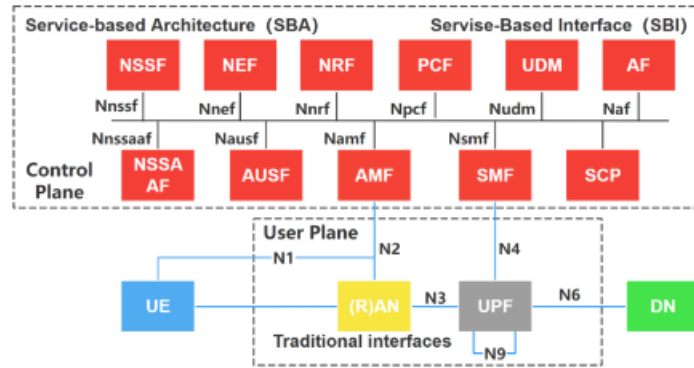


Figure 3: 5G core network standard architecture (credit: Zeng Z [11])

As we see in Figure 3, the control plane consists of smaller functional components, namely network functions. In the control plane, the network functions are unified via the service-based architecture [11]. Each network function uses its own service-based interface to communicate with other network functions, but each network function is decoupled from other network functions in terms of business functions [11].

Last, we will discuss some network functions in terms of functionality. The Access and Mobility Management function (AMF) is the endpoint for messages from the Radio Access Network (RAN) [6]. The AMF also manages the registration procedure of the UE with the Unified Data Management function (UDM) [4, 8]. Furthermore, the AMF is also responsible for authenticating UEs by obtaining authentication vectors from the Authentication Service Function (AUSF) [4, 8]. The AUSF obtains the authentication vectors from the UDM [4, 8].

The AMF is also responsible for managing the security contexts of the UE, and it also acts as a proxy between the UE and other network functions [4]. Next to its role in the authentication and registration procedures of the UE, the AUSF is also responsible for providing security parameters to protect the integrity of the UE update procedure [4].

The functionality of the UDM includes, but is not limited to, managing data for access authorization, data network profiles and UE registration [4]. The UDM leans on the AMF, AUSF and SMF for its functionality because the relevant data for the UDM becomes available through these network functions [4]. The UDM also has the functionality to decrypt the subscription concealed identifier (SUCI) to reveal the subscription permanent identifier (SUPI) of the subscriber [6, 4]. The SUPI is a generic name for the mobile subscriber's identity in 5G networks. The SUPI is a unique identifier of the user's identity in the 5G network [6]. The SUCI is a protected version of the SUPI. The SUCI is a SUPI that is encrypted using asymmetric cryptography [6].

Lastly, as we see in Figure 3, the service communication proxy is also part of the control plane, although the SCP is not a network function. The SCP is deployed to provide routing control and resilience, among other functionalities [4]. We will discuss the SCP in more detail in chapter 6.

Further technical specifications for all the network functions are defined in the 3GPP specifications [9].

## 2.3 Intrusion Detection Systems

In chapter 6, we will discuss ways to upgrade the service-based architecture of the 5G Core network with an intrusion detection system (IDS). Therefore, in this section, we will explain what an IDS is and the existing approaches to intrusion detection.

An IDS is a key security measure of each network infrastructure; IDSs defend such a network by detecting and blocking attack-related network traffic. We also distinguish between software-based IDSs and hardware-based IDSs [12]. Because 5G core networks use SDN and SBA technologies, we will focus only on software-defined IDSs. It is also stated that the implementation of software-based IDSs could cause delays in the network, especially when exposed to high-speed traffic [12]. This is something we will discuss more in chapter 6, especially because the use cases of 5G networks are almost intolerant to network delay, but at the same time they have a high requirement for security.

Moving on to the known approaches towards intrusion detection, we distinguish four major approaches: misuse-based, anomaly-based, policy-based and hybrid intrusion detection [13]. In the remainder of this section, we will go briefly over each approach.

A misuse-based approach to intrusion detection detects attacks by comparing incoming traffic to previously known attacks. Known attacks or patterns of known attacks are stored as signatures in the database of the system in which the IDS is deployed, hence the name signature-based IDS [14, 13]. Signature-based IDSs detect known attacks with a relatively low false-positive rate, but this approach fails when it comes to zero-day attacks or attacks that are unknown to the target system [14, 13]. This approach also fails if the attacker manages to change the attack vector in such a way that the IDS will not recognize it. Because the signature stored in the target system is not general enough to match all the attack vectors of a specific attack scenario [13]. This makes the maintenance of a signature-based IDS tedious because the deployer needs to make sure that the signature database is up-to-date with novel attacks and that the signatures are general enough to cover their variants [13].

An anomaly-based approach to intrusion detection is based on the modeling of network traffic using machine learning techniques [14, 13]. In the training phase, the IDS models normal traffic by learning patterns in traffic behavior. In the test phase, or deployment, the IDS labels deviation from the learned traffic as an anomaly or intrusion, hence the name anomaly-based IDS [14, 13]. Anomaly-based IDSs support the detection of attacks that can not be detected by signature-based IDSs, namely unknown and novel attacks [14, 13].

The first drawback of this approach is that it is quite challenging to implement for real-time usage because of the fast-evolving network traffic behavior combined with the limited amount of computational resources [14]. The second drawback is the risk of overfitting because of the high-dimensional network traffic data and the complex model of an IDS. This can lead to high false-positive rates [14, 13].

Another interesting approach is policy-based intrusion detection. The idea behind this approach is to cover the biggest limitations of signature-based and anomaly-based IDSs by solving two major issues: Detection of novel attacks and the right classification of normal unseen behavior [13]. Policy-based IDSs decide if a certain behavior is malicious or not by imposing a set of rules [13]. Although this approach seems well-rounded, it has some challenges. First, a security specialist has to design effective and consistent policies. These policies should also be logically correct throughout the whole system to avoid any interpolicy or intrapolicy conflict. Another challenge is to implement the rules in a proper sequential order; otherwise, the policy can cause a deadlock or feedback loop situation [13].

The last approach is hybrid intrusion detection. This approach fuses the previous approaches into a single IDS. This approach gives a better performance by combining the strengths and complementing the weaknesses of the previous techniques [13]. But, in the situation where one technique considers a scenario to be an attack scenario and another technique does not, it is still challenging to resolve this conflict [13].

In this thesis, we are implementing attack detection against a denial-of-service attack on the UDM following a misuse-based approach. Our attack detection system analyzes network traffic in a simulated real-time setting and blocks malicious traffic. Malicious traffic is detected by comparing a sniffed network packet with network packets that were sniffed earlier and stored in our system.

After discussing IDSs, in the next section we will briefly explain what the STRIDE methodology is about and explain its relevance for our thesis.

## 2.4 STRIDE Methodology

With the advancement of technology, cyber threats also evolve. To be able to effectively classify the evolving threats, we use the STRIDE classification. The STRIDE methodology for threat classification was originally developed by Microsoft with the goal of describing and categorizing cyber threats according to the attacker's goals [6, 1].

The threats are categorized based on the goals of the attacker: spoofing of user or device identity, tampering, repudiation, information disclosure, denial of service and elevation of privilege [6]. In this thesis, we are focusing on the attacker's goal of denial of service. Each component in the system on which we are performing a threat assessment, 5G Core Systems, is exposed to a certain category of threats [6].

5G systems can be divided into the following components: external entities or interactors; processes; data and key storage; data flow; and devices. Except for external entities and interactors, all the components of 5G networks are exposed to denial-of-service attacks [6]. Attackers perform a denial-of-service attack by making a server on the target system temporarily unusable, thereby depriving valid users of service on the target system [6]. The appropriate security controls against these attacks are availability and redundancy, so the improvement of reliability and availability of the target system [6]. However, in our case, we are only focusing on the data flow component because an attack on this component implies an attack on the network level of the target system. So, an IDS is the right tool to defend against these attacks [15].

### 3 Methodology

] In this chapter, we will focus on the methodology and technical setup used for our thesis. In the first section, we will present our technical setup. In the second section, we will focus on our implementation. We will present our detection logic, defense framework, and attack framework. And in the last section, we will provide instructions on how to run an attack-defense simulation using the discussed technical setup and frameworks.

As discussed in section 1.3, we have three main practical contributions and one theoretical contribution. The first two practical contributions build up to the third one, namely, attack detection against a denial-of-service attack on the UDM, by analyzing network traffic in real-time and blocking malicious traffic. Meaning, we are planning to attack the UDM by executing our attack script that generates valid and invalid registration requests to the UDM. As we will discuss in section 3.2.1, invalid registration requests are sent to the UDM to simulate a denial-of-service attack against the UDM. So, the attack vector to the UDM is a denial-of-service attack, and the general attack vector is a denial-of-service attack to the 5G core network via the UDM. The earlier two practical stages will be discussed in chapter 5. In chapter 4, we will present the results of all the practical achievements of our thesis.

#### 3.1 Technical Setup

In this section, we will discuss the technology stack used for our thesis. We will list all the technologies and explain the added value of each technology to our thesis. Second, we will show how we combined those technologies into a working setup. We will leave the explanation of the logic behind the scripts for the next section.

To list our technologies, we will use a top-down approach. Meaning, the order of the listed technologies is in the same order if one needs to replicate the setup. The technologies used for our setup are listed below.

1. Windows 11 [16] : This is the operating system that we used on which we hosted our practical setup. In this thesis, we will call this the Host-Machine.
2. Oracle VM VirtualBox 6.1 [17] : Virtualbox is a technology that makes it possible to run virtual machines on a particular host machine [17]. In the Host-Machine, we used the Oracle VirtualBox to run our virtual machines.
3. Ubuntu 64-bit [18] : Ubuntu is a Linux operating system from the Debian family [18]. We used Ubuntu as an operating system for our virtual machines. We deployed two virtual machines: The CoreNetwork and the AttackVM. The CoreNetwork serves as host for our 5G Core network. The AttackVM serves as a launchpad for our simulated attack and defense. The technical specifications of the CoreNetwork can be found in Appendix 9.10, and the specification of the AttackVM can be found in Appendix 9.11. Network settings that make it possible for the two virtual machines to communicate are presented in Appendix 9.12.
4. Docker [19] : Docker is a tool that makes application deployment possible regardless of the system environment [19]. We use Docker to deploy open5Gs.
5. Open5Gs [20] : Open5Gs is an open source implementation of the 5G Core network [20]. Open5Gs allows us to set up our own 5G Core network on which we conduct experiments.

6. Iptables [21] : Iptables is an administration tool for manipulating firewall rules on Linux systems [21]. We use Iptables to add firewall rules to our AttackVM, such that traffic gets routed to our defense script. More on this in the next section.

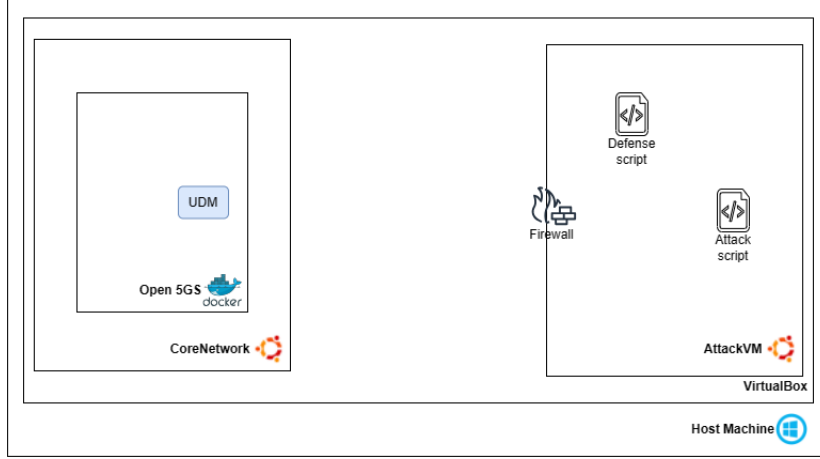


Figure 4: The final setup of our experimental environment.

Figure 4 gives a visual representation of our setup. In the next section, we will discuss the development of the scripts used to simulate a denial-of-service attack on the UDM and a defense against such an attack.

## 3.2 Development

Our development comprises two parts: Attack and defense. We will explain each part in separate subsections. In the last subsection, we will provide a concrete guide for running our implemented attack and defense simulation.

### 3.2.1 Attack Framework

Before explaining how the attack and non-attack scripts work, we will describe the requirements for our attack and visualize our attack framework. Later in this subsection, we will also explain what we mean by attack and non-attack scenarios.

As mentioned earlier, the attack consists of the attack script communicating with the UDM by sending valid and invalid registration requests. More precisely, as shown in Figure 4, the attack script in the AttackVM should be able to send valid and invalid requests to the UDM that is running in an open5Gs instance that is hosted in the CoreNetwork. For a successful execution of the attack, the virtual machine CoreNetwork should adhere to the specifications shown in Appendix 9.10. The virtual machine AttackVM should adhere to the specifications shown in Appendix 9.11. And, the network settings for the CoreNetwork and AttackVM should adhere to the configuration shown in Appendix 9.12. More detailed instructions for the execution of the attack are discussed in section 3.3.

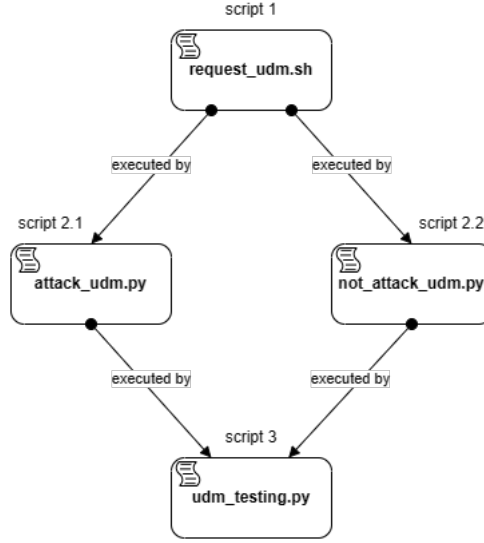


Figure 5: Scripts of the attack framework visualized in a hierarchical orders.

The visualization of our attack framework is shown in Figure 5. To run our simulation, we run the ‘udm\_testing.py’ script, also called script 3. As shown in Figure 5, this script is implemented using the ‘attack\_udm.py’ and ‘not\_attack\_udm.py’ scripts, also called scripts 2.1 and 2.2 respectively. Those two scripts are implemented using the ‘request\_udm.sh’ script, script 1. In the remainder of this subsection, we will go over each script describing the implementation.

- **Script 1: ‘request\_udm.sh’**

This shell script takes as input the IP address of the UDM, the port number and a SUCI-id. The script uses these parameters to construct an HTTP/2 request to the UDM, expecting an authentication vector as a response from the UDM if the provided SUCI is already registered with the UDM. If the provided SUCI is not stored in the database, a 404 return code is expected as a response from the UDM. Because, as discussed in chapter 2, the SUCI is a concealed user identifier that is stored in the UDM.

Before running the script, we manually added a UE instance to our running open5Gs instance using the Web UI of open5Gs, this is shown in Appendix 9.17. Hereby, the UDM is expected to behave as described in the 3GPP specifications [9].

We simulate an attack scenario by executing script-1 often in succession, provided with a SUCI that is not stored in the UDM. This approach ensures that the UDM is not overwhelmed with requests that are not adding a functional value to the UE, such as providing an authentication vector. This is the basis of a denial-of-service attack. A non-attack scenario is simulated by executing script-1 provided with a SUCI that is manually stored in the UDM.

We run script 1 with the following command:

```
‘./request_udm.sh 198.168.56.101 7777 suci-0-001-01-0000-0-0-1234567895’
```

The source code for script 1 can be found in Appendix 9.6.

- **Script 2.1: ‘attack\_udm.py’ and Script 2.2: ‘not\_attack\_udm.py’**

To run script 1 many times successively, we need to form automation. That brings us to these Python scripts. Both scripts take as input the number of requests that need to be generated by script 1. Both scripts execute script 1 in a for-loop according to the amount given as input parameters.

The input parameters for script 1 are already known and constant because we are working with our own technical setup as described in the previous section. So, the input parameters for script 1 are hard-coded in both Python scripts. However, for script 2.1 the SUCI-id does not need to be hard-coded, at each iteration, script 2.1 generates an invalid SUCI that is used for the execution of script 1.

We run script 2.1 using the following command:

```
‘./attack_udm.py <amount_of_requests>’
```

We run script 2.2 using the following command:

```
‘./not_attack_udm.py <amount_of_requests>’
```

The source code for script 2.1 can be found in Appendix 9.3, and the source code for script 2.2 can be found in Appendix 9.5.

- **Script-3: ‘udm\_testing.py’**

In a real-world scenario, valid and invalid authentication requests are mixed together. To simulate this scenario, we implemented script 3 that allows for the execution of the attack and non-attack scripts a specified number of times. In other words, script 3 executes scripts 2.1 and 2.2 a specified number of times.

Script 3 takes two input parameters: the number of valid requests and the number of invalid requests. The first parameter suggests the number of times that script-2.2 should be executed by script-3. And the second parameter submits the number of times that script-2.1 should be executed by script-3. The main logic behind script 3 is that at each iteration, a pseudo-random choice is made between script 2.1 and script 2.2. This way, script 3 ensures that these executions occur by chance; the overall count of legitimate and invalid requests does not occur in sequence.

We run script 3 using the following command:

```
‘./udm_testing.py <amount_of_valid_requests><amount_of_invalid_requests>’
```

The source code for script 3 can be found in Appendix 9.8.

### 3.2.2 Detection Logic

In this section, we will provide a logical, non-technical explanation of the logic behind the defense we implemented, which is presented in the flowchart in Appendix 9.7.

Our detection logic is based on the fact that we have to decide as early as possible if the sniffed TCP packet should be dropped or not. We base our decision sequentially on the following four criteria:

1. Is the sniffed packet part of the intercore communication?
2. Is the sniffed packet an ingoing or outgoing packet?
3. Is the sniffed packet part of an earlier detected TCP conversation?
4. Is the sniffed packet part of a valid or invalid registration request to the UDM?

The first decision criterion checks if the sniffed packet is part of a network conversation between the UDM and another network function. If that is the case, we will not examine the packet further because the scope of our implemented defense lies in the communication between the UDM and a potential attacker not residing in the 5G core network. So, the packet will not be dropped in this case. Otherwise, the sniffed packet will be examined further based on the second criterion.

The second decision criterion checks if the source of the sniffed packet is the UDM or an IP address outside the 5G core network. If the source of the sniffed packet is the UDM, then the sniffed packet is treated as an outgoing packet; otherwise, the sniffed packet is treated as an incoming packet. In the case of an incoming packet, we check if the source IP address is already blacklisted in our system. If that is the case, we drop the sniffed packet. In the case of an outgoing packet, the sniffed packet will be examined further based on the third criterion.

The third decision criterion checks if the sniffed packet is part of a TCP conversation that is already detected by our script. We perform this check by looking at the following four data points in the sniffed packet: source IP, destination IP, source port, and destination port. If these data points are new to our system, the sniffed packet will be accepted, and the four data points with the payload will be stored in a data structure in our system. Otherwise, the sniffed packet will be examined further based on the fourth criterion.

The fourth decision criterion checks if the sniffed packet is part of a valid or non-valid registration request to the UDM. If the packet is part of a valid request, then the sniffed packet will be accepted. Otherwise, the sniffed packet will be dropped if the destination IP of the sniffed packet is already known as malicious in our system and a certain attack threshold has been exceeded from the same destination IP. As this is a non-technical elaboration of our detection logic, a more technical discussion, especially on the third and fourth criteria, is provided in section 5.4.

### 3.2.3 Defense Framework

In this subsection, we will discuss our defense framework. First, we will discuss our defense framework abstract. Second, we will dive into the logic of our defense script. Lastly, we will show how to run our defense framework.

The defense framework consists of our defense script and a set of IP table rules that route incoming packets from the UDM and outgoing packets to the UDM to our defense script. Our defense script examines the intercepted packet and decides if it should be forwarded to its destination or dropped. The Iptables rules that need to be added are presented later in this subsection.

A visualization of our defense framework is shown in Figure 6. As we can see in Figure 6, the numbers on the arrow indicate the order of communication when a request is sent from the attack script in the AttackVM to the UDM in the CoreNetwork. Below, we will enumerate the steps taken in the communication between the attack script and the UDM:

1. The attack script generates a request with the UDM as its destination. The generated request is intercepted by the firewall.
2. The firewall passes the request to the defense script.
3. If the request is not dropped by the defense script, it is sent to its destination: the UDM in the CoreNetwork.
4. The UDM generates a response with the attack script as its destination. The generated response is intercepted by the firewall in the AttackVM.
5. The firewall passes the request to the defense script.
6. If the response is not dropped by the defense script, it is sent to its destination: the UDM in the CoreNetwork.

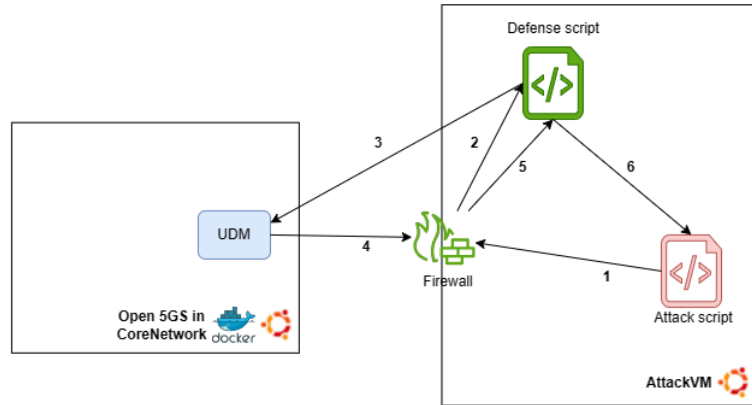


Figure 6: The defense framework in relation with our final setup. The defense framework is colored green, and the attack framework is colored red. The numbered arrows indicate the order of communication when a request is sent by the attack script.

The ability to manipulate network traffic in a simulated real-time setting is based on the use of two Python packets: Python Scapy and Python NetfilterQueue. Python Scapy is a library for network packet manipulation written in Python [22]. We use Scapy for deeper examination of intercepted network packets. Python NetfilterQueue is a library that examines network packets by connecting to the `libnetfilter_queue` Linux subsystem [23]. This makes sure that Python NetfilterQueue can drop, alter, mark or accept packets that are matched by an Iptables rule in Linux [23]. So, the Iptables rules we add to our AttackVM before running our defense script make sure that specific traffic gets redirected to our script. Then, we examine those packets using Python Scapy. And last, we accept or drop the intercepted packet using Python NetfilterQueue.

The decision to drop or forward a packet is made based on our decision flowchart, which is represented in Appendix 9.7. The flowchart is based on the fact that the network conversation between the UDM and the UE is a TCP conversation, this is discussed more in chapter 5. The flowchart is also designed keeping in mind that the decision to drop or accept a TCP packet should be made as soon as possible. The details and the idea behind our implementation are presented in chapter 5. The source code for the defense script is presented in Appendix 9.8.

Another point to mention is that the implementation of our defense framework is inspired by the fact that each network has its own limit on the number of invalid registration requests to the UDM it can handle. As discussed earlier, the tolerance for latency is dependent on the main use case of the network, so it is up to the user to decide how many invalid requests the network can take before it is considered an attempted denial-of-service attack. Our implementation and experimentation—more on this in chapter 4—enforce a ratio of one invalid request against a maximum of four invalid requests (1/5). In chapter 8, we make a suggestion for future work regarding the ratio definition in a network.

Last, we will present the steps for the execution of our defense script, the following steps need to be taken:

1. Add the following Iptables rules to the AttackVM
  - Rule 1:  
`'sudo iptables -I INPUT -s <UDM IP>-j NFQUEUE --queue-num <queue number used in defense script>'`
  - Rule 2:  
`'sudo iptables -I OUTPUT -d <UDM IP>-j NFQUEUE --queue-num <queue number used in defense script>'`
2. Run the defense script using the following command:  
`'sudo ./netfilter_queue_intercept.py'`

In the next subsection, we will concretize our methodology by showing exact steps that need to be taken to run an attack and defense simulation if provided with our setup.

### 3.3 Simulation

Before providing instructions on how to run an attack-defense simulation, we need to list our assumptions for the system settings.

Assumptions:

- A technical environment is set up as described in subsection 3.1. The operating system of the Host Machine is trivial.
- Python Scapy is installed on AttackVM.
- Python NetfilterQueue is installed on AttackVM.
- All the scripts discussed in this chapter are provided.

If the above assumptions hold, the instructions listed below can be followed:

- **Step 1:** Turn on the CoreNetwork virtual machine.
- **Step 2:** Run the open5Gs dockerized instance in the terminal of the CoreNetwork virtual machine.
- **Step 3:** Turn on the AttackVM virtual machine.
- **Step 4:** Run the defense framework as described in subsection 3.2.3 from the terminal of the AttackVM virtual machine.
- **Step 5:** Run the ‘udm\_testing.py’ script with the desired amount of valid and invalid requests to the UDM using the following command in the terminal of the AttackVM virtual machine:

– ‘./udm\_testing.py <amount\_of\_valid\_requests><amount\_of\_invalid\_requests>’

In the next chapter, we will manually test our environment and provide the generated results. Also, we will test our earlier phases of development and provide the generated results. Details and discussion on the earlier phases of development are provided in chapter 5.

## 4 Testing and Results

In this chapter, we will test our implementation and present our findings. We will conduct experiments on our implementation of the four stages of development.

The first stage focuses on visualizing valid and invalid authentication requests to the UDM. The main objective of the first stage is to get a realistic image of the behavior of the script presented in Appendix 9.18 and to examine the response of the UDM towards our valid and invalid requests. The implementation of the first stage is presented in Appendix 9.15. The second stage focuses on attack detection by examining recorded network traffic. The implementation of the second phase is presented in Appendix 9.16. The third stage focuses on attack detection by examining network traffic in a simulated real-time scenario. The implementation of the third phase is presented in Appendix 9.17. And the final phase focuses on attack detection and prevention by examining network traffic in a simulated real-time scenario. The implementation of the final phase is presented in Appendix 9.19.

The results of our experiments regarding each stage of development are presented in separate subsections. The technical details of each development stage are discussed in more detail in chapter 5.

Before presenting our findings, we want to reiterate how we named the main components of our setup:

- **CoreNetwork:**
  - Linux virtual machine where the open5Gs is deployed.
  - In our experiments, this is abbreviated as: C.N.
- **AttackVM:**
  - Linux virtual machine, from which we simulate an attack scenario.
  - In our experiments, this is abbreviated as: A.VM.
- **Host Machine:**
  - Windows Subsystem Linux, on which the virtual machines are running and from which we will run some attacks.
  - In our experiments, this is abbreviated as: H.M.

Our defense framework is deployed in the CoreNetwork virtual machine in the first three stages. In the last stage, however, the defense framework is deployed in the AttackVM due to technical reasons that are mentioned in the practical discussion. The deployment of the attack framework is shown in each experiment.

## 4.1 Visualization

In this section, we will test the implementation of the first stage of development: visualizing valid and non-valid requests to the UDM using recorded network traffic. The results are presented as plots, with the x-axis presenting the time offset and the y-axis presenting the validity of the requests. Table 1 shows more details about this experiment.

### 4.1.1 Stage 1: Experiments and results

Test No.	Attack Framework	Total requests	Valid requests	Invalid requests	Results
Test 1.1	C.N	100	50	50	Figure 7
Test 1.2	C.N	500	350	150	Figure 8

Table 1: Experiment stage 1

Notes:

- All the commands were executed from the CoreNetwork.
- Traffic capture for test 1: `udm_testing_stage-1_test-1.pcap` [24].
- Traffic capture for test 2: `udm_testing_stage-1_test-2.pcap` [24].
- The y-axis in Figures 7 and 8 represents the HTTP return codes of the requests. A return code of "200" suggests a response to a valid request, and a return code "404" suggests a response to an invalid request [9].
- Commands used:
  - Test 1.1: `./udm_testing_corevm.py 50 50'`
  - Test 1.2: `./udm_testing_corevm.py 350 150'`

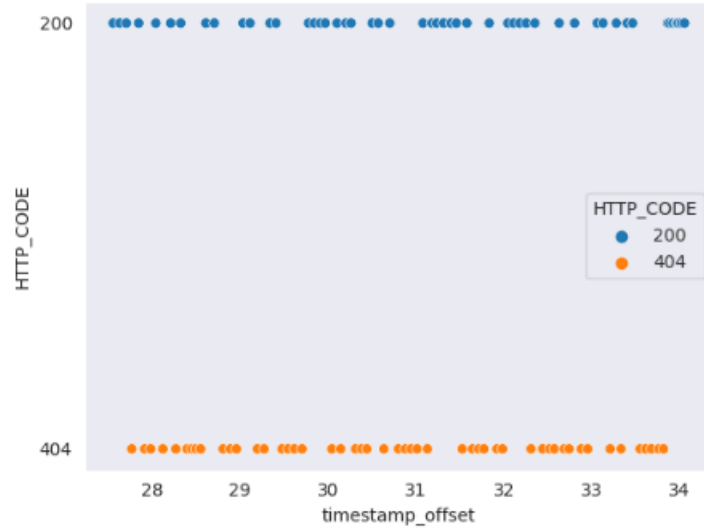


Figure 7: Results of Test 1.1

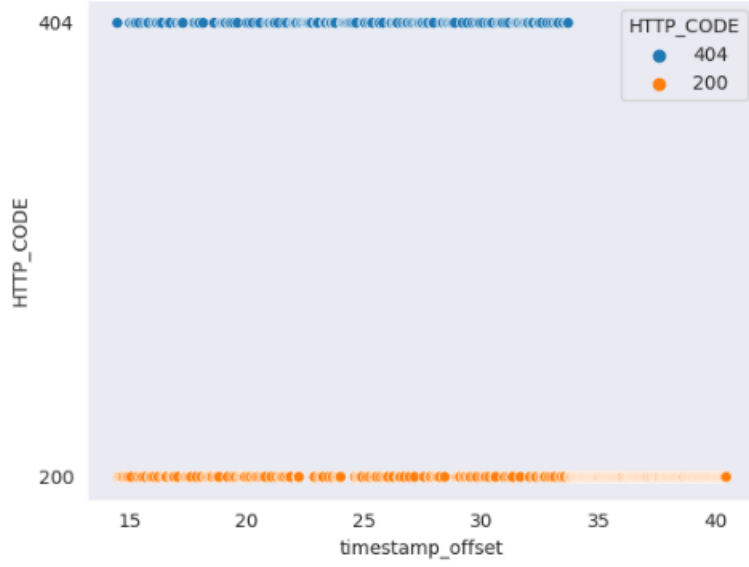


Figure 8: Results of Test 1.2

#### 4.1.2 Discussion

The results of the visualization presented in Figures 7 and 8 indeed show that the valid and non-valid requests are indeed executed in a pseudo-random order. This confirms that our attack script simulates a real-world scenario where valid and invalid registration requests are mixed together. Also, we can see that the UDM generates only two HTTP codes in its response to a valid or invalid registration request: HTTP code 200 and HTTP code 404. This behavior of the UDM is already defined in the 3GPP specifications [9]. But now we have proved that the UDM in our experimental environment behaves the way it is expected to.

## 4.2 Attack detection by examining recorded traffic

In this section, we will test the implementation of the second stage of development: attack detection for the UDM using recorded network traffic. The results are presented as plots, with the x-axis presenting the time offset of the malicious requests and the y-axis presenting the source of those requests. Table 2 shows more details about this experiment.

### 4.2.1 Stage 2: Experiments and results

Test No.	Attack Framework	Total requests	Valid requests	Invalid requests	Results
Test 2.1	C.N	1500	1199	301	Figure 9
Test 2.2	C.N, A.VM, H.M	1500	1050	450	Figure 10

Table 2: Experiment stage 2

Notes:

- Traffic capture for test 1: `udm_testing_stage-2_test-1.pcap` [24].
- Traffic capture for test 2: `udm_testing_stage-2_test-2.pcap` [24].
- The commands for test 2.1 were executed from the CoreNetwork.
- The commands for test 2.2 were executed from the CoreNetwork, AttackVM and Host Machine. This was done semi-simultaneously and manually.
- Commands used:
  - Test 2.1:
    - \* CoreNetwork: `./udm_testing_corevm.py 1199 301`
  - Test 2.2:
    - \* CoreNetwork: `./udm_testing_corevm.py 350 150`
    - \* AttackVM: `./udm_testing.py 350 150`
    - \* HostMachine: `./udm_testing.py 350 150`

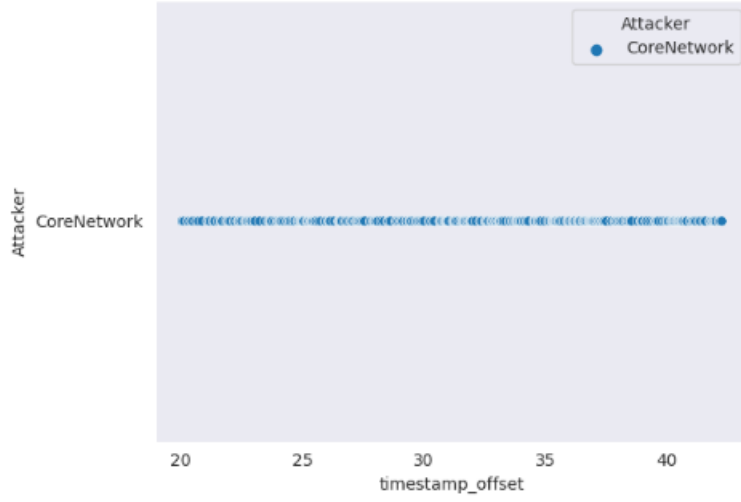


Figure 9: Results of Test 2.1

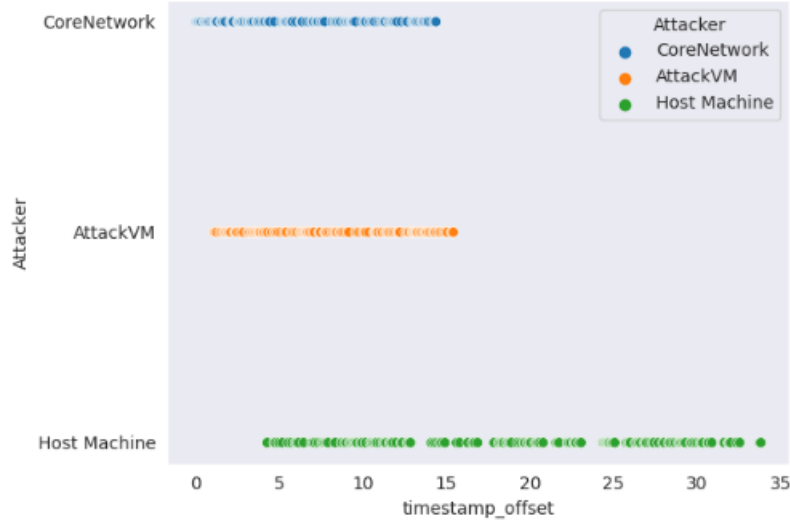


Figure 10: Results of Test 2.2

#### 4.2.2 Discussion

An important point to discuss is the advantage of analyzing recorded network traffic. This stage offers the opportunity to train deep machine learning algorithms with the recorded network traffic, acting as training data. This will result in the ability to perform attack detection with a low false-positive rate. As discussed in the chapter 2, attack detection with a low false-positive rate is one of the desired performance attributes of an IDS. Using deep machine learning for attack detection is a suggestion for future work, we will discuss this further in chapter 8.

### 4.3 Attack detection by examining real-time traffic

In this section, we will test the implementation of the third stage of development: attack detection for the UDM in a simulated real-time setting. The results are presented as statistics that show the values of some global variables used in our implementation. Table 3 shows more details about this experiment.

#### 4.3.1 Stage 3: Experiments and results

Test No.	Attack Framework	Total requests	Valid requests	Invalid requests	Results
Test 3.1	C.N	50	39	11	Figure 11
Test 3.2	C.N, A.VM, H.M	300	150	150	Figure 12
Test 3.3	C.N, A.VM, H.M	2100	1200	900	Figure 13
Test 3.4	C.N, A.VM, H.M	3600	2400	1200	Figure 14

Table 3: Experiment stage 3

Notes:

- The commands for test 3.1 were executed from the CoreNetwork.
- The commands for tests 3.2, 3.3 and 3.4 were executed from the CoreNetwork, AttackVM and Host Machine. This was done semi-simultaneously.
- Commands used:
  - Test 3.1:
    - \* CoreNetwork: './udm\_testing\_corevm.py 39 11'
  - Test 3.2:
    - \* CoreNetwork: './udm\_testing\_corevm.py 50 50'
    - \* AttackVM: './udm\_testing.py 50 50'
    - \* HostMachine: './udm\_testing.py 50 50'
  - Test 3.3:
    - \* CoreNetwork: './udm\_testing\_corevm.py 400 300'
    - \* AttackVM: './udm\_testing.py 400 300'
    - \* HostMachine: './udm\_testing.py 400 300'
  - Test 3.4:
    - \* CoreNetwork: './udm\_testing\_corevm.py 800 400'
    - \* AttackVM: './udm\_testing.py 800 400'
    - \* HostMachine: './udm\_testing.py 800 400'

	Attacker	Total Requests Detected	Invalid Requests Detected	Valid Requests Detected
0	CoreNetwork	51	11	40

Figure 11: Results of Test 3.1

	Attacker	Total Requests Detected	Invalid Requests Detected	Valid Requests Detected
0	CoreNetwork	102	52	50
1	Host Machine	100	50	50
2	AttackVM	104	50	54

Figure 12: Results of Test 3.2

	Attacker	Total Requests Detected	Invalid Requests Detected	Valid Requests Detected
0	CoreNetwork	689	300	389
1	Host Machine	688	291	397
2	AttackVM	704	301	403

Figure 13: Results of Test 3.3

	Attacker	Total Requests Detected	Invalid Requests Detected	Valid Requests Detected
0	CoreNetwork	1190	400	790
1	Host Machine	1188	397	791
2	AttackVM	1208	412	796

Figure 14: Results of Test 3.4

Test No.	Expected Requests ( $ER$ )	Detected Requests ( $DR$ )	$Deviation =  ER - DR $
Test 3.1	50	51	1
Test 3.2	300	306	6
Test 3.3	2100	2081	19
Test 3.4	3600	3586	14

Table 4: Experiment stage 3

### 4.3.2 Discussion

This experiment, in particular, challenges the capabilities of our implementation and our running UDM instance. This is because this experiment has a factor that simulates a real-world setting: simulated real-time attacks from multiple sources. The experiment conducted in the fourth section lacks the multiple source component due to technical reasons discussed in chapter 5. And the experiment conducted in the second section lacks the real-time component, as it was in an early stage of development.

Figures 11, 12, 13, and 14 show a deviation in the number of detected valid and invalid requests from the number of valid and invalid requests that are executed by the commands in our experiment. Apart from technical reasons and the relatively unstable nature of TCP traffic, two reasons come to mind. The first one is that our implementation is not formally correct and generates false positives and false negatives, which result in the deviation in the detected number of requests, as shown in the results. We did not provide a proof of soundness and correctness for our implementation, as it is out of scope for our thesis.

The second reason is that we successfully attacked the UDM with a denial-of-service attack, which means that the UDM is too overwhelmed to process registration requests, regardless of their validity. Table 4 shows the deviation in detected requests from each test by comparing the data of Table 3 with the data in the corresponding result figures. We can conclude from Table 4 that there is a correlation between the size of our tests and the size of the deviation in detected requests. However, we need formal proof for our implementation to be able to conclude that we successfully attacked the UDM with a denial-of-service attack.

## 4.4 Attack detection and prevention in real-time

In this section, we will test the implementation of the last stage of development: attack detection and prevention for the UDM in a simulated real-time setting. The results are presented as statistics that show the values of some global variables used in our implementation. The results of this experiment are screenshots from the terminal, because the idea behind this stage of implementation is that information regarding attempted attacks should be shown in real-time while the defense framework is running. Table 5 shows more details about this experiment.

### 4.4.1 Stage 4: Experiments and results

Test No.	Attack Framework	Total requests	Valid requests	Invalid requests	Results
Test 4.1	A.VM	13	10	3	Table 6
Test 4.2	A.VM	130	100	30	Table 6

Table 5: Experiment stage 4 (final stage)

Notes:

- The commands for the tests were executed from the AttackVM.
- Commands used:
  - Before the tests were conducted, we added the Iptables rules with the following commands:
    - \* `sudo iptables -I INPUT -s 192.168.56.102 -j NFQUEUE --queue-num 0`
    - \* `'sudo iptables -I OUTPUT -d 192.168.56.102 -j NFQUEUE --queue-num 0'`
  - '192.168.56.102' is the IP address of CoreNetwork.
  - Test 4.1:
    - \* AttackVM: `'./udm_testing.py 10 3'`
  - Test 4.2:
    - \* AttackVM: `'./udm_testing.py 100 30'`

Test No.	Sent requests	Successful requests	Communication with attacker blocked after:
Test 4.1	10 valid + 3 invalid	1 valid + 1 invalid	2 requests
Test 4.2	100 valid + 30 invalid	5 valid + 1 invalid	6 requests

Table 6: Results experiment stage 4

#### 4.4.2 Discussion

The experimental setting used for our thesis is relatively small compared to real-world settings. We can state this against our defensive and offensive framework; Both frameworks use very little resources in computing power and memory compared to real-world settings. So, the attacks and defense performed are on a larger scale than in a real-world setting. This means that we are only able to speculate about the advantages and disadvantages of our implementation in a real-world deployment scenario by using our implementation and results shown in this chapter.

The results show that our implemented attack detection mechanism is effective, but again, we can not prove that it will also be effective in a real-world deployment setting. However, we can conclude, using the results we showed in this chapter, that we build a foundational attack detection solution against a denial-of-service attack to the UDM that can be polished and built upon for real-world deployment.

#### 4.5 Conclusion

In this chapter, we showed our practical contribution to attack detection against denial-of-service attacks of the UDM. We did this by analyzing recorded network traffic in the first two sections, and by analyzing network traffic in a simulated real-time setting in the third and fourth sections. As we can see in the previous sections, after each experiment, we provide a short discussion about the results. However, there is one common point that we need to mention regarding the experiments and results discussed above.

As mentioned in chapter 3, we set the ratio of 1/5 for valid to invalid registration requests to the UDM. This ratio sets the boundary for the share of invalid requests the UDM can respond to before considering an invalid request as an attempt at a denial-of-service attack. As we can see in the second, third and fourth experiments, we challenge our implementation on this ratio: we have one invalid request above the limit to see if we can detect this request as a malicious request, and with that, treat the source of such a request as an attacker. As we can see, the results in Figures 9, 11, and Table 6 show that the attacks were successfully detected.

## 5 Practical Discussion

In this chapter, we will discuss earlier stages of development and infrastructure. Before the final stage, there were three stages of development, each with its own shortcomings and challenges. And each stage is an improvement on the previous stage. We will discuss the stages one by one. At the end, we will look at the final stage and discuss the shortcomings of this implementation.

### 5.1 The First Stage

At the first stage of development, our focus lay on visualizing valid and non-valid requests to the UDM, for that, we used the 5g-trace-visualizer tool [25]. At this stage, our setup consists of an Ubuntu 64-bit virtual machine that hosts an open5Gs instance in a dockerized environment. This is the same virtual machine we call the CoreNetwork in the chapter 3. The setup is shown more clearly in Figure 15.

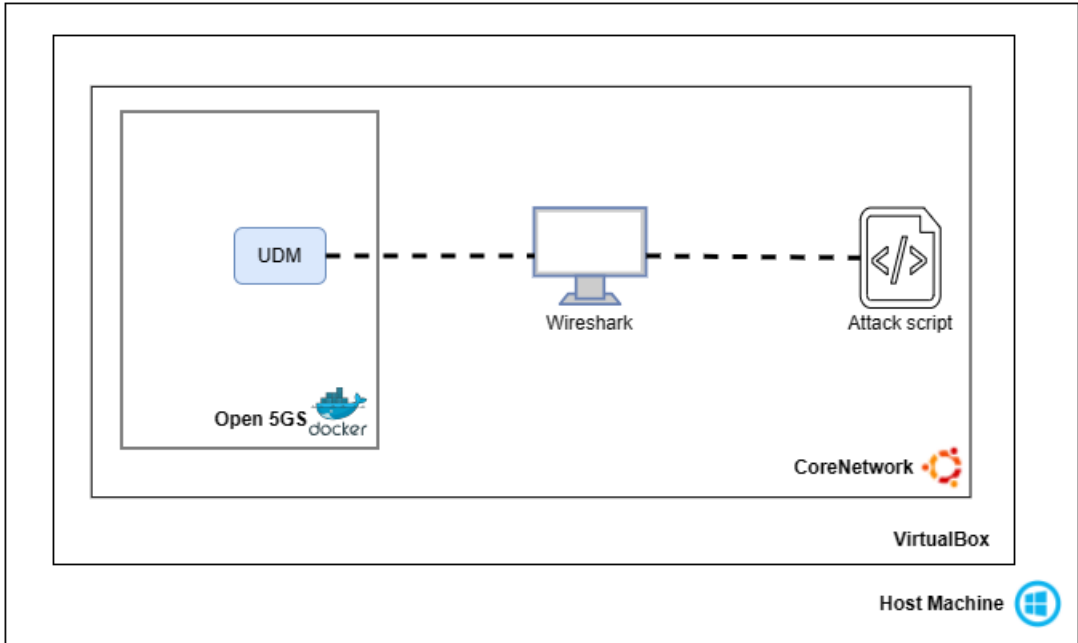


Figure 15: Experimental setup at the first stage of development. The dashed line represents the flow of network traffic between the attack script and the UDM.

The traffic we wanted to visualize was pre-recorded using Wireshark, which was installed on the CoreNetwork virtual machine, as shown in Figure 15. So we had to provide a pcap-file as input to our implemented program for visualization. The recorded traffic contained all the network traffic running in our open5Gs instance. The traffic we are interested in is only the responses received from the UDM to our terminal. So, we had to filter the network data accordingly before visualizing the valid and invalid requests.

The attack script did exactly the same as in the final version. The used IP address for the UDM was the same UDM IP address that was provided by the .env file of the Docker project, as shown in Appendix 9.1. The attack and non-attack scripts were executed from the terminal in the CoreNetwork virtual machine. The purpose of this stage is to plot the valid and invalid requests to the UDM against time. This is shown in more detail in section 4.1. The source code of the attack and non-attack scripts is shown in Appendices 9.2 and 9.4, respectively. These two scripts can both be executed using the `_udm_testing_corevm.py` script presented in Appendix 9.18.

The terminal of the CoreNetwork virtual machine acted as a random UE who tried to subscribe directly to the network by communicating with the UDM directly. This is shown by following the TCP conversation in the captured traffic using Wireshark. Figure 16 shows an example of this: the terminal shows exactly the same payload as shown in the TCP stream in the Wireshark window.

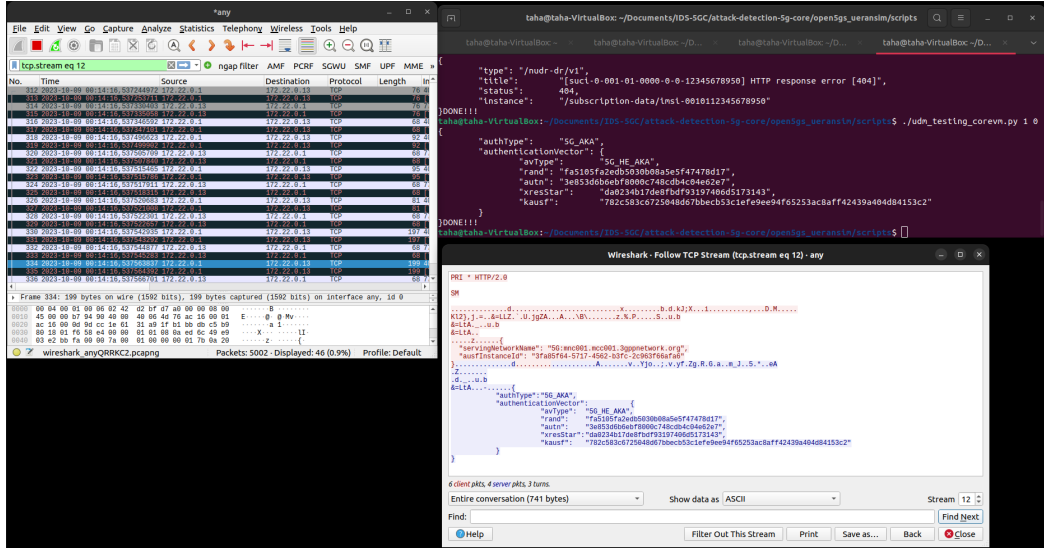


Figure 16: TCP conversation between the UDM and the terminal of the CoreNetwork virtual machine.

In this stage, our implementation was done in an IPython Notebook instead of a Python script, as shown in the chapter 3. IPython Notebook is a Python programming interface that provides a more interactive Python development environment [26]. The implementation of this stage can be found in Appendix 9.15.

## 5.2 The Second Stage

In this stage, the setup and scripts used are identical to those used in the previous stage. But, in this stage, we dive a bit deeper by slicing the recorded traffic into batches using our own implementation of a sliding window. This is done after filtering the recorded traffic in the same way as we did in the first stage.

Each batch contains traffic generated every five seconds. In each batch, we extract the ratio of non-valid requests to total requests. If the ratio exceeds a certain limit, which is given by the user, then a warning message is displayed containing the source IP of the possible attack. This limit can, for example, be determined when the user sees that the network's performance will be affected by the flood of non-valid requests.

At the end, a plot is made showing the source of non-valid requests against time. In our case, we have only one attacker's IP. The attack script is executed from the terminal. But, our implementation is also suitable for multiple-source attacks. The implementation of this stage can be found in Appendix 9.16.

At this stage, the biggest shortcoming is that we work on a pcap-file instead of live network traffic. This diminishes the purpose of an IDS, and as discussed earlier, an IDS should be able to detect an attack in real time. This brings a whole new set of challenges to development, which we will address in the next stage.

## 5.3 The Third Stage

Because of the shortcomings mentioned in the previous stage, in this stage we changed our approach. We used Python Scapy instead of the 5g-trace-visualizer. Scapy allows us to examine traffic online and offline. Online suggests in real-time simulation, thus without providing a pcap-file as input to the script. Offline examination proposes the examination of a pcap-file. At this stage, we still choose to conduct an offline examination of the network traffic between the UDM and our UE, the terminal on the CoreNetwork virtual machine. Our traffic examination will be done on the TCP layer because this is the lowest and most practical network layer for traffic interception and examination in a real-time scenario. The simulation of traffic interception and examination will be discussed in the next section.

Leaving the 5g-trace-visualiser behind brought up a new challenge. The Transport Layer Protocol (TCP) transmits data over a network connection as a stream of octets, which are divided over multiple TCP packets [27]. This means that we need to group TCP packets that belong to the same TCP stream manually, which we will explain further in this subsection. By solving this challenge, we set the foundation for the final stage: examining and intercepting traffic in a simulated real-time scenario.

After intercepting the network traffic with Wireshark, as shown in the previous stages, we extract the TCP traffic. After filtering out other network traffic, we only keep the traffic between the UDM and IP addresses outside our Open 5GS instance. Those other IP addresses can be seen as potential attackers. However, in our practical setting, we only deal with a limited number of potential attackers, so we believe our implementation should be generic and applicable on a larger scale.

After obtaining the relevant network traffic for our setting, group the TCP packets that belong to the same TCP stream. The most efficient way to do this is to group the packets based on the 5-tuple (protocol, source-IP, destination-IP, source-port, destination-port). However, we already extracted the TCP packets, so we can now group the packets based on the 4-tuple (source-IP, destination-IP, source-port, destination-port). We use the dictionary data structure because of its mutable nature. The key in our dictionary is the 4-tuple, and the value is the list of TCP packets that adhere to the 4-tuple.

Now that we have all the TCP packets that share the same conversation attached to the corresponding 4-tuple, we can extract the complete payload of each conversation. For this, we allocate a new dictionary. We call this dictionary ‘raw\_data’. After iterating through ‘raw\_data’ and extracting the stored bytes in the ‘Raw’ header of the TCP packets, we decode the bytes in ‘utf-8’ and ignore the bytes that can not be decoded. This results in the keys ‘raw\_data’ being the 4-tuple and the value being the complete payload of the TCP conversation in a human-readable string.

After the decoding, we want to detect the attack ratio for this set of packets. If the HTTP response is 200 (non-attack), then the data contains an authentication vector, and if the HTTP response is 404 (attack scenario), then the data contains a “status: 404” parameter, as defined in the 3GPP specifications of 5G Systems [9].

Last, we are using the dictionary ‘raw\_data’ to extract the attack / total ratio and compare it to a user-defined limit ratio. If the limit is reached, then we print a warning message containing the IP of the attacker. The implementation of this stage can be found in Appendix 9.17.

This stage made sure that we were comfortable handling the TCP traffic and also helped us gain a better understanding of how the UDM communicates with external IP addresses. That is why we consider this stage foundational for the final stage. In the next section, we will discuss the shortcomings of this stage when it comes to real-time attack detection and address how we solved them. Also, we will discuss the shortcomings of the final product itself.

## 5.4 The Final Stage

As discussed in the previous section, our implementation of the previous stage has some shortcomings when it comes to real-time attack detection.

The first shortcoming is that the script sniffs a whole pcap-file, in a real-time setting, we don’t perform a packet capture but a packet interception. The second shortcoming is that the script decides if a device communicating with the UDM behaves as an attacker after looping several times over captured data. Looping over data is not applicable in a real-time attack detection scenario.

The third shortcoming is that the script decides if a device communicating with the UDM behaves as an attacker based on a response the UDM gives to the device instead of an incoming request from the attacker. As discussed in chapter 2, an IDS should prevent an attack from happening. Which means that in a real-time attack detection scenario, incoming malicious packets should be blocked before being able to generate a response from the target, the UDM in our case. And the last shortcoming is that the script only observes the traffic, we need to be able to act on a potential attacker. In other words, we should block malicious packets, and communication between the attacker and target should not be possible after detection of misbehavior.

To solve the mentioned shortcomings, a few challenges arise. The first one is that we can not work with a pcap-file, so we need to probe a UDM interface to sniff the traffic. The second challenge is to be able to decide based on the current packet if it should be dropped or not. The third challenge arises from the fourth issue, which is that we should be able to drop an incoming request to the UDM.

As shown in Figure 17, each network function has its own network interface that is visible on Wireshark. So we had to try out each interface and see if the traffic consisted of the IP addresses of the UDM and my terminal. We would know that we had the right interface when we saw a huge increase in traffic from the terminal to the UDM, if we sent a relatively large number of requests from the terminal to the UDM.

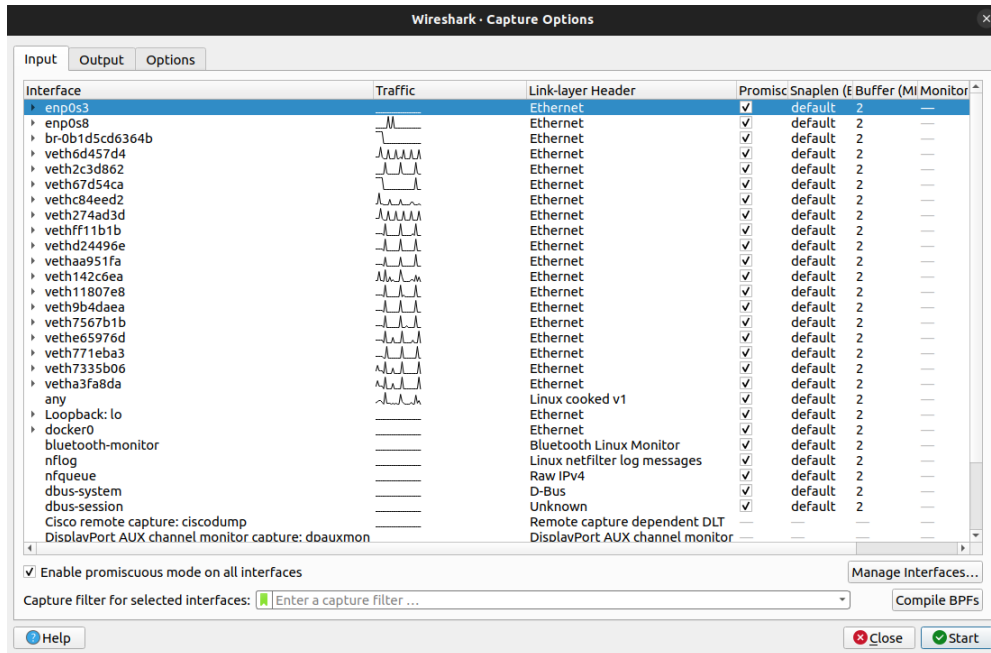


Figure 17: Visible network interfaces in Wireshark while running open5Gs in Docker.

To tackle the second challenge, we had to redesign our logic. We had to use global variables and get rid of loops in our program. We did not need to change our data structure because a Python dictionary is a dynamic data structure where we can store our relevant packets and access data without the need to loop through the data structure. Additionally, we need to perform our packet analysis in request-response pairs. A request should be judged based on previous responses. This is done by blacklisting malicious IP addresses and preventing communication to the UDM from those IP addresses. This way, our implementation will be more suitable in cases where attacks come from multiple sources.

Also, our logic is based on trying to decide if a packet should be dropped or not as soon as possible because, in a real-time setting, attack detection should be as effective as possible. This all resulted in the flow chart presented in Appendix 9.7 and discussed in chapter 3.

To tackle the third challenge, we had to pivot our practical setup. First, we had to let go of Scapy as a tool for traffic interception. As presented in chapter 3, we needed to add firewall rules using Iptables and use Python NetfilterQueue instead of Scapy. Second, running open5Gs in a Docker instance on our CoreNetwork virtual machine causes a lot of technical problems when it comes to traffic interception via the libnetfilter\_queue Linux subsystem. The discussion about these technical issues is environment-specific and out of scope for our thesis. However, the solution to these technical issues caused a change in our technical setup: we had to transform the setup shown in Figure 4 to the setup shown in Figure 15.

Another change was also made in the IP address that we use to communicate with the UDM. Because we communicate with the UDM, which is deployed in the CoreNetwork virtual machine, from the AttackVM virtual machine using the network settings shown in Appendix 9.12. And because the CoreNetwork virtual machine is configured to apply packet forwarding from the native network interface to the network interface of the running Docker network. We can communicate directly with the UDM by communicating with the IP address of the CoreNetwork virtual machine. This communication mode is visualized in Figure 18.

Before tackling the third challenge, we implemented a premature solution, solving only the first two challenges. The source code for this premature implementation can be found in Appendix 9.19. This premature solution is tested as the third stage of development in chapter 4. We chose to treat this premature solution as the third stage of development in the experimental phase because this implementation, as stated earlier, acts as the main foundation for our final implementation. The source code for the final implementation can be found in Appendix 9.14.

Moving on to the issues of our current implementation, the first one is that there is no garbage collection implemented. After we blacklist an IP address, we should be able to remove the previous traffic because that has no relevance to the scope of our research.

The second issue is that there is no persistent storage implemented for our blacklisted IP addresses. The downside of this is that if, for some unseen reason, our program shuts down, then all the blacklisted IP addresses will be removed from memory and will be able to communicate again with the UDM. In other words, the attackers get a new chance to perform an attack.

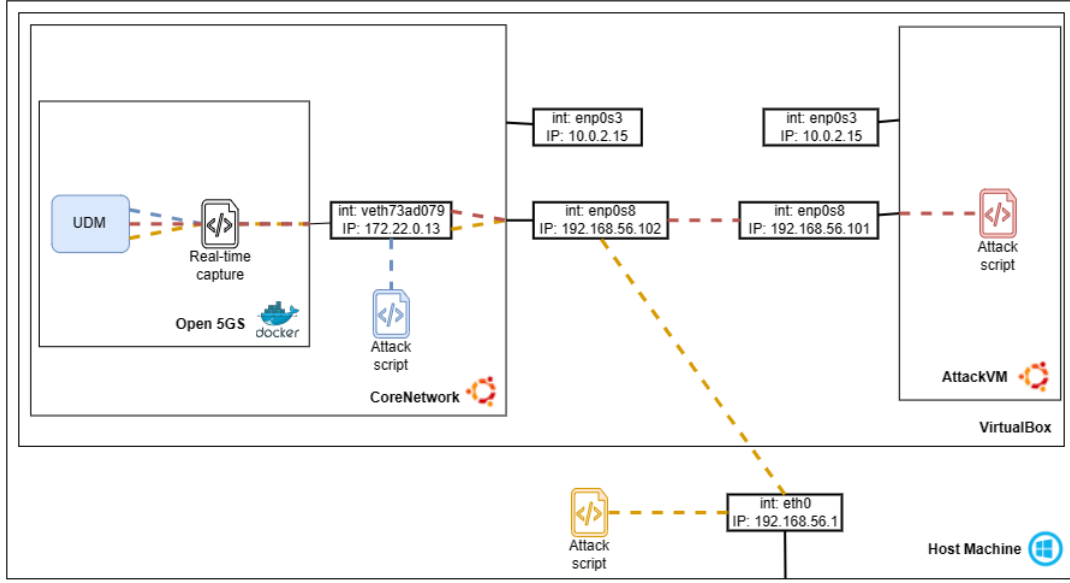


Figure 18: Experimental setup for real-time multiple source attack detection showing the network interfaces, abbreviated as 'int', and IP addresses involved. Only the relevant network interfaces of the Host Machine and open5Gs are shown. The colored dashed lines represent the flow of network traffic between the respective colored attack script and the UDM. The script for real-time capture is presented in Appendix 9.19.

The third issue is that we sniff incoming and outgoing traffic to the UDM from one NetfilterQueue interface. This can prove inefficient over time because there can be a lot of overhead if we deal with a huge amount of network traffic in a relatively short amount of time, which is more likely to happen in practice. The separation of incoming and outgoing traffic seems like a viable solution, but it also has a downside: we need to share the blacklisted IP addresses between the two NetfilterQueue interfaces. This increases the chance for a deadlock situation because two separate instances, the NetfilterQueue instances, modify the same data structures at the same time.

Last, we want to mention that all the source code used for our thesis can be found on our GitLab page [24].

## 6 Theoretical Discussion

In this section, we will technically discuss our thesis. First, we will dive a little deeper into the Service Communication Proxy. Then we will discuss the deployment strategies of an IDS for the UDM to protect it against denial-of-service attacks. Last, we will discuss the deployment strategies of an IDS in the 5G core networks by generalizing our discussion about the IDS deployment in the UDM. To be able to keep this section to the point of deploying an IDS, we will be simplifying the explanation of some technical components of our discussion. These will be the communication between network functions, the functionality of the Service Communication Proxy, and the registration process of the UE with the UDM.

### 6.1 Service Communication Proxy

As discussed in chapter 2, the 5G core network uses a service-based architecture (SBA). The SBA is a system architecture whose functionality is achieved by a set of network functions providing services to other network functions in the system. The service-based interface (SBI) is an interface where the service operations of network functions are invoked [8]. So, a NF producer provides its service to a NF consumer via the SBI. Figure 19 shows a graphical representation.

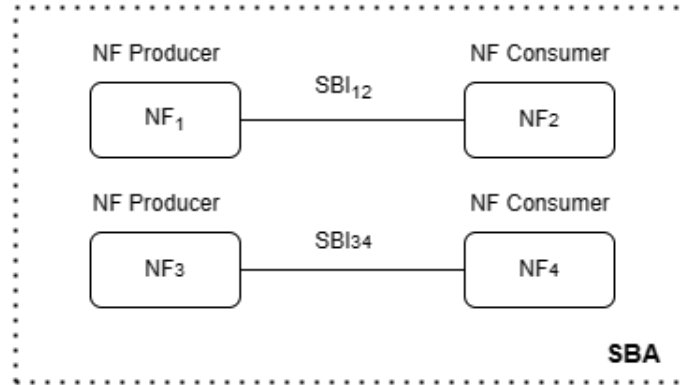


Figure 19: A NF producer communicating over a SBI to a NF consumer as part of a SBA.

This brings us to the Service Communication Proxy (SCP). The SCP is necessary for indirect communication between network functions and was introduced by 3GPP Release 16 [28]. Unlike the original implementation, where the NF-consumer and NF-producer communicate directly, the new communication modes introduce the SCP as a mediator between the service consumers and producers [8, 28]. The SCP itself is not a network function, as it neither provides nor consumes any service from any network function [8, 28]. Other than indirect communications, some functions of the SCP are delegated discovery, load balancing, and secure communication between the network functions [8]. To simplify the working of the SCP, the consumer NF sends an HTTP request to the producer NF, and the producer NF sends an HTTP response back to the consumer NF [29]. The mentioned request and response are routed via the SCP to their destination [29]. Figure 20 shows a simplification of the role of the SCP between network functions.

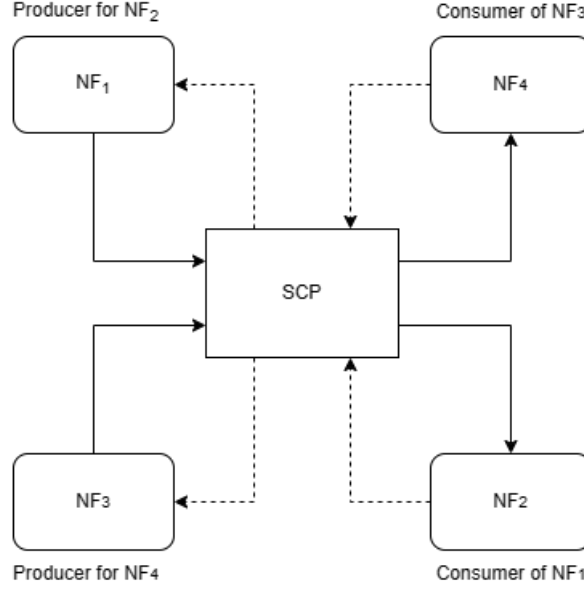


Figure 20: A NF producer communicates with a NF consumer via the SCP. Solid arrows indicate HTTP Requests. Dashed arrows indicate HTTP responses.

In this thesis, we will only discuss the load balancing functionality and leave other functionalities of the SCP out of the discussion, as they don't contribute to the deployment of an IDS in the 5GC.

As we see in Figure 20, the SCP mediates between multiple network functions simultaneously, and so the NF-consumer requests should be handled in a load-balancing manner [29]. The SCP performs load balancing between different NF producers based on three service attributes: Priority, capacity and load [29]. Shetty et al., state that the NF producers with the lowest priority are given precedence over other NF producers, and if multiple NF producers have the same priority, the SCP load balances between the producers according to Equation 6.1 [29].

$$\text{NF\_traffic\_share} = \text{NF traffic capacity} - (\text{NF traffic capacity} * \text{NF traffic load factor})(1)$$

We will keep this equation in mind for our discussion on the deployment of an IDS. In the next section, we will discuss the deployment of an IDS to protect the UDM from DoS attacks by looking at two attack scenarios. But first, we will briefly discuss the performance metrics of an IDS. Then, we will briefly discuss the re-registration process from the UE to the UDM. These two briefly discussed points will help us in the main discussion regarding the IDS deployment in the UDM.

## 6.2 IDS deployment for the UDM

To be able to argue if an IDS can be effective if deployed in a certain place in the 5GC, we need to theoretically evaluate the performance of the deployed IDS. The objectives for the performance of an IDS are a broad detection range, economy in resource usage, and resilience to stress [30].

An IDS with a broad detection range means that the IDS is able to distinguish with high precision between an intrusion and normal behavior, if that is not the case, then many intrusions will escape detection [30]. The second performance objective is efficient use of system resources like CPU time, main memory and disk space [30]. If an IDS consumes too many resources, it can cause a lot of latency in the system and become impractical. The last performance objective is that the IDS should function correctly when the system is under stressful conditions, such as very high-level computing activity [30]. This is because a network is more vulnerable under stressful conditions, and so a correct defense is needed the most. Based on these three objectives, we will discuss the deployment scenarios of an IDS for the UDM.

Before discussing the deployment scenarios of an IDS, we will first take a look at the registration procedure of the User Equipment (UE) with the UDM. For the scope of this discussion, we only need to know two factors. First, through which components of the 5G core network goes the traffic required for the registration process. Second, if the registration process is vulnerable to denial-of-service attacks.

There are several procedures for registration at the UDM: initial registration, periodic registration, mobility registration, and emergency registration [4]. In this discussion, we will not go into what each of these registration procedures entails, but we will rather focus on what generalizations we can draw from these procedures and how that connects to the deployment of an IDS.

The first generalization is about the components of the 5G network that are involved in the registration procedure in relation to the UDM, namely, UE, gNB, AMF, UDM and the AUSF [6, 4]. According to [31], the Network Repository Function (NRF) is also involved. Because of the previously discussed communication method between the network functions, we can state that the SCP is also involved. Figure 21 shows which components of the 5G Core network are needed for the registration process from the UE to the UDM. These are the components we should keep in mind when deploying an IDS to protect the UDM.

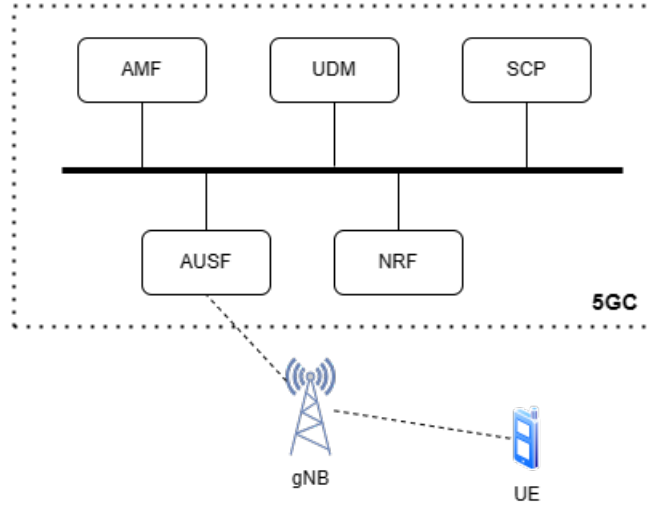


Figure 21: Components of the 5G network that are involved in the registration process of the UE to the UDM.

Second, it is nowhere stated that the UE has a limit on how many times it can repeat the registration procedure in a certain time frame. This means that a malicious UE can theoretically repeat the registration procedure as many times as possible with the intent of overloading the UDM. So, theoretically, the registration procedure of the UE with the UDM is an attack vector for a denial-of-service attack.

This concludes that the UDM is vulnerable to DoS attacks from two attack vectors:

1. Direct communication with the UDM with no involvement of other components of the 5GC. This attack vector is tested practically in this thesis.
2. Repeated execution of the registration procedure by the UE for the UDM. This attack vector is concluded theoretically from existing literature and has not been tested practically yet.

In the upcoming two subsections, we will discuss deployment strategies for an IDS to protect the UDM against the mentioned threat scenarios.

### 6.2.1 IDS deployment for the UDM: First attack scenario

Against the first attack vector, we only have one approach to the deployment of an IDS. And against the second attack vector, we have two approaches to the deployment of an IDS to protect the UDM against a denial-of-service attack.

As discussed previously, the execution of a DoS attack using the first attack vector does not involve another entity of the 5G Core network, except for the attacker UE and the UDM. This means that all the traffic necessary for the attack goes strictly between the two entities, this is shown in Appendix 9.9 with a screenshot of a network capture between the malicious UE (our terminal) and the UDM. In this case, the right approach to deployment is to place the IDS in the UDM, as shown in Figure 22.

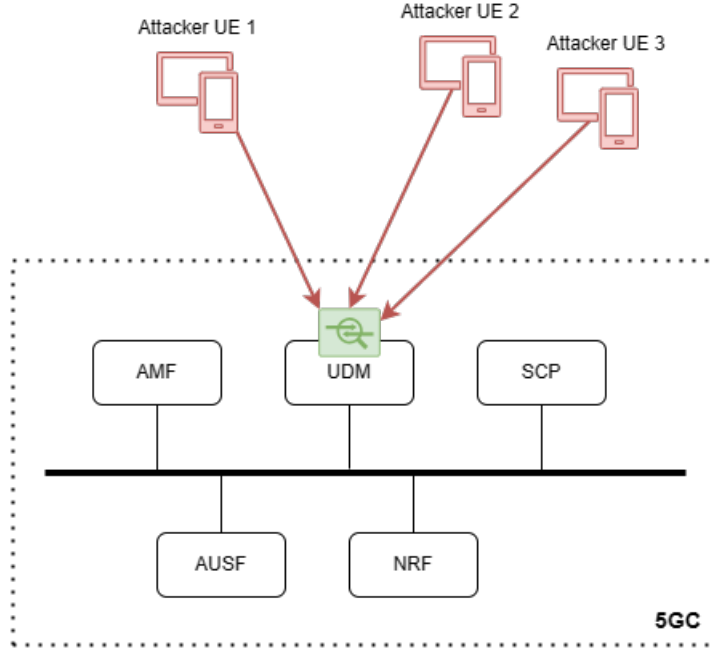


Figure 22: Three attacker UEs attacking the UDM by direct communication from outside the 5GC. The UDM is protected by an IDS.

For this attack scenario, this approach seems practical. The first argument is that the deployed IDS has a good economy of resource usage because it is only focused on network traffic to and from IP addresses outside the 5G Core network and ignores all other traffic. This means that the IDS will not interfere with the functionality of the UDM. Because the service consumers and service providers of the UDM only contain components inside the 5G Core network [8, 9].

The second argument is that the deployed IDS can be easily maintained and upgraded if we want to implement a defense mechanism against another threat towards the UDM, specifically from outside the 5GC.

A counterargument can be made that the IDS may not be stress-resilient. But as we saw in our implementation in chapter 3, the user can define the limit of the ratio of invalid UDM requests that can be considered an attack attempt. This is useful because the busier the network, the easier it becomes for the attacker to perform a denial-of-service attack. So a viable solution is for the user to update the ratio of invalid requests according to how busy the 5G Core network is. This ratio is also dependent on the configuration and performance requirements of the network where the IDS is deployed.

### 6.2.2 IDS deployment for the UDM: Second attack scenario

In this subsection, we will discuss how we can deploy an IDS on the UDM against a denial-of-service attack by repeating the registration process.

To decide where to deploy an IDS, we have to determine through which entity the network traffic goes that we want to examine. In this case, we have two entities: The UDM and the SCP. The UDM is the entity that we want to defend. The SCP because, as discussed earlier, all the network traffic of the 5G Core network goes through the SCP. And therefore also the traffic that we want to monitor to protect the UDM.

So, against this attack, we have two approaches to deploying an IDS:

1. Deploy the IDS in the UDM.
2. Deploy the IDS in the SCP.

The first approach, as shown in Figure 23, is quite comparable to the approach discussed in the previous subsection. In contrast to the argument made in favor of the previous deployment approach about the desired economy in resource usage of the deployed IDS, here we have to disagree. By repeating the registration process of the UDM, the attacker abuses the normal functionality of the entities involved in the registration process, including the UDM. The UDM is attacked in this scenario only via communication with internal entities of the 5G Core network, as mentioned in [9]. So we can call this attack a form of an internal attack.

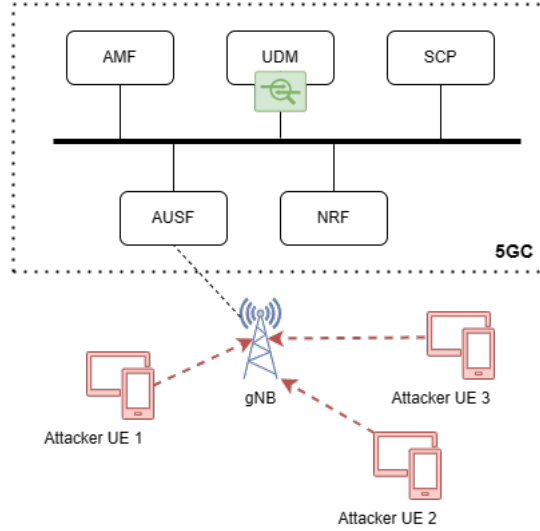


Figure 23: Three attacker UEs attacking the UDM by repeating the registration process to the UDM. An IDS is deployed in the UDM.

In this case, the deployed IDS will have to examine all the network traffic of the UDM, meaning that the IDS will cost the UDM in resources and capacity and increase the load on the UDM. As discussed earlier in Equation 6.1, the SCP decides the share of network traffic that will be passed to a network function based on the load and capacity of the network function. In our case with the UDM, the SCP will assign a lower share of the network traffic to the UDM. Which will result quicker in a lower performance of the UDM and, by induction, a lower performance of the 5G Core network.

An argument in favor of this approach is that the deployed IDS in this case will have a broader detection range because it can offer protection against the first and second attack scenarios. However, in practice, the 5G Core network is mostly deployed using Kubernetes [29]. This means that you can defend the UDM from external traffic by configuring the deployment environment of the 5GC to not expose the UDM to the internet. And so dedicate the deployment of an IDS to protecting the UDM from internal attacks. Such practical decisions are specific to the use case of the specific 5G Core network.

Moving on to the second approach, deploying an IDS in the SCP as shown in Figure 24. As discussed in section 6.1, the SCP acts as a mediator between network functions in 5G core networks. This implies that the SCP is an entity in the 5GC that has access to all the network traffic in the 5GC, which shapes the argument that the deployment of an IDS in the SCP is a viable solution. Because the deployed IDS should have access to all the traffic related to the registration procedure of potential malicious UEs to the UDM to be able to have the desired functionality.

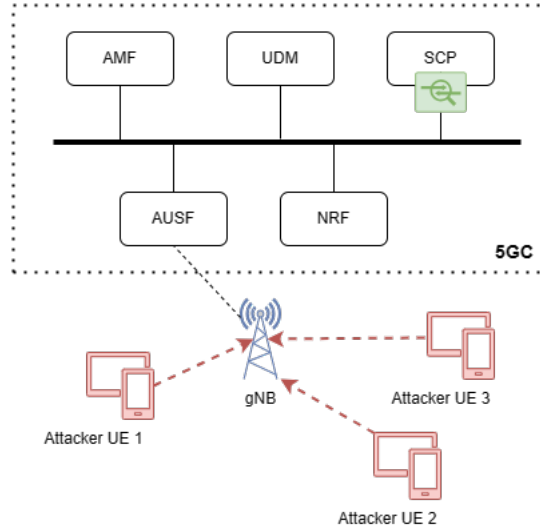


Figure 24: Three attacker UEs attacking the UDM by repeating the registration process to the UDM. An IDS is deployed in the SCP.

However, the downside of this approach is that the IDS will have to filter out a lot of network data that it does not need. The irrelevant network data for the deployed IDS will act as noise data. To be able to measure the effect that this will have on the performance of the deployed IDS, the noise ratio, needs to be measured in a separate experiment. More precisely, we need to measure the share of network traffic that the SCP assigns to the UDM (Equation 1). Because the lower the traffic share, the more noise data will go through the IDS. Which means the IDS should filter out a lot of data, which can slow down the functionality of the SCP itself and therefore increase the latency of the whole 5G Core network. How much latency the 5G Core network can afford is specific to the size of the network and the use case of the network. For example, performing brain surgery from a distance does not tolerate any form of latency compared to a simple phone call.

The upside of deploying an IDS in the SCP is that you can easily add functionalities involving traffic from other network functions in case a new threat emerges. The deployed IDS can be upgraded without changing the entire security architecture of the core network.

A counterargument against both approaches can be made that, by deploying the IDS only in the SCP or only in the UDM, we will have a single point of failure. A theoretical solution would be to divide the defense algorithm into two layers: the first layer is deployed in the SCP, and the second layer is deployed in the UDM. But practical research is needed to determine if deploying two IDSs is resource-efficient and practical for maintenance and upgrades.

Reflecting on both attack scenarios and the three discussed deployment strategies, it is definitely needed to deploy an IDS in the UDM for protection against the first attack scenario, which is an external attack targeting the UDM directly. For the second attack scenario, practical testing is needed to determine which deployment strategy is more efficient.

In the next section, we will theoretically discuss the deployment of an IDS in the 5G core network in general. We will discuss what factors to keep in mind when deciding where to deploy an IDS in the 5GC.

### 6.3 IDS deployment for the 5GC

Real-world deployment of the 5GC follows a cloud-native paradigm, mostly implemented using Kubernetes [29]. This has the advantage that service deployments are more agile, resilient and resource-efficient [29]. However, this introduces a disadvantage from a security standpoint, namely an expanded attack surface for the attacker. From a defensive point of view, to keep the 5G Core network secure, we also need to make sure that the deployment environment, such as Kubernetes, is secure. How to make that a reality is out of scope for this thesis. In this section, we will only focus on the theoretical aspect of deploying an IDS in the 5GC.

When designing security solutions for 5G networks, it is critical to achieve a balance between security and the performance of the network [31]. The focus should lie on enhancing both factors: network security and network performance. Both areas are equally important [31]. This also applies to the deployment of an IDS in the 5G core network. To be able to determine the impact of a deployed IDS on network performance and if the impact is acceptable, practical experiments need to be conducted and the specific use cases of the network need to be determined. As discussed earlier, some use cases are less tolerant of a decline in network performance than others.

For the deployment of an IDS in the 5GC, we first need to determine if we are defending against an attack that makes use of internal or external traffic. Internal network traffic is traffic that flows only through one of the service-based interfaces of the 5GC. External network traffic is traffic that communicates with a network function or the SCP without the use of the Service-Bases Interface of the 5GC. It is like direct communication with a network function without the involvement of another network function, as discussed earlier. Considering the real-world deployment of the 5G Core network, a possible attack vector for an attacker is to gain access to the 5GC through an exposed port of a network function or the SCP to the internet. This case is theoretically solvable by configuring the deployment environment securely.

As concluded earlier, the SCP has access to all the internal network traffic of the 5GC because it acts as a load balancer and mediator of communication between the network functions. So if we want to deploy an IDS to defend one or more network functions, the intuitive approach would be a deployment in the SCP, as shown in Figure 25.

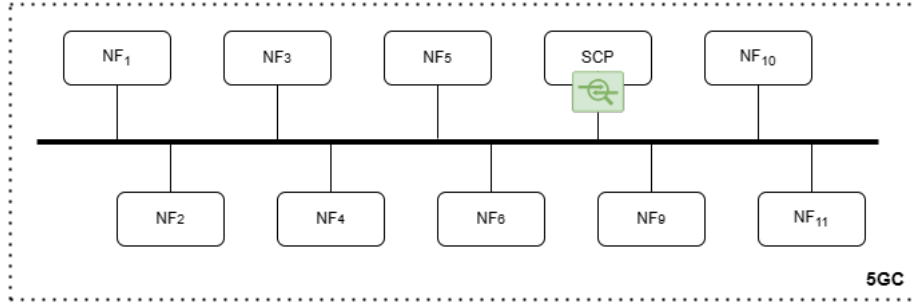


Figure 25: Generic case of an IDS deployment for the 5GC in the SCP.

For this deployment approach, we can generalize the arguments made in the previous section regarding the IDS deployment in the SCP to defend the UDM from an internal attack. The argument in favor of this approach is the scalability of the IDS. In this case, we can safely say that this approach allows for adding defense mechanisms to the IDS against threats targeted at other network functions. This way, the deployed IDS has the potential to be the first layer of defense for all network functions in the 5G core network.

The first argument against this approach is that the IDS would have to filter out a lot of noise data, which affects the detection range of the deployed IDS negatively. However, if we consider scalability to be an essential property for our IDS, then theoretically we can state that the proportion of noise network traffic will decline as the scaled IDS would need more network data to detect several threats. For an exact determination of this claim, practical experiments are needed.

The second argument against this approach is that the defense architecture of the 5G core network would have a single point of failure. A possible solution would be to decentralize the defense mechanisms across the network functions in the 5GC.

Lastly, we will discuss the scenario of deploying an IDS to defend a network function from malicious external traffic, as shown in Figure 26. Previously in this chapter, we discussed the scenario of defending the UDM against an external denial-of-service attack, and we then suggested the approach of placing an IDS in the UDM.

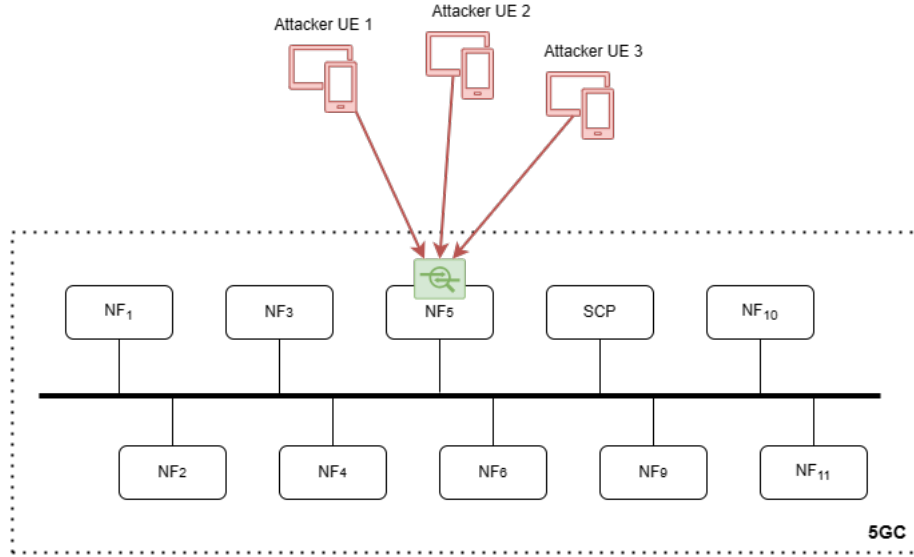


Figure 26: Generic case of an IDS deployment for the 5GC in a NF that is attacked externally.

The first argument mentioned in favor of this approach is the efficient resource usage of the IDS. The second argument is the scalability of the IDS to defend against external attacks. These two arguments can be generalized to any network function. The counterargument mentioned was the possible lack of stress resilience in the deployed IDS, which holds also for the case of defending any network function.

Reflecting on both attack scenarios and the proposed approaches for IDS deployment in the 5G core network, each approach has its own advantages and disadvantages. Another aspect to keep in mind is that one attack scenario does not exclude another. Namely, external and internal attacks are both threats to the network functions in the 5GC [6]. That is why, in reality, the deployed IDS should be able to defend against both kinds of threats to a network function and to the 5GC. So, for the security architecture of the 5GC, a combination of both approaches should be considered.

## 7 Conclusion

From our contribution in general, we can draw both a practical and a theoretical conclusion. In this chapter, we will draw a practical and a theoretical conclusion for our thesis. We will do this by providing a reflection on our contributions and answering our main research question and sub-research questions.

As shown in chapter 4, we implemented four stages of attack detection against the UDM. We also showed that our implementations are effective in our relatively small, experimental setting. We also discussed the fact that we can not prove that our implementation will be effective in a real-world setting. In chapter 5, we discussed the advantages and the shortcomings of the implementation of each stage.

However, we conclude that our implementation in the final stage is the best one that can serve as a foundation for implementing an IDS that protects the UDM from denial-of-service attacks. This is because our implementation in the final stage detects and blocks potential attackers in a simulated real-time setting, which is a functional requirement of an IDS [14].

In chapter 6, we provided our theoretical contribution. As stated, there are many approaches to deploying an IDS in the 5G core network. Each way of deployment is beneficial under certain attack scenarios we want our IDS to defend against, and we should consider what assets in the 5G core network we want to defend. This leads us back to our research questions and sub-questions.

Answering our first sub-research question, we conclude that the UDM is the most important asset in the 5G core network. As discussed in chapter 2, the UDM manages data for access authorization, data network profiles and UE registration. This makes the UDM an attractive asset to target. A denial-of-service attack on the UDM leads, theoretically, to a denial-of-service attack on the core network.

After concluding that the UDM is the most important asset of the 5G core network, we can answer our second sub-research question: What is the best way to defend the UDM using an IDS?

As discussed in chapter 6, deploying an IDS to defend the UDM depends on the attack scenario we want to defend the UDM against. We also discussed that the SCP and the amount of noise traffic data play a significant role in deployment strategy. Practical experiments are needed to determine the ratio of noise traffic data to relevant traffic data in the SCP and its effect on the performance of the IDS and the core network in general.

So, to determine the best way to defend the UDM, practical experiments on the SCP are needed. These are also the technical considerations that need to be kept in mind when deploying an IDS, which answers our third sub-research question: What practical considerations do you need to keep in mind when deploying an intrusion detection system?

Lastly, we will answer our main research question: How can we upgrade the 5G Core network with an intrusion detection system?

To be able to determine how we can upgrade the 5G core network with an IDS, we should state the attack scenario we want to defend against and the assets we want to defend in the 5G core network. When these two factors are established, we should determine how tolerant our network is to latency. As discussed earlier, this depends on the specific use case of the network. Another technical factor is the role of the SCP in relation to the assets in the 5G core network we want to defend, and latency in the network is deployed with an IDS. In the next chapter, we make a suggestion for future work that includes practical experiments on the SCP in relation to other assets in the core network. When these technical considerations are done, we can conclude the best way to deploy an IDS in the 5G core network.

## 8 Future Work

In this chapter, we will discuss suggestions for future work. First, we will suggest how our current implementation can be improved by reiterating the shortcomings mentioned in chapter 5. Second, we will provide suggestions for future work that will contribute to a more substantiated deployment strategy for an IDS in the 5G core network.

To reiterate our practical discussion, we mentioned a few shortcomings in our implementation. The first one is that our implementation does not contain a garbage collection mechanism. The second shortcoming is that the blacklisted IPs are not stored using persistent storage, like a database management system (DBMS), but using system memory. And the third shortcoming is that our implementation sniffs traffic from one NetfilterQueue instance. So, our first suggestion for future work is to improve our implementation by fixing the mentioned shortcomings.

Salahdine et al., mentions a proposal for the use of deep reinforcement learning techniques to mitigate denial-of-service attacks over 5G core networks [32]. This leads to our second suggestion for future work, to improve our implementation with deep reinforcement learning algorithms.

Moving on to the suggestions for future work that will provide more substantiation to the deployment strategy of an IDS in the 5GC. The first one that comes to mind is to test if the SCP breaks the TLS connection between network functions and is thereby able to read the content of the messages sent and received. If that is the case, then it is highly likely that the deployed IDS will need to process less network traffic to identify a threat. This will have a positive impact on the resource usage of the deployed IDS.

After theoretically discussing a denial of service on the UDM by exploiting the registration of the UE to the UDM in chapter 6. We would suggest simulating this attack practically and researching if there are more points at which an IDS could be deployed other than the UDM and the SCP to prevent such an attack.

Another point that needs to be researched further to be able to determine where an IDS should be deployed in 5GC is the ratio of network traffic between UDM and total network traffic going through the SCP. If this process is repeated with all the network functions in the 5GC, we would be able to predict the proportion of network data for a network function against the total network data that goes through the SCP more accurately. As discussed in chapter 6, when deploying an IDS to protect a network function, the network traffic of other network functions that are not relevant is treated as noise data. If the ratio of the noise data can be predicted beforehand, then this will assist in the deployment of an IDS in the SCP.

As shown in chapter 3 and discussed in chapter 6, our implementation allows the user to manually determine the ratio of invalid UDM requests against valid UDM requests. Our last suggestion for future work is to implement a mechanism for this user-defined ratio to be adjusted automatically according to how busy the 5GC network is. This will take a series of simulated denial-of-service attacks. The objective is to find out how much network traffic is needed for network performance to slow down. From there, the correlation between normal network activity and network performance can be defined. And this will be a component of defining the ratio of invalid UDM requests automatically.

To summarize this chapter, we will list our six suggestions for future work below:

1. Improve our implementation by patching up our discussed shortcomings.
2. Improve our defense technique with deep reinforcement learning.
3. Test if the SCP breaks up the TLS connection and can read the contents of messages.
4. Simulate a denial-of-service attack on the UDM by exploiting the registration process.
5. Determine the ratio of network traffic for each network function against the total network traffic going through the SCP.
6. Automatically adjust the ratio of invalid UDM requests against valid UDM requests according to how busy the 5GC network is.

## 9 Appendix

### 9.1 .env file open5Gs

```
1 DEFAULT_CONFIG_PATH=/open5gs/install/default_configs/
2 OPEN5GS_IMAGE=radixsecurity/open5gs
3 MONGO_IMAGE=radixsecurity/mongo
4 UERANSIM_IMAGE=radixsecurity/ueransim
5 TIMEZONE_MNT=/etc/timezone
6 MCC=001
7 MNC=01
8 TEST_NETWORK=172.22.0.0/24
9 DOCKER_HOST_IP=192.168.1.223
10 MONGO_IP=172.22.0.2
11 HSS_IP=172.22.0.3
12 PCRF_IP=172.22.0.4
13 SGWC_IP=172.22.0.5
14 SGWU_IP=172.22.0.6
15 SGWU_ADVERTISE_IP=172.22.0.6
16 SMF_IP=172.22.0.7
17 UPF_IP=172.22.0.8
18 UPF_ADVERTISE_IP=172.22.0.8
19 MME_IP=172.22.0.9
20 AMF_IP=172.22.0.10
21 AUSF_IP=172.22.0.11
22 NRF_IP=172.22.0.12
23 UDM_IP=172.22.0.13
24 UDR_IP=172.22.0.14
25 DNS_IP=172.22.0.15
26 RTPENGINE_IP=172.22.0.16
27 MYSQL_IP=172.22.0.17
28 FHSS_IP=172.22.0.18
29 ICSCF_IP=172.22.0.19
30 SCSCF_IP=172.22.0.20
31 PCSCF_IP=172.22.0.21
32 SRS_ENB_IP=172.22.0.22
33 NR_GNB_IP=172.22.0.60
34 NR_UE_IP=172.22.0.70
35 UE1_IMEI=356938035643803
36 UE1_IMEISV=4370816125816151
37 UE1_IMSI=001011234567895
38 UE1_KI=A9AF4F6CAF3AF8BC7E570b08CAFED00D
39 UE1_OP=11111111111111111111111111111111
40 UE1_AMF=8000
41 OAI_ENB_IP=172.22.0.25
42 WEBUI_IP=172.22.0.26
43 PCF_IP=172.22.0.27
44 NSSF_IP=172.22.0.28
45 BSF_IP=172.22.0.29
46 ENTITLEMENT_SERVER_IP=172.22.0.30
47 OSMOMSC_IP=172.22.0.31
48 OSMOHLR_IP=172.22.0.32
49 SMSC_IP=172.22.0.33
50 PCAP_TRACE=true
```

## 9.2 Source Code: attack\_udm\_original.py

```
1 #!/usr/bin/env python3
2
3 import subprocess
4 import os
5 import sys
6
7 if len(sys.argv) != 2:
8     print("Usage: ./attack_udm_original.py <amount_of_requests>")
9     sys.exit()
10 else:
11     # Try to convert input to integer.
12     try:
13         length = int(sys.argv[1])
14     except ValueError:
15         print("Usage: ./attack_udm_original.py <amount_of_requests>")
16         sys.exit()
17
18 udm_ip = ' 172.22.0.13'
19 udm_port = ' 7777'
20 suci_id = ' suci-0-001-01-0000-0-0-1234567895'
21
22 attack_commands = list()
23 half_command = './request_udm.sh' + udm_ip + udm_port
24 # Create attack commands:
25 for i in range (length):
26     invalid_suci = suci_id + str(i)
27     att_command = half_command + invalid_suci
28     attack_commands.append(att_command)
29
30 # Execute the created attack commands.
31 for att_comm in attack_commands:
32     os.system(att_comm)
33
34 print()
```

### 9.3 Source Code: attack\_udm.py

```
1 #!/usr/bin/env python3
2
3 import subprocess
4 import os
5 import sys
6
7 if len(sys.argv) != 2:
8     print("Usage: ./attack_udm.py <amount_of_requests>")
9     sys.exit()
10 else:
11     # Try to convert input to integer.
12     try:
13         length = int(sys.argv[1])
14     except ValueError:
15         print("Usage: ./attack_udm.py <amount_of_requests>")
16         sys.exit()
17
18 udm_ip = ' 192.168.56.102' #Ip of CoreNetwork virtual machine.
19 udm_port = ' 7777'
20 suci_id = ' suci-0-001-01-0000-0-0-1234567895'
21
22 attack_commands = list()
23 half_command = './request_udm.sh' + udm_ip + udm_port
24 # Create attack commands:
25 for i in range (length):
26     invalid_suci = suci_id + str(i)
27     att_command = half_command + invalid_suci
28     attack_commands.append(att_command)
29
30 # Execute the created attack commands.
31 for att_comm in attack_commands:
32     os.system(att_comm)
33
34 print()
```

## 9.4 Source Code: not\_attack\_udm\_original.py

```
1 #!/usr/bin/env python3
2
3 import subprocess
4 import os
5 import sys
6
7 if len(sys.argv) != 2:
8     print("Usage: ./not_attack_udm_original.py <amount_of_requests>")
9     sys.exit()
10 else:
11     # Try to convert input to integer.
12     try:
13         length = int(sys.argv[1])
14     except ValueError:
15         print("Usage: ./attack_udm_original.py <amount_of_requests>")
16         sys.exit()
17
18 udm_ip = ' 172.22.0.13'
19 udm_port = ' 7777'
20 suci_id = ' suci-0-001-01-0000-0-0-1234567895'
21
22 command = './request_udm.sh' + udm_ip + udm_port + suci_id
23
24 # Execute the created commands.
25 results = list()
26 for i in range (length):
27     os.system(command)
28
29 print()
```

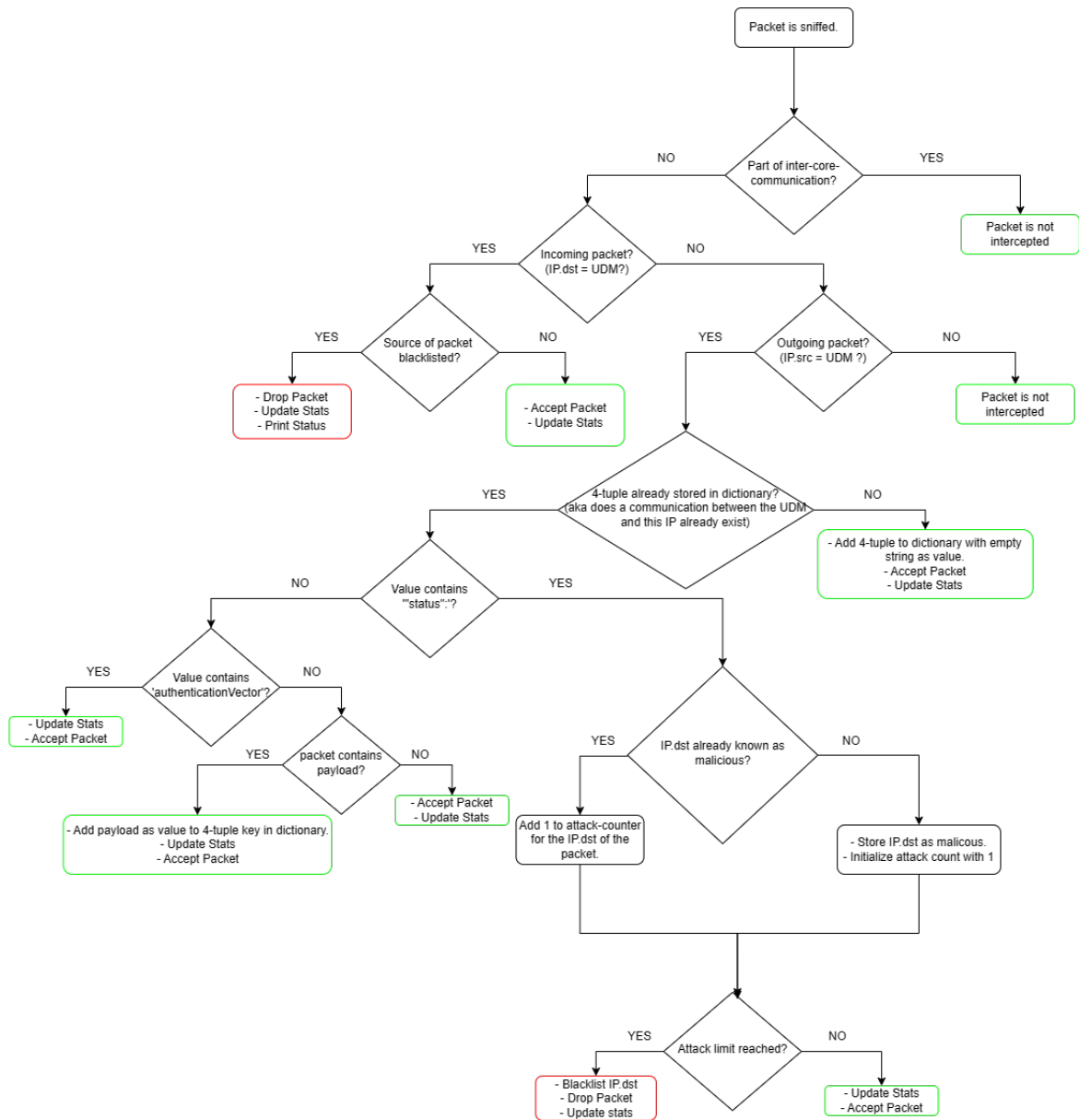
## 9.5 Source Code: not\_attack\_udm.py

```
1 #!/usr/bin/env python3
2
3 import subprocess
4 import os
5 import sys
6
7 if len(sys.argv) != 2:
8     print("Usage: ./not_attack_udm.py <amount_of_requests>")
9     sys.exit()
10 else:
11     # Try to convert input to integer.
12     try:
13         length = int(sys.argv[1])
14     except ValueError:
15         print("Usage: ./attack_udm.py <amount_of_requests>")
16         sys.exit()
17
18 udm_ip = ' 192.168.56.102' #Ip of CoreNetwork virtual machine
19 udm_port = ' 7777'
20 suci_id = ' suci-0-001-01-0000-0-0-1234567895'
21
22 command = './request_udm.sh' + udm_ip + udm_port + suci_id
23
24 # Execute the created commands.
25 for i in range (length):
26     os.system(command)
27
28 print()
```

## 9.6 Source Code: request\_udm.sh

```
1 #!/bin/bash
2 if [ $# -ne 3 ]; then
3     printf "Usage $0 <UDM_IP> <PORT> <ID>\n ID=(IMSI SUCI)\n"
4     exit
5 fi
6 UDM_IP=$1
7 PORT=$2
8 ID=$3
9
10 curl -X 'POST' --http2-prior-knowledge \
11     "http://$UDM_IP:$PORT/nudm-ueau/v1/$ID/security-information/generate-
12     auth-data" \
13     -H 'accept: application/json' \
14     -H 'Content-Type: application/json' \
15     -d '{
16     "servingNetworkName": "5G:mnc001.mcc001.3gppnetwork.org",
17     "ausfInstanceId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
18 }'
```

## 9.7 Flowchart for Defense Script



## 9.8 Source Code: udm\_testing.py

```
1  #!/usr/bin/env python3
2
3  import subprocess
4  import os
5  import sys
6  import random
7
8  _valid_req = None
9  _invalid_req = None
10
11 if len(sys.argv) != 3:
12     print("Usage: ./udm_testing.py <amount_of_valid_requests> <
13         amount_of_invalid_requests>")
14     sys.exit()
15 else:
16     # Try to convert input to integer.
17     try:
18         _valid_req = int(sys.argv[1])
19         _invalid_req = int(sys.argv[2])
20     except ValueError:
21         print("Usage: ./udm_testing.py <amount_of_valid_requests> <
22             amount_of_invalid_requests>")
23         sys.exit()
24     pass
25
26 _valid_req_left = _valid_req > 0
27 _invalid_req_left = _invalid_req > 0
28 _choice = "None"
29
30 # Choose randomly which script to execute until one of the given amounts
31 # is exhausted.
32 while _valid_req_left and _invalid_req_left:
33     _choice = random.choice(["attack", "non_attack"])
34
35     if _choice == "attack":
36         _invalid_req -= 1
37         os.system("./attack_udm.py 1")
38         _invalid_req_left = _invalid_req > 0
39     elif _choice == "non_attack":
40         _valid_req -= 1
41         os.system("./not_attack_udm.py 1")
42         _valid_req_left = _valid_req > 0
43
44 # When one of the given amounts is exhausted, execute the right script
45 # with left amount.
46 if _valid_req_left:
47     os.system("./not_attack_udm.py " + str(_valid_req))
48 elif _invalid_req_left:
49     os.system("./attack_udm.py " + str(_invalid_req))
```

## 9.9 Network Capture of traffic between the UDM and Malicious UE.

The screenshot displays the Wireshark network capture interface. The top menu bar includes File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Wireless, Tools, and Help. The toolbar contains various icons for file operations, capture control, and analysis. The packet list pane shows a series of captured packets, with the selected packet (334) highlighted. The packet details pane shows the structure of the selected packet, including the Ethernet II header, Internet Protocol Version 4 header, and Transmission Control Protocol header. The packet bytes pane shows the raw hex and ASCII data of the selected packet.

No.	Time	Source	Destination	Protocol	Length	Info
312	2023-10-09 08:14:18.537246972	172.22.0.1	172.22.0.13	TCP	76	40396 → 7777 [EST] Seq=64256 Win=0 Len=0 MSS=1460 SACK_PERM=1
313	2023-10-09 08:14:18.537250111	172.22.0.1	172.22.0.13	TCP	76	[TCP Out-Of-Order] TCP Port number is reserved 40396 → 7777 [RST]
314	2023-10-09 08:14:18.537250493	172.22.0.13	172.22.0.1	TCP	72	7777 → 40396 [RST, ACK] Seq=64256 Win=0 Len=0 MSS=1460
315	2023-10-09 08:14:18.537250908	172.22.0.13	172.22.0.1	TCP	76	[TCP Out-Of-Order] 7777 → 40396 [RST, ACK] Seq=64256 Win=0 Len=0 MSS=1460
316	2023-10-09 08:14:18.537251322	172.22.0.1	172.22.0.13	TCP	68	40396 → 7777 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=33832970
317	2023-10-09 08:14:18.537347181	172.22.0.1	172.22.0.13	TCP	68	[TCP Dup ACK 316] 40396 → 7777 [ACK] Seq=1 Ack=1 Win=64256
318	2023-10-09 08:14:18.537496623	172.22.0.1	172.22.0.13	TCP	92	40396 → 7777 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=24 TSval=39
319	2023-10-09 08:14:18.537599908	172.22.0.1	172.22.0.13	TCP	92	[TCP Retransmission] 40396 → 7777 [PSH, ACK] Seq=1 Ack=1 Win=64256
320	2023-10-09 08:14:18.537607690	172.22.0.13	172.22.0.1	TCP	68	7777 → 40396 [ACK] Seq=1 Ack=25 Win=65152 Len=0 TSval=6619996
321	2023-10-09 08:14:18.537607690	172.22.0.13	172.22.0.1	TCP	68	[TCP Dup ACK 320] 7777 → 40396 [ACK] Seq=1 Ack=25 Win=65152
322	2023-10-09 08:14:18.537615469	172.22.0.1	172.22.0.13	TCP	92	40396 → 7777 [PSH, ACK] Seq=25 Ack=1 Win=64256 Len=27 TSval=3
323	2023-10-09 08:14:18.537615469	172.22.0.1	172.22.0.13	TCP	92	[TCP Retransmission] 40396 → 7777 [PSH, ACK] Seq=25 Ack=1 Win=64256
324	2023-10-09 08:14:18.537617911	172.22.0.13	172.22.0.1	TCP	68	7777 → 40396 [ACK] Seq=1 Ack=52 Win=65152 Len=0 TSval=6619996
325	2023-10-09 08:14:18.537618316	172.22.0.13	172.22.0.1	TCP	68	[TCP Dup ACK 324] 7777 → 40396 [ACK] Seq=1 Ack=52 Win=65152
326	2023-10-09 08:14:18.537620683	172.22.0.1	172.22.0.13	TCP	81	40396 → 7777 [PSH, ACK] Seq=52 Ack=1 Win=64256 Len=13 TSval=3
327	2023-10-09 08:14:18.537621008	172.22.0.1	172.22.0.13	TCP	81	[TCP Retransmission] 40396 → 7777 [PSH, ACK] Seq=52 Ack=1 Win=64256
328	2023-10-09 08:14:18.537622301	172.22.0.13	172.22.0.1	TCP	83	7777 → 40396 [ACK] Seq=1 Ack=85 Win=65152 Len=0 TSval=6619996
329	2023-10-09 08:14:18.537622687	172.22.0.13	172.22.0.1	TCP	83	[TCP Dup ACK 328] 7777 → 40396 [ACK] Seq=1 Ack=85 Win=65152
330	2023-10-09 08:14:18.537624635	172.22.0.1	172.22.0.13	TCP	192	40396 → 7777 [RST, ACK] Seq=85 Ack=1 Win=64256 Len=129 TSval=1
331	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	184	[TCP Out-Of-Order] 7777 → 40396 [RST, ACK] Seq=85 Ack=1 Win=64256
332	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	83	7777 → 40396 [ACK] Seq=1 Ack=184 Win=65152 Len=0 TSval=6619996
333	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	68	[TCP Dup ACK 332] 7777 → 40396 [ACK] Seq=1 Ack=184 Win=65152
334	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	195	40396 → 7777 [PSH, ACK] Seq=184 Ack=1 Win=64256 Len=131 TSval=1
335	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	195	[TCP Retransmission] 40396 → 7777 [PSH, ACK] Seq=184 Ack=1 Win=64256
336	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	68	7777 → 40396 [ACK] Seq=1 Ack=325 Win=64896 Len=8 TSval=6619996
337	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	68	[TCP Dup ACK 336] 7777 → 40396 [ACK] Seq=1 Ack=325 Win=64896
338	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	82	7777 → 40396 [PSH, ACK] Seq=1 Ack=325 Win=64896 Len=16 TSval=1
339	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	82	[TCP Retransmission] 7777 → 40396 [PSH, ACK] Seq=1 Ack=325 Win=64896
340	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	68	40396 → 7777 [ACK] Seq=325 Ack=16 Win=64256 Len=8 TSval=33832
341	2023-10-09 08:14:18.537624635	172.22.0.13	172.22.0.1	TCP	68	[TCP Dup ACK 340] 40396 → 7777 [ACK] Seq=325 Ack=16 Win=64256
342	2023-10-09 08:14:18.537738613	172.22.0.1	172.22.0.13	TCP	77	40396 → 7777 [PSH, ACK] Seq=325 Ack=16 Win=64256 Len=9 TSval=1

Frame 334: 199 bytes on wire (1592 bits), 199 bytes captured (1592 bits) on interface any, id 0










```

0000  00 04 00 01 00 00 02 42 42 bf d7 a0 00 00 00 00  .....S.....
0010  45 00 00 07 45 00 40 00 40 08 4d 78 ac 16 00 01  .....J..Mv....
0020  ac 16 00 0d 0d cc 1e 87 31 a9 1f b1 b4 db cb b9  ....x.....I...
0030  06 10 01 16 50 e4 00 00 01 01 08 0a ed 8c 49 e9  ....X.....I...
0040  03 e2 bf fa 00 00 7a 06 01 00 00 00 01 7b 0a 20  ....z.....{...
0050  20 22 73 65 72 76 09 0e 67 4e 65 74 77 0f 72 6b  "servin gNetwork










```

Packets: 24607 · Displayed: 46 (0.2%) Profile: Default

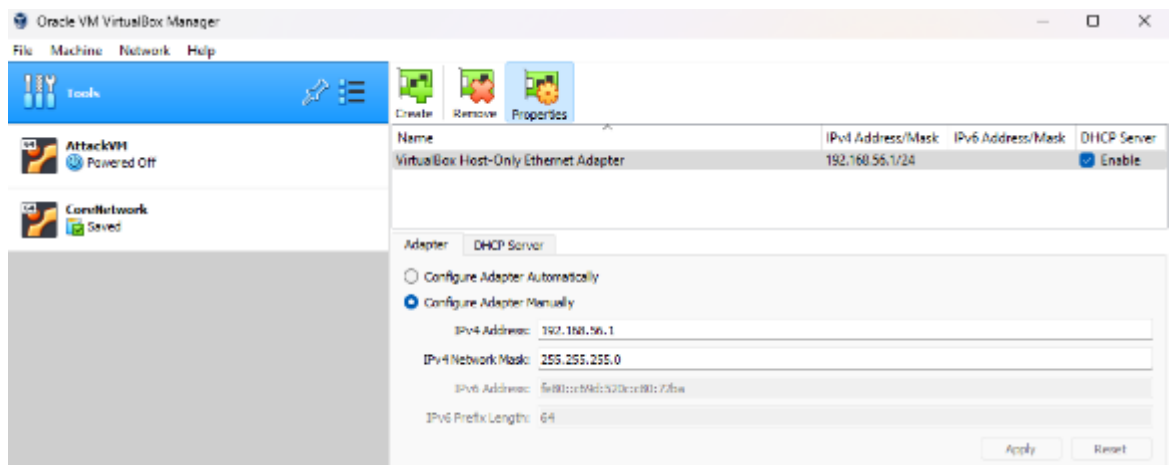
## 9.10 Technical specifications: virtual machine CoreNetwork.

 <b>General</b>	
Name:	CoreNetwork
Operating System:	Ubuntu (64-bit)
<hr/>	
 <b>System</b>	
Base Memory:	5094 MB
Processors:	3
Boot Order:	Floppy, Optical, Hard Disk
Acceleration:	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
<hr/>	
 <b>Display</b>	
Video Memory:	128 MB
Graphics Controller:	VMSVGA
Acceleration:	3D
Remote Desktop Server:	Disabled
Recording:	Disabled
<hr/>	
 <b>Storage</b>	
Controller:	IDE
IDE Primary Device 0:	[Optical Drive] CoreNetwork-disk001.iso (50,59 MB)
Controller:	SATA
SATA Port 0:	CoreNetwork-disk002.vdi (Normal, 35,00 GB)
<hr/>	
 <b>Audio</b>	
Host Driver:	Windows DirectSound
Controller:	ICH AC97
<hr/>	
 <b>Network</b>	
Adapter 1:	Intel PRO/1000 MT Desktop (NAT)
Adapter 2:	Intel PRO/1000 MT Desktop (Host-only Adapter, 'VirtualBox Host-Only Ethernet Adapter')
<hr/>	
 <b>USB</b>	
USB Controller:	OHCI
Device Filters:	0 (0 active)
<hr/>	
 <b>Shared folders</b>	
	None
<hr/>	
 <b>Description</b>	
	None

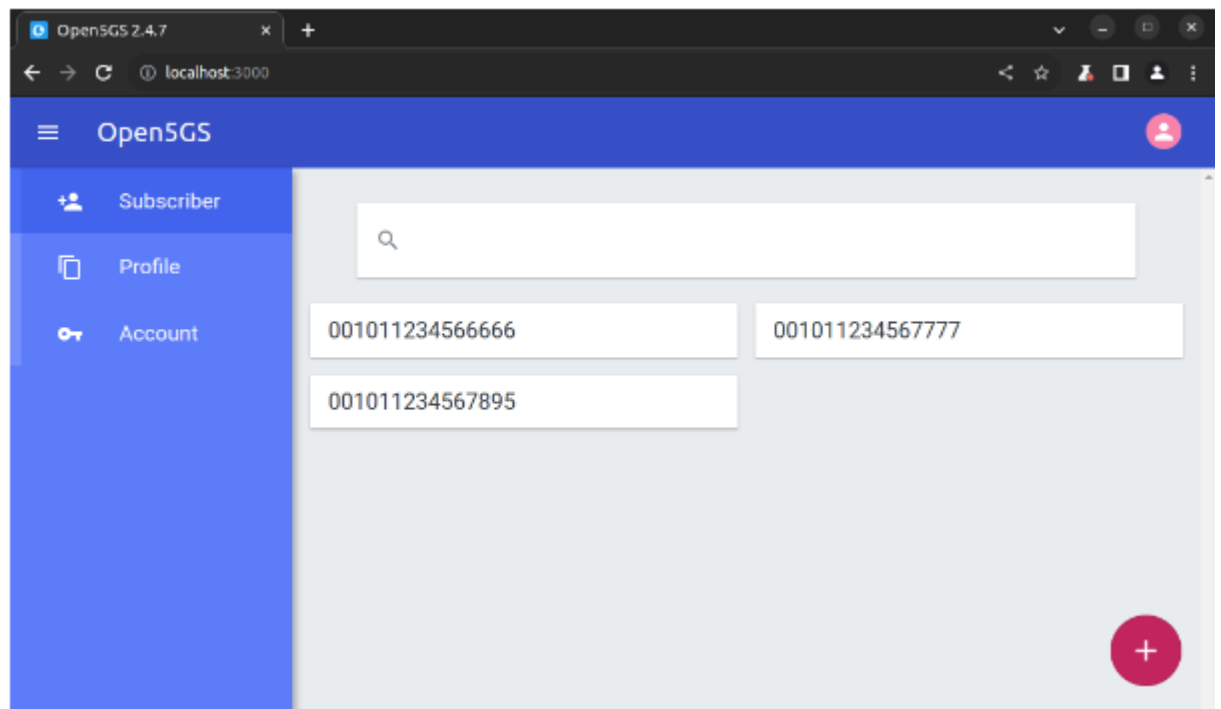
## 9.11 Technical specifications: virtual machine AttackVM.

	<b>General</b>
Name:	AttackVM
Operating System:	Ubuntu (64-bit)
	<b>System</b>
Base Memory:	4096 MB
Processors:	2
Boot Order:	Floppy, Optical, Hard Disk
Acceleration:	VT-x/AMD-V, Nested Paging, KVM Paravirtualization
	<b>Display</b>
Video Memory:	128 MB
Graphics Controller:	VMSVGA
Acceleration:	3D
Remote Desktop Server:	Disabled
Recording:	Disabled
	<b>Storage</b>
Controller:	IDE
IDE Primary Device 0:	[Optical Drive] AttackDetection-disk001.iso (50,59 MB)
Controller:	SATA
SATA Port 0:	AttackDetection-disk002.vdi (Normal, 30,00 GB)
	<b>Audio</b>
Host Driver:	Windows DirectSound
Controller:	ICH AC97
	<b>Network</b>
Adapter 1:	Intel PRO/1000 MT Desktop (NAT)
Adapter 2:	Intel PRO/1000 MT Desktop (Host-only Adapter, 'VirtualBox Host-Only Ethernet Adapter')
	<b>USB</b>
USB Controller:	OHCI, EHCI
Device Filters:	0 (0 active)
	<b>Shared folders</b>
	None
	<b>Description</b>
	None

## 9.12 Network Settings for CoreNetwork and AttackVM.



## 9.13 SUCIs of UEs registered in the Web UI in open5Gs.



## 9.14 Source Code: netfilter\_queue\_intercept.py

```
1  #!/usr/bin/env python3
2
3  # Generic imports
4  import pandas as pd
5  import scapy.all as sp
6  from collections import defaultdict
7  import netfilterqueue as nf
8
9  # Global Variables
10 black_list = [] #List where the known attacker IP's are stored.
11 packets = dict() # Dictionary with 4-tuples as key and payload as value.
12 invalid_req_per_IP = dict() # Dictionary with detected source IP as key
    and number of invalid requests as value.
13 valid_req_per_IP = dict() # Dictionary with detected source IP as key and
    number of valid requests as value.
14 total_req_per_IP = dict() # Dictionary with detected source IP as key and
    number of total requests as value.
15 ratio = 1 / 5
16
17
18 # All the IP's of the 5G core {This is all from the .env file provided}
19 _MONGO_IP='172.22.0.2'
20 _HSS_IP='172.22.0.3'
21 _PCRF_IP='172.22.0.4'
22 _SGWC_IP='172.22.0.5'
23 _SGWU_IP='172.22.0.6'
24 _SMF_IP='172.22.0.7'
25 _UPF_IP='172.22.0.8'
26 _MME_IP='172.22.0.9'
27 _AMF_IP='172.22.0.10'
28 _AUSF_IP='172.22.0.11'
29 _NRF_IP='172.22.0.12'
30 _UDM_IP='172.22.0.13'
31 _UDR_IP='172.22.0.14'
32 _DNS_IP='172.22.0.15'
33 _RTPENGINE_IP='172.22.0.16'
34 _MYSQL_IP='172.22.0.17'
35 _FHSS_IP='172.22.0.18'
36 _ICSCF_IP='172.22.0.19'
37 _SCSCF_IP='172.22.0.20'
38 _PCSCF_IP='172.22.0.21'
39 _SRS_ENB_IP='172.22.0.22'
40 _NR_GNB_IP='172.22.0.60'
41 _NR_UE_IP='172.22.0.70'
42 _OAI_ENB_IP='172.22.0.25'
43 _WEBUI_IP='172.22.0.26'
44 _PCF_IP='172.22.0.27'
45 _NSSF_IP='172.22.0.28'
46 _BSF_IP='172.22.0.29'
47 _ENTITLEMENT_SERVER_IP='172.22.0.30'
48 _OSMOMSC_IP='172.22.0.31'
49 _OSMOHLR_IP='172.22.0.32'
50 _SMSC_IP='172.22.0.33'
51 _5g_core_ips = [_MONGO_IP, _HSS_IP, _PCRF_IP, _SGWC_IP, _SGWU_IP, _SMF_IP,
    _UPF_IP, _MME_IP, _AMF_IP, _AUSF_IP, _NRF_IP, _UDM_IP, _UDR_IP,
    _DNS_IP, _RTPENGINE_IP, _MYSQL_IP, _FHSS_IP, _ICSCF_IP, _SCSCF_IP,
    _PCSCF_IP, _SRS_ENB_IP, _NR_GNB_IP, _NR_UE_IP, _OAI_ENB_IP, _WEBUI_IP,
    _PCF_IP, _NSSF_IP, _BSF_IP, _ENTITLEMENT_SERVER_IP, _OSMOHLR_IP,
    _OSMOMSC_IP, _SMSC_IP]
52
53 def packet_callback (nf_packet):
54     global black_list
55     global packets
```

```

56 global limit
57 global mal_ips_count
58 global valid_req_per_IP
59 global ratio
60 # Because of the Iptables rules set before running this script, assume
    for now:
61 # - The sourceIP of incoming packet is the UDM
62 # - The destinationIP of outgoing packet is the UDM.
63 _coreVM_IP = '192.168.56.102'
64 _attackVM_IP = '192.168.56.101'
65
66 sc_packet = sp.IP(nf_packet.get_payload())
67 _packet_to_UDM = sc_packet['IP'].dst == _coreVM_IP and 'TCP' in
sc_packet
68 _packet_from_UDM = sc_packet['IP'].src == _coreVM_IP and 'TCP' in
sc_packet
69
70 if _packet_to_UDM:
71     _src_ip = sc_packet['IP'].src
72     # Check if the source o the packet is blacklisted, if so drop the
the packet.
73     if _src_ip in black_list:
74         _mal_packets = invalid_req_per_IP[_src_ip]
75         nf_packet.drop()
76
77         # Update Statistics
78         if _src_ip in total_req_per_IP:
79             total_req_per_IP[_src_ip] += 1
80         else:
81             total_req_per_IP[_src_ip] = 1
82
83         # Print message
84         print ("PACKET " + str(_mal_packets) + " BLOCKED FROM " + str(
_src_ip))
85
86     else:
87         # No need to examine the packet further, because we are not
interested in examining the contents
88         # of the requests to the UDM. We are only interested in
examining the contents
89         # of the responses to the UE's for the UDM. So we accpet the
packet.
90         nf_packet.accept()
91
92         # Update statistics
93         # _ip = sc_packet['IP'].src
94         if _src_ip in total_req_per_IP:
95             total_req_per_IP[_src_ip] += 1
96         else:
97             total_req_per_IP[_src_ip] = 1
98
99 elif _packet_from_UDM:
100     # According to the 3GPP-specs, the UDM doesn't send an initial
request to the UE.
101     # This means we can safely assume that the 4-tuple of the TCP
conversation already exists in the dict.
102     # So we are only interested in the payload of the packet (if there
is any).
103     # The payload will tell us if the UDM is responding to a
legitimate UE or not.
104     # If it is not, we are going to check if the this UE already
exceeded the limit of non-valid requests.
105     # If that is the case, the UE will be blacklisted.
106
107     _src_ip, _dst_ip, _src_port, _dst_port = sc_packet['IP'].src,
sc_packet['IP'].dst, sc_packet['TCP'].sport, sc_packet['TCP'].dport

```

```

108     _key = (_src_ip, _dst_ip, _src_port, _dst_port)
109     if _key in packets:
110         _payload = packets[_key]
111         if '"status":' in _payload:
112
113             # Update statistics
114             _mal_ip = sc_packet['IP'].dst
115             if _mal_ip in invalid_req_per_IP:
116                 invalid_req_per_IP[_mal_ip] += 1
117             else:
118                 invalid_req_per_IP[_mal_ip] = 1
119
120             if _mal_ip in total_req_per_IP:
121                 total_req_per_IP[_mal_ip] += 1
122             else:
123                 total_req_per_IP[_mal_ip] = 1
124
125             # Check if the ratio is reached.
126             # It is not always the case that a malicious IP previously
127             # sent valid requests.
128             # So we need to check that first to prevent runtime errors
129
130             _cur_ratio = None
131             if valid_req_per_IP.get(_mal_ip) == None:
132                 _cur_ratio = invalid_req_per_IP[_mal_ip] / 1
133             else:
134                 _cur_ratio = invalid_req_per_IP[_mal_ip] /
135                 valid_req_per_IP[_mal_ip]
136
137             if _cur_ratio >= ratio:
138                 _mal_packets = invalid_req_per_IP[_mal_ip]
139
140                 if _mal_ip not in black_list:
141                     black_list.append(_mal_ip)
142                     print(str(_mal_ip) + " BLACK-LISTED after " + str(
143                         _mal_packets) + " malicious packets, all communication will be blocked
144                         .")
145
146                 nf_packet.drop() # Prevent the packet from reaching
147                 its destination
148                 print ("PACKET " + str(_mal_packets) + " BLOCKED FROM
149                 " + str(_mal_ip))
150             else:
151                 nf_packet.accept() # No reason to block packet, so
152                 accept the packet.
153
154             elif "authenticationVector" in _payload:
155                 # Update statistics
156                 if _dst_ip in valid_req_per_IP:
157                     valid_req_per_IP[_dst_ip] += 1
158                 else:
159                     valid_req_per_IP[_dst_ip] = 1
160
161                 if _dst_ip in total_req_per_IP:
162                     total_req_per_IP[_dst_ip] += 1
163                 else:
164                     total_req_per_IP[_dst_ip] = 1
165
166                 nf_packet.accept() # Packet is a response to a valid
167                 request, so accept the packet.
168             else:
169                 if 'Raw' in sc_packet:
170                     _value = sc_packet['Raw'].load.decode('utf-8', errors=
171                     'ignore')
172                     packets[_key] += _value

```

```

164
165         # Update Statistics
166         if _dst_ip in total_req_per_IP:
167             total_req_per_IP[_dst_ip] += 1
168         else:
169             total_req_per_IP[_dst_ip] = 1
170
171         nf_packet.accept() # No reason to block packet, so accept
the packet.
172     else:
173         packets[_key] = ''
174         nf_packet.accept() # No reason to block packet, so accept the
packet.
175
176         # Update Statistics
177         if _dst_ip in total_req_per_IP:
178             total_req_per_IP[_dst_ip] += 1
179         else:
180             total_req_per_IP[_dst_ip] = 1
181
182     else:
183         nf_packet.accept() # No reason to block packet, so accept the
packet.
184         _dst_ip = sc_packet['IP'].dst
185
186         # Update Statistics
187         if _dst_ip in total_req_per_IP:
188             total_req_per_IP[_dst_ip] += 1
189         else:
190             total_req_per_IP[_dst_ip] = 1
191
192
193 queue = nf.NetfilterQueue()
194 queue.bind(0, packet_callback)
195
196 try:
197     # Start intercepting packets
198     print ('IDS started ... ')
199     queue.run()
200 except KeyboardInterrupt:
201     # Stop the packet interception gracefully if the user interrupts the
program
202     _blacklisted_ips = list(set(black_list))
203     print("\nIDS stopped running, the following IPs are blacklisted: ")
204     for _ip in _blacklisted_ips:
205         print(str(_ip))
206
207 # Cleanup and restore
208 queue.unbind()

```

## 9.15 Source Code: plotting\_udm\_requests.ipynb

```
1 # Generic imports
2 import pandas as pd
3 import plotly
4 import plotly.graph_objects as go
5
6 # 5G visualization logic
7 import trace_plotting
8 import logging
9 import re
```

Listing 1: Code cell 1

```
1 # Wireshark trace with 5GC messages
2 wireshark_trace = 'traces/udm_testing_stage-1_test-2.pcap'
```

Listing 2: Code cell 2

```
1 if isinstance(wireshark_trace, list):
2     output_name_files = wireshark_trace[0]
3 else:
4     output_name_files = wireshark_trace
5 output_name_files = '.'.join(output_name_files.split('.')[0:-1])
6
7 # DEBUG logging level for big traces so that you can see if processing is
8 # stuck or not
9 packets_df = trace_plotting.import_pcap_as_dataframe(
10     wireshark_trace,
11     http2_ports = "32445,5002,5000,32665,80,32077,5006,8080,3000,8081,7777",
12     wireshark_version = 'latest',
13     logging_level=logging.INFO,
14     remove_pdml=True)
```

Listing 3: Code cell 2

```
1 procedure_df, procedure_frames_df = trace_plotting.
2 calculate_procedure_length(packets_df)
```

Listing 4: Code cell 3

```
1 # Get needed columns for own data frame before applying filters:
2 pcap_data_df = procedure_frames_df[['ip_src', 'ip_dst', 'timestamp_offset',
3     'summary_raw', 'HTTP_TYPE']].copy()
```

Listing 5: Code cell 4

```
1 # Filters:
2 _src_is_udm = pcap_data_df['ip_src'] == '172.22.0.13'
3 _dst_is_terminal = pcap_data_df['ip_dst'] == '172.22.0.1'
4 _is_rsp = pcap_data_df['HTTP_TYPE'] == 'rsp'
5 _filter = _src_is_udm & _dst_is_terminal & _is_rsp
6
7 # Apply filters
8 pcap_data_df_filtered_1 = pcap_data_df[_filter]
9 pcap_data_df_filtered = pcap_data_df_filtered_1[pcap_data_df_filtered_1.
10     summary_raw.str.contains('^HTTP/2 [245][0][0134] rsp.', regex=True, na
11     =False)]
```

Listing 6: Code cell 5

```

1 # Apply sanity check.
2 total_req_test_1 = 100
3 total_req_test_2 = 500
4 total_req = total_req_test_2
5 if total_req == pcap_data_df_filtered.shape[0]:
6     print ('Sanity passed!')
7 else:
8     print ('Sanity failed!')

```

Listing 7: Code cell 6

```

1 # Create own column with HTTP response code.
2 # Extract code from summary_raw column with value.split(' ')[1]
3 # Store value in new column.
4
5 def _extract_code(_packet):
6     _summary_raw = _packet['summary_raw']
7     _code = _summary_raw.split(' ')[1]
8     return _code
9
10 pcap_data_df_filtered['HTTP_CODE'] = pcap_data_df_filtered.apply (lambda
    _packet: _extract_code(_packet), axis=1)

```

Listing 8: Code cell 7

```

1 # Plot the http_code of the requests against time.
2 from matplotlib import pyplot as plt
3 import seaborn as sns
4
5 sns.set_style('dark')
6 sns.scatterplot(data=pcap_data_df_filtered, x=pcap_data_df_filtered['
    timestamp_offset'], y=pcap_data_df_filtered['HTTP_CODE'], hue='
    HTTP_CODE', legend='full');

```

Listing 9: Code cell 8

## 9.16 Source Code: detecting\_udm\_attacks.ipynb

```
1 # Generic imports
2 import pandas as pd
3 import plotly
4 import plotly.graph_objects as go
5 from matplotlib import pyplot as plt
6 import seaborn as sns
7
8 # 5G visualization logic
9 import trace_plotting
10 import logging
11 import re
```

Listing 10: Code cell 1

```
1 _MONGO_IP='172.22.0.2'
2 _HSS_IP='172.22.0.3'
3 _PCRF_IP='172.22.0.4'
4 _SGWC_IP='172.22.0.5'
5 _SGWU_IP='172.22.0.6'
6 _SMF_IP='172.22.0.7'
7 _UPF_IP='172.22.0.8'
8 _MME_IP='172.22.0.9'
9 _AMF_IP='172.22.0.10'
10 _AUSF_IP='172.22.0.11'
11 _NRF_IP='172.22.0.12'
12 _UDM_IP='172.22.0.13'
13 _UDR_IP='172.22.0.14'
14 _DNS_IP='172.22.0.15'
15 _RTPENGINE_IP='172.22.0.16'
16 _MYSQL_IP='172.22.0.17'
17 _FHOSS_IP='172.22.0.18'
18 _ICSCF_IP='172.22.0.19'
19 _SCSCF_IP='172.22.0.20'
20 _PCSCF_IP='172.22.0.21'
21 _SRS_ENB_IP='172.22.0.22'
22 _NR_GNB_IP='172.22.0.60'
23 _NR_UE_IP='172.22.0.70'
24 _OAI_ENB_IP='172.22.0.25'
25 _WEBUI_IP='172.22.0.26'
26 _PCF_IP='172.22.0.27'
27 _NSSF_IP='172.22.0.28'
28 _BSF_IP='172.22.0.29'
29 _ENTITLEMENT_SERVER_IP='172.22.0.30'
30 _OSMOMSC_IP='172.22.0.31'
31 _OSMOHLR_IP='172.22.0.32'
32 _SMSC_IP='172.22.0.33'
33 _5g_core_ips = [_MONGO_IP, _HSS_IP, _PCRF_IP, _SGWC_IP, _SGWU_IP, _SMF_IP,
                 _UPF_IP, _MME_IP, _AMF_IP, _AUSF_IP, _NRF_IP, _UDM_IP, _UDR_IP,
                 _DNS_IP, _RTPENGINE_IP, _MYSQL_IP, _FHOSS_IP, _ICSCF_IP, _SCSCF_IP,
                 _PCSCF_IP, _SRS_ENB_IP, _NR_GNB_IP, _NR_UE_IP, _OAI_ENB_IP, _WEBUI_IP,
                 _PCF_IP, _NSSF_IP, _BSF_IP, _ENTITLEMENT_SERVER_IP, _OSMOHLR_IP,
                 _OSMOMSC_IP, _SMSC_IP]
```

Listing 11: Code cell 2

```

1 # Wireshark trace with 5GC messages
2 wireshark_trace = 'traces/udm_testing_stage-2_test-1.pcap'

```

Listing 12: Code cell 3

```

1 if isinstance(wireshark_trace, list):
2     output_name_files = wireshark_trace[0]
3 else:
4     output_name_files = wireshark_trace
5 output_name_files = '.'.join(output_name_files.split('.')[0:-1])
6
7 # DEBUG logging level for big traces so that you can see if processing is
8   stuck or not
9 packets_df = trace_plotting.import_pcap_as_dataframe(
10     wireshark_trace,
11     http2_ports = "32445,5002,5000,32665,80,32077,5006,8080,3000,8081,7777",
12     wireshark_version = 'latest',
13     logging_level=logging.INFO,
14     remove_pdm1=True)

```

Listing 13: Code cell 4

```

1 procedure_df, procedure_frames_df = trace_plotting.
   calculate_procedure_length(packets_df)

```

Listing 14: Code cell 5

```

1 # Get needed columns for own data frame before applying filters:
2 pcap_data_df = procedure_frames_df[['ip_src', 'ip_dst', 'timestamp_offset',
   'summary_raw', 'HTTP_TYPE']].copy()

```

Listing 15: Code cell 6

```

1 # This function extracts other IPs that the UDM communicated with outside
2   the core network.
3 def _filter_ips():
4     _other_ips = []
5     for _ip in pcap_data_df['ip_dst']:
6         if _ip not in _5g_core_ips:
7             _other_ips.append(_ip)
8
9     _result = list(set(_other_ips))
10
11     return _result

```

Listing 16: Code cell 7

```

1 # Filters:
2 _src_is_udm = pcap_data_df['ip_src'] == '172.22.0.13'
3 _dst_outside_5gc = _filter_ips()
4 _dst_is_not_5g = pcap_data_df['ip_dst'].isin(_dst_outside_5gc)
5 _is_rsp = pcap_data_df['HTTP_TYPE'] == 'rsp'
6 _filter = _src_is_udm & _dst_is_not_5g & _is_rsp
7
8 # Apply filter
9 pcap_data_df_filtered_1 = pcap_data_df[_filter]
10 pcap_data_df_filtered = pcap_data_df_filtered_1[pcap_data_df_filtered_1.
   summary_raw.str.contains('^HTTP/2 [245][0][0134] rsp.', regex=True, na
   =False)]

```

Listing 17: Code cell 8

```

1 # Apply sanity check.
2 total_req_test_1 = 1500
3 total_req_test_2 = 1500
4 total_req = total_req_test_2
5 if total_req == pcap_data_df_filtered.shape[0]:
6     print ('Sanity passed!')
7 else:
8     print ('Sanity failed!')

```

Listing 18: Code cell 9

```

1 # Create own column with HTTP response code.
2 # Extract code from summary_raw column with value.split(' ')[1]
3 # Store value in new column.
4
5 def _extract_code(_packet):
6     _summary_raw = _packet['summary_raw']
7     _code = _summary_raw.split(' ')[1]
8     return _code
9
10 pcap_data_df_filtered['HTTP_CODE'] = pcap_data_df_filtered.apply (lambda
    _packet: _extract_code(_packet), axis=1)

```

Listing 19: Code cell 10

```

1 # Here we loop through the UDM requests stored in: pcap_data_df_filtered
2 # First we split the dataframe into batches, the batch size is determined
  by the time passed.
3 # We split the requests with the following logic.
4 _start_time = pcap_data_df_filtered['timestamp_offset'].iloc[0].round()
5 _end_time = pcap_data_df_filtered['timestamp_offset'].iloc[-1].round() +
  1.0
6
7
8 # Sliding window implementation.
9 # Returns a list of dataframes for further analysis.
10 def _sliding_window (df, col, _start, _end):
11     _result = []
12     for i in range(int(_start-1.0), int(_end + 1.0)):
13         _tmp_batch_1 = df[df[col] <= i + 1.0]
14         _tmp_batch_2 = _tmp_batch_1[_tmp_batch_1[col] >= i]
15         _result.append(_tmp_batch_2)
16         _tmp_batch_3 = df[df[col] <= i + 1.5]
17         _tmp_batch_4 = _tmp_batch_3[_tmp_batch_3[col] >= i + 0.5]
18         _result.append(_tmp_batch_4)
19
20     return _result
21
22 batches = _sliding_window(pcap_data_df_filtered, 'timestamp_offset',
    _start_time, _end_time)

```

Listing 20: Code cell 11

```

1 # Loop through the batches produced by sliding window -> Inside the loop:
2 # Check amount of requests.
3 # Check amount of 404 codes
4 # Calculate if the user defined ratio is not exceeded.
5
6 _user_defined_ratio = 1 / 5
7
8 for _batch in batches:
9     if not _batch.empty:
10         _amnt_404 = _batch[_batch.HTTP_CODE == '404'].shape[0]
11         _total = _batch.shape[0]
12         _current_ratio = _amnt_404 / _total
13         if _current_ratio >= _user_defined_ratio:
14             _attackers = _batch['ip_dst'].unique()
15             for _att in _attackers:
16                 print('BREACH INCOMING FROM', _att)

```

Listing 21: Code cell 12

```

1 # Iterate through filtered pcap file.
2 # If response code is 404 then add to new dataframe.
3
4 # We reset the indices of the pcap dataframe.
5 pcap_filtered = pcap_data_df_filtered.reset_index()
6 _probed_pcap = []
7 for _index, _req in pcap_filtered.iterrows():
8     if _req['HTTP_CODE'] == '404':
9         _probed_pcap.append(_req)
10
11 # Convert the result into a data frame.
12 probed_pcap = pd.DataFrame(_probed_pcap)

```

Listing 22: Code cell 13

```

1 # This function renames the Column 'ip_dst' to 'Attacker', because those
2 # IPs are the detected attackers.
3 # Some of the IPs are known, we give them a readable name. IPs that are
4 # unknown will show in the plot as an IP address.
5 def rename_columns(probed_pcap):
6     probed_pcap = probed_pcap.rename(columns={'ip_dst': 'Attacker'})
7
8     if (probed_pcap == '192.168.56.1').any().any():
9         probed_pcap = probed_pcap.replace('192.168.56.1', 'Host Machine')
10
11     if (probed_pcap == '192.168.56.101').any().any():
12         probed_pcap = probed_pcap.replace('192.168.56.101', 'AttackVM')
13
14     if (probed_pcap == '172.22.0.1').any().any():
15         probed_pcap = probed_pcap.replace('172.22.0.1', 'CoreNetwork')
16
17     return probed_pcap

```

Listing 23: Code cell 14

```

1 # Plot2: x-axis:= time_stamp, y-axis: attack_source
2 probed_pcap_renamed = rename_columns(probed_pcap)
3 sns.set_style('dark')
4 sns.scatterplot(data=probed_pcap_renamed, x=probed_pcap_renamed['
    timestamp_offset'], y=probed_pcap_renamed['Attacker'], hue='Attacker',
    legend='full');

```

Listing 24: Code cell 15

## 9.17 Source Code: scapy\_packet\_handling.ipynb

```
1 # All the IP's of the 5G core {This is all from the .env file provided}
2 _MONGO_IP='172.22.0.2'
3 _HSS_IP='172.22.0.3'
4 _PCRF_IP='172.22.0.4'
5 _SGWC_IP='172.22.0.5'
6 _SGWU_IP='172.22.0.6'
7 _SMF_IP='172.22.0.7'
8 _UPF_IP='172.22.0.8'
9 _MME_IP='172.22.0.9'
10 _AMF_IP='172.22.0.10'
11 _AUSF_IP='172.22.0.11'
12 _NRF_IP='172.22.0.12'
13 _UDM_IP='172.22.0.13'
14 _UDR_IP='172.22.0.14'
15 _DNS_IP='172.22.0.15'
16 _RTPENGINE_IP='172.22.0.16'
17 _MYSQL_IP='172.22.0.17'
18 _FHSS_IP='172.22.0.18'
19 _ICSCF_IP='172.22.0.19'
20 _SCSCF_IP='172.22.0.20'
21 _PCSCF_IP='172.22.0.21'
22 _SRS_ENB_IP='172.22.0.22'
23 _NR_GNB_IP='172.22.0.60'
24 _NR_UE_IP='172.22.0.70'
25 _OAI_ENB_IP='172.22.0.25'
26 _WEBUI_IP='172.22.0.26'
27 _PCF_IP='172.22.0.27'
28 _NSSF_IP='172.22.0.28'
29 _BSF_IP='172.22.0.29'
30 _ENTITLEMENT_SERVER_IP='172.22.0.30'
31 _OSMOMSC_IP='172.22.0.31'
32 _OSMOHLR_IP='172.22.0.32'
33 _SMSC_IP='172.22.0.33'
34 _5g_core_ips = [_MONGO_IP, _HSS_IP, _PCRF_IP, _SGWC_IP, _SGWU_IP, _SMF_IP,
    _UPF_IP, _MME_IP, _AMF_IP, _AUSF_IP, _NRF_IP, _UDM_IP, _UDR_IP,
    _DNS_IP, _RTPENGINE_IP, _MYSQL_IP, _FHSS_IP, _ICSCF_IP, _SCSCF_IP,
    _PCSCF_IP, _SRS_ENB_IP, _NR_GNB_IP, _NR_UE_IP, _OAI_ENB_IP, _WEBUI_IP,
    _PCF_IP, _NSSF_IP, _BSF_IP, _ENTITLEMENT_SERVER_IP, _OSMOHLR_IP,
    _OSMOMSC_IP, _SMSC_IP]
```

Listing 25: Code cell 1

```
1 # Generic imports
2 import pandas as pd
3 import scapy.all as sp
4 from collections import defaultdict
```

Listing 26: Code cell 2

```
1 # Wireshark trace with 5GC messages
2 wireshark_trace = 'traces/udm_req_200.pcap'
```

Listing 27: Code cell 3

```
1 # Load the packet capture.
2 # We filter on TCP packets, because that is the conversations we are
  interested in.
3 pcap_packets = sp.sniff(filter='tcp', offline=wireshark_trace)
```

Listing 28: Code cell 4

```

1 # Filter out packets with source-ip or destination-ip of the UDM
2 _udm_packets = []
3
4 for _pac in pcap_packets:
5     if _pac['IP'].src == _UDM_IP or _pac['IP'].src == _UDM_IP:
6         _udm_packets.append(_pac)
7
8 # Now we have to filter the udm communication that was not with a network
9 # function in the 5G Core.
10 # So we have to filter out intra-5G Core communication.
11 # We are going to make a new list that only contains inter-5G
12 # communication with UDM.
13 inter_core_udm_com = []
14
15 # The 4-tuples contain: src.ip, dst.ip, src.port, dst.port
16 # This list will be used to get group a tcp conversations.
17 four_tuples = []
18 for _udm_pac in _udm_packets:
19     if _udm_pac is not None:
20         if _udm_pac['IP'].src not in _5g_core_ips or _udm_pac['IP'].dst
21         not in _5g_core_ips:
22             inter_core_udm_com.append(_udm_pac)
23
24             # Define the 4-tuple:
25             _src_ip, _dst_ip, _src_port, _dst_port = _udm_pac['IP'].src,
26             _udm_pac['IP'].dst, _udm_pac['TCP'].sport, _udm_pac['TCP'].dport
27             _tuple = (_src_ip, _dst_ip, _src_port, _dst_port)
28             four_tuples.append(_tuple)
29
30 # Remove duplicates from the list of 4 tuples.
31 four_tuples = list(set(four_tuples))

```

Listing 29: Code cell 5

```

1 # Here we are going to group the packets on 5 characteristics:
2 # IP.src, IP.dst, TCP.sport, TCP.dport
3 # We already filtered our desired protocol, which TCP.
4
5 # First we make a dictionary with the tuples as key.
6 grouped_packets = dict()
7 for _tuple in four_tuples:
8     grouped_packets.update([(_tuple, [])])
9
10 # Then we iterate through the packets and identify the 4 tuple of the
11 # packet, so we added in the right place in the dictionary.
12 for _pac in inter_core_udm_com:
13     _src_ip, _dst_ip, _src_port, _dst_port = _pac['IP'].src, _pac['IP'].
14     dst, _pac['TCP'].sport, _pac['TCP'].dport
15     _tuple = (_src_ip, _dst_ip, _src_port, _dst_port)
16     # Update dictionary
17     grouped_packets[_tuple].append(_pac)

```

Listing 30: Code cell 6

```

1 # Now we have all the TCP packets that share the same conversation
  attached to the corresponding 4-tuple.
2 # We can now extract the Raw-data of each conversation, we make a
  dedicated dictionary for that.
3 raw_data = dict()
4 for _tuple in four_tuples:
5     raw_data.update([(_tuple, '')])
6
7 # Iterate through the dictionary and extract Raw-data on bytes.
8 # We are decoding the raw data in 'utf-8' and ignoring characters that can
  not be decoded.
9 # This is the approach for now, because the information we are looking for
  is decoded in utf-8.
10 for _4_tuple in grouped_packets:
11     for _pac in grouped_packets.get(_4_tuple):
12         if 'Raw' in _pac:
13             _new_data = _pac['Raw'].load.decode('utf-8', errors='ignore')
14             raw_data[_4_tuple] = raw_data[_4_tuple] + _new_data

```

Listing 31: Code cell 7

```

1 # Now we want to detect the attack ratio for this set of packets.
2 # As we see, if the HTTP-response is 200 (non-attack), then the data
  contains an 'authenticationVector'
3 # And if the HTTP-response is 404 (attack-scenario), then data contains a
  "status: 404" parameter.
4
5 # For now we are going to use the dictionary to extract:
6 # attack / total - ratio and compare it by an user-defined limit
7 # source of attack.
8
9 # To be consistent and efficient we will initialize a new dictionary with:
10 # {'4-tuple' : 'case'}
11 # 'case' -> Is the ip responsible for an attack-scenario? 'attack', 'non-
  attack' string format
12
13 communication = dict()
14 for _tuple in four_tuples:
15     communication.update([(_tuple, '')])
16
17 for _4_tuple in raw_data:
18     _contains_vector = "authenticationVector" in raw_data[_4_tuple]
19     _contains_404 = '"status:"' in raw_data[_4_tuple]
20     if _contains_404:
21         communication[_4_tuple] = 'attack'
22     elif _contains_vector:
23         communication[_4_tuple] = 'non-attack'
24     else:
25         print('Something went wrong...')

```

Listing 32: Code cell 8

```

1 total = len(communication.keys())
2 _user_defined_ratio = 1 / 5
3 _attack = 0
4 for _com in communication:
5     # Check if current conversation is an attack scenario.
6     if communication[_com] == 'attack':
7         _attack = _attack + 1
8
9     # Check at each detected attack check if the limit is exceeded.
10    _cur_ratio = _attack / total
11    if _cur_ratio >= _user_defined_ratio:
12        _attacker = _com[1]
13        print('BREACH INCOMING FROM', _attacker)

```

Listing 33: Code cell 9

## 9.18 Source Code: udm\_testing\_corevm.py

```
1  #!/usr/bin/env python3
2
3  import subprocess
4  import os
5  import sys
6  import random
7
8  _valid_req = None
9  _invalid_req = None
10
11 if len(sys.argv) != 3:
12     print("Usage: ./udm_testing_corevm.py <amount_of_valid_requests> <
13         amount_of_invalid_requests>")
14     sys.exit()
15 else:
16     # Try to convert input to integer.
17     try:
18         _valid_req = int(sys.argv[1])
19         _invalid_req = int(sys.argv[2])
20     except ValueError:
21         print("Usage: ./udm_testing_corevm.py <amount_of_valid_requests> <
22             amount_of_invalid_requests>")
23         sys.exit()
24     pass
25
26 _valid_req_left = _valid_req > 0
27 _invalid_req_left = _invalid_req > 0
28 _choice = "None"
29
30 # Choose randomly which script to execute until one of the given amounts
31 # is exhausted.
32 while _valid_req_left and _invalid_req_left:
33     _choice = random.choice(["attack", "non_attack"])
34
35     if _choice == "attack":
36         _invalid_req -= 1
37         os.system("./attack_udm_original.py 1")
38         _invalid_req_left = _invalid_req > 0
39     elif _choice == "non_attack":
40         _valid_req -= 1
41         os.system("./not_attack_udm_original.py 1")
42         _valid_req_left = _valid_req > 0
43
44 # When one of the given amounts is exhausted, execute the right script
45 # with left amount.
46 if _valid_req_left:
47     os.system("./not_attack_udm_original.py " + str(_valid_req))
48 elif _invalid_req_left:
49     os.system("./attack_udm_original.py " + str(_invalid_req))
```

## 9.19 Source Code: realtime\_capture.ipynb

```
1 # All the IP's of the 5G core {This is all from the .env file provided}
2 _MONGO_IP='172.22.0.2'
3 _HSS_IP='172.22.0.3'
4 _PCRF_IP='172.22.0.4'
5 _SGWC_IP='172.22.0.5'
6 _SGWU_IP='172.22.0.6'
7 _SMF_IP='172.22.0.7'
8 _UPF_IP='172.22.0.8'
9 _MME_IP='172.22.0.9'
10 _AMF_IP='172.22.0.10'
11 _AUSF_IP='172.22.0.11'
12 _NRF_IP='172.22.0.12'
13 _UDM_IP='172.22.0.13'
14 _UDR_IP='172.22.0.14'
15 _DNS_IP='172.22.0.15'
16 _RTPENGINE_IP='172.22.0.16'
17 _MYSQL_IP='172.22.0.17'
18 _FHOSS_IP='172.22.0.18'
19 _ICSCF_IP='172.22.0.19'
20 _SCSCF_IP='172.22.0.20'
21 _PCSCF_IP='172.22.0.21'
22 _SRS_ENB_IP='172.22.0.22'
23 _NR_GNB_IP='172.22.0.60'
24 _NR_UE_IP='172.22.0.70'
25 _OAI_ENB_IP='172.22.0.25'
26 _WEBUI_IP='172.22.0.26'
27 _PCF_IP='172.22.0.27'
28 _NSSF_IP='172.22.0.28'
29 _BSF_IP='172.22.0.29'
30 _ENTITLEMENT_SERVER_IP='172.22.0.30'
31 _OSMOMSC_IP='172.22.0.31'
32 _OSMOHLR_IP='172.22.0.32'
33 _SMSC_IP='172.22.0.33'
34 _5g_core_ips = [_MONGO_IP, _HSS_IP, _PCRF_IP, _SGWC_IP, _SGWU_IP, _SMF_IP,
    _UPF_IP, _MME_IP, _AMF_IP, _AUSF_IP, _NRF_IP, _UDM_IP, _UDR_IP,
    _DNS_IP, _RTPENGINE_IP, _MYSQL_IP, _FHOSS_IP, _ICSCF_IP, _SCSCF_IP,
    _PCSCF_IP, _SRS_ENB_IP, _NR_GNB_IP, _NR_UE_IP, _OAI_ENB_IP, _WEBUI_IP,
    _PCF_IP, _NSSF_IP, _BSF_IP, _ENTITLEMENT_SERVER_IP, _OSMOHLR_IP,
    _OSMOMSC_IP, _SMSC_IP]
```

Listing 34: Code cell 1

```
1 # Generic imports
2 import pandas as pd
3 import scapy.all as sp
4 from collections import defaultdict
5
6 # Global Variables
7 black_list = [] #List where the known attacker IP's are stored.
8 packets = dict() # Dictionary with 4-tuples as key and payload as value.
9 invalid_req_per_IP = dict() # Dictionary with detected source IP as key
    and number of invalid requests as value.
10 valid_req_per_IP = dict() # Dictionary with detected source IP as key and
    number of valid requests as value.
11 ratio = 1 / 5
```

Listing 35: Code cell 2

```

1 def packet_handling (packet):
2     global black_list
3     global packets
4     global limit
5     global mal_ips_count
6     global ratio
7
8     # If packet belongs to the intra-core coommunication, we ignore the
9     packet.
10    if packet['IP'].src in _5g_core_ips and packet['IP'].dst in
11    _5g_core_ips:
12        pass # Ignore packet.
13
14    # If packet is an incoming packet to the UDM.
15    elif packet['IP'].dst == _UDM_IP:
16        # Then we check if the src-IP of the packet is blacklisted.
17        # If that is the case, we need then to block the packet from
18        reaching the UDM.
19        if packet['IP'].src in black_list:
20            #TODO: Implement a way to block the from reaching the UDM.
21            print ('BREACH INCOMING FROM', packet['IP'].src)
22
23            return
24        else:
25            # No need to examine further, because are not interested in
26            examining the contents
27            # of the requests to the UDM. We are only interested in
28            examining the contents
29            # of the responses to the UE's for the UDM.
30            # So we ignore the packet.
31            pass
32
33    # If packet is an outgoing TCP-packet from the UDM.
34    elif packet['IP'].src == _UDM_IP:
35        # According to the API, the UDM doesn't send an initial request to
36        the UE.
37        # This means we can safely assume that the 4-tuple of the TCP
38        conversation already exists in the dict.
39        # So we are only interested in the payload of the packet (if there
40        is any).
41        # The payload will tell us if the UDM is responding to a
42        legitimate UE or not.
43        # If it is not, we are going to check if the this UE already
44        exceeded the limit of non-valid requests.
45        # If that is the case, the UE will be blacklisted.
46
47        _src_ip, _dst_ip, _src_port, _dst_port = packet['IP'].src, packet[
48        'IP'].dst, packet['TCP'].sport, packet['TCP'].dport
49        _key = (_src_ip, _dst_ip, _src_port, _dst_port)
50        if _key in packets:
51            _payload = packets[_key]
52            if '"status":' in _payload:
53                _mal_ip = packet['IP'].dst
54                if _mal_ip in invalid_req_per_IP:
55                    invalid_req_per_IP[_mal_ip] += 1
56                else:
57                    invalid_req_per_IP[_mal_ip] = 1
58
59                # Check if the ratio is reached.
60                # It is not always the case that a malicious IP previously
61                sent valid requests.
62                # So we need to check that first to rprevent runtime errors
63                .
64
65                _cur_ratio = None
66                if valid_req_per_IP.get(_mal_ip) == None:
67                    _cur_ratio = invalid_req_per_IP[_mal_ip] / 1

```

```

54         else:
55             _cur_ratio = invalid_req_per_IP[_mal_ip] /
valid_req_per_IP[_mal_ip]
56
57             if _cur_ratio >= ratio:
58                 black_list.append(_mal_ip)
59
60             elif "authenticationVector" in _payload:
61                 # Add one valid request to this IP
62                 _ip = packet['IP'].dst
63                 if _ip in valid_req_per_IP:
64                     valid_req_per_IP[_ip] += 1
65                 else:
66                     valid_req_per_IP[_ip] = 1
67
68             else:
69                 if 'Raw' in packet:
70                     _value = packet['Raw'].load.decode('utf-8', errors='
ignore')
71                     packets[_key] += _value
72                 else:
73                     pass # Ignore packet.
74             else:
75                 packets[_key] = ''
76         else:
77             pass
78
79 _interface = 'veth73ad079' # UDM interface, this changes at every run if
our Open 5GS instance.
80 sniffer = sp.sniff(iface=_interface, filter='tcp', prn=packet_handling)

```

Listing 36: Code cell 3

```

1 # This function maps the Ip addresses of the Column 'Attacker' to domain
names of the attacker if known.
2 # Some of the IPs are known, we give them a readable name. IPs that are
unknown will show in the plot as an IP address.
3 def rename_columns (results):
4     results = results.rename(columns={'Attacker': 'Attacker'})
5
6     if (results == '192.168.56.1').any().any():
7         results = results.replace('192.168.56.1', 'Host Machine')
8
9     if (results == '192.168.56.101').any().any():
10         results = results.replace('192.168.56.101', 'AttackVM')
11
12     if (results == '172.22.0.1').any().any():
13         results = results.replace('172.22.0.1', 'CoreNetwork')
14
15     return results

```

Listing 37: Code cell 4

```

1 # Results
2
3 # Gather from the valid and invalid requests data the total requests of
  the attackers.
4 total_requests = dict()
5 for _key in invalid_req_per_IP:
6     try:
7         total_requests[_key] = invalid_req_per_IP[_key] + valid_req_per_IP
          [_key]
8     except KeyError:
9         print ('The attacker did not send any valid requests')
10
11 _total_requests_df = pd.DataFrame(list(total_requests.items()), columns=['
    Attacker', 'Total Requests Detected'])
12 _invalid_requests_df = pd.DataFrame(list(invalid_req_per_IP.items()),
    columns=['Attacker', 'Invalid Requests Detected'])
13 _valid_requests_df = pd.DataFrame(list(valid_req_per_IP.items()), columns
    =['Attacker', 'Valid Requests Detected'])
14
15 # Merge the two Dataframes in to one DataFrame
16 _result_1 = pd.merge(_total_requests_df, _invalid_requests_df, on='
    Attacker')
17 _result_2 = pd.merge(_result_1, _valid_requests_df, on='Attacker')
18 # Map the IP addresses into domain names where possible.
19 result = rename_columns(_result_2)

```

Listing 38: Code cell 5

```

1 result

```

Listing 39: Code cell 6

## References

- [1] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirović, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. *CCS*, 14, 2018.
- [2] Fortinet. Mobile security. <https://www.fortinet.com/resources/cyberglossary/mobile-security>.
- [3] Forbes. What is mobile security. <https://www.forbes.com/advisor/business/what-is-mobile-security/>.
- [4] Qiang Tang, Orhan Ermis, Cu D. Nguyen, Alexandre De Oliveira, and Alain Hirtzig. A systematic analysis of 5g networks with a focus on 5g core security. *IEEE Access*, 10:18298–18319, 2022.
- [5] Zhihong Tian, Yanbin Sun, Shen Su, Mohan Li, Xiaojiang Du, and Mohsen Guizani. Automated attack and defense framework for 5g security on physical and logical layers. 2019.
- [6] Gerrit Holtrup, William Lacube, ; Dimitri, Percia David, Alain Mermoud, ; G  r  me Bovet, and Vincent Lenders. 5g system security analysis. 2021.
- [7] Microsoft. Microsoft threat modeling tool threats. <https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats>.
- [8] Rajaneesh Sudhakar Shetty. *5G Mobile Core Network: Design, Deployment, Automation, and Testing Strategies*. Springer, 1 2021.
- [9] TSGC. Ts 129 503 - v15.2.1 - 5g; 5g system; unified data management services; stage 3 (3gpp ts 29.503 version 15.2.1 release 15), 2019.
- [10] Gheyath Mustafa Zebari, Dilovan Asaad Zebari, and Adel Al-zebari. Fundamentals of 5g cellular networks: A review — journal of information technology and informatics. *Journal of Information Technology and Informatics (JITI)*, 1:1–5, 4 2021.
- [11] Zhichao Zeng and Hong Zhang. Research on lightweight core network solutions for 5g private networks. pages 349–355. Association for Computing Machinery, 5 2023.
- [12] Mehdi Bahrami, Mohammad Bahrami, Booshehr Branch, Iran Bahrami, and ; Shayan. An overview to software architecture in intrusion detection system. *International Journal of Soft Computing And Software Engineering (JSCSE)*, 1:2251–7545, 2011.
- [13] Nancy Agarwal and Syed Zeeshan Hussain. A closer look at intrusion detection system for web applications. 2018.
- [14] Buse Gul Atli and Alexander Jung. Online feature ranking for intrusion detection systems. 6 2018.
- [15] Jaydip Sen. An agent-based intrusion detection system for local area networks. *International Journal of Communication Networks and Information Security (IJCNIS)*, 2, 2010.
- [16] Microsoft. Make the everyday easier. <https://www.microsoft.com/en-us/windows/windows-11>.
- [17] Oracle. Virtualbox. <https://www.virtualbox.org/>.

- [18] Canonical Ltd. Modern enterprise open source. <https://ubuntu.com/>.
- [19] Docker Inc. Build secure software from the start. <https://www.docker.com/>.
- [20] Open5GS. Open5gs. <https://open5gs.org/>.
- [21] Rusty Russell. iptables(8) - linux man page. <https://linux.die.net/man/8/iptables>.
- [22] Scapy. Python scapy. <https://scapy.net/>.
- [23] PyPI. Netfilterqueue 0.1. <https://pypi.org/project/NetfilterQueue/0.1/>.
- [24] Taha Hammouchi. Attack detection 5g core. <https://gitlab.science.ru.nl/thammouchi/attack-detection-5g-core>.
- [25] Telekom. 5g-trace-visualizer. <https://github.com/telekom/5g-trace-visualizer>.
- [26] IPython. The jupyter notebook. <https://ipython.org/notebook.html>.
- [27] IETF. Transmission control protocol. <https://www.ietf.org/rfc/rfc793.txt>.
- [28] Volker Kleinfeld, Göran Hall, and Amarisa Robison. Indirect communication for service-based architecture in 5g core. Technical report.
- [29] Rajaneesh Shetty, Anil Jangam, and Ananya Simlai. Intelligent strategies for overload detection handling for 5g network. pages 135–140. Institute of Electrical and Electronics Engineers Inc., 2021.
- [30] Nicholas J Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A Olsson. A methodology for testing intrusion detection systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 22:719, 1996.
- [31] Seungchan Woo, Jaehyoung Park, Soonhong Kwon, Kyungmin Park, Jonghyun Kim, and Jong-Hyouk Lee. Simulation of data hijacking attacks for a 5g-advanced core network. pages 538–542. IEEE, 6 2023.
- [32] Fatima Salahdine, Tao Han, and Ning Zhang. Security in 5g and beyond recent advances and future challenges. *SECURITY AND PRIVACY*, 6, 1 2023.