

Thesis for the Degree of Bachelor of Science in Engineering –  
Computer Network Engineering, 15.0 credits

# EVALUATING ZEEK AND SURICATA FOR INTRUSION DETECTION IN 5G CORE NETWORKS

Jonas Sjöström  
[jsm20004@student.mdu.se](mailto:jsm20004@student.mdu.se)  
Lars Körnings  
[lks19001@student.mdu.se](mailto:lks19001@student.mdu.se)

Examiner: Johan Åkerberg  
Mälardalen University, Västerås, Sweden

Supervisor: Maryam Vahabi  
Mälardalen University, Västerås, Sweden

Company Supervisor: Hans Byström  
Ericsson, Stockholm, Sweden

## Abstract

As organizations rely more on digital infrastructure, protecting their networks has become a top priority as cyber threats continue to grow. To manage the vast amount of data and potential risks, organizations use monitoring tools like Zeek and Suricata to detect and respond to suspicious network activity.

This thesis explores the potential of Zeek and Suricata to improve incident monitoring in a 5G Core Network (5GCN). Specifically, we assess their effectiveness in monitoring the N4 and Service Based Interfaces, focusing on detecting Denial-of-Service attacks via Packet Forwarding Control Protocol (PFCP) session deletion requests and TLS handshake failure events caused by protocol version mismatch, respectively. To achieve this, we conduct an experiment in a simulated 5GCN that evaluates the ability of Zeek and Suricata to detect these incidents, aiming to provide valuable insights into improving network visibility and incident detection.

Our results show that both Zeek and Suricata can effectively detect TLS handshake failures due to protocol version mismatch. However, neither tool can directly identify incidents on the PFCP protocol, as their base versions lack built-in support for it. Instead, they detect only the underlying UDP traffic. We concluded that the detection capabilities of these tools depend on the protocols involved in the incident. While common protocols like TLS allow for sufficient monitoring, specialized protocols like PFCP offer limited detection when incidents occur at or above the TCP/UDP layer.

## Table of Contents

<b>1. Introduction .....</b>	<b>7</b>
<b>2. Background.....</b>	<b>9</b>
2.1. <i>Zeek</i> .....	9
2.2. <i>Suricata</i> .....	9
2.3. <i>3rd Generation Partnership Project</i> .....	9
2.3.1. 5G .....	10
2.3.2. 5G Core.....	10
2.3.3. N4 Interface .....	11
2.3.4. Session Management Function .....	11
2.3.5. User Plane Function .....	11
2.3.6. Packet Forwarding Control Protocol.....	11
2.3.7. Service-Based Interface .....	11
2.3.8. Network Repository Function.....	12
2.4. <i>Free5GC</i> .....	12
2.4.1. UERANSIM.....	12
2.5. <i>TCPdump</i> .....	12
2.6. <i>Transport Layer Security</i> .....	13
2.6.1. JA3.....	14
2.7. <i>Scapy</i> .....	14
<b>3. Related work.....</b>	<b>15</b>
3.1. <i>Security Analysis of Critical 5G Interfaces</i> .....	15
3.2. <i>Threatening the 5GC via PFCP DoS attacks: the case of blocking UAV communications..</i> 15	15
3.3. <i>Which open-source IDS? Snort, Suricata or Zeek?</i> .....	15

---

<b>4. Problem formulation.....</b>	<b>16</b>
<b>5. Method.....</b>	<b>17</b>
<b>6. Ethical and Societal Considerations .....</b>	<b>18</b>
<b>7. Experimental setup and execution.....</b>	<b>19</b>
7.1. <i>Setting up the virtual machine .....</i>	19
7.2. <i>Setting up and configuring 5GCN.....</i>	19
7.3. <i>Installing and configuring Zeek and Suricata .....</i>	19
7.4. <i>Simplified threat analysis.....</i>	19
7.4.1.     PFCP session deletion request.....	20
7.4.2.     TLS handshake failure .....	20
7.5. <i>Executing the attacks and analyzing them.....</i>	21
7.6. <i>Comparison with background traffic and successful TLS handshakes .....</i>	22
<b>8. Results .....</b>	<b>23</b>
8.1. <i>TLS handshake failure attack.....</i>	23
8.1.1.     Zeek .....	23
8.1.2.     Suricata .....	24
8.2. <i>PFCP session deletion request attack.....</i>	26
8.2.1.     Zeek .....	26
8.2.2.     Suricata .....	26
8.3. <i>Analysis.....</i>	27
8.3.1.     TLS handshake failure attack .....	27
8.3.2.     PFCP session deletion request attack .....	29

---

<b>9. Discussion</b>	30
<b>10. Conclusions</b>	31
<b>11. Future Work</b>	32
<b>References</b>	33
<b>Appendices</b>	36
<i>A – Setting up and configuring 5GCN</i>	36
<i>B – Installation commands for Zeek and Suricata</i>	38
<i>C – Modifications made to the suricata.yaml file</i>	39
<i>D – TLS handshake failure attack script</i>	40
<i>E – Configuration changes made to the NRF file</i>	41
<i>F – PFCP session deletion request attack script</i>	42
<i>G – Mirror container script</i>	43
<i>H – The execution process</i>	44
<i>I – Simulate background traffic script</i>	45
<i>J – Successful TLS handshake script</i>	47

## List of Figures

Figure 1 – Overview of the 5GS .....	10
Figure 2 - The 5GC architecture .....	10
Figure 3 - The full execution process of both attacks .....	21

## List of Tables

Table 1 – Log event in conn.log describing the relevant fields and their values captured from the TLS attack script .....	23
Table 2 – Describes the meaning of each letter in the history field from the conn.log file .....	24
Table 3 – Log event in ssl.log describing the relevant fields and their values captured from the TLS attack script .....	24
Table 4 – Log event in eve.json describing the relevant fields for the event_type frame and their values captured from the TLS attack script .....	25
Table 5 - Log event in eve.json describing the relevant fields for the event_type tls and their values captured from the TLS attack script.....	25
Table 6 - Log events in eve.json describing the relevant fields for the event_type flow and their values captured from the TLS attack script .....	25
Table 7 - Log event in conn.log describing the relevant fields and their values captured from the PFCP attack script .....	26
Table 8 – Log events in eve.json describing the relevant fields and their values captured from the PFCP attack script .....	27

## 1. Introduction

Organizations are becoming increasingly reliant on digital infrastructure, making the protection of these networks a top priority. As cyber threats become more complex, the need for effective network security approaches grows. To manage the sheer volume of data and potential threats, organizations rely on powerful monitoring tools like Zeek and Suricata. These tools provide security teams with detailed insights into network activity, allowing them to detect and respond to suspicious behavior [1], [2]. Zeek is well known for its ability to monitor network traffic and detect malicious behavior, while Suricata excels in high-performance network intrusion detection and prevention. Together, these tools enable security teams to maintain visibility into network behavior and respond quickly to emerging threats.

This thesis investigates the potential to improve incident monitoring in a 5G Core Network (5GCN) by using the tools Zeek and Suricata. We chose these tools due to their comprehensive documentation and the extensive available online resources. We aim to evaluate the effectiveness of these tools in detecting incidents on the N4 and Service Based Interfaces (SBI) within a 5GCN, with a primary focus on Denial-of-Service (DoS) attacks via Packet Forwarding Control Protocol (PFCP) session deletion requests and Transport Layer Security (TLS) handshake failure events due to protocol version mismatch, respectively. We chose these incidents for their ease of execution and potential impact on the 5GCN. Unauthorized TLS traffic might serve as an entry point to compromise a network function, while the PFCP DoS attacks may prevent the end user from being routed through the 5GCN. This evaluation is intended for administrators of 5GCNs, offering valuable insights into how Zeek and Suricata can enhance network visibility and improve incident detection methods.

To effectively evaluate these tools within a 5GCN, we will use scientific methods: literature review and experimentation. The literature review provides the necessary knowledge to conduct the experiment by helping us understand the tools and environment we are setting up. As for the experiment, we evaluate the detection capabilities of Zeek and Suricata inside a simulation tool that emulates a 5GCN. The focus is specifically on their ability to identify the incidents DoS attacks via PFCP session deletion requests on the N4 interface and TLS handshake failure events caused by protocol version mismatch on the SBI interfaces.

The results show that the ability to which Zeek and Suricata can monitor the SBI and N4 interfaces in a 5GCN depends on the protocol involved in the incident. Common protocols like TLS allow for adequate detection, while specialized protocols such as PFCP provide limited detection at or above the TCP/UDP layer. However, these results assume that the experimental environment has not fully implemented all the security recommendations from the 3rd Generation Partnership Project (3GPP). Implementing these security recommendations in the monitored environment could potentially affect the experimental result.

---

We have derived these results by building upon previous related work, including [3], where the authors reviewed the security measures recommended by various standards and the research literature for critical 5G System interfaces. Additionally, they analyzed the potential vulnerabilities and threats each interface could face if proper security measures were not implemented. Another key study we based our work on is [4], in which the authors designed and conducted various DoS attacks using the PFCP protocol, assessing their impact on communication between two simulated drone swarms. Lastly, [5] offers a comprehensive performance analysis of Snort, Suricata, and Zeek, examining the impact of the packet capturing module and pattern-matching algorithm on hardware utilization and traffic throughput.

The thesis begins with an introduction to Zeek and Suricata, the 3GPP and its subdivisions, the Free5GC environment used for our experiment, and the TLS protocol. This is followed by comparison with earlier publications, the formulation of our research questions, and the associated method. It then continues with a discussion of the ethical and social concerns of our thesis, the execution of our experiment, and presentation, explanation and analysis of the results. The thesis concludes with a discussion of the implications of the results, the conclusions drawn from the experiment, and suggestions for improvements that could lead to new research.

## 2. Background

In this section, we present the essential and fundamental knowledge related to the subject we will investigate in this study. This includes insights into the network security tools Zeek and Suricata, 3GPP and its associated subdivisions, the Free5GC environment we used to conduct our experiment, and TLS protocol.

### 2.1. Zeek

Zeek is a passive, open-source network traffic analyzer widely used as a Network Security Monitor (NSM) [1]. It helps operators investigate suspicious or malicious activities while also supporting various traffic analysis tasks beyond security, such as performance monitoring and troubleshooting. This is accomplished through its ability to generate comprehensive logs that accurately capture network activity. These logs include a thorough record of every observed connection, along with application-layer details.

Zeek runs on commodity hardware, making it a cost-effective alternative to proprietary solutions [1]. Unlike traditional signature-based Intrusion Detection Systems (IDS), the scripting language of Zeek enables more advanced techniques for detecting malicious activity, such as anomaly detection and behavioral analysis.

### 2.2. Suricata

Suricata is a free and open-source high-performance network threat detection engine. It emphasizes security, usability, and efficiency driven by a fast-evolving, community-led development model [2]. With capabilities in real-time IDS, inline intrusion prevention, NSM, and offline packet capture (PCAP) analysis, Suricata provides extensive network analysis functionality. It accomplishes this by utilizing a comprehensive signature language to identify known threats, policy violations, malicious behavior, and traffic anomalies based on various rulesets during inspection [6]. This includes capturing and storing TLS certificates through the TLS parser in Suricata, enabling analysis of most aspects of TLS exchanges directly within the ruleset language. With full PCAP capture support, it offers seamless analysis capabilities, making it an integral part of any NSM ecosystem.

### 2.3. 3rd Generation Partnership Project

3GPP brings together seven telecommunications standards development organizations (ARIB, ATIS, CCSA, ETSI, TSDSI, TTA, and TTC) referred to as "Organizational Partners" [7]. These partners offer their members a collaborative environment to create the reports and specifications that define 3GPP technologies. 3GPP specifications include cellular telecommunications technologies such as radio access, core networks, and service capabilities, providing a complete system framework for mobile communications.

---

### 2.3.1. 5G

The Fifth Generation of Mobile Telephony, known as 5G, is one of the technologies defined by 3GPP [8]. It was functionally finalized in June 2018 and fully specified by September 2019. The 5G system is structured similarly to previous generations, using the same core elements: User Equipment (UE), which includes a Mobile Station and a USIM, the Radio Access Network (NG-RAN), and the 5GCN, as shown in Figure 1.

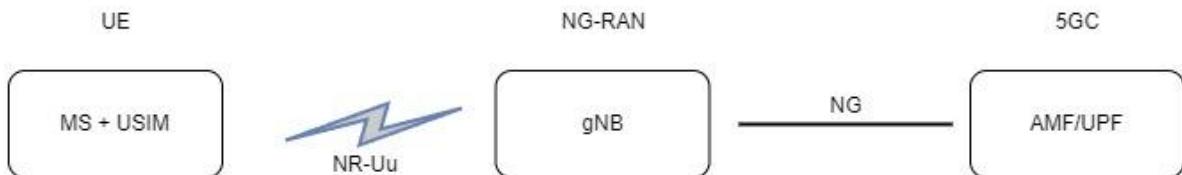


Figure 1 – Overview of the 5GS.

### 2.3.2. 5G Core

The 5GC architecture is based on a "Service-Based Architecture" (SBA) framework, where components are defined as "Network Functions" (NFs) rather than traditional network entities [9]. Through interfaces within a common framework, each NF provides services to other authorized NFs or any "consumers" allowed to access them. This SBA approach enhances modularity and reusability. Figure 2 shows the basic 5GC architecture.

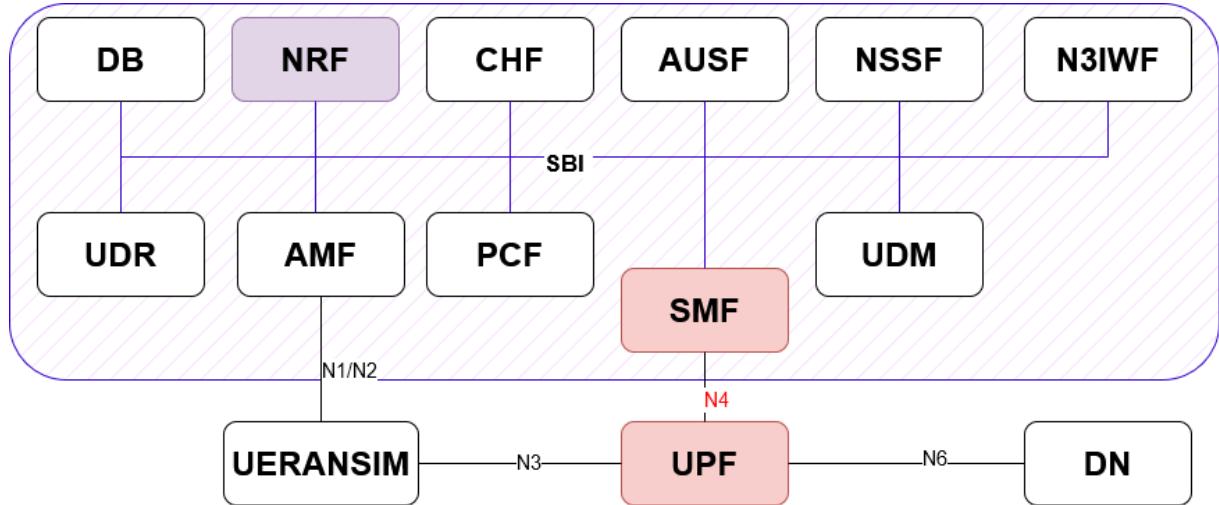


Figure 2 - The 5GC architecture.

In the following sections, we will discuss the highlighted NFs and their corresponding interfaces that are relevant to this thesis.

### 2.3.3. N4 Interface

The N4 interface connects the Session Management Function (SMF) and the User Plane Function (UPF) in the 5GCN [3]. It plays a key role in managing Protocol Data Unit (PDU) sessions, directing traffic to the UPF, and reporting events like PDU establishment to the SMF. Additionally, it transmits lawful intercept targets and packet filtering/forwarding templates.

### 2.3.4. Session Management Function

The SMF manages session management, coordination, and the selection of the UPF serving the UE. It assigns IP-addresses during the setup of the PDU session for UEs [3]. It also supplies the UPF with the necessary Quality of Service (QoS) parameters. While the SMF connects to multiple interfaces, we will specifically focus on the N4 interface in this thesis.

### 2.3.5. User Plane Function

The UPF is a key part in data transmission within the 5GCN, connecting to the Data Network (DN) in the overall 5G architecture [11]. As a central NF within the 5GCN, the UPF manages critical data processing tasks such as packet routing and forwarding, packet inspection, and QoS management. The UPF has four key interfaces: N3, N4, N6, and N9. However, in this study we will focus on the N4 interface. The free5GC implements the UPF in two components: the control plane and the user plane. The control plane is managed by GO-UPF for the N4 interface, while the user plane is managed by GTP5G for the N3, N6, and N9 interfaces. GO-UPF configures the setting for GTP5G, a crucial communication protocol within the 5GCN responsible for transferring user data between the UE and the DN.

### 2.3.6. Packet Forwarding Control Protocol

PFCP is a protocol for communication between the SMF and UPF in the 5GCN [10]. It manages the setup and reconfiguration of user sessions, including deletion. The PFCP protocol operates on top of the UDP protocol on port 8805.

### 2.3.7. Service-Based Interface

The SBI interfaces enable communication between NFs within the 5GCN through an API connection [3]. At the application layer, all communication over these interfaces uses RESTful APIs, employing Hypertext Transfer Protocol version 2 with JavaScript Object Notation (JSON) methods [12]. An NF producer uses the SBI interfaces to provide services to authorized NF consumers [3]. Each service is made up of a service producer (server) and a service consumer (client), with each service focused on carrying out a specific function [12].

---

In general, any NF consumer can access a service, provided they have the necessary permissions. This design promotes straightforward extensibility and reuse.

### 2.3.8. Network Repository Function

The Network Repository Function (NRF) is a NF within the 5GCN, responsible for managing the discovery of NRF services and their endpoint addresses through the NRF bootstrapping service [13]. It also supports service discovery by handling NF Discovery Requests from NF instances and providing information about other discovered NF instances to the requesting NF. Additionally, the NRF stores profiles of available NF instances and the services they support. Furthermore, it provides notifications to subscribed NF service consumers regarding newly registered, updated, or deregistered NF instances along with their potential NF services.

## 2.4. Free5GC

Free5GC, the world's first open-source 5GCN software, was developed in 2019 under the Linux Foundation and focuses on the development of core networks for 5G mobile systems [14], [15]. Its primary goal is to implement the 5GCN as outlined in 3GPP Release 15 and beyond. The main contributors to the project are currently from National Yang Ming Chiao Tung University. Free5GC was initially designed to provide resources for academic research in communications networks. However, the software's compatibility with any mobile phone or base station adhering to the international 5G standard has made it widely adopted for product testing and proof-of-concept trials across the global industry.

### 2.4.1. UERANSIM

UERANSIM is an advanced open-source simulator for 5G UE and RAN [16]. In simple terms, it can be seen as a simulation for a 5G mobile phone (UE) and a base station (gNodeB). The project is designed for testing the 5GCN and exploring the 5G System. UERANSIM is also the world's first and only open-source implementation of 5G Standalone UE and gNodeB.

## 2.5. TCPdump

TCPdump provides a description of the contents of packets on a network interface that match a given Boolean expression [17]. Each description is preceded by a timestamp, which by default is displayed as hours, minutes, seconds, and fractions of a second. TCPdump can also be executed with the `-w` flag to save the packet data to a file for later analysis. Additionally, using the `-i` flag will allow it to capture packets on the specific interface listed.

## 2.6. Transport Layer Security

The main purpose of TLS is to establish a secure communication channel between two peers, ensuring authentication, confidentiality, and integrity [18]. For authentication, the server is always authenticated, while client authentication is optional. In terms of confidentiality, data exchanged over the channel after establishment is accessible only to the communicating endpoints. While TLS does not conceal the length of transmitted data, it allows endpoints to pad TLS records to obscure lengths, which provides better protection against traffic analysis. Finally, regarding integrity, data transmitted through the channel is protected against unauthorized modifications, ensuring that any tampering by attackers is detectable.

TLS consists of two main components, a handshake protocol and a record protocol [18]. However, this thesis focuses on the handshake protocol, which verifies the identities of the communicating parties, determines cryptographic modes and parameters, and generates shared keying material. It is designed to resist tampering, ensuring that an active attacker cannot coerce the peers into negotiating different parameters than they would agree to if the connection were not under attack. A TLS handshake occurs whenever a user accesses a website over HTTPS or during other HTTPS communications, such as API calls [19]. The TLS version 1.3 handshake process consists of the following steps:

- Client Hello – The client initiates communication by sending a "client hello" message to the server. This includes the protocol version, a randomly generated client value (client random), and a list of supported cipher suites. Additionally, the client hello message contains the parameters needed for calculating the premaster secret.
- Server Generates Master Secret - After receiving the client hello, the server uses its own random value (server random) to compute the master secret.
- Server Hello and "Finished" - The server responds with a "server hello" message, which includes its certificate, digital signature, server random, and the selected cipher suite. With the master secret already computed, the server also sends a "Finished" message.
- Final steps and client "Finished" - The client validates the server's certificate and digital signature. It then generates the master secret and finalizes the TLS handshake by sending the "Finished" message, thereby establishing secure symmetric encryption.

A TLS handshake takes place after a TCP connection has been established through a TCP handshake. TCP supports two types of connection releases: abrupt connection release and graceful connection release [20]. An abrupt connection release occurs when a Reset (RST) segment is sent. This segment forces the termination of the connection, either by one TCP entity or when one user closes both directions of data transfer. For a graceful connection release, the connection remains open until both parties have properly closed their respective

---

sides. A graceful connection release happens when a packet with the FIN bit set is sent, which allows each host to independently release its side of the connection. This is done through the Four-way FIN handshake [21]. It consists of the following steps:

1. It starts with either the client or the server sending the FIN flag to request the termination of the connection.
2. The recipient of the FIN flag will then respond by sending an ACK flag as an acknowledgment of the termination request.
3. Next, it will send its own FIN flag to signal the closure to the other side.
4. Finally, the side that receives the last FIN flag will send an ACK flag as the final acknowledgment of the proposed connection closure.

### 2.6.1. JA3

JA3 is a technique for fingerprinting TLS clients by analyzing specific fields in the client hello packet [22]. It extracts the decimal values of the bytes for the following fields: version, accepted ciphers, extensions, elliptic curves, and elliptic curve formats. These values are concatenated in order, with fields separated by commas and values within each field separated by hyphens. The resulting string is then hashed using MD5 to generate a compact, shareable 32-character fingerprint. After creating JA3, a similar approach is applied to fingerprint the server side of the TLS handshake by extracting the same decimal values of the bytes from the server hello packet.

### 2.7. Scapy

Scapy is a Python-based network library that can sniff and dissect Ethernet packets, as well as create its own custom packets, allowing users to scan, probe, and attack networks [23]. With support for a long list of protocols, Scapy claims to be able to replace a variety of other specialized programs. The strength of Scapy lies in its very fine granularity, where you can see every part of a packet and modify it in a free-form manner. In this thesis we use one of the modules in Scapy to write a program to DoS the UPFs PFCP server.

### 3. Related work

In this section, we aim to address some of the threats identified in earlier publications and explore whether these threats can be mitigated in a 5GCN, thereby bridging the gap between 5G security research, IDS research, and real-world applications. Specifically, we have analyzed three relevant papers [3], [4], and [5], in order to find the most prevalent threats in a 5GCN, while considering the main objective and methodologies on these research works.

#### 3.1. Security Analysis of Critical 5G Interfaces

To get a reference for what threats are prevalent in a 5GCN, we have looked at the research paper [3]. It is a comprehensive analysis of threats based on the interface and protocols used in the 5G network. Their analysis is based on the STRIDE model, which is an industry standard threat analysis model developed by Microsoft to identify security issues on a design level. For our thesis, we used the threats they identified as prevalent on the SBI and N4 interfaces as the basis for evaluating Zeek and Suricata.

#### 3.2. Threatening the 5GC via PFCP DoS attacks: the case of blocking UAV communications

In [4], the authors created and executed a variety of DoS attacks using the PFCP protocol and evaluated the impact of these attacks on communication between two simulated drone swarms. In their discussion, they suggest mitigating some of the security shortcomings by cross-referencing NF logs. In our work, we plan to implement a similar DoS attack, which we are confident will be effective based on the findings in [4]. However, instead of focusing on their suggestion to cross-reference NF logs, we examine how such attacks can be detected at the network layer. Since both our and their experiments were conducted in a simulated environment, our thesis and their paper share several similarities in testing methodology. The key difference is that we focus on detecting attacks at the network layer, and our thesis is not limited to just PFCP DoS attacks.

#### 3.3. Which open-source IDS? Snort, Suricata or Zeek?

In [5], they did comprehensive performance analysis of Snort, Suricata and Zeek. In their testing, they focused on how the packet capturing module and pattern matching algorithm impacted both the hardware requirement, utilization and traffic throughput. They found that different capturing engines had a noticeable but limited impact on hardware utilization with the biggest difference being that some older versions had some problem with packet loss. In their work they do not look at any attacks, triggered rules or event detection. In comparison we focus on specific attacks and their detection.

---

## 4. Problem formulation

This thesis explores the potential for improving incident monitoring in a 5GCN using the tools Zeek and Suricata. We focus on evaluating how effectively Zeek and Suricata can identify incidents on the N4 and SBI interfaces in a 5GCN, focusing specifically on DoS attacks via PFCP session deletion request and TLS handshake failure events caused by protocol version mismatch, respectively. By demonstrating how Zeek and Suricata can detect these incidents in a 5GCN, we seek to highlight the challenges associated with the network's increased complexity and identify more effective methods for incident detection. We hope to do this by answering the following questions:

RQ1: To what extent can Zeek and Suricata monitor the N4 and SBI interfaces in a 5G core network?

RQ1.1: Can Zeek and Suricata provide the necessary detection for DoS attacks via PFCP session deletion requests on the N4 interface?

RQ1.2: Can Zeek and Suricata provide the necessary detection for TLS handshake failures on the SBI interfaces?

We chose to have one main question and two sub-questions, as the sub-questions serve as metrics to evaluate the main question. These research questions are derived from the simplified threat analysis we conducted, as outlined in Section 7.4. Research question 1 will be answered based on the results of sub-questions 1.1 and 1.2.

The main objective is to assess the detection capabilities of Zeek and Suricata within a 5GCN. However, this study has several limitations that may affect the results and the conclusions drawn. Firstly, the test environment may not fully reflect all possible real-world network conditions, which could impact the validity of the results. Secondly, the scope and depth of our analysis and conclusions are limited by time constraints as there are several other threats that we were not able to explore. Additionally, we only evaluate the base versions of these tools and do not consider any add-ons or extensions. Lastly, our analysis and conclusions are based solely on the tools Zeek and Suricata, and may not apply to other security tools, such as Snort.

## 5. Method

In this thesis, we employ scientific methods of literature review and experimentation. Both methods will be carried out in tandem due to the scarcity of literature in this field and the extensive setup required for our experimental environment. This is because working on both methods side by side can give us a fuller picture of the research problem, blending theoretical insights with real-world findings.

The literature review is conducted to establish the foundational knowledge necessary for understanding the tools and environment that we set up in the experiment. This involves reviewing related works for Zeek, Suricata and 5GCNs, as well as their official documentation. The experiment will be designed based on insights from the literature review, focusing on deploying Zeek and Suricata as functional tools in a simulated 5GCN for evaluation.

Engineers often use the research method experiment when investigating how a technical system or process functions under different circumstances [24]. The technical system or process under investigation can be analyzed based on one or more variables. This method is particularly useful when the goal is to identify cause-and-effect relationships between variables. In an experiment, the independent variable represents the cause, while the dependent variable reflects the effect. In this thesis, the independent variables are the PCAP-files scanned into Zeek and Suricata and the dependent variables are the SBI and N4 interfaces.

## 6. Ethical and Societal Considerations

In this thesis we explore how the tools Zeek and Suricata can be used to protect and monitor traffic going through and into a 5GCN. Since we are generating our own data solely for the experiment, data collection is not a concern for us. However, if our experiment were implemented in the real world, we can see some privacy concerns around interfaces where public/customer data is transported. Since the other interfaces are for administration of the base station, monitoring those have no ethical concerns as far as we can see.

There are some further concerns about how the conclusions in this thesis can be used to point out novel ways to track and monitor users on the network with either good or bad intentions. No matter the intentions, we believe it is better to raise awareness about the potential uses, both positive and negative, to hopefully open doors to mitigate the harmful effects and make the most of the beneficial ones.

## 7. Experimental setup and execution

This section details the hardware, software, and their configurations used in our experiment. It also describes the virtual environment setup for the 5GCN, the chosen interfaces and their corresponding monitored incidents, as well as their execution and comparison with normal behavior.

### 7.1. Setting up the virtual machine

We set up the 5GCN on a virtual machine running the Xubuntu distribution of Linux. The virtual machine is allocated 8 cores, 32 GB memory and 100 GB of storage.

### 7.2. Setting up and configuring 5GCN

To conduct our experiment, we need to set up a virtual environment for the 5GCN. We are using the Free5GC open-source project to implement the 5GCN. However, there are some prerequisites required for it to work properly. First, we need a kernel module called GTP5G, which is needed to run the UPF. This module has compatibility issues with newer versions of Linux, which is why we chose the recommended Ubuntu version 20.04 [25]. Additionally, we need Docker Engine and Docker Compose v2 to run the Free5GC containers and to launch the Free5GC stack, respectively. After installing the prerequisites, we initialize and launch the Free5GC stack by building and running the docker images from GitHub. Refer to Appendix A for the full list of commands we are using to download these prerequisites and set up and configure the Free5GC stack.

### 7.3. Installing and configuring Zeek and Suricata

We install Zeek and Suricata as outlined in their official documentation [26], [27]. We edit the *suricata.yaml* file to specify what type of information should be captured from the network activity and to detect TCP traffic on the relevant ports used in the network. This is so we gather as much relevant information as possible, ensuring comprehensive detection and logging of all relevant events. For Zeek, this is not applicable, as it dynamically adapts to determine which logs should be generated with the information displayed in these logs being predefined. The installation commands for Zeek and Suricata are detailed in Appendix B, while Appendix C outlines the modifications made to the *suricata.yaml* file.

### 7.4. Simplified threat analysis

Before conducting our experiment to evaluate Zeek and Suricata in a 5GCN, a simplified threat analysis is necessary to identify security threats that pose a risk to the 5GC interfaces. This task involves reviewing scientific literature on these threats and selecting two 5GC interfaces, each with one relevant incident, to monitor within our network.

---

We chose to monitor the N4 interface because of its role in handling subscription and user data, which may fail to be routed through the 5GCN if attacks occur on the interface. Regarding the SBI interfaces, we chose to monitor them because attacks on these interfaces could allow malicious actors to gain unauthorized access and exploit them. Our choice of incidents is based on the security threats that pose a risk to the interface and its feasibility to carry out the corresponding attacks.

We decided to monitor DoS attacks via PFCP session deletion requests on the N4 interface and TLS handshake failure events caused by protocol version mismatch on the SBI interfaces. Both incidents came from [3], where the authors emphasized that the N4 interface is vulnerable to attacks on PFCP protocol. One of the attacks they mentioned is DoS via PFCP session deletion request, which was identified in [4] and served as the basis for the PFCP part of our experiment. For TLS, no SBI interfaces were mentioned to be vulnerable to attacks on the TLS protocol. However, the organizations 3GPP, ETSI and IETF recommend using TLS for mitigating eavesdropping, tampering, and message forgery. Therefore, we decided to investigate TLS handshake failure events due to protocol version mismatch in this thesis, noting that this attack is the most feasible for us to execute. Section 7.5 provides details on the execution of these attacks.

#### 7.4.1. PFCP session deletion request

To generate PFCP session deletion requests, we execute a Bash script based upon the paper [4] as detailed in Appendix F. The script sends a PFCP session deletion request from a spoofed source address to the NF UPF.

#### 7.4.2. TLS handshake failure

To trigger a TLS handshake failure in the 5GCN, we execute a Bash script from the Docker host against the NF NRF, which can be found in Appendix D. The script attempts to establish a secure connection from Docker host using TLS version 1.2 to the NRF, which uses TLS version 1.3. This causes a TLS handshake failure due to the mismatch in the TLS protocol versions. However, in Free5GC's implementation of the 5GCN, the TLS protocol dynamically adapts to the client's version of the protocol. To disable this automatic protocol version negotiation and enforce TLS version 1.3, we edit a config file for the NRF to set the minimum version of the TLS protocol version to 1.3. Refer to Appendix E for details on the configuration changes made to the NRF file.

## 7.5. Executing the attacks and analyzing them

When executing the attacks, we must ensure that the correct container is selected for the event we aim to monitor, as each NF has its own container. However, the virtual Ethernet (*veth*) interfaces created by Docker are unsuitable for traffic monitoring with Zeek and Suricata, which is why we set up the *eth0* interface on the Docker host to mirror traffic from the *veth* interfaces, as detailed in Appendix G. For the TLS handshake failure attack, we monitor the NF NRF, as it acts as a server in the TLS handshake process due to its functionality of handling registration for other NFs. For the PFCP session deletion request attack, we monitor the NF UPF, since the attack script targets the PFCP sessions created by the UPF. Additionally, for the PFCP session deletion request attack, we enter the UERANSIM container to add two interfaces for the subscribers. This step is needed because it creates PFCP sessions that we can later observe being deleted by the attack. For PFCP, we let the attack script run for approximately 10 seconds. With TLS, we execute the same attack script five times. Figure 3 illustrates both the setup of our 5GCN and the execution process for the attacks, as detailed in Appendix H.

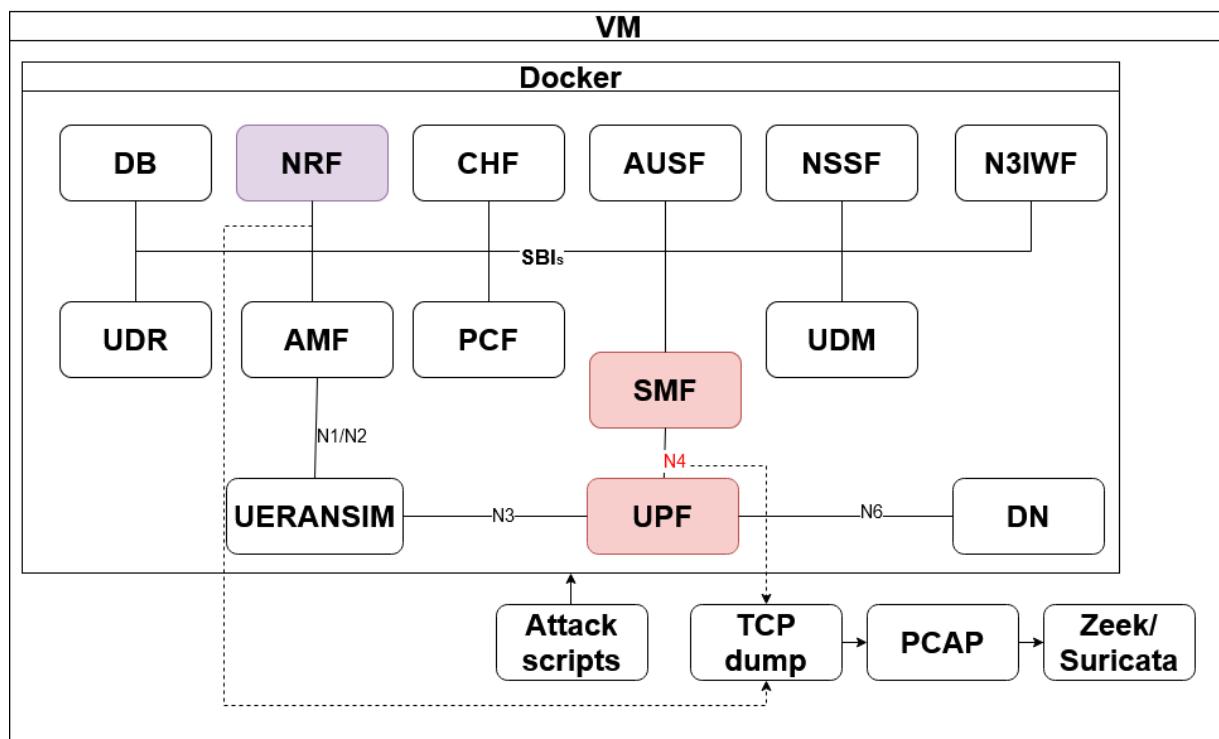


Figure 3 - The full execution process of both attacks.

## 7.6. Comparison with background traffic and successful TLS handshakes

We run a custom script to simulate background traffic, as shown in Appendix I, to compare with the log events generated by the attack scripts. This helps us identify log events that are uniquely generated by the attack scripts. Additionally, we run a script that results in successful TLS handshakes, as detailed in Appendix J, to compare the log events of TLS handshake failures. This allows us to establish a baseline for how the logs should look.

## 8. Results

The attack scripts we executed in our simulated 5GCN have generated results in the form of log events in both Zeek and Suricata. We will present and explain these results with accompanying tables, followed by an in-depth analysis.

### 8.1. TLS handshake failure attack

In this section, we present the results from the attack script, which triggered TLS handshake failures observed in both Zeek and Suricata. Although we executed the attack script five times, we will focus on the log events generated from a single execution to avoid redundancy in the explanation, as all five runs of the attack script produced identical results.

#### 8.1.1. Zeek

Zeek generated two log files, *conn.log* and *ssl.log*, each containing a unique log event from packets captured by the TLS attack script. Table 1 presents the relevant fields and values from the log event in the *conn.log* file, which records IP, TCP, UDP, and ICMP connection details. Table 3 displays the log event in *ssl.log*, showing key fields and values from the TLS handshake. Though the value from the field *service* displays *ssl*, it is a generic term that applies to TLS as well.

Field	Description	Value
proto	Transport layer protocol used for the connection	tcp
service	Application protocol ID sent over the connection	ssl
conn_state	Connection state, in this case is SF = “Normal establish and termination”	SF
history	Records the state history of connections as a string of letters. Client-side letters are capitalized, while server-side letters are in lowercase. Refer to Table 2 for the meaning of each letter.	ShADadFf

Table 1 – Log event in *conn.log* describing the relevant fields and their values captured from the TLS attack script.

Letter	Meaning
S	A SYN without the ACK bit set
H	A SYN+ACK (“handshake”)
A	A pure ACK
D	Packet with payload (“data”)
F	Packet with FIN bit set

**Table 2 – Describes the meaning of each letter in the history field from the conn.log file.**

Field	Description	Value
last_alert	Last alert that was seen during the connection.	protocol_version
established	Indication if the session was fully established.	false
ssl_history	Shows which types of packets were received in which order.	C1

**Table 3 – Log event in ssl.log describing the relevant fields and their values captured from the TLS attack script.**

In the *ssl\_history* field, client-side letters are capitalized, while server-side letters are in lowercase. In this case, does the letter “C” stand for client hello packet, and the letter “l” for an alert. This suggests that the server generated an alert after receiving the hello packet from the client.

### 8.1.2. Suricata

Suricata generated three separate log events in the *eve.json* log file from the packets captured by the TLS attack script. This log displays event logs such as alerts, anomalies, and protocol specific records in JSON format. Table 4, 5 and 6 show the log events for the *event\_types*: *frame*, *tls* and *flow* in the *eve.json* file, respectively. These tables describe the relevant fields for each *event\_type* with their corresponding values.

Field	Description	Value
event_type	Indicates the log type	frame
frame.type	The type of frame	stream
frame.direction	The direction of the frame	toserver

**Table 4 – Log event in eve.json describing the relevant fields for the event\_type frame and their values captured from the TLS attack script.**

Field	Description	Value
event_type	Indicates the log type	tls
tls.version	The TLS version used	TLSv1
tls.ja3.hash	The ja3 hash used for fingerprinting TLS clients.	fbe7e189e37a07ee33706f86bc746344

**Table 5 - Log event in eve.json describing the relevant fields for the event\_type tls and their values captured from the TLS attack script.**

Field	Description	Value
event_type	Indicates the log type	flow
flow.state	The flow's state	closed
flow.reason	Mechanism that triggered the flows termination	shutdown

**Table 6 - Log events in eve.json describing the relevant fields for the event\_type flow and their values captured from the TLS attack script.**

There are also logs created for the *event\_type frame*, where the *frame.type* had the following values in the order: *hdr*, *pdu* and *data*. Additionally, a log event exists for the *frame.type alert* with the *frame.direction toclient*, which appeared after the log event *frame.type data* with the *frame.direction toclient*. This means that 9 log events were generated, 4 from the client-side and 5 from the server-side. Typically, there would also be a field for *tls.ja3s.hash* from the server. However, that field is missing, which we will explain in Section 8.3.1.

## 8.2. PFCP session deletion request attack

In this section, we present the results from the attack script that generated PFCP session deletion requests as observed in both Zeek and Suricata.

### 8.2.1. Zeek

Zeek generated a unique log event in the *conn.log* file from the packets captured by the PFCP attack script. The result from the *conn.log* file is shown in Table 7, which describes the relevant fields with their corresponding values.

Field	Description	Value
id.orig_h	The IP address of the system that originated the connection.	192.168.0.17
proto	Transport layer protocol used for the connection	udp
conn_state	Connection state, in this case is SF = “Normal establish and termination”	SF
history	Records the state history of connections as a string of letters. Refer to Table 2 for the meaning of each letter.	Dd

Table 7 - Log event in *conn.log* describing the relevant fields and their values captured from the PFCP attack script.

### 8.2.2. Suricata

Suricata generated a unique log event in the *eve.json* file from the packets captured by the PFCP attack script. Table 8 displays the *eve.json* file results, highlighting the relevant fields and their corresponding values.

Field	Description	Value
event_type	Indicates the log type	flow
proto	Transport layer protocol used for the connection	UDP
app_proto	Application layer protocol used for the connection	failed
flow.state	The flow's state	established
flow.reason	Mechanism that triggered the flows termination	shutdown

Table 8 – Log events in eve.json describing the relevant fields and their values captured from the PFCP attack script.

### 8.3. Analysis

In this section, we delve deeper into the results and provide an analysis of them.

#### 8.3.1. TLS handshake failure attack

For Zeek, our primary results stem from Table 3, which indicates that the TLS handshake failed due to protocol version mismatch. This is clear from the *last\_alert* field, which displays the value *protocol version*, indicating that the TLS protocol versions of the client and server do not match. Additionally, the *ssl\_history* field corroborates this mismatch. There we can see that the server triggered an alert after the client sent its Hello packet, further suggesting TLS protocol version mismatch. Another indication of a TLS handshake failure in Table 3, though not due to a protocol version mismatch, is found in the *established* field. This field shows us that the TLS connection was not established, as displayed by the value *false*.

There are also some results from Table 1 for Zeek, however, they refer more to the TCP part of the TLS handshake as seen from the *proto* field. We can observe that the TCP handshake was successful and then terminated for some reason related to TLS, which can be observed by the fields *conn\_state* and *service* with the *SF* and *tls*, respectively. However, it is unknown what happened with the attempted TLS connection that caused the termination of the TCP connection via a graceful connection release. Additionally, the *history* field further supports this outcome. There we can observe that the client and server sent each other a packet with the FIN bit set, indicating the termination of the TCP connection by both hosts.

For Suricata, the results are not as clear compared to Zeek. When we presented the results from the three separate log events in the *eve.json* file, we said that the *event\_type frame* generated an alert from the server after the frame types: *stream*, *hdr*, *pdu* and *data* had been processed. This means that the server was unable to send its hello packet to the client, indicating either cipher suite incompatibility or protocol version mismatch. In Table 5, the *event\_type tls* points to the same results. Specifically, the field *tls.ja3s.hash* from the server side, which should be present, is missing. This indicates that JA3s could not derive the decimal values of the bytes from the server hello packet, further suggesting a cipher suite incompatibility or protocol version mismatch. However, the *tls.version* field indicates that TLS version 1.0 is being used, which is not possible, as our environment is programmed in Go and has this version disabled by default. [28]. This suggests a potential protocol version mismatch, as successful TLS handshakes in our environment typically display either TLS 1.2 or 1.3.

On the other hand, the *event\_type flow* in Table 6 from Suricata might indicate the same result, but it is not as clearly presented as the other *event\_types*: *frame* and *tls*. It shows that the *flow.state* is *closed*, however, there are several ways to terminate a flow, either through an RST or the Four-way FIN handshake [29]. For the field *flow.reason* with the value *shutdown*, Suricata team member Victor Julién [30] explains, “shutdown just means the flow was evicted during engine shutdown instead of during normal runtime.” This aligns with the Suricata documentation, which states that the Stream engine monitors TCP connections, a component of the TLS protocol [29]. Interestingly, when executing the same attack procedure with live packets in an online environment, rather than using offline packets from PCAP-files, the result from *flow.reason* changes to *timeout*. *Timeout* is considered normal behavior among all the possible values for the *flow.reason* field. We believe this is due to the stream engine and PCAP-file processing the time differently as the stream engine uses real time instead of the PCAP timestamps. However, we cannot be certain, as the documentation does not explain why the stream engine shuts down before the timeout is triggered when processing PCAP-files.

### 8.3.2. PFCP session deletion request attack

For Zeek, Table 7 shows no indication of PFCP protocol being used between the two hosts. This aligns with the absence of the *service* field in the log entry, which does not list the PFCP protocol, unlike the TLS log events in Zeek, which can identify the application layer protocol TLS. The reason Zeek is unable to detect PFCP packets in the network is the lack of an adequate detection algorithm implementation to recognize them, as the PFCP protocol is a highly specific communication protocol used exclusively in 5GCN between the SMF and UPF. The log entry from Zeek could only detect the underlying UDP protocol as shown from the field *proto* indicating the “*udp*” value. The connection state for this conversation is marked as *SF* in the field *conn\_state*, which stands for “normal establishment and termination” of the connection. However, since UDP is a stateless protocol, it leaves the duty of managing the connection state of the conversations to higher-level protocols [31]. In this case, *SF* means that Zeek has observed what it considers to be a "normal exchange" of data for UDP. However, from the field *history*, we observe that a data packet was sent from the spoofed source address to the NF UPF and back. The packet sent to the UPF is likely a PFCP session deletion request, as the source IP address, 192.168.0.1, as indicated by the field *id.orig\_h*, does not exist in our network and is configured in the PFCP attack script to send the session deletion request to the NF UPF. This suggests that the returned packet is most likely a confirmation of the deleted PFCP session. This means that Zeek, in its base version, could not detect the session deletion request from the PFCP protocol.

For Suricata, as shown in Table 8, the results are similar. It detects only the underlying UDP protocol as seen from the *proto* field and fails to recognize the PFCP protocol, as indicated by “*failed*” in the field *app\_proto*. However, compared to Zeek, we can simply observe that the connection is established when traffic is seen from both sides through the *flow.state* field [29]. Interestingly, for the field *flow.reason* with “*shutdown*” value, this behavior does not make sense, as the stream engine is responsible only for monitoring TCP connections. We believe this behavior stems from the same cause as the *shutdown* value observed in the *flow.reason* field during the analysis of the TLS portion of the experiment.

## 9. Discussion

Our results show that both tools can detect TLS traffic, as demonstrated in Section 8.3.1. However, neither tool currently supports direct detection of the PFCP protocol, as detailed in Section 8.3.2. Instead we could only identify the underlying protocol, UDP. Further research and development are needed for more precise PFCP detection. While there may be ongoing efforts to develop PFCP detection, we have not had the opportunity to explore them due to time constraints. Additionally, it is recommended to encapsulate the PFCP protocol in an IPsec tunnel due to its lack of built-in security, which should be considered alongside an evaluation of how these security measures affect detection. Lastly, no signatures were triggered from Suricata for either TLS or PFCP, likely due to the default ruleset not supporting our specific attacks.

These results are noteworthy for anyone who might be exposed to TLS attacks, including any 5GCN owner. Regarding our PFCP findings, these are primarily relevant to 5GCN owners but may also be applicable to owners of other 3GPP-standard networks with PFCP traffic. For these groups, we have demonstrated how Zeek and Suricata can potentially improve the security of 5GCNs. However, since our work does not include any cost-benefit analysis and has a very limited scope, further investigation is necessary to fully evaluate the practicality of these tools. With this, we believe that we have successfully explored how Zeek and Suricata can improve incident monitoring within a 5GCN. Additionally, we have highlighted some of the challenges posed by the complex 5GCN. However, some questions remain regarding how efficient these tools are at monitoring 5GCNs, particularly in terms of performance and coverage of other 5GCN-standard protocols.

In this thesis, we have answered the two research questions and taken a step toward addressing RQ1. However, further testing and evaluation are necessary to provide a comprehensive answer to RQ1. Furthermore, if Zeek and/or Suricata are to be used as security tools for the 5GCN, broader questions should be examined to evaluate their utility in a 5G network as a whole and how they could potentially work with other tools. With this, we have taken a step toward mitigating some of the potential threats identified in earlier works. Our results for TLS may be useful outside our specific environment, while they also suggest that at least some 3GPP protocols specific to mobile networks have limited support for security tools. However, further investigation is required to confirm this.

---

## 10. Conclusions

This study investigated the possibility of improving incident monitoring in a 5GCN using the tools Zeek and Suricata. We aimed to explore the extent to which Zeek and Suricata can monitor the N4 and SBI interfaces in a 5GCN by assessing their effectiveness in detecting DoS attacks via PFCP session deletion requests and TLS handshake failure events caused by protocol version mismatch, respectively.

For the TLS handshake failure event, Zeek was able to detect the incident on the N4 interface through a single, clear log event. Suricata also detected the TLS handshake failure event on the N4 interface but required piecing together information from multiple log events.

For DoS attacks via PFCP session deletion request, Zeek was unable to identify the incident on the PFCP protocol. However, Zeek did detect the underlying protocol UDP. Similarly, the results from Suricata for the PFCP session deletion requests event showed that it failed to identify the PFCP protocol, only recognizing UDP traffic between both sides. To detect the PFCP protocol, additional security monitoring tools are needed, or an investigation into externally developed Zeek scripts and Suricata signatures should be conducted.

The conclusion to be drawn is that the extent to which Zeek and Suricata can monitor the N4 and SBI interfaces in a 5GCN depends on the incidents being monitored. If the attack or event includes a common protocol such as TLS, the tools may provide adequate detection. However, for specialized protocols like PFCP, the tools offer limited detection if the attacks or events occur at or above the TCP/UDP layer. If the attacks or events occur below the TCP/UDP layer, Zeek and Suricata may provide no detection at all.

It is also worth noting that we believe Zeek offers better coverage in scenarios where no signatures are triggered by Suricata. However, using both tools together might improve overall detection of potential attacks and events in the monitored network.

## 11. Future Work

Although this thesis has primarily centered around the tools Zeek and Suricata within a virtualized environment, further research incorporating additional security tools for evaluation could offer deeper insights into the field and their effectiveness in detecting network incidents. Additionally, future research utilizing physical equipment that mirrors the architecture of corporate environments could improve the validity of the findings for real-world applications. Furthermore, exploring the monitoring of other events, such as the reuse of old OAuth 2.0 access tokens, attempts to downgrade to less secure protocols, and reconnaissance attacks like port scanning, could provide valuable insights into the level of detection these security tools can offer.

As mentioned in section 10, the base versions of Zeek and Suricata are unable to detect anything related to the PFCP protocol. This limitation may also apply to other 5GCN-standard protocols that Zeek and Suricata are not configured to detect. To the best of our knowledge, there are no existing add-ons or extensions for Zeek and Suricata that address the incidents involving the PFCP protocol. Therefore, a thorough exploration of developing custom scripts and signatures for both tools is essential to enable the detection of such incidents.

## References

- [1] "About Zeek," *Book of Zeek*, Jan. 15 2025. [Online]. Available: <https://docs.zeek.org/en/master/about.html>
  - [2] "Suricata," *Suricata*, Jan. 10 2021. [Online]. Available: <https://web.archive.org/web/20210110224318/https://suricata-ids.org/>
  - [3] M. Mahyoub, A. AbdulGhaffar, E. Alalade, E. Ndubisi and A. Matrawy, "Security Analysis of Critical 5G Interfaces," *IEEE Communications Surveys & Tutorials*, vol. 26, no. 4, pp. 2382-2410, March 2024. [Online] Available: <https://ieeexplore.ieee.org/document/10472310>
  - [4] G. Amponis *et al.*, " Threatening the 5G core via PFCP DoS attacks: the case of blocking UAV communications," *EURASIP Journal on Wireless Communications and Networking*, vol. 2022, no. 1, Dec. 2022. [Online]. Available: <https://jwcn-eurasipjournals.springeropen.com/articles/10.1186/s13638-022-02204-5>
  - [5] A. Waleed, A. F. Jamali, A. Masood, "Which open-source IDS? Snort, Suricata or Zeek , " *Computer Networks*, vol. 213, Aug. 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S1389128622002420>
  - [6] "Features," *Suricata*, [Online], Available: <https://suricata.io/features/>
  - [7] " Introducing 3GPP," *The 3<sup>rd</sup> Generation Partnership Project*, [Online], Available: <https://www.3gpp.org/about-us/introducing-3gpp>
  - [8] A. Sultan, " 5G System Overview," *The 3<sup>rd</sup> Generation Partnership Project*, Oct. 11 2022. [Online] Available: <https://www.3gpp.org/technologies/5g-system-overview>
  - [9] *3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Release 15 Description; Summary of Rel-15 Work Items (Release 15)*, 21.915, 2019.
  - [10] "PFCP protocol," *Nokia*, Dec. 20 2024. [Online]. Available: <https://documentation.nokia.com/mag-c/24-3/books/pfcp/pfcp-protocol.html>
-

- [11] "UPF Design Document," *Free5GC*, March 13 2024. [Online]. Available: <https://free5gc.org/doc/Gtp5g/design/>
  - [12] "Service Based interfaces," *Nokia*, Dec. 13 2024. [Online]. Available: <https://documentation.nokia.com/mag-c/23-10-1/books/cpf/service-based-interfaces.html>
  - [13] *3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; System architecture for the 5G System (5GS); Stage 2 (Release 19)*, 23.501, 2024.
  - [14] "World's Leading Open Source Mobile Packet Core, free5GC, Moves Under Linux Foundation to Provide Open Source Alternatives Across 5G Deployments," *The Linux Foundation*, Sept.16 2024 [Online] Available: <https://www.linuxfoundation.org/press/worlds-leading-open-source-mobile-packet-core-free5gc-moves-under-linux-foundation-to-provide-open-source-alternatives-across-5g-deployments>
  - [15] Free5GC, "free5gc / free5gc," *Github*, 2024. [Online] Available: <https://github.com/free5gc/free5gc>
  - [16] A. Güngör, "aligungr / UERANSIM," *Github*, 2022. [Online], Available: <https://github.com/aligungr/UERANSIM>.
  - [17] "tcpdump(1) man page," *tcpdump*, Dec. 02, 2024. [Online] Available: <https://www.tcpdump.org/manpages/tcpdump.1.html>.
  - [18] E.Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," *rfc-editor*, Aug. 2018. [Online], Available: <https://www.rfc-editor.org/rfc/rfc8446>.
  - [19] "What happens in a TLS handshake? SSL handshake," *Cloudflare*, [Online] Available: <https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/>
  - [20] "TCP Connection Termination," *GeeksforGeeks*, Oct. 03, 2024. [Online] Available: <https://www.geeksforgeeks.org/tcp-connection-termination/>
  - [21] "Why TCP Connect Termination Need 4-Way-Handshake?," *GeeksforGeeks*, Apr. 19, 2022. [Online] Available: <https://www.geeksforgeeks.org/why-tcp-connect-termination-need-4-way-handshake/>
-

- [22] J. Althouse, "TLS Fingerprinting with JA3 and JA3S," *salesforce*, [Online], Available: <https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967/>
- [23] "Introduction," *scapy 2.6.1 documentation*, Jan. 09, 2025. [Online] Available: <https://scapy.readthedocs.io/en/latest/introduction.html>
- [24] K. Säfsten och M. Gustavsson, *Forskningsmetodik, för ingenjörer och andra problemlösare*. Lund: Studentlitteratur AB, 2019.
- [25] "Recommended Environment," *free5gc*, [Online] Available: <https://free5gc.org/guide/Environment/?h=hardware#recommended-environment>
- [26] "Binary Packages," *Zeek*, Dec. 16, 2024. [Online] Available: <https://docs.zeek.org/en/lts/install.html#binary-packages>
- [27] "Quickstart guide," *Suricata*, [Online] Available: <https://docs.suricata.io/en/latest/quickstart.html>
- [28] "TLS 1.0 and 1.1 disabled by default client-side," *go*, [Online] Available: <https://go.dev/doc/go1.18#tls10>
- [29] "Suricata.yaml." *Suricata*, [Online] Available: <https://docs.suricata.io/en/latest/configuration/suricata-yaml.html>
- [30] S. Steinbiss, "Documentation #6404," *openinfosecfoundation*, [Online], 2023. Available: <https://redmine.openinfosecfoundation.org/issues/6404>
- [31] "conn.log," *Zeek*, Jan. 15, 2025. [Online] Available: <https://docs.zeek.org/en/master/logs/conn.html>

## Appendices

### A – Setting up and configuring 5GCN

Here is the list of commands we used to set up and config the 5GCN. This process includes installing GTP5G kernel module, setting up Docker Engine and Compose v2, installing and launching the Free5GC stack, and finally configuring the UERANSIM container.

GTP5G kernel module:

- sudo apt install git make gcc
- git clone -b v0.8.10 https://github.com/free5gc/gtp5g.git
- cd gtp5g
- make
- sudo make install

Docker Engine and Compose v2:

- sudo apt-get install ca-certificates curl
- sudo install -m 0755 -d /etc/apt/keyrings
- sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
- sudo chmod a+r /etc/apt/keyrings/docker.asc
- echo \ "deb [arch=\$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \ \$(. /etc/os-release && echo "\$VERSION\_CODENAME") stable" | \ sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
- sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
- sudo usermod -aG docker ndr
- sudo reboot now

Free5GC:

- git clone -b v3.4.2 https://github.com/free5gc/free5gc-compose.git
  - cd free5gc-compose
  - cd base
  - git clone --recursive -j `nproc` https://github.com/free5gc/free5gc.git
  - cd ..
  - make all
  - docker compose -f docker-compose-build.yaml build
-

- docker compose -f docker-compose-build.yaml up

UERANSIM:

- sudo nano config/gnbcfg.yaml
  - linkIp: “UERANSIM container IP-address”
  - ngapIp: “UERANSIM container IP-address”
  - gtpIp: “UERANSIM container IP-address”
  - amfConfigs:
    - - address: “amf container IP-address”
- sudo nano config/uecfg.yaml
  - gnbSearchList:
    - - “amf container IP-address”
- ./nr-ue -c config/uecfg.yaml

Configuring the correct IP-addresses for the *linkIp*, *ngapIp*, *gtpIp* and *amfConfigs:address* parameters is crucial in the configuration file for the UERANSIM container. We edit the *linkIp*, *ngapIp* and *gtpIp* parameters to specify the IP-address of the UERANSIM container, and we also edit the *amfConfigs:address* parameter to specify the IP-address of the AMF container. Then we run the UE inside the UERANSIM container.

## B – Installation commands for Zeek and Suricata

Below is the list of commands we used to install the tools Zeek and Suricata.

Zeek:

- echo 'deb http://download.opensuse.org/repositories/security:/zeek/xUbuntu\_20.04/ /' | sudo tee /etc/apt/sources.list.d/security:zeek.list
- curl -fsSL https://download.opensuse.org/repositories/security:zeek/xUbuntu\_20.04/Release.key | gpg --dearmor | sudo tee /etc/apt/trusted.gpg.d/security\_zeek.gpg > /dev/null
- sudo apt update && sudo apt upgrade -y
- sudo apt install zeek-lts -y
  - select option - “Internet Site“

Suricata:

- sudo apt-get install software-properties-common -y
  - select 8 - Europe
  - select 49 - Stockholm
- sudo add-apt-repository ppa:oisf/suricata-stable
  - press enter
- sudo apt install suricata jq -y

## C – Modifications made to the *suricata.yaml* file

Below are the configuration changes made to the *suricata.yaml* file:

outputs:

- eve-log:

- types:

- frame:

- enabled: yes

- stats:

- enabled: no

pcap-file:

- checksum-checks: no

app-layer:

- tls:

- detection-ports:

- dp: 443, 8000

## D – TLS handshake failure attack script

The TLS handshake failure attack script we used in our experiment.

```
import ssl
import socket

context = ssl.SSLContext(ssl.PROTOCOL_TLS) # Specifies to use the TLS protocol
context.maximum_version = ssl.TLSVersion.TLSv1_2 # Sets the maximum version of TLS
context.verify_mode = ssl.CERT_NONE # No certificate is requested from the client

# Creates a socket connection with the NRF, wraps the created Python socket and returns an
# instance of the SSL socket class, which is then utilized to print the version of TLS that was
# used in the successful connection. Otherwise, the SSLError catches the protocol version
# mismatch error and prints an error message.

try:
    with socket.create_connection(('10.100.200.12', 8000)) as sock:
        with context.wrap_socket(sock, server_hostname='10.100.200.12') as secure_sock:
            print(f"Connected successfully using: {secure_sock.version()}")
except ssl.SSLError as ssl_err:
    print(f"SSL Error occurred: {ssl_err}")
```

## E – Configuration changes made to the NRF file

To implement the configuration changes, we modified the *~/free5gc-compose/base/free5gc/NFs/nrf/internal/sbi/server.go* file to set the minimum version of the TLS protocol to 1.3. We did this by adding the following lines of code to row 55, 56 and 57:

```
s.httpServer.TLSConfig = &tls.Config {  
    MinVersion: tls.VersionTLS13,  
}
```

To rebuild the modified file, we installed the Go programming language and executed the *make nrf* command to build the NRF image. This process generated a binary file located in *~/free5gc/compose/base/free5gc/bin*, which we added as a volume in the *~/free5gc-compose/docker.compose.build* file. This build file runs that binary file on the boot up of the NRF container.

## F – PFCP session deletion request attack script

The PFCP session deletion request attack script we used in our experiment.

```
from scapy.all import *
from scapy.contrib.pfcpc import *

SEID = 0x0
SEQ = 101
try:

    while True:

        print(SEID)

        # PFCP packet containing the session deletion request.
        pfcpc_request = PFCPC(
            version=1
            ,seid=SEID,
            seq=SEQ,
            message_type = "session_deletion_request"
        )

        # Information about where to send the PFCP-packet, the protocol to use, and the
        # corresponding port number.
        full=IP(src="192.168.0.17", dst="10.100.200.10")/UDP(sport=8805,
        dport=8805)/pfcpc_request

        #Ignoring checksums
        full[UDP].chksum = None

        #Sends the packet
        send(full)
        SEQ = SEQ + 1
        SEID = SEID + 1

    except KeyboardInterrupt:

        print("KeyboardInterrupt!!")
```

---

## G – Mirror container script

The script we used to mirror traffic from the UPF/NRF to the *eth0* interface on the Docker host. Replace the word 'container' with the container you want to mirror, either UPF or NRF.

```
# Finds the corresponding veth interface for the interface inside the selected container.  
for i in /sys/class/net/veth*/ifindex;  
do  
    a=$(cat $i);  
    b=$(docker exec -it 'container' cat /sys/class/net/eth0/iflink|tr -cd '[:digit:]');  
    test $a = $b && echo $i|cut -d/ -f5 && interface=$(echo $i|cut -d/ -f5);  
  
done  
  
# Creates the interface.  
sudo modprobe dummy  
sudo ip link add eth0 type dummy  
sudo ip addr add 192.168.1.10/24 brd + dev eth0 label eth0:0  
sudo ip link set dev eth0 up  
  
# Mirrors the traffic from the selected container to the interface created.  
sudo tc qdisc add dev $interface ingress  
sudo tc filter add dev $interface parent ffff: protocol all prio 2 u32 match u32 0 0 flowid 1:1  
action mirred egress mirror dev eth0  
sudo tc qdisc replace dev $interface parent root handle 10: prio  
sudo tc filter add dev $interface parent 10: protocol all prio 2 u32 match u32 0 0 flowid 10:1  
action mirred egress mirror dev eth0
```

## H – The execution process

We launch the 5GCN and then mirror the appropriate NF to correspond with the attack that will be executed. When mirroring the appropriate NF to the corresponding attack that will be executed, we use the tcpdump to capture all the packets from a specific interface into a PCAP-file. We select the interface by the -i flag and use the -w flag to specify where our PCAP-file should be stored. From there, we execute the attack script we want to monitor. When the attack is complete, we scan the PCAP-file for the attack we want to analyze into Zeek and Suricata. For Suricata, the flags -k, -c, and -r represent the following: disable checksum checking, specifying the location of the configuration file, and the listed PCAP-file to run, respectively. For Zeek, the flags -C and -r instruct Zeek to ignore invalid IP checksums and to perform the default analysis on the listed PCAP file. Additionally, we include the *LogAscii::use\_json=T* option when scanning PCAP files with Zeek to generate logs in JSON format. This step is unnecessary for Suricata, as its log files are in JSON format by default. Below are the steps for the commands we used for the execution process:

1. cd free5gc-compose  
# Goes into the folder where we can launch the Free5GC stack.
2. docker compose -f docker-build.yaml up  
# Launch the Free5GC stack.
3. bash nrf.bash / upf.bash  
# Scripts used to mirror the traffic, depending on the event we aim to monitor. NRF for TLS and UPF for PFCP.
4. sudo tcpdump -i eth0 -w tls.pcap / pfcpcap  
# Captures all packets from the eth0 interface into a PCAP file. As explained in step 3, the PCAP file name should be chosen carefully, as it will be used in step 6.
5. python tls.py / pfcpc.py  
# Executes the attack script. Select only one script to run for the corresponding monitored event.
6. sudo suricata -k none -c /etc/suricata/suricata.yaml -r tls.pcap / pfcpcap  
# Scan the PCAP-file from step 3 into Suricata for analysis.  
./././bin/zeek -C -r tls.pcap / pfcpcap LogAscii::use\_json=T  
# Scan the PCAP-file, from step 3 into Zeek for analysis.

## I – Simulate background traffic script

[Link](#) to the original creator of the script with comments explaining the code. Below is our simplified version of the script, which we used to simulate background traffic in our environment.

```
#!/bin/bash

speedtest () {
    dlspeed=$(echo -n "scale=2; " && curl --interface uesimtu0 --connect-timeout 1 http://$1/$fileName -w "%{speed_download}" -o $fileName -s | sed "s/,/./g" && echo "/1048576");
    echo "$dlspeed" | bc -q | sed "s/$/ MB/sec;/s/^/tDownload Speed\: /";
    ulspeed=$(echo -n "scale=2; " && curl --interface uesimtu0 --connect-timeout 1 -F "file=@$fileName" http://$1/webtests/ul.php -w "%{speed_upload}" -s -o /dev/null | sed "s/,/./g" && echo "/1048576");
    echo "$ulspeed" | bc -q | sed "s/$/ MB/sec;/s/^/tUpload speed\: /";
}

fileName="100mb.test";
ls "$fileName" 1>/dev/null 2>/dev/null;
if [ $? -eq 0 ]
then
    echo "$fileName already exists, remove it or rename it";
    exit 1;
fi

metDependencies=1;
type curl 1>/dev/null 2>/dev/null;
if [ $? -ne 0 ]
then
    echo "curl is not installed, install it to continue by typing:"
    echo "sudo apt install curl"
    metDependencies=0 ;
fi
type bc 1>/dev/null 2>/dev/null;
if [ $? -ne 0 ]
then
    echo "bc is not installed, install it to continue by typing:"
    echo "sudo apt install bc"
    metDependencies=0;
fi
if [ $metDependencies -eq 0 ]
then
    exit 1;
```

```
fi
```

```
while true  
do
```

```
echo "-----Speed test-----";  
echo "Testing North America locations";  
curl https://www.google.com/search?q=[1980-1986] --interface uesimtun0 -w  
"%{time_connect},%{time_total},%{speed_download},%{size_download},%{  
url_effective}\n" -o /dev/null -s  
echo "Speedtest from Seattle, Washington, USA [ generously donated by  
http://ramnode.com ] on on a shared 1 Gbps port";  
speedtest 23.226.231.112;  
echo "Speedtest from Los Angeles, California, USA [ generously donated by  
http://ramnode.com ] on on a shared 1 Gbps port";  
speedtest 168.235.78.99;  
echo "Speedtest from New York City, New York, USA [ generously donated by  
http://ramnode.com ] on on a shared 1 Gbps port";  
speedtest 168.235.81.120;  
echo "Speedtest from Atlanta, Georgia, USA [ generously donated by  
http://ramnode.com ] on on a shared 1 Gbps port";  
speedtest 192.73.235.56;  
echo "Speedtest from Atlanta, GA, USA [ generously donated by  
http://hostus.us ] on a shared 1 Gbps port";  
speedtest 162.245.216.241;  
  
echo -e "\nTesting EU locations";  
echo "Speedtest from Paris, France on a shared 1 Gbps port";  
speedtest "4ii18b4g67f03cdecaw9nusv.getipaddr.net";  
echo "Speedtest from Alblasserdam, Netherlands [ generously donated by  
http://ramnode.com ] on on a shared 1 Gbps port";  
speedtest 185.52.0.68;  
  
echo -e "\nTesting Australian locations";  
echo "Speedtest from Sydney, Australia on a shared 1 Gbps port";  
speedtest 103.25.58.8:3310;
```

```
unlink $fileName;  
rm 100mb.test ;
```

```
done
```

---

## J – Successful TLS handshake script

The Successful TLS handshakes script we used in our experiment.

```
import ssl
import socket

context = ssl.SSLContext(ssl.PROTOCOL_TLS) # Specifies to use the TLS protocol
context.maximum_version = ssl.TLSVersion.TLSv1_3 # Sets the maximum version of TLS
context.verify_mode = ssl.CERT_NONE # No certificate is requested from the client

# Creates a socket connection with the NRF, wraps the created Python socket and returns an
# instance of the SSL socket class, which is then utilized to print the version of TLS that was
# used in the successful connection. Otherwise, the SSLError catches the protocol version
# mismatch error and prints an error message.

try:
    with socket.create_connection(('10.100.200.12', 8000)) as sock:
        with context.wrap_socket(sock, server_hostname='10.100.200.12') as secure_sock:
            print(f"Connected successfully using: {secure_sock.version()}")
except ssl.SSLError as ssl_err:
    print(f"SSL Error occurred: {ssl_err}")
```