OULUN AMMATTIKORKEAKOULU

Vladimir Surtaev

An Automated Log Analyser for MACsec Protected Fronthaul Connections within a Base Station

An Automated Log Analyser for MACsec Protected Fronthaul Connections within a Base Station

Vladimir Surtaev
Bachelor’s Thesis
Spring 2024
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author(s): Vladimir Surtaev
Title of the thesis: An Automated Log Analyser for MACsec Protected Fronthaul Connections within a Base Station
Thesis examiner(s): Teemu Leppänen
Term and year of thesis completion: Spring 2024 Pages: 53

The increasing use of MACsec (Media Access Control Security) for securing fronthaul connections necessitates efficient log analysis for debugging and testing purposes. The topic of this thesis was to improve Rain - Nokia’s base station log analysis framework to analyse MAC layer protected fronthaul connections. This addition would reduce the time spent by developers on manual log analysis during the debugging allowing them to concentrate on feature development work more efficiently.

Logging is an important aspect of software development as by recording the information about system during runtime into logs, its behaviour can be reconstructed later. When debugging is needed, manual log analysis is flexible (as it is done by humans), but it consumes more effort, compared to automated analysis, especially when the same process needs to be repeated multiple times. And MACsec protected connections is something that needs to be analysed regularly.

To develop the analyser, it was required to strengthen the knowledge in MAC security standard, as well as its implementation in a base station to identify log messages that would provide useful information. Also, preparational work involved studying Rain platform interfaces and analyser development guidelines. Then it was required to design and develop such analyser that will be capable of processing textual logs, extracting the information about secured connections, and presenting it to users.

The result of this work is an automated log analysis program that makes analysis of MACsec protected fronthaul connections easier and efficient. That program is capable of processing logs produced by different software versions, has a modular architecture allowing developers to extend it for covering complex scenarios. Analysis results are presented within the familiar to developers Rain platform's graphical user interface. The thesis not only contributes a valuable tool for analysing MACsec connections but also deepens understanding of base station log analysis and Rain platform development.

Keywords: MACsec, Fronthaul, Automated Log Analysis, Logging.

PREFACE

I would like to express my gratitude to Ari-Pekka Taskila and Petri Kangas for their invaluable guidance and support throughout the development of this thesis. I am also grateful to the Oulu L1 DSP team for providing such an interesting thesis topic. My thanks extend to Jesse Nieminen for his professional advice on the subject, which proved to be highly beneficial.

Furthermore, I would like to acknowledge all those who offered their support in some way during this project.

I am particularly grateful to Nokia Oyj and the Oulu L1 DSP team for providing an exceptional opportunity to apply my knowledge in a practical setting. The experience and knowledge gained during this process have established an excellent foundation for the starting of my professional career.

CONTENTS

ABBREVIATIONS

API	Application Programmable Interface
BBU	Baseband Unit
BTS	Base transceiver station
CAK	Connectivity Association Key
CKN	Connectivity Association Key Name
CPRI	Common Public Radio Interface
DU	Distributed Unit (used interchangeably with BBU)
eCPRI	evolved CPRI
JSON	JavaScript Object Notation
HTTP	Hyper Text Transfer Protocol
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
MAC	Media Access Control layer
MACsec	IEEE 802.1AE security standard
MKA	MACsec Key Agreement
RRH	Remote Radio Head
RU	Radio Unit (used interchangeably with RRH)
SAK	Secure Association Key
SW	Software
TLDA	Technical Log Decoder and Analyzer
UML	Unified Modelling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 INTRODUCTION

The increasing demand for secure and reliable mobile data communication creates a need in robust fronthaul connections within Radio Access Networks (RAN). To address security concerns, Nokia has been developing MACsec (Media Access Control Security) for fronthaul traffic protection. This involves multiple system elements working together on encryption key negotiation, protection setup, traffic encryption, etc. However, troubleshooting potential issues requires analysing relevant data extracted from base station logs, a process that is currently manual, time-consuming, and inefficient.

This thesis presents the development of a MKA analyser - log analyser tool, specifically designed to analyse MACsec protected fronthaul connections. It aims to improve the troubleshooting process for developers by automating data extraction, analysis, and reporting. By automating these tasks, the analyser reduces significant time and effort required for manual analysis, allowing developers to focus on core development activities.

2 PROJECT BACKGROUND

2.1 Project goals

The current method of manually analysing base station logs for troubleshooting MACsec protected fronthaul connection is time-consuming and inefficient. This project aims to develop an automated log analyser to address this challenge.

Fronthaul traffic protection mechanisms, developed by Nokia, involve multiple system components collaborating on such activities as encryption key negotiation, protection setup, and traffic encryption. Extracting relevant data from the base station logs formed by these components, including key negotiation details, protection setup, and traffic encryption information requires significant manual effort for analysis and report generation in a human-readable format.

This project focuses on automating these tasks. Understanding the complexity of MAC security and fronthaul interfaces in Radio Access Networks (RAN) is crucial for efficient data extraction and analysis. Chapter 3 delves deeper into these underlying technologies. Furthermore, a strong understanding of logging as well as automated log analysis processes is essential. This topic is explored in detail within Chapter 4.

2.2 Project environment and requirements

The project started with a study of the existing fronthaul traffic protection architecture and the information logged during its operation. The primary objective is to analyse base station logs and produce reports that detail the status of fronthaul connections in a clear, human-readable format.

The Rain platform, widely used within Nokia for analysing base station logs, was chosen to host the developed analyser plugin. A detailed introduction to the Rain platform is provided in Chapter 5. Following platform selection, description of the design and development of the analyser provided in Chapter 6. Throughout this thesis, examples may be modified or omitted to ensure company confidentiality.

3 MACSEC IN RADIO ACCESS NETWORKS

3.1 Fronthaul in Radio Access Networks.

Fronthaul is a critical component in modern mobile networks. It is defined as the fibre connection between the Baseband Unit (BBU) and the Remote Radio Head (RRH) (See *FIGURE 1*). Fronthaul originated with 4G networks with the intention of moving radios closer to the antennas, separating the signal processing between radios and baseband units. Its significance has further increased with the advent of 5G technology, where it is crucial for supporting advanced features for example massive MIMO (Multiple Input Multiple Output) and lower latency. In modern networks, fronthaul plays a crucial role because, depending on the configuration, it helps balance the latency, throughput, and reliability of the network. (1.)

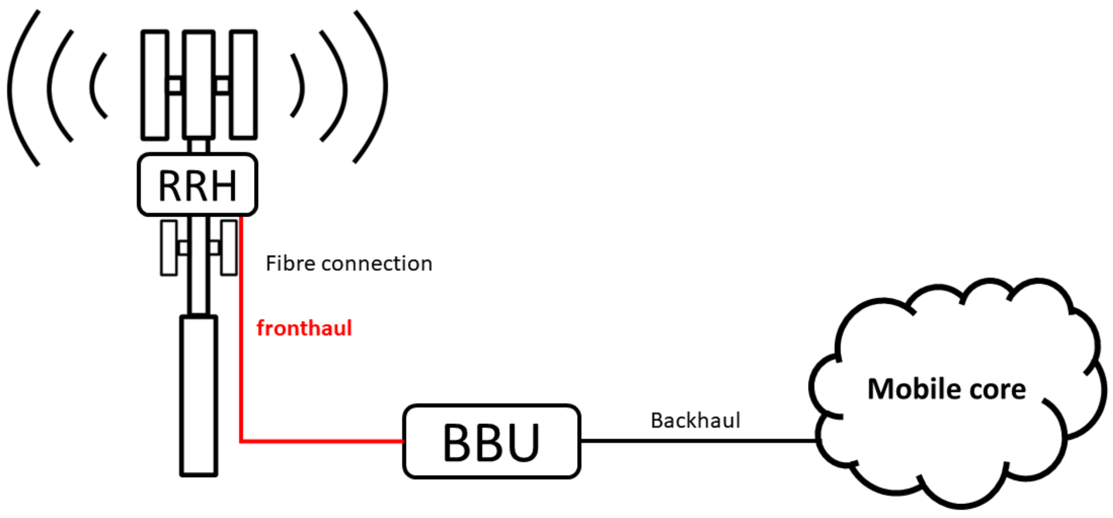


FIGURE 1. Base station site. Adopted from (2).

There are various deployment options for Radio Access Networks (RAN), such as Distributed RAN (D-RAN), Centralised RAN (C-RAN), virtualised RAN (vRAN) and Open RAN (See *FIGURE 2*). with the latter two gaining more interest nowadays (3). In these architectures, the heavy-duty processing tasks are performed by Baseband Units, which are required for network management and signal processing. Remote Radio Heads are located at the cell site (on towers or rooftops) and are responsible for transmitting and receiving radio signals. And in all cases BBUs and RRHs are being connected via fronthaul link(s), which makes the fronthaul a crucial component in a modern RANs.

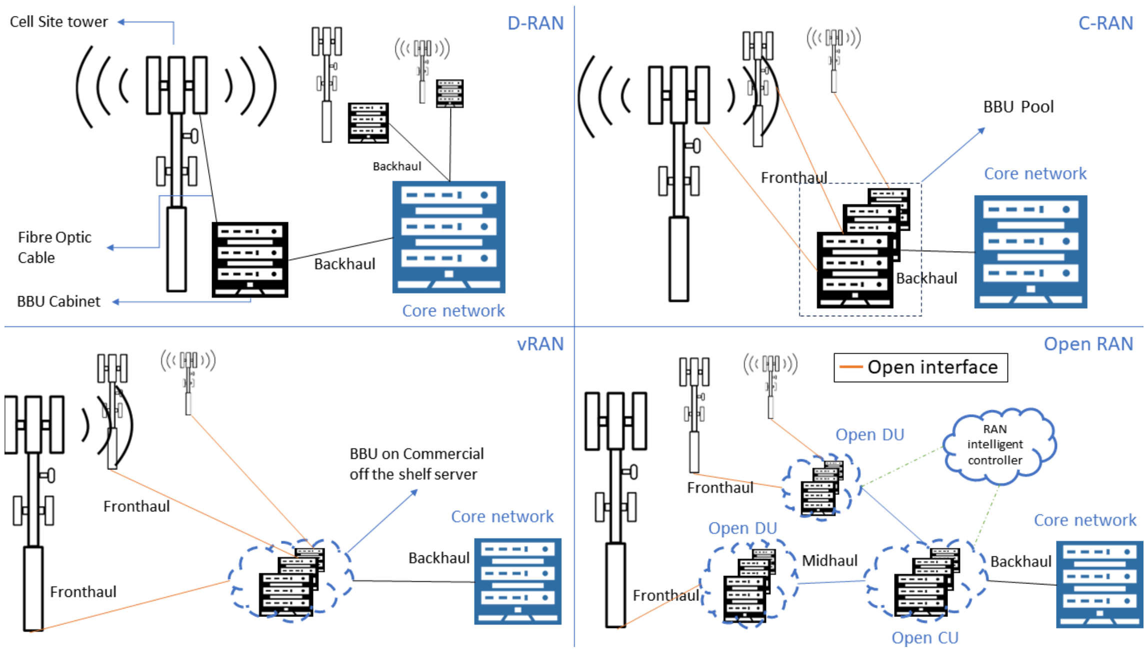


FIGURE 2. D-RAN, C-RAN, vRAN and Open RAN deployment variants. Adopted from (3).

A single BB Unit may have multiple fronthaul connections to Radio Units via single or multiple ports. The signals in fronthaul link carry critical data such as In-phase and Quadrature (I/Q) data, control, and management information. I/Q data represents the actual content of radio signals, such as voice and data.

Traditionally, fronthaul connections operate on fibre optic cables due to their high capacity and low latency. Standard interfaces such as the Common Public Radio Interface (CPRI) and the Enhanced CPRI (eCPRI) are used in fronthaul links for data transmission. eCPRI is designed to be more efficient and flexible compared to CPRI, which is suitable for 5G's varied and dynamic needs. Another notable difference between these two protocols – data transfer in eCPRI is implemented with packet-based protocols employing Ethernet or IP. (1.)

What comes to fronthaul in RAN, there are multiple requirements and challenges to be considered. Firstly, fronthaul connections need a support of extremely high bandwidths to accommodate the large volume of data being transmitted between RRHs and BBUs. Transition of mobile networks to newer technologies such as 5G and 6G amplifies this need because those technologies utilize higher frequencies and more bandwidth-intensive applications, which results in a significant increase in data traffic. Secondly, fronthaul should perform with low latency to maintain high-quality, real-time communication services, which is essential to ensure efficient communication between BBUs and RRHs. (4, 5.)

To sum up, fronthaul connections face significant challenges, primarily needing high bandwidth for increased data traffic and low latency for efficient, real-time communication between RRHs and BBUs, especially with the advent of 5G and 6G technologies.

3.1.1 Security vulnerabilities of fronthaul

Despite their critical role of fronthaul connections in mobile networks, they are susceptible to various security vulnerabilities which can significantly impact integrity and reliability of communication services. Unauthorised access is a significant threat, when an attacker may gain access to the Open Fronthaul Ethernet. By exploiting vulnerabilities at the L1 physical layer interface, attackers can potentially intercept or disrupt the data flow between the BBUs and RRHs.

Another way to compromise mobile network security is a sophisticated false base station attack, when an attacker could set up a rogue RU that imitates legitimate part of the mobile network. This type of attack could deceive mobile devices into connecting to a fraudulent network, enabling the attacker to intercept or manipulate the communication.

Moreover, the risk of Man-In-The-Middle (MITM) attacks in fronthaul connections is particularly concerning. In such kind of attacks, an intruder could insert themselves between the BBU and RRHs to intercept, relay, and potentially alter the In-phase and Quadrature (I/Q) data and control messages being transmitted. Attacks could lead to serious issues, including eavesdropping, data theft, service disruptions, and even network manipulation. (6.)

Considering these vulnerabilities, securing fronthaul connections is not just about maintaining data confidentiality and integrity - it is also about ensuring the availability and reliability of mobile networks. Therefore, implementing robust security measures for encryption and authentication is crucial to protect against unauthorised access, false base station attacks, and MITM threats, therefore safeguarding critical communication infrastructure in modern mobile networks.

In conclusion, fronthaul is a vital link in modern mobile networks, ensuring that core network can effectively communicate with antennas serving end-users. As mobile technology continues to evolve, importance of efficient, high-capacity, and low-latency fronthaul connections will only increase. Adding protection at this level would introduce an extra layer of security to the mobile networks. MACsec technology, which is used to protect transport layer traffic, can be applied to address identified fronthaul vulnerabilities.

3.2 MACsec (IEEE 802.1AE)

In the constantly evolving landscape of network security, the integrity and confidentiality of data that being transported have become extremely important**.** As malicious entities are continuously developing to intercept and exploit data, the implementation of robust security measures is crucial.

Media Access Control Security (MACsec) is a security technology defined by IEEE 802.1AE. Its primary purpose is to secure communication across the Ethernet links, ensuring that data is protected from intrusion, tampering, and eavesdropping. By encrypting and authenticating all traffic at the MAC layer, MACsec provides a level of security that is both robust and versatile.

MACsec operates by securing data on a point-to-point basis. It employs sophisticated encryption techniques to ensure that each packet of data is rendered unreadable to unauthorised users. Key to this process is the usage of security keys, which are exchanged and managed using the MACsec Key Agreement protocol (MKA).

Additionally, MACsec provides integrity checks to ensure that the data has not been tampered with during transit. Furthermore, as it operates on the MAC layer, it is transparent to most of the network, making it highly compatible and scalable with existing infrastructure.

There are three pivotal elements in MACsec operation: the Connectivity Association Key Name (CKN), the Connectivity Association Key (CAK), and the Secure Association Key (SAK). When two peers initiate a MACsec-secured communication, they first need to be configured with a pair of secrets. The CAK is an actual secret key that both peers possess, enabling them to encrypt and decrypt the data, while the CKN serves as a unique identifier for the CAK. The SAK, generated from the CAK, is a dynamic and operational key that encrypts, and decrypts data packets sent between peers. The CAK can be thought of as a master key that births a series of SAKs, each encrypting different sets of data or sessions (7.)

Once peers agree on the CKN and CAK through the MACsec Key Agreement protocol (MKA), a crucial phase ensues — the derivation of the Secure Association Key (SAK). One of the peers should become a Key Server to generate and distribute SAK as shown on the *FIGURE 1*. The usage of a dynamically generated SAK not only enhances security by regularly refreshing the encryption keys, but also ensures that the encrypted data remains indecipherable to the unauthorised entities.

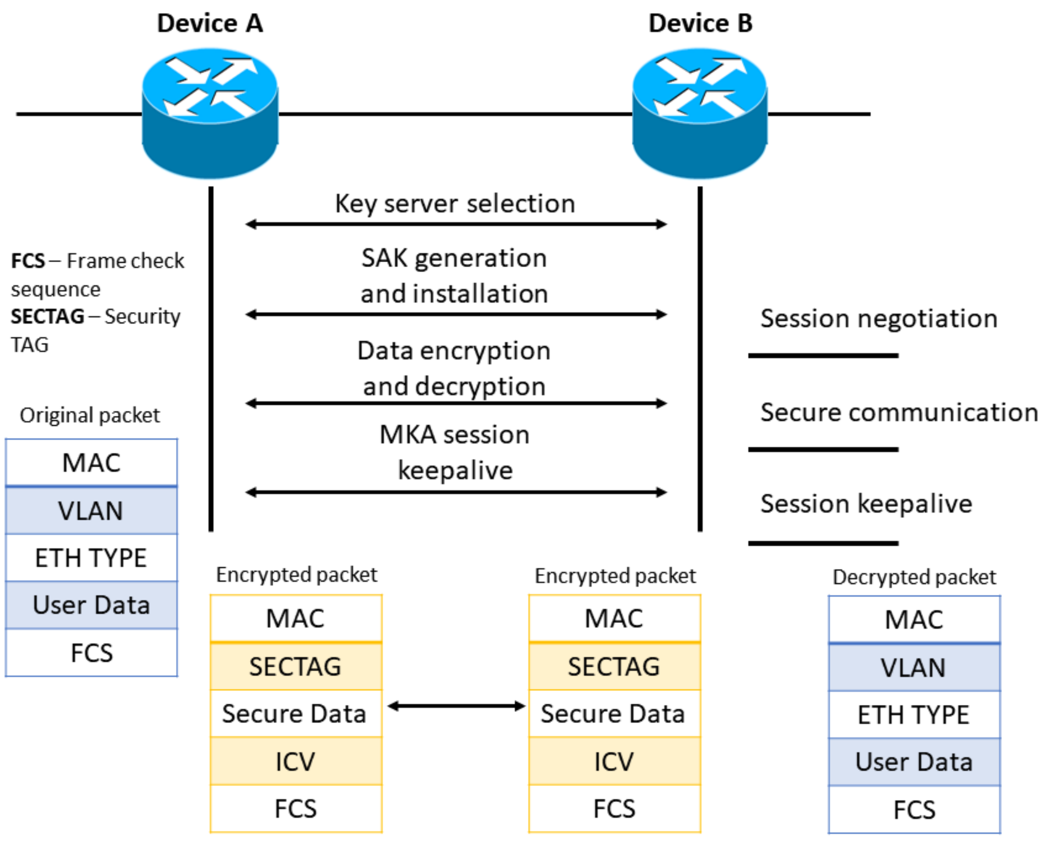


FIGURE 3. MACsec interaction process. Adopted from (8).

After the MKA Session has been negotiated, secure communication can start. As sending node transmits data, it encrypts payload with the SAK, effectively encoding readable information into a protected format. This encrypted data, as it traverses the network, is obfuscated, and rendered unintelligible, therefore mitigating the risk of interception and unauthorised access by malicious actors. Upon receipt, the corresponding node employs the SAK to decrypt the payload, reverting it to its original, comprehensible state.

Furthermore, MACsec attaches an Integrity Check Value (ICV) to each data packet (See "Encrypted packet" on the *FIGURE 3*). It verifies the data's integrity upon reaching its destination. Should the ICV indicate any discrepancy, it signals that the data might have been tampered with en route, thus alerting the system to potential security breaches.

Through the interaction of CAK, SAK, and CKN, coupled with integrity verification via the ICV, MACsec provides a robust, confidential, and reliable channel for data transmission. This algorithm operates with seamless efficiency, offering an impenetrable stronghold for data whilst in transit.

Since the payload is being encrypted when traversing between network nodes MACsec prevents a Man-In-The-Middle (MITM) from being able to intercept and read or alter the data in a transit. The payload encryption also prevents an attacker with physical access from deciphering the data without the appropriate encryption keys. Moreover, by implementing a strong authentication mechanism, ensuring that only authenticated nodes can join the Network, this way an attacker cannot setup a rogue node without knowing the encryption keys.

By providing protection against these potential attacks MACsec becomes an excellent choice for securing a fronthaul connection, which has the similar security concerns identified (9).

On the other hand, MACsec has drawbacks to be considered.
One of the main challenges is the complexity involved in key management. Efficiently managing and distributing pre-shared keys in a dynamic and large network can be daunting and may require additional infrastructure. Furthermore, while MACsec secures data on the local area network, it does not provide end-to-end encryption through the entire data path if the data traverses beyond the local network, such as across the internet. This means additional security measures may be necessary for complete protection.

In conclusion, MACsec emerges as a compelling and effective choice for securing MAC layer connections in the realm of network security by providing an encryption of the interface between network peers and preventing from rogue node or MITM attack. Despite challenges associated with key management and the scope of encryption, strengths of MACsec in protecting data within local domain are undeniable. Its ability to safeguard communication against a variety of threats makes it a good choice for protecting the fronthaul connection between the Baseband Unit and Radio Unit.

3.3 IEEE 802.1X Supplicant for MKA management

For the implementation of MACsec protection, a software implementation of an IEEE 802.1X supplicant (hereafter referred to as 'supplicant') has been chosen to manage MKA connections.

The supplicant, as defined in the IEEE 802.1X standard, is for port-based Network Access Control and provides an authentication mechanism for devices wishing to attach to a Local Area Network (LAN) or Wireless Local Area Network (WLAN). This protocol defines three key actors: the supplicant, which seeks access to the network; the authenticator, which controls physical access to the network resources; and the authentication server, which verifies the credentials provided by the supplicant and informs the authenticator. (10.)

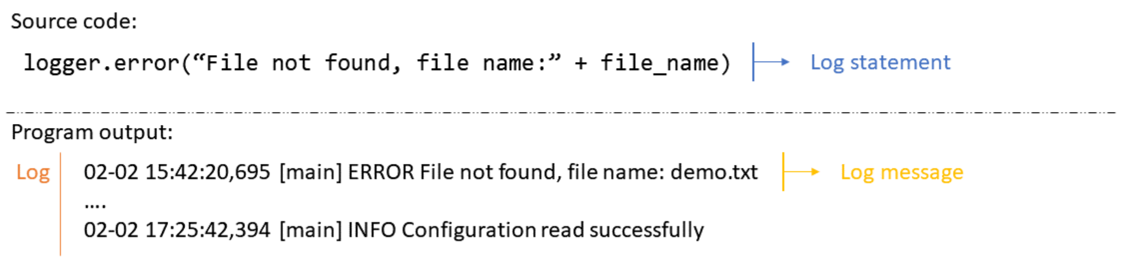
While supplicant defined in abovementioned standard has nothing to do with MACsec (because MACsec can be enabled after supplicant has been authorised in a network), its software implementation can handle the key agreement, setup, and activation of MACsec on Ethernet links. It is also capable of negotiating with other devices on the network and propagating the SAKs to other MACsec-enabled peers. This added capability makes supplicant a good choice for implementing a fronthaul protection.

To secure traffic between Radio Units (RU) and the Baseband Unit (BBU), a similar setup operates on both ends. The Supplicant, running on both the RU and BBU, manages MACsec and is governed by a Nokia software that monitors all connections and hardware configurations.

The Supplicant generates numerous logs related to MKA information on a particular interface. Additionally, BBU and RU software produces logs concerning connection management and messages sent to the Supplicant, among other details. Analysing these logs is crucial, as the aggregated information can assist in reconstructing real-life events and support investigations when necessary.

4 LOGGING IN SOFTWARE DEVELOPMENT

"Logging is the process of recording application actions and state to a secondary interface" (11). Keeping records of application behaviour helps in monitoring and reconstructing of its execution. To perform logging developer needs to insert a statement into the program's source code. That statement will produce a record of the runtime behaviour of that program during its execution (12). All the records or log messages produced by the program form the log which can tell about the program's way of acting. Refer to the *FIGURE 4* showing the terms in an example.


Source code:

```
logger.error("File not found, file name:" + file_name)
```

Program output:

```
02-02 15:42:20,695 [main] ERROR File not found, file name: demo.txt
....
02-02 17:25:42,394 [main] INFO Configuration read successfully
```

FIGURE 4. Logging related terms in an example. Adopted from (12).

Logging allows developers to reconstruct the program's behaviour as well as detect and address anomalies. Also, logs of programs are particularly useful in investigation processes because sometimes it can be difficult to reproduce a problem that appeared during the program execution. Another reason that makes it difficult to directly reproduce the problem is that customers may not want to share the inputs that lead to faulty behaviour due to confidentiality reasons. Logs on the other hand can be easily shared between customers, testers, and developers. All the above proves that logging is vital in software development. (13.)

The rest of this chapter will in detail introduce the purposes of logging, key logging terms as well as automated analysis to familiarize the reader with the basic ideas on the topic.

4.1 Core Purposes of Logging

The main purposes of logging in a software development can be categorized into three major categories, by the information which is being stored and extracted from logs: Operations, Design, and Operations (14). The rest of this section describes each of them in details.

Operational knowledge represents information about running system and relates to the most prevailing way of logging utilization. Previously mentioned examples of logging applications for faulty behaviour identification and a root cause analysis are included in operations. Information contained in logs is not only useful during the investigation of past events, but it can be used to audit the program during its runtime too. Analysing the logs in real time can not only help detect anomalies in the early stages but also predict them and mitigate potential problems. (14.)

The Design category focuses on extracting information about the internal structure of the application. In the context of mobile networks such could be for example information about data usage, network information, connection quality, and so on.

The last Design category aims to extract information about the internal structure of the application. Extracting such information could benefit in discovering the workflows inside the application by following records of the events happening. Validating the logs with the set of requirements can assess the security compliance of an application. Analysing the data about interactions with a system and failures that happened are tools for quality analysis that aim to improve the system's usability. (14.)

While the above demonstrates the importance and usefulness of logging due to its ability to provide various information, this is only true if logging is implemented correctly (12). Sometimes, developers fail to design log statements correctly from the outset, and examining logs later may not contain the desired information, causing developers to update the statement (13). This highlights that logging is not only important for the information it provides but also for the dedication and effort invested in its design.

4.2 Log messages

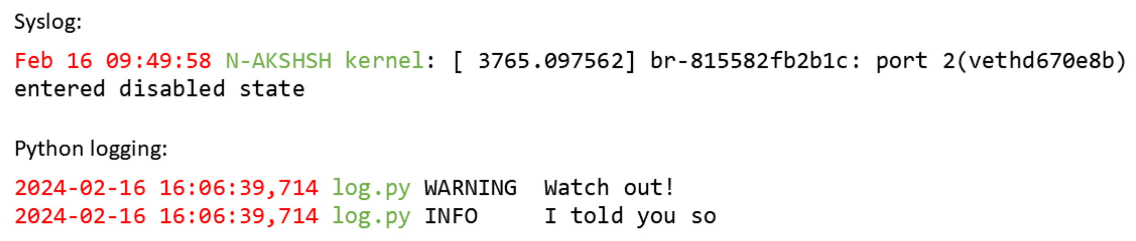
This chapter focuses on explaining different concepts associated with log messages, so the reader would understand the terminology and ideas used further in this paper.

As already mentioned above, the log is formed from multiple messages produced by the program during its execution. Those messages are being generated by the log statements inserted into the program's source code usually with the help of logging libraries or systems exposing an API. Application can define different conditions to produce a log message, for example, an incoming TCP request can trigger the server application to log the information about this request. (15).

4.2.1 Message format

Typically, log messages content timestamp, source of the message, and the data. These basic components are present in most of the log entries produced by different systems and libraries. The timestamp identifies the time of an event occurrence, the source indicates the application or component that produced the log entry, and the data itself is the message (15).

Log data can be structured or unstructured and the way it appears depends on the framework or standard that was selected to be used, as well as the format can be dictated by the vendor of a device or software (15).



Syslog:

```
Feb 16 09:49:58 N-AKSHSH kernel: [ 3765.097562] br-815582fb2b1c: port 2(vethd670e8b)
entered disabled state
```

Python logging:

```
2024-02-16 16:06:39,714 log.py WARNING  Watch out!
2024-02-16 16:06:39,714 log.py INFO     I told you so
```

FIGURE 5. Log messages from different tools. Timestamp marked with red, source marked with green.

There are examples of textual log messages shown in *FIGURE 5*. The first message was taken from the system log produced by syslog (standard for message logging) of a personal computer that runs the Linux operating system. Two following messages were produced by the simple Python program which utilized a standard **logging** library.

Both types of messages include key elements mentioned earlier but with some noticeable differences. While humans can easily interpret different timestamp formats (e.g., March 31st, 2024, 31/03/2024, or 03/31/2024), programs require complex algorithms to extract the day, month, and year from non-standardized formats.

In addition to the textual format, logs can also be represented in a structured formats such as JSON or XML, or even stored in a binary format. Such formats make automated analysis easier since data is available in a structured way, but this comes with a trade-off. For human beings reading text files is easier than browsing nested XML data and at the same time, reading XML logs is much easier than reading the contents of a binary file. Developers need to understand the purpose of the log when deciding upon its format. (15; 12.)

4.2.2 Logging Levels

Sometimes, the amount of log messages can be overwhelming, and it is crucial to distinguish between different messages. Messages can display information needed only for development purposes, general information, errors, and as might be guessed the severity of various kinds of messages varies (15). As can be noticed from previous examples, logging errors directly affecting the software operation is much more critical than logging about the variable state change. The logging levels help developers and users filter the messages to find relevant information. The general logging levels as well as their relative severity are shown in *TABLE 1*.

TABLE 1. Typical logging levels. Adopted from (15).

Severity	Name	Purpose
1	Debug	Development or troubleshooting. Help developers identify problems or inspect program on a deeper level
2	Informational	Inform users or administrators about something occurred
3	Warning	Inform that something is missing or needed, but does not impact program’s operation

4	Error	Inform about error occurred
5	Alert	Inform that important event occurred and action needs to be taken

TABLE 1 illustrates basic logging levels, but each system or framework can select its own names, severities, and categories for logging levels. For example, the Python **logging** library does not have an Alert level, but at the same time defines a Critical level which has higher severity than the Error level (16). The syslog protocol defines two more logging levels: Notice and Emergency, where Notice represents normal but significant condition and Emergency indicates that system is unusable (17). When implementing a logging statement and selecting a logging level developers need to follow the rules of the logging system or framework used.

4.3 Saving logs

Until this point, usage of the phrase "log file" has been avoided, because means of storing the logs go far beyond the file and there are multiple factors that affect developers' decisions. There are multiple ways to organize information, and indeed using JSON format would result in JSON files, the same goes with XML and text formats. But things become more interesting as the program's or system's complexity grows.

As the log's size grows, developers might want to split it into several files that are distinguished by components that produce it. Another scenario of having multiple files is caused by the need to produce logs in multiple formats (e.g. binary, textual, and so on). If there is a necessity to share logs between different components, the logging can be configured in a centralized manner. In such architecture logs from different components are collected in one location (usually log server). If some logs need to be regularly accessible, those would be stored in a database, or distributed storage system such as Hadoop. It is difficult to use a phrase "log file(s)" when your logs are stored on multiple machines. (15.)

Sometimes, storing all the logs produced by a large system is resource-inefficient and useless. If something has been working for 5 years correctly and breaks one day, there is no need to analyse all the recorded logs. Also, storing logs produced by a complex system for 5 years would consume a lot of storage space. One approach developers have produced is log rotation. Logs can be rotated: moved, archived, or deleted when they grow too long (15). This would help save space by keeping the relevant information. The logs that were collected after the system failure and that contain the most relevant information about the system state before the crash are called snapshot logs (18).

4.4 Manual Log Analysis

As was mentioned in 4.1, there can be lots of useful information stored in logs. However, this information needs to be extracted from them. Log message analysis or lot analysis combines a set of actions to extract valuable data from logs. Manual log analysis is one of the simplest and quickest methods to examine logs. Reading logs line by line or message by message is difficult, but it still works when there are no tools for log analysis developed. This is a frustrating process, so operating systems such as Unix and Linux usually provide tools for making it easier. (15.)

There are three main actions that make manual analysis easier: log filtering, log reformatting, and log summarization.

Such tool as grep present in most Linux distributions allows filtering the log messages by a given pattern. It helps when a log contains information about multiple events that occurred and messages that are crucial for analysis are drawn in the “noise”.

After logs are filtered, those can be reformatted to modify the way information is displayed - awk is the tool that can be used for it. Awk is a programming language that allows writing scripts to extract data from textual sources. It can be used for both reformatting and summarization because it allows manipulating the output format (15).

Other tools such as Sort and Uniq can help in presenting information in a structured and concise manner.

The abilities of filtering, reformatting, and summarizing make manual analysis easier, but it should not be relied upon as a main mechanism for log analysis. This way of extracting data does not scale up and it becomes difficult to apply manual analysis for large logs. In addition, analysis often requires correlating logs from various sources. It can be done manually, but again it is a waste of time if it is done on a regular basis.

4.5 Automated Log Analysis

As the system's complexity increases so increases the amount of log messages produced by this system. Manually browsing logs in attempts to find anomalies and gain insights needs to be done from time to time (during an investigation, for example) but it is not efficient if it is done regularly. This is where automated analysis comes into play. Automated log analysis can be defined as a set of actions utilizing software tools and techniques to extract valuable data from logs in an automated manner.

Machine log analysis can be separated into four main stages: Filtering, Normalization, Correlation, and Action. In the filtering stage, the analysing program decides if a message is relevant and filters it out in case it is not relevant. This selective approach helps lower the system load. If the message passes the filtering stage, it is normalized – mapped to a generic format (for example transforming timestamps represented by string into the machine-understandable structure). Normalized logs are processed by an analyser program to extract meaningful data. After the analyser gets insights from the program, it takes action (for example, inform the administrator, send an email, backup results, and output results). (15.)

The analysis can be run after the program finishes its execution or during the runtime. The post-mortem analysis can be used to detect errors that occurred and find their root causes. The runtime analysis allows monitoring of a program's behaviour in real life, it may also help to resolve issues at early stages, as well as prevent them. (14.)

Automated analysis has its benefits and drawbacks compared to manual analysis. Two main benefits are endurance and speed. Automated analysis, compared to manual, can be run without human involvement 24 hours every day. and it would take less time and human efforts (15). However, these benefits can be achieved if the analysis algorithm is known and implemented into analyser. If instruction set for data extraction changes, algorithms need to be updated. There are Machine learning approaches in automated analysis developed for anomaly detection that does not require human involvement, however, those algorithms are not widely used in industry yet (19).

To summarize this chapter, logging is a process of recording application actions and states to a secondary interface. It is used for various purposes, including debugging, monitoring, and security. Logs can be structured or unstructured, and they can be stored in a variety of formats. Logging levels are used to distinguish between different messages, and automated log analysis can be used to extract useful information from logs without human efforts.

5 RAIN PLATFORM

Rain is a cloud-based log analysis and collaboration platform used internally at Nokia to analyse base station logs. It provides developers and testers with a browser-based interface for examining log files. Another feature simplifying the examination process allows uploading the base station logs from environments developers used to (e.g. directly from the hard drive, from the command line interface of the development environment, or even with HTTP URL). Rain is targeting to provide easy collaboration between people – analysis results and findings can be easily shared with another person by providing a URL. (20.)

Rain is a powerful platform for log analysis; its developers have already integrated it with other internal tools and continue this work further. Rain implements both manual and automated analysis.

For the manual analysis, Rain incorporates a log viewer, which allows effortlessly browse large log files. It also allows users to filter log messages, find patterns, highlight interesting lines and words, and share them with others. The example appearance of the log viewer is shown in *FIGURE 6*.

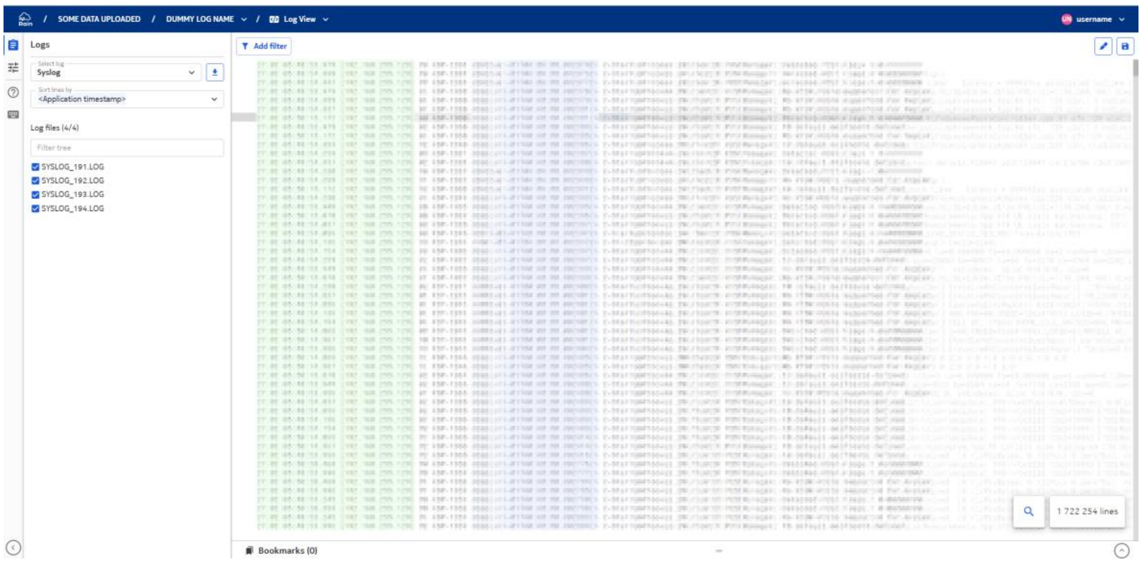
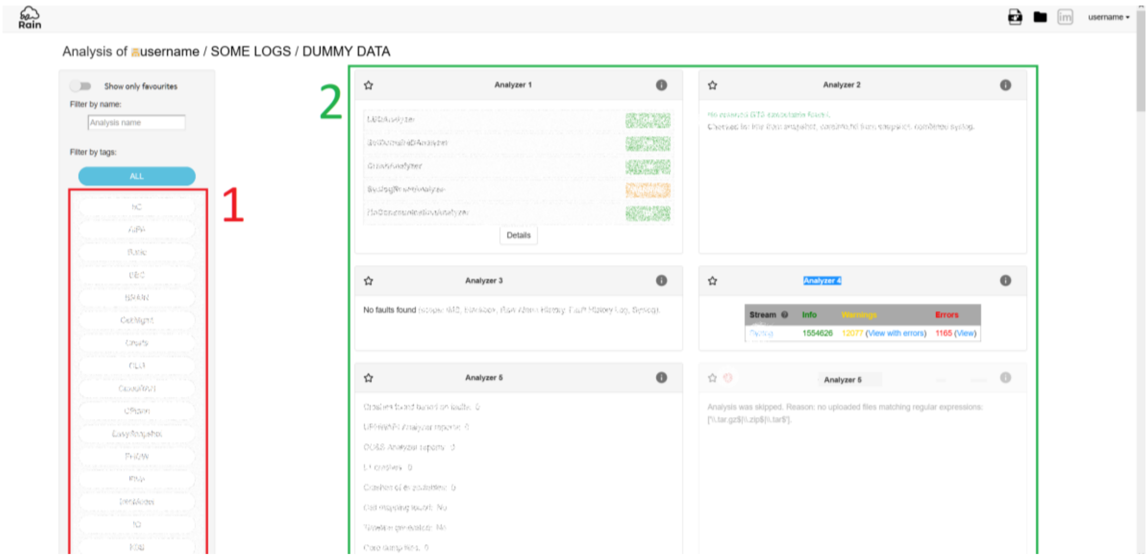


FIGURE 6. Log viewer incorporated into Rain.

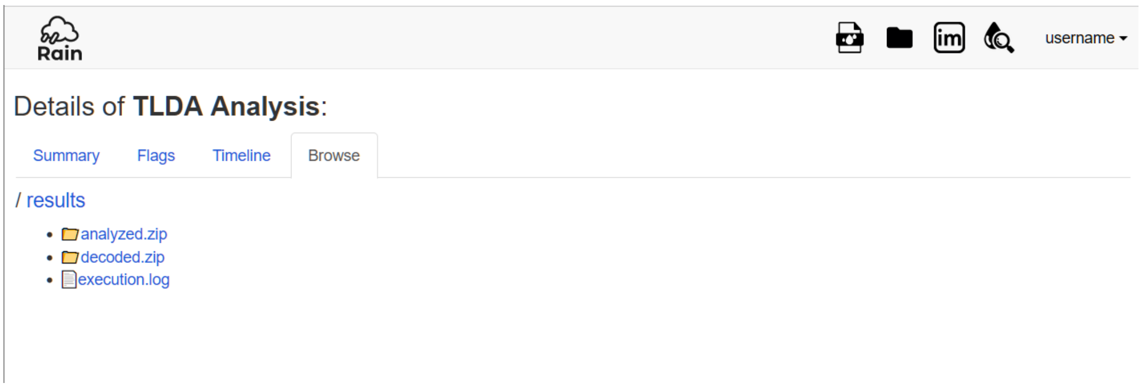
Rain automatically triggers the analysis process for uploaded logs. Rain allows running multiple plugins and integrates custom analysers for specific needs, which is important for this project. (20). This allows developers to implement analyser that fulfil their needs and easily accessible by them and others. As a browser-based tool, Rain can display analyser results visually making the investigation process easier. All the analysers' summaries can be found on the dashboard as shown in *FIGURE 7*. Users can filter relevant analysers by tag as marked under number 1, as well as search analyser with a name. Users can also click the details button to see more analysis information, for example, *FIGURE 8* shows a details view of TLDA (Technical Log Decoder and Analyser) analysis results, which allows users to download relevant files.



No faults found

Analysis was skipped. Reason: no uploaded files matching regular expressions: ['\.tar.gz$|\.zip$|\.tar$'].

FIGURE 7. Rain analysers dashboard example with filtering by tags (1) and analysis summaries (2).



Details of **TLDA Analysis**:

/ results

- analyzed.zip
- decoded.zip
- execution.log

FIGURE 8. Detailed view of one of the Rain analysers.

Analyser consists of two parts: backend and frontend. The backend part's purpose is to analyse the source data and provide analysis results. Frontend is responsible for displaying those results in a browser window. Rain provides a tool to assist during the development process which can initialize the structure, prepare the data from the real case, and run an analyser. (21).

Python programming language is used for backend implementation, and Rain provides a library that allows reading files using official API as well as using other analysers results (21). Python is a convenient choice due to its ease of use, efficiency, and wide range of data manipulation tools. Since each analyser is run as a container, it can pack any non-standard dependency into the image and use it during the development process.

The backend analyser should provide *results.json* and *details.json* to be used by the frontend. The first file should contain high-level information about the analysis, whereas the second file should provide details about the analysis results and data extracted. Optionally analyser can produce other files that can be read by the frontend or downloaded by users.

Frontend presents data using traditional browser technologies: HTML, CSS, and JavaScript. To ease the development, it uses the React framework by default, but it can be anything else if developer wants to. Frontend should implement two components: Summary and Detailed views Those would be used later by the Rain to display analysis results as shown in *FIGURE 7* and *FIGURE 8*.

Besides memory and time efficiency, the analyser should be backward compatible, meaning that it should correctly process logs collected with different software versions (22). It can be easily achieved if the analyser deals with hardware logs, which are standardised, but it is another story if working with developer-defined log statements, as those may be changed between different software revisions, as was noted in 4.1. The chapter 6.2 of development process would explain how this problem was addressed during analyser development.

6 DEVELOPING AN ANALYSER

After plunging into research about logging and MACsec specifications, development process started. This chapter will provide detailed description of requirements which were set for this project, design decisions which were made, overview of implementation and testing processes, as well as optimization of an analyser and its integration into Rain testing platform.

6.1 Specifications

There was a list of requirements set for this project. Generally speaking, the analyser should take an input log, extract the information about MKA connections and present it. Following paragraphs explain each phase in details.

The analyser should take a base station log as an input. This base station log contains multiple logs from various base station components, so program should identify relevant files and process them.

The analyser should extract information about MKA connections. For each of MKA connections it should detect MAC addresses of Radio Unit and Distributed Unit and list of events which are relevant for this connection. For each of the events it should provide a timestamp indicating event occurrence time, software component that produced the log message about this event, as well as MAC addresses of connection participants to map event to the connection. Each event should also contain information about a file name and message number to find an associated log file and log message in case manual analysis is needed.

The analyser should present an extracted information in a user-friendly manner on the Rain platform.
Moreover, it should also highlight potential errors that has happened during the lifetime of secure connection.

In addition to functional requirements, the analyser should comply with performance and resource limitations set by the Rain platform.

To conclude, a lot of aspects were considered for making development process strict and well-aligned. The more requirements are discussed beforehand, the more organised workflow is going to be fulfilled.

6.2 Design

Design has started addressing the problem set in the chapter 5 - Rain analyser should be backward compatible. As fronthaul traffic protection might require changes into implementation, log messages can be modified in the future. It does not only affect the analyser implementation which should be designed to support various formats of log message. Adding processing for a new log message would be difficult too since analyser's source code is stored in its own repository.

Following design requirements, "Technical Log Decoder and Analyser" (TLDA) is synchronized with software producing logs. TLDA is integrated into Rain platform (and was already shown on *FIGURE 8*) - it takes the same input as all the other analysers, and results it produces can be utilised by other plugins. It does not only allow integrating analysers into it, but also it selects correct version of an analyser based on the software version specified in logs. However, analysis results are saved as files, so it is difficult to implement interactable interface using only TLDA, that is why the Rain analyser is needed (23).

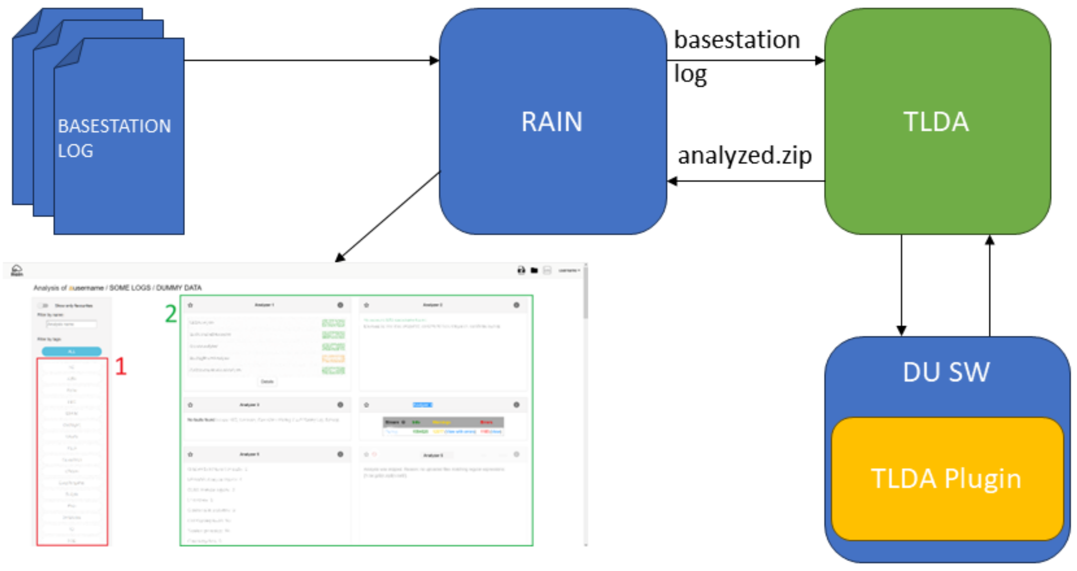


FIGURE 9. High-level analyser architecture.

Analyser's implementation was divided between Rain platform and TLDA. The latter tool takes the same input as any other Rain analyser, so implementing plugin for it is aligned with functional requirements.

Architecture explaining their relation is shown on *FIGURE 9*. Once base-station logs are submitted to the Rain platform, they are passed to the TLDA. TLDA runs the analyser plugin which is synchronized with software component. After the analysis is completed in TLDA side, the Rain analyser takes the data from *analyzed.zip* file, which contains analysis results. As a result, those are displayed on the Rain platform.

6.3 TLDA Plugin

The analyser plugin running in TLDA is responsible for main processing and data extraction from logs. Its architecture should be easy to understand, extend and maintain. Since it deals with raw log data it should also meet performance and memory consumption requirements.

The architecture presented on *FIGURE 10* from work (24) became an inspiration for the one that will be used in this project. It separates analyser program on multiple modules from which the important ones are Parsing, Inference Engine (also known as Analyser), Interface Engine and Data Manager. Parsing is responsible for extracting the data from raw log files into machine understandable data structures managed by Data Manager. Data Manager can store data in SQLite database and in computer's memory. After the information was extracted, Inference Engine starts. In connection with Mind Maps scripts (implemented with the language developed by article authors) it performs analysis on the extracted data. Interface Engine is then responsible for presenting the results to a user. (24.)

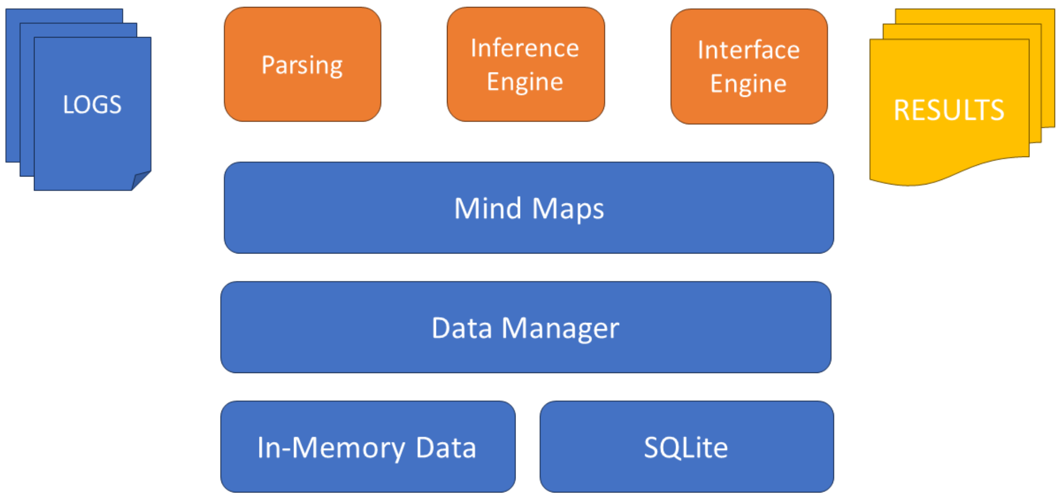


FIGURE 10. Analyser high-level architecture. Adopted from (24).

Presented model was updated to fulfil the requirements for this project. Parsing step will remain and be implemented in a *parser* module. Inference Engine will be implemented as an *analyzer* module. Data Management will be implemented in *cache* module and would use only In-Memory data for storing the information. As it was discussed in previous chapter, TLDA does not have user interface, so instead of Interface Engine there would be function which exports relevant data from *cache* to files. These files would be later used by Rain plugin. The final architecture of the analyser is shown on *FIGURE 11*. Module *datatypes* contains dataclasses and datatypes definitions to store extracted data and run static checks on code.

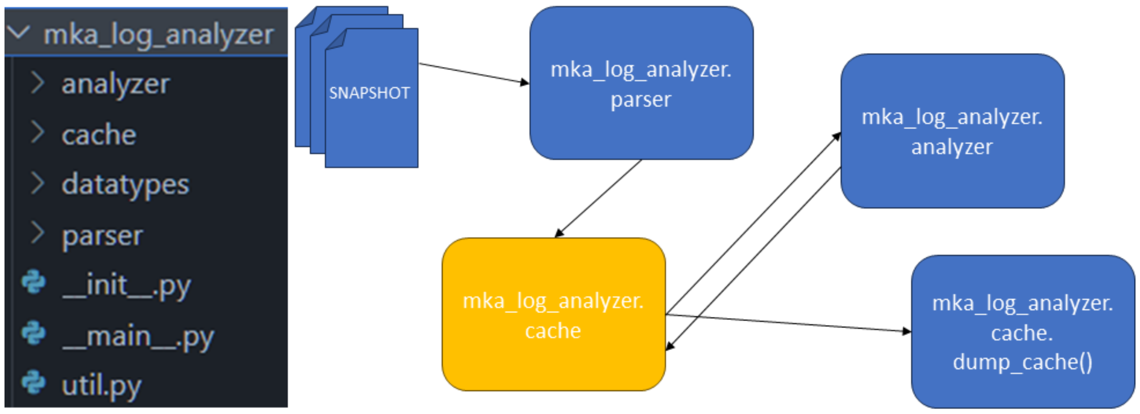


FIGURE 11. TLDA mka_log_analyzer folders structure and plugin architecture.

6.3.1 Parser Design

The parser is crucial parts of the analyser. It is responsible for converting human readable textual data into machine-understandable data structures that can be used on analysis stage. Its design affects the extendibility, maintainability, and performance.

To extract the data from logs, the parser needs to go through all the messages at least once. If the number of relevant messages is small and constant, it would be fine to match every message to a list of pre-defined patterns. But such approach is not suitable if the number of patterns is large and can extend. Personal study has shown that the number of patterns affects the complexity of the program at least linearly (script has utilized simple linear expressions without backtracking) see *FIGURE 12*. As the result, the number of regular expressions applied to each line should be optimized.

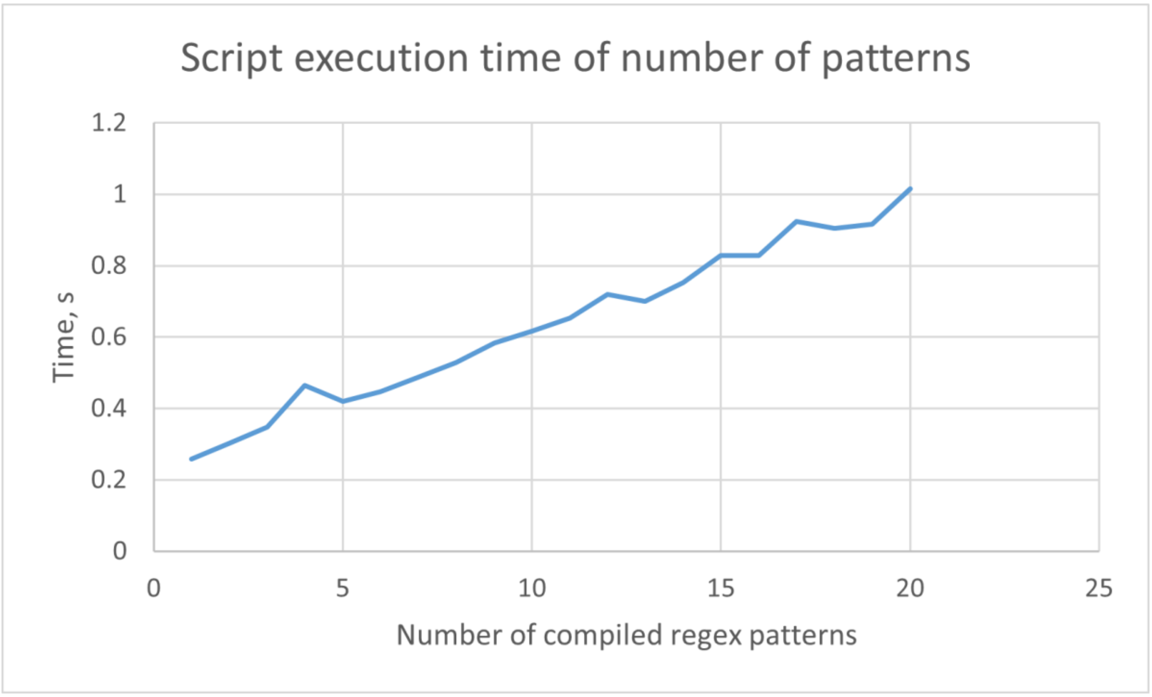
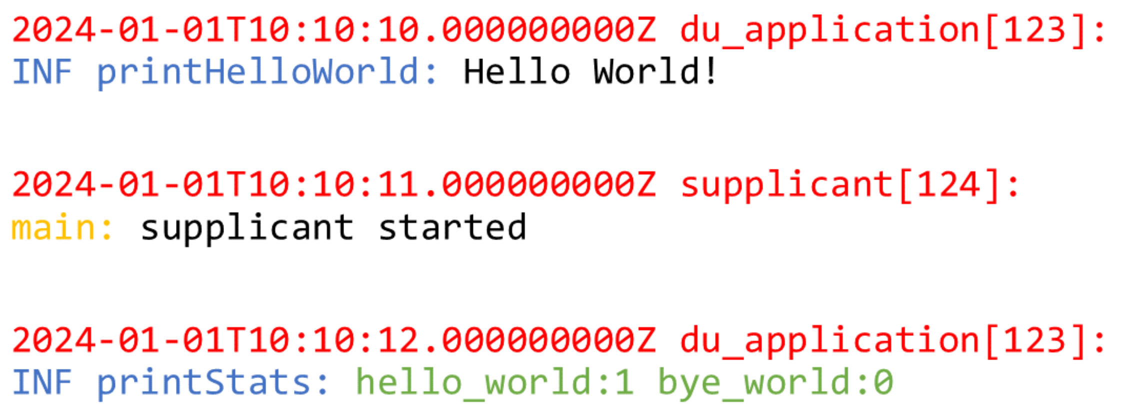


FIGURE 12. Local studies on 150000 lines files with compiled Python regular expressions. Python version 3.8.18

As it was mentioned before, the analyser should process log messages that were generated by different software components such as Distributed Unit, IEEE 802.1X supplicant, Radio Unit applications. *FIGURE 13* shows example log messages. There are common structures that can be detected from those messages. All of them have general information such as timestamp, application name, and process id (highlighted with red) that does not depend on application. Application-specific information has common parts such as logging level and function name (highlighted with blue and yellow) too. Then function-specific information can be also parsed differently: it can be just text (i.e. “Hello World!” or “supplicant started”) or structured data (e.g. number of prints in the third message).



```
2024-01-01T10:10:10.000000000Z du_application[123]:
INF printHelloWorld: Hello World!

2024-01-01T10:10:11.000000000Z supplicant[124]:
main: supplicant started

2024-01-01T10:10:12.000000000Z du_application[123]:
INF printStats: hello_world:1 bye_world:0
```

FIGURE 13. Example log messages.

It is also important to note that all information about message (timestamp, application, process id, function name, message, or other data) is important as it is describing the message. The straightforward approach would require defining different structures for each message. As it was described earlier, it is not the optimal method.

Solution selected in this project is implement parser as a tree-like structure (see *FIGURE 14*) and it relies on assumption that meaningful information can be extracted from a prefix of the message (first prefix is generic data, then application-specific data, then function data, etc.). The core of the parser tree will be a ParserNode class responsible for data extraction and passing the rest of the message to its children-nodes. This way the amount of regular expression matches will be optimal for each line.

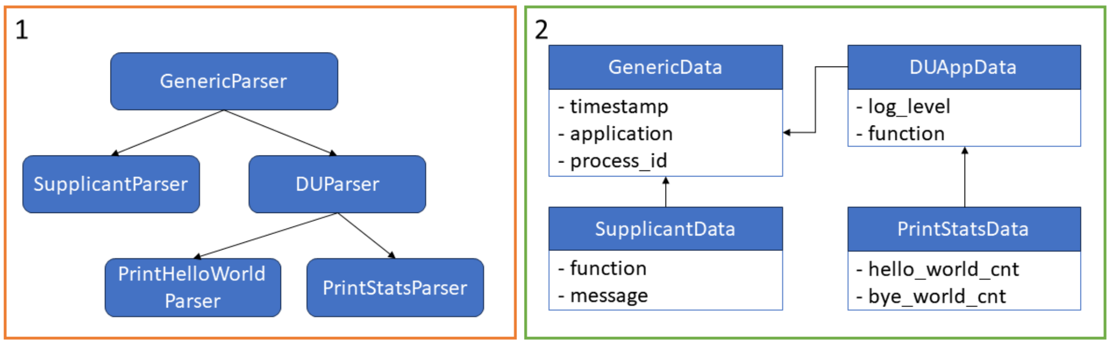
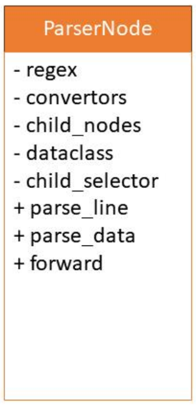


FIGURE 14. Example of a parser tree configured for an example log messages (1) and corresponding dataclasses (2).

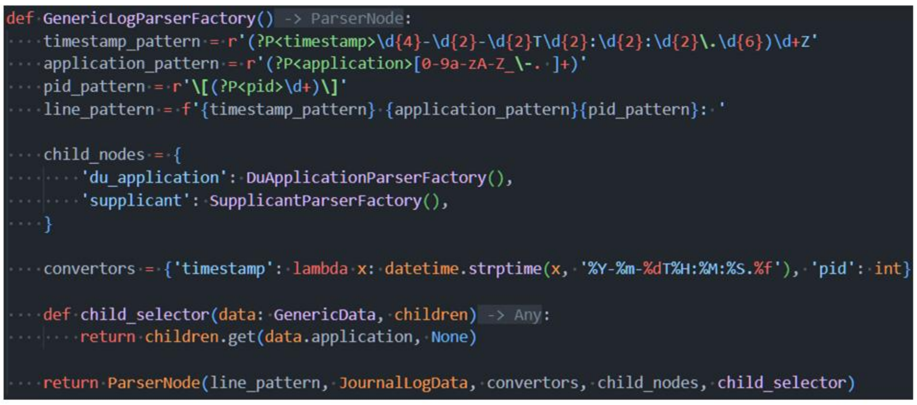
To store extracted data, it was decided to use **dataclasses** python library. Alternative structure to store the information could be Python dictionary, but dataclasses were selected, because using attributes instead of keys requires less syntax in source code and most importantly, it allows static typing checks on a source code. Static checks help catch errors before running actual program. Additionally, dataclasses support inheritance as normal Python classes, allowing developers to avoid copying common attributes across classes declarations and making it easier to define relations between parent and child classes.

A Unified Modelling Language (UML) class diagram of ParserNode is shown on *FIGURE 15*. Its attributes consist of a *regex* – compiled regular expression to extract information, *convertors* – an associative array containing functions to convert a type of extracted string, *child_nodes* – an associative array containing children ParseNode classes, *dataclass* – an associated dataclass to store extracted data, and *child_selector* – a function that selects a child node based on extracted data. Using compiled regular expressions is more efficient in case pattern needs to be used multiple times, because regular expression engine needs to parse it only once (25). Another note is that ParserNode would try to feed the rest of the message to every child node in case *child_selector* function is not provided.

1



2



```python
def GenericLogParserFactory() -> ParserNode:
    timestamp_pattern = r'(?P<timestamp>\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}\.\d{6})\d+Z'
    application_pattern = r'(?P<application>[0-9a-zA-Z_\-. ]+)'
    pid_pattern = r'\[(?P<pid>\d+)\]'
    line_pattern = f'{timestamp_pattern} {application_pattern}{pid_pattern}: '

    child_nodes = {
        'du_application': DuApplicationParserFactory(),
        'supplicant': SupplicantParserFactory(),
    }

    convertors = {'timestamp': lambda x: datetime.strptime(x, '%Y-%m-%dT%H:%M:%S.%f'), 'pid': int}

    def child_selector(data: GenericData, children) -> Any:
        return children.get(data.application, None)

    return ParserNode(line_pattern, JournalLogData, convertors, child_nodes, child_selector)
```

FIGURE 15. ParserNode UML diagram (1) and GenericParserFactory function definition (2).

ChildNode class also defines multiple methods. *parse_line* method takes a log message in a string format, *parse_data* takes a log message in a dataclass format (usually provided by a parent class). Both methods return *Null* if data cannot be extracted. Otherwise, both methods pass an associated dataclass with extracted data to child nodes of the ParserNode class. If data extraction succeeds in child nodes, the ParserNode would return data extracted by its child, it would return own dataclass otherwise.

To give an example on how parsing process goes, this paragraph describes scenario where the third message from *FIGURE 13* would be given as an input to the example parser configured on *FIGURE 14*. Firstly, GenericLogParser will extract timestamp, application name and process id from the message and put those into object of type GenericLogData. Then it will pass extracted data to the DUParser that will extract logging level and function name. Then DUAppData object will be passed to the PrintStatsParser which will extract prints statistics. Since latter parser does not define any children nodes and it successfully extracts data, parser tree will return PrintStatsData object with the attribute values from *TABLE 2*.

TABLE 2. PrintStatsData attributes' values extracted from example log message.

Attribute	Value
timestamp	01 Jan 2024 10:10:12
application	du_application
process_id	123

log_level	INF
function	printStats
hello_world_cnt	1
bye_world_cnt	0

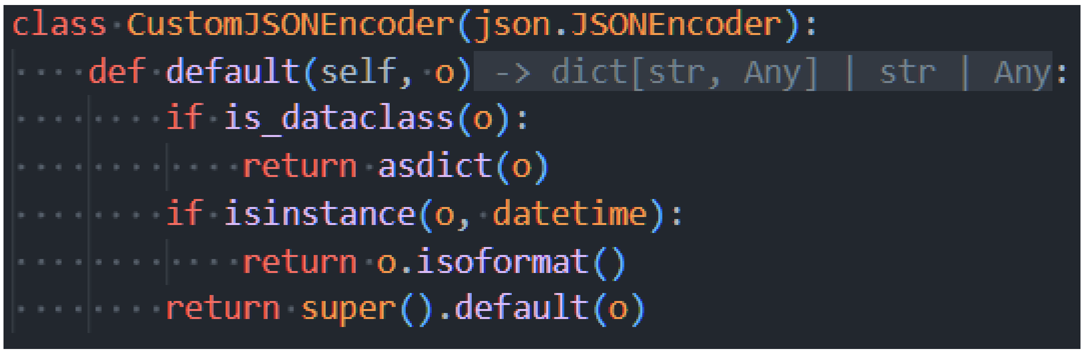
After the log message has been parsed, it is saved to the cache.

6.3.2 Cache

The cache is designed to use In-Memory data structures. It is implemented by a singleton Cache class in *cache* module. A singleton pattern ensures a class only has one instance and provides a lobal access point for it (26). Methods of this class allow performing basic operations on data such as storing new object, retrieving, deleting, and updating existing ones. Data stored in cache is grouped by its type, and each object is saved under unique key. This key can be defined by user or assigned automatically based on number of objects which are already stored. Cache class relies on python dictionaries – associative arrays that use named keys to identify objects stored in them.

Additionally, Cache class has a *hash* static method that calculates the hash value of provided object. Those values can be used as keys if user want to store only unique objects. When processing multiple log files, same message can appear several times, so ignoring duplicates helps save the resources and keep data clear.

The *cache* module also has a function *dump_cache* that can export data stored in Cache into JSON files. Implementation of this function required to extend a standard JSONEncoder class from *json* module to support dataclasses (see *FIGURE 16*). This function will be later used to save analysis results stored in cache into multiple JSON files that would be delivered from TLDA to the Rain analyser.



```python
class CustomJSONEncoder(json.JSONEncoder):
    def default(self, o) -> dict[str, Any] | str | Any:
        if is_dataclass(o):
            return asdict(o)
        if isinstance(o, datetime):
            return o.isoformat()
        return super().default(o)
```

FIGURE 16. JSONEncoder class extended to work with dataclasses.

After all the logs processed, and information from them extracted, the analysis phase starts.

6.3.3 Analysis

Analysis is implemented in an *analyzer* module. This module was designed to be modular, so every developer can implement a custom plugin by inheriting BaseAnalyzer class. This plugin will be able to declare the ones it depends on and have full access to cache object when called.

The core of analysis process is AnalyzerPluginManager class. An instance of this class is responsible for running plugins in the correct order according to their dependencies.

After instantiation of this class, plugins should be registered using *register_analyzer* method that will store plugin into dictionary by the name of the class. Before running the analysis, AnalyzerPluginManager instance would run a dependency check on registered plugins that will report if some analyser has defined unregistered plugin as its dependency. It would also run a depth-first search – one of the basic graphs traversing algorithms - to detect a cyclic dependency (27). After all the checks finished, plugins would be sorted using a modification of graphs algorithm called breadth-first search (27). Then AnalyzerPluginManager would execute every plugin in order.

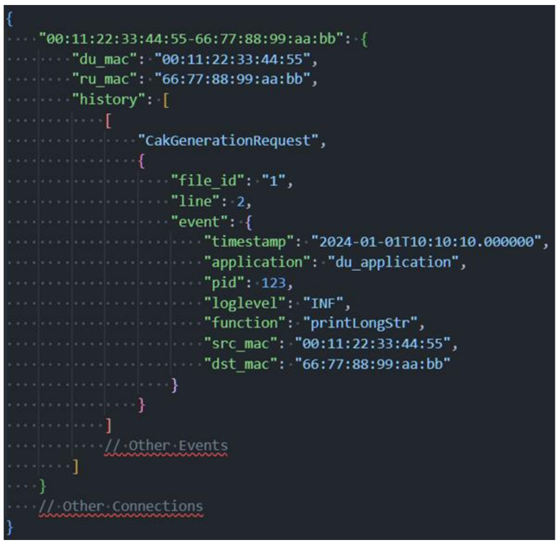
In the scope of this project there has been a plan to implement the following plugins:

- PrintLongStrAnalyzer – plugin to extract data that was spread across multiple log messages using printLongStr function. Messages produced by this function contain data about multiple events, so PrintLongStringAnalyzer can be inherited to provide information about relevant events printed using this function.

- CAKGenerationRequestAnalyzer – plugin inheriting PringLongStrAnalyzer that extracts information about Connectivity Association Key generation request – event prior to connection establishment.
- ConnectionSetupReqAnalyzer – plugin inheriting PrintLongStrAnalyzer that extracts information about connection setup requests and parameters provided for it.
- ConnectionDeleteReqAnalyzer - plugin inheriting PrintLongStrAnalyzer that extracts information about connection delete requests and parameters provided for it.
- MKAConnectionsAnalyzer – plugin that groups all extracted events by the connection they belong to. This plugin also sorts the events by the time of their occurrence. After plugin finished its execution, grouped events are saved to the cache.
- DuRuMacAnalyzer – plugin that detects the MAC addresses of Distributed Unit and Radio Unit between which MACsec connection has been established. Some types of log messages contain this information, so it is being extracted and propagated to the dataclasses containing overall information about the connection.

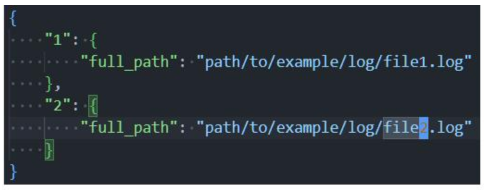
After the analysis phase is completed, the *dump_cache* function is called to save information about MKA connections and source files from which message was extracted into MKAConnection.json and FileReference.json files respectively (examples are shown on *FIGURE 17*). This concludes the role of TLDA analysis. After finishing the execution its results will be compressed into *analyzed.zip* and can be accessed by the Rain analyser.

1



```
{
    "00:11:22:33:44:55-66:77:88:99:aa:bb": {
        "du_mac": "00:11:22:33:44:55",
        "ru_mac": "66:77:88:99:aa:bb",
        "history": [
            [
                "CakGenerationRequest",
                {
                    "file_id": "1",
                    "line": 2,
                    "event": {
                        "timestamp": "2024-01-01T10:10:10.000000",
                        "application": "du_application",
                        "pid": 123,
                        "loglevel": "INF",
                        "function": "printLongStr",
                        "src_mac": "00:11:22:33:44:55",
                        "dst_mac": "66:77:88:99:aa:bb"
                    }
                }
            ]
            // Other Events
        ]
    }
    // Other Connections
}
```

2



```
{
    "1": {
        "full_path": "path/to/example/log/file1.log"
    },
    "2": {
        "full_path": "path/to/example/log/file2.log"
    }
}
```

FIGURE 17. An example of MKAConnection.json (1) and FileReference.json (2) contents.

6.4 Rain analyser

The analyser that neds to be implemented for Rain platform is responsible for presenting the results from TLDA plugin to the user. The analyser's implementation consists of backend to pre-process the data and frontend to present it.

6.4.1 Frontend

Frontend is the "face" of MKA analyser, and it is the main interaction point for user. Its appearance should be easy to understand and provide all the information clearly. The main use case for this analyser is to investigate the lifetime of secure connections that were established (or attempted to be established) in a base station. To achieve this, it was decided to display interactive timeline where all the events can be browsed and filtered.

With regards to a chapter about Rain platform, frontend implementation should provide two components. The first one is execution summary – this component will appear on the dashboard view of the platform. The second one is detailed view – developers can decide what to put there.

For execution summary view it was decided to display all the extracted connections as well as statistics of events. An example on *FIGURE 18* shows that connection with id **00:11:22:33:44:55-66:77:88:99:aa:bb** had 4135 events reported with informational and zero events with error logging levels. After clicking the connection id, user would be forwarded to the detailed view.

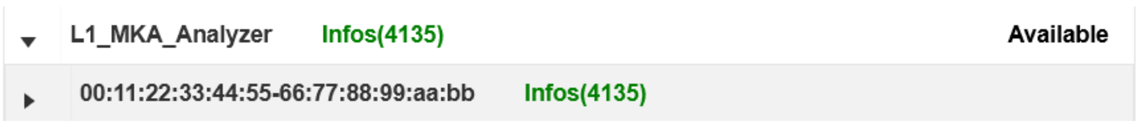
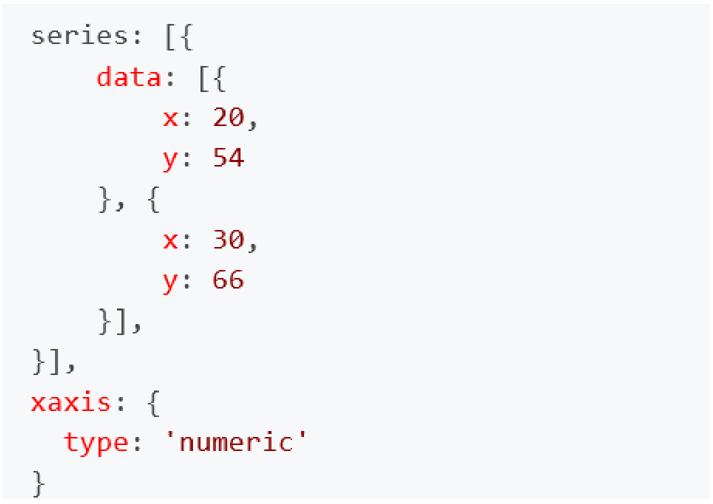


FIGURE 18. Example of MKA analyser's summary view.

As per detailed view, it should display the timeline with all the events visible as markers. When the marker is hovered with mouse pointer, a tooltip should appear and provide detailed information about the event. If the marker is clicked, a modal window should open displaying detailed information about the event, as well as providing a link to the source file from which this event has been extracted. Detailed view should also contain selectors that would filter the data by connection id, application name, logging level, and event name.

It was decided to use ApexCharts.js framework for data visualization. It provides a highly customizable interactive data visualization components that can be integrated into the web application (28). The line chart with timeline x axis and categorial y axis was selected to display extracted events.

To draw a chart ApesCharths.js requires data to be in a specific format (See *FIGURE 19*) that differs from the one provided by the TLDA analyser (*FIGURE 17*). Reformatting could be done on a clients-side, but it will increase the load on user's device, and it is not needed, since there is a backend of the Rain analyser. Data preprocessing would be described in detail in the section 6.4.2.

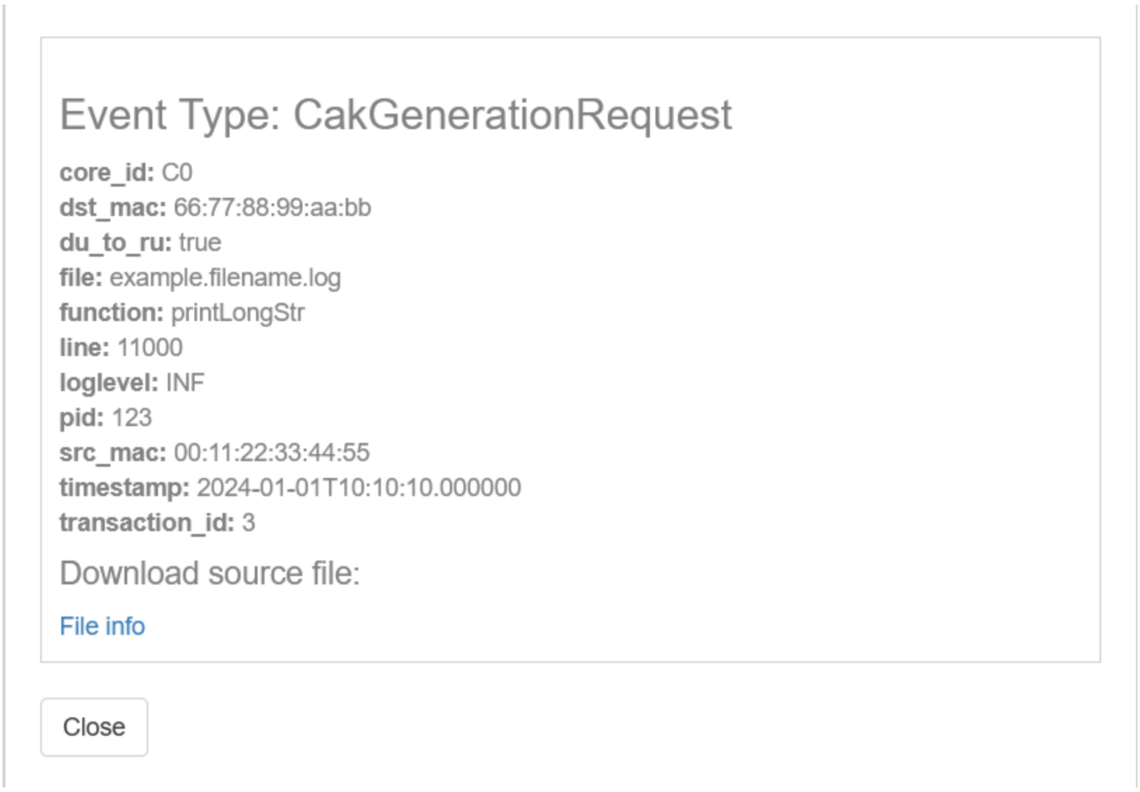


```
series: [{
    data: [{
        x: 20,
        y: 54
    }, {
        x: 30,
        y: 66
    }],
}],
xaxis: {
  type: 'numeric'
}
```

FIGURE 19. ApexCharts.js series format (28).

Frontend was implemented using React framework because it has steep learning curve and comes preconfigured within analyser skeleton provided by Rain. To fulfil the results presentation's requirements, the following components were created for a detailed view of the MKA analyser:

- **<MkaEvent />** - this component is used to display detailed information about the event when a marker clicked (*FIGURE 20*). It is implemented as a modal window that appear above other components. It also contains a link that allows downloading the source log file.



Event Type: CakGenerationRequest

core_id: C0
dst_mac: 66:77:88:99:aa:bb
du_to_ru: true
file: example.filename.log
function: printLongStr
line: 11000
loglevel: INF
pid: 123
src_mac: 00:11:22:33:44:55
timestamp: 2024-01-01T10:10:10.000000
transaction_id: 3

Download source file:

File info

FIGURE 20. An example of <MkaEvent /> component.

- **<MkaFilterBar />** - this component contains selectors which filter data that is being displayed on the chart (*FIGURE 21*). It is implemented using several <Select /> components from react-select library.

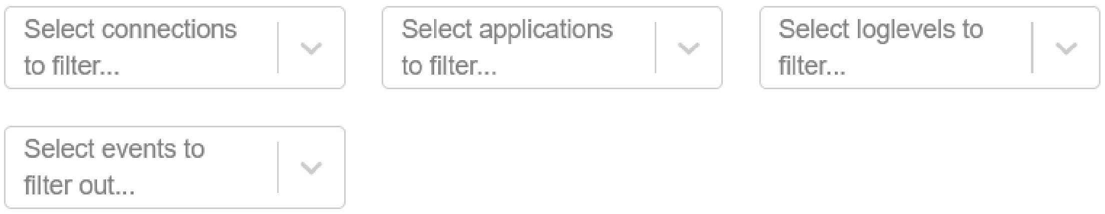

Select connections to filter...
Select applications to filter...
Select loglevels to filter...
Select events to filter out...

FIGURE 21. An example of <MkaFilterBar /> component.

- **<LineChart />** - this component is on top of <ReactApexChart /> component and responsible for providing the configuration and styles for this chart component (*FIGURE 22*). It defines a custom behaviour for zoom action, because by default, zoom resets if any of filters is changed. It configures function that would be called once a marker is clicked. Also, it configures the tooltip which appears on the marker hovering. Among others, <LineChart /> component configures such zoom and pan tools provided by ApexCharts, but not enabled by default.

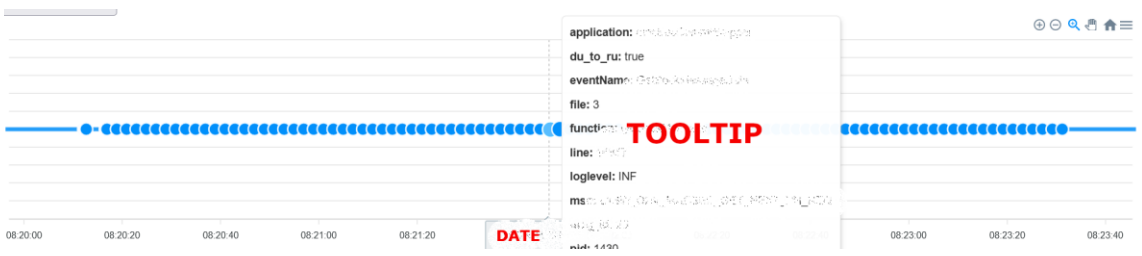


FIGURE 22. An example of <LineChart /> component.

- **<MkaTimeline />** - this component combines all the other to present analysis results to the Rain user in analyser's detailed view (*FIGURE 23*). Besides displaying other components, it is responsible for filtering the series data and exporting filtered data into JSON file.

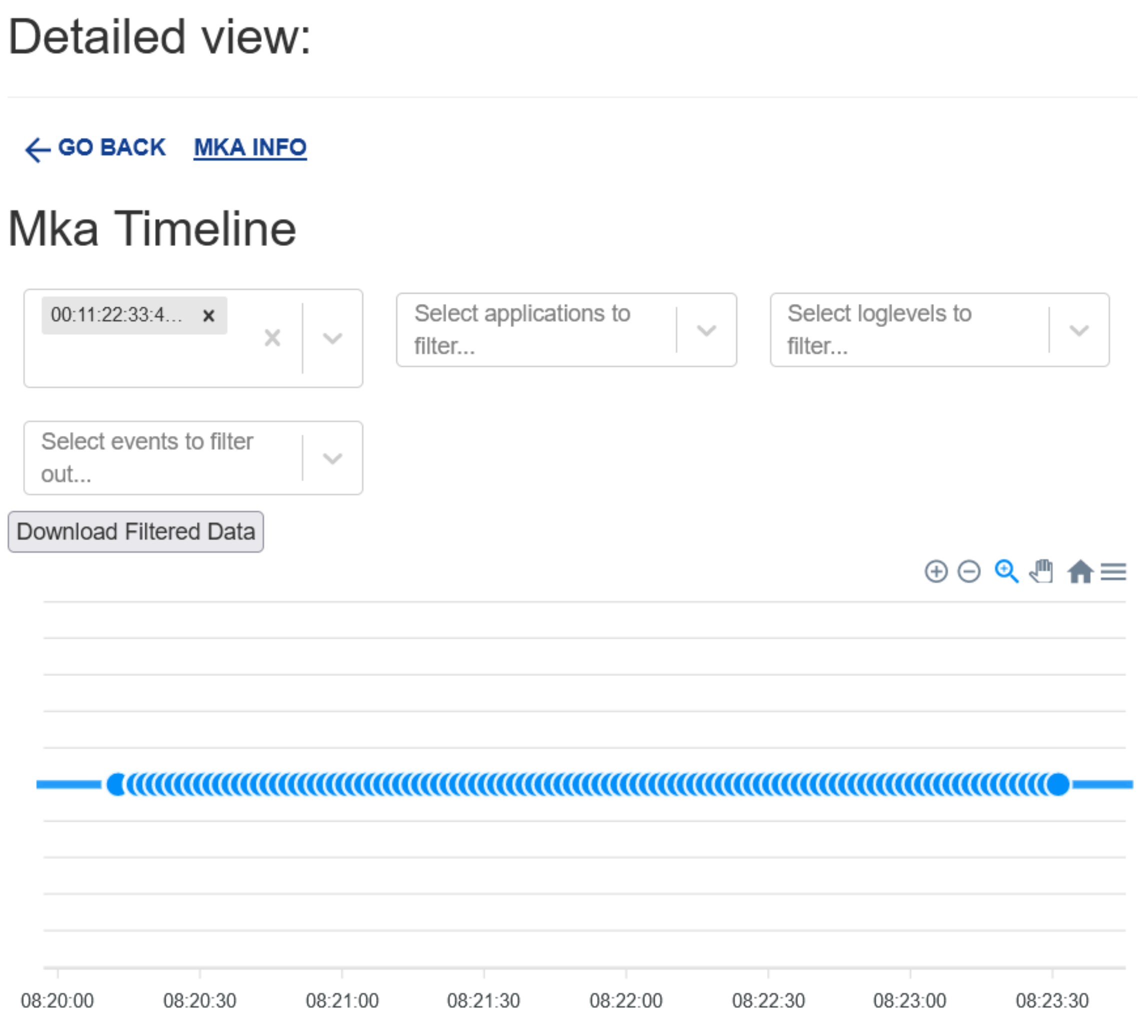


FIGURE 23. An example of <MkaTimeline /> component.

6.4.2 Backend

Data coming from the TLDA analyser cannot be used by frontend directly, thus backend is responsible for changing the data format. Rain also provides possibility to download file from the uploaded logs, so backend should map those files with ones used by TLDA during parsing.

Its implementation is not complicated. After defining the TLDA analyser as a dependency, Rain platform will automatically run the MKA analyser only when TLDA results are available. Backend needs to read the relevant files from the archive using **zipfile** library, change their structure to match the requirements of ApexCharts.js (*FIGURE 19*). Besides that, backend extracts unique values of all software components involved into log formation, all the event types and connection ids, and provides those in a dedicated structures to be used by filter selectors in a frontend.

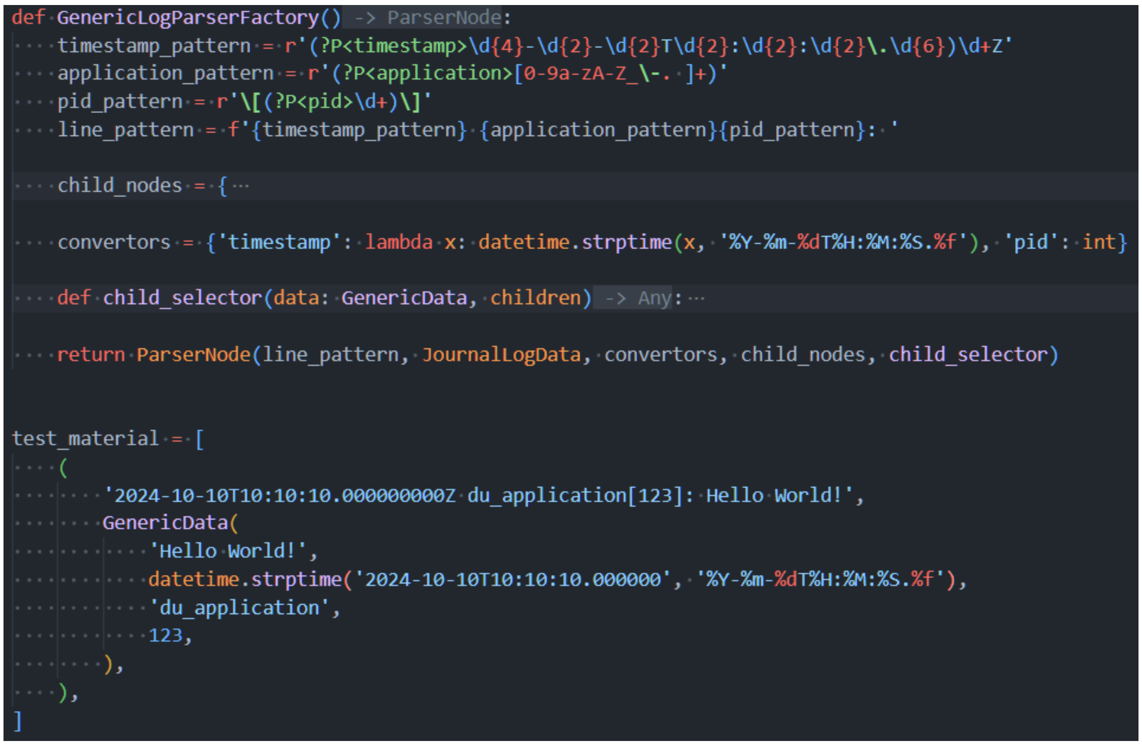
To save the results, backend is required to provide three files: *analysis.log* with analyser's log messages for development and troubleshooting purposes, *results.json* that will be used by execution summary view, and *details.json* that will be used by detailed view.

6.5 Testing

Testing is an important part of development process. Not only does it help to verify that implementation working correctly, but also reduces the time needed for implementing.

During TLDA analyser implementation, there were multiple unit tests implemented that helped to validate regular expressions of parser nodes, as well as check end-to-end results. Testing played a huge role in Rain analyser development too.

Parser tree of the TLDA analyser contains a log of regular expressions that leave a large room for a mistake. The decision to test every parser node's regular expression helped to save a lot of developing time. Test script was implemented using **pytest** library and designed in a way that it automatically collects test materials from the modules that define various parser nodes. This way developers do not need to manually add new testcase for new parser node implemented, instead they just need to add a **test_material** array into the same module (See *FIGURE 24*).



```python
def GenericLogParserFactory() -> ParserNode:
    timestamp_pattern = r'(?P<timestamp>\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}\.\d{6})\d+Z'
    application_pattern = r'(?P<application>[0-9a-zA-Z_\-. ]+)'
    pid_pattern = r'\[(?P<pid>\d+)\]'
    line_pattern = f'{timestamp_pattern} {application_pattern}{pid_pattern}: '

    child_nodes = {…

    convertors = {'timestamp': lambda x: datetime.strptime(x, '%Y-%m-%dT%H:%M:%S.%f'), 'pid': int}

    def child_selector(data: GenericData, children) -> Any: …

    return ParserNode(line_pattern, JournalLogData, convertors, child_nodes, child_selector)


test_material = [
    (
        '2024-10-10T10:10:10.000000000Z du_application[123]: Hello World!',
        GenericData(
            'Hello World!',
            datetime.strptime('2024-10-10T10:10:10.000000', '%Y-%m-%dT%H:%M:%S.%f'),
            'du_application',
            123,
        ),
    ),
]
```

FIGURE 24. Example test material for GenericLogParser.

Such tests allow testing the regular expression only on the part of the line that is relevant to the parser. In case developer needs to test parser tree from the beginning to the end, they should modify another testing script and provide a complete input line there, along with target dataclass.

MKA analyser implementation also contains tests for Cache and AnalyzerPluginManager classes. Those scripts test basic functions of the classes securing the implementation even better. Analyser plugins also have their unit tests that provide a pre-configured cache, run the analyser, and check its results.

Both frontend and backend of Rain analyser have been tested too. Since backend was implemented using Python, **unittest** module was used for testing. For testing the frontend components Jest (framework for JavaScript testing) was used.

Backend is secured by unit tests which allows testing different functions of the implementation. It also relies on mocks to control some libraries calls (including zip archive manipulations, input output operations, etc). By providing input and comparing output with expected one these tests ensure that backend operates correctly.

Frontend has unit tests too. Those are used to check functions that support the implementation (i.e. filtering or sorting) and React functional components. By mounting those components in a virtual environment, test functions verify that correct HTML elements are used. Also, Jest supports a snapshot testing – once some component is mounted, its HTML code is saved, and later test runs compare an updated version against this snapshot. Updating the snapshot requires developer consciously compare new version of HTML code, preventing accidental modifications into the codebase.

Besides unit tests, the MKA analyser was tested on the development instance of Rain platform. It is provided to developers, to test their analysers with real data on a real platform without interrupting original instance used by many people.

6.6 Optimization

After the TLDA analyser was developed and tested, its memory consumption was high, even though it was withing the limitations of analysing platform. There was an extra study conducted to increase the performance of the analyser.

The largest object consuming the runtime memory is the Cache instance. It stores all the data extracted from logs and exists throughout the analyser's life cycle.

First action that reduced the analyser's memory consumption on a real data test was adding a filter for some of the messages. Originally, all the data successfully extracted by parser tree will be saved. Even if the log message was irrelevant for analysis, it would be saved because generic log or application-specific parsers will successfully extract the information. Additional filtration by datatype extracted from the message, has reduced the size of the occupied memory almost by 50%.

Another action was targeting the data that is being stored in cache. Dataclasses which were chosen to store the information extracted from log messages are not optimized by default. As all Python classes they store their attributes inside the dictionaries that consume a lot of memory resources, due to their ability to extend dynamically. It also affects the speed of access, because accessing object's attributes involves reference call to get the dictionary storing those attributes.

Slots on the other hand, allow developers to use class attributes directly stored in the class object, without using the dictionary (29). It helps to reduce size of the object and access speed.

Slots are perfectly fit into the implementation, because dataclasses are defined to store fixed amount of information and do not require to be extended dynamically. Also, there are many dataclass objects, so reducing the size of each one a little helps saving resources used by the analyser. Employing the slots to store the log messages data has improved the memory consumption by another 16%.

Overall memory consumption improvement was around 61% compared to the original implementation. This optimisation adds a room for further analyser improvements.

7 RESULTS AND DISCUSSION

This chapter describes the results reached during the development of the MKA log analyser for the Rain platform. It covers functionalities delivered by the analyser, explores its limitations and improvement possibilities, and summarises the project's achievements.

During the course of the project, complex yet modular architecture was designed to address defined requirements and avoid version conflicts. Finally, two analysers were implemented to work together, analysing information about MKA connections from logs.

TLDA analyser was designed to be easily extendable and maintainable. Analyser was also optimised to consume less resources, leaving a space for improvement possibilities in the future. Analyser was integrated into TLDA tool and is capable of analysing the logs produced by new software builds.

Following this, Rain analyser was developed and integrated into the platform. This user-focused analyser integrates with the Rain platform, presenting clear visualizations of MKA connection details on a timeline. Users can filter events to narrow the analysis scope and download filtered data for further investigation.

While current implementation offers significant functionalities, there are possibilities for further development of MKA analyser. Radio Unit logs processing was originally planned to be implemented in the scope of this project, but this information is not processed currently. Thanks to the design of TLDA parser, functionality can be extended to cover those. To add, analyser can be extended to cover more complex scenarios such as SAK rekeying events that are crucial for maintaining secure communication.

Even though the current frontend version of Rain analyser presents detailed information about MKA connections, it can also be the subject for improvement. In addition to displaying all the connection events, it could present connection timing and statistics for the most recent connection or attempt of establishing one.

Technical documentation on TLDA analyser as well as specification for MKA analyser were also implemented for the Nokia's intranet in the scope of the project. This thesis work can be used internally for familiarising with analyser's architecture.

To conclude, the MKA log analyser is a valuable tool for Rain platform users. It empowers them to efficiently analyse MKA connections, identify potential issues, and gain insights into the secure communication processes within the base station. This project successfully achieved its main goal of developing a tool that operates within the Rain platform to analyse MKA connection information and present it in a user-friendly manner. Beyond implemented functionalities, development process itself was supported with a deep dive into the technical domain, design considerations, and communication with supporting teams. The MKA log analyser has the potential to significantly improve development workflows and troubleshooting processes.

8 CONCLUSION

This thesis successfully addressed the challenge of time-consuming manual analysis for troubleshooting MACsec protected fronthaul connections. The result is a valuable automated log analyser plugin for the Rain platform, named the MKA log analyser.

The analyser offers significant benefits such as reduced analysis time, improved efficiency, and clear insights. This saves developers time form time-consuming manual analysis, allowing them to focus on core development activities. Presenting MKA connection details in a user-friendly manner in the familiar Rain platform empowers developers to make conclusions faster during the fault investigation process.

The modular design of the MKA analyser lays the groundwork for future enhancements. Potential areas for improvement include processing Radio Unit logs and handling complex scenarios, for example SAK rekeying events. The provided technical documentation facilitates future development and adoption of the analyser.

Overall, the MKA log analyser is a valuable contribution to the troubleshooting toolkit for developers working with MACsec protected fronthaul connections. It improves efficiency and empowers developers to ensure secure communication within the base station network.

REFERENCES

1. VIAVI. What Is Fronthaul? Search date 09.01.2024, https://www.viavisolutions.com/en-us/fronthaul
2. techplayon.com. What is RRH (Remote Radio Head), How it is connected to BBU (Base Band Unit). Search date 09.01.2024, https://www.techplayon.com/rrh-remote-radio-head-connected-bbu-base-band-unit/
3. telecompedia.net. D-RAN, C-RAN, vRAN and Open RAN. Search date 09.01.2024, https://telecompedia.net/ran/
4. Pizzinat, Anna, Chanclou, Philippe, Saliou, Fabienne & Diallo, Thierno. 2015. Things You Should Know About Fronthaul. Journal of Lightwave Technology, 33(5), 1077–1083. Search date: 09.01.2024, https://doi.org/10.1109/jlt.2014.2382872
5. Peng, Mugen, Wang, Chonggang, Lau, Vincent & Poor, H. Vincent. 2015. Fronthaul-constRained cloud radio access networks: insights and challenges. IEEE Wireless Communications, 22(2), 152–160. Search date: 09.01.2024, https://doi.org/10.1109/mwc.2015.7096298
6. Liyanage, Madhusanka, Braeken, An, Shahabuddin, Shahriar & Ranaweera, Pasika. 2023. Open RAN security: Challenges and Opportunities. Journal of Network and Computer Applications 214 (2023) 103621. Search date: 09.01.2024, https://doi.org/10.1016/j.jnca.2023.103621
7. IEEE Standard for Local and metropolitan area networks-Media Access Control (MAC) Security. IEEE Std 802.1AE™ 2018. Search date: 09.01.2024 https://doi.org/10.1109/IEEESTD.2018.8585421
8. support.huawei.com. NetEngine 8000 M14K Configuration Guide – MACsec Configuration. 2022. Search date 11.01.2024, https://support.huawei.com/enterprise/en/doc/EDOC1100279274/d122743e/macsec-configuration
9. www.comcores.com. O-RAN Fronthaul Security using MACsec. 2023. Search date 11.01.2024, https://www.comcores.com/wp-content/uploads/2023/12/MACsec-Security-for-O-RAN-Fronthaul.pdf
10. IEEE Standard for Local and Metropolitan Area Networks--Port-Based Network Access Control. IEEE Std 802.1X™ 2020. Search date: 11.01.2024, https://doi.org/10.1109/ieeestd.2020.9018454
11. Eberhardt, Colin. The Art of Logging. 2014. Search date: 20.02.2024, https://www.codeproject.com/Articles/42354/The-Art-of-Logging
12. Shenghui, Gu, Guoping, Rong, He, Zhang & Haifeng Shen. 2023. Logging Practices in Software Engineering: A Systematic Mapping Study. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 49, NO. 2, FEBRUARY 2023. Search date: 20.02.2024, https://doi.org/10.1109/TSE.2022.3166924

13. Yuan, Ding, Park, Soyeon & Zhou, Yuanyuan. 2012. Characterizing logging practices in open-source software, 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2012, pp. 102-112. Search date: 20.02.2024, https://doi.org/10.1109/ICSE.2012.6227202
14. Korzeniowski, Łukasz & Goczyła, Krzysztof. 2022. Landscape of Automated Log Analysis: A Systematic Literature Review and Mapping Study, in IEEE Access, vol. 10, pp. 21892-21913, 2022. Search date: 20.02.2024, https://doi.org/10.1109/ACCESS.2022.3152549
15. Chuvakin, Anton, Schmidt, Kevin & Phillips, Chrisopher. 2013. Logging and Log Management. Elsevier, Inc. ISBN: 978-1-59749-635-3
16. Working group. logging — Logging facility for Python. Official python v3.12.2 documentation. Search date: 26.02.2024, https://docs.python.org/3/library/logging.html#logging-levels
17. Gerhards, R., "The Syslog Protocol", RFC 5424, DOI 10.17487/RFC5424, March 2009, Search date: 26.02.2024, https://www.rfc-editor.org/info/rfc5424
18. Hitachi working group. Collecting the Snapshot Log. Search date: 01.03.2024, https://itpfdoc.hitachi.co.jp/manuals/3020/30203Y1110e/EY110026.HTM
19. Zhuangbin, Chen, Jinyang, Liu, Wenwei, Gu, Yuxin, Su & Michael, R. Lyu. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. 2021. Search date: 01.03.2024, https://doi.org/10.48550/arXiv.2107.05908
20. Rain team. README.md from Rain project repository. Available in Nokia Intranet. Search date: 27.02.2024.
21. Rain team. create-and-maintain-analyzer.md from Rain project repository. Available in Nokia Intranet. Search date: 27.02.2024.
22. Discussion with Tuomas Tolonen (Nokia). Date: 07.09.2023.
23. Technical Log Decoder and Analyser team. Developing plugins. Available in Nokia intranet. Search date: 05.03.2024.
24. Jayathilake, Dileepa. Towards structured log analysis. 2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE). Search date: 05.03.2024, https://doi.org/10.1109/jcsse.2012.6261962
25. Working group. re – Regular expression operation. . Official python v3.12.2 documentation. Search date: 06.03.2024, https://docs.python.org/3/library/re.html#re.compile
26. Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. 1994. Addison-Wesley. ISBN: 0-201-63361-2
27. Cormen, Thomas, Charles, Leiserson, Rivest, Ronald & Stein, Clifford. Introduction to Algorithms fourth edition. 2022. Massachusetts Institute of Technology. ISBN: 9780262046305

28. ApexChart.js contributors. ApexCharts documentation. Search date: 07.03.2024, https://apexcharts.com/docs/installation/
29. Working group. UsingSlots. Search date: 08.03.2024, https://wiki.python.org/moin/UsingSlots