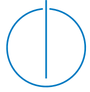
Department of Informatics
Technical University of Munich

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Decentralized Feature Processing on Resource-Constrained Devices for Network Anomaly Detection

Manuel Lorenz Herbert Ehler

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Decentralized Feature Processing on Resource-Constrained Devices for Network Anomaly Detection

Dezentralisierte Feature-Verarbeitung auf ressourcenbeschränkten Geräten zur Netzwerkanomaliedetektion

Author:	Manuel Lorenz Herbert Ehler
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Marcel von Maltitz, M. Sc. Stefan Liebald, M. Sc. Simon Bauer, M. Sc. Dr. rer. nat. Holger Kinkelin
Date:	September 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching b. München, September 15, 2018	
Location, Date	Signature

Abstract

Nowadays IT trends like the Internet of Things and Big Data induce a couple of consequences for related computing networks. Those include the steady growth of the networks in their number of components, the resulting and increasing dependencies between those components as well as more traffic in general and interfaces to the outside. Summarizing, this leads to more complex networks and thereby accompanying side-effects such as increased attacks and intrusions. Their advance makes it hard to detect, analyze and ward them off in time with traditional countermeasures. Conventional firewalls for instance are not able to inspect malicious network traffic in depth. However, as due to the aforementioned evolutions especially the security, safety and functional integrity of those networks within critical infrastructure are crucial, a modern network intrusion detection system is required, in order to early and reliably identify deviations from usual behavior and traffic.

The application of such a system in particular needs a mechanism for extraction and analysis of network communication characteristics (called features), which often proves to be difficult in safety- and security-critical networks though. This primarily is due to the characteristics of those networks as they appear in e.g. cars and aircraft. Of particular importance is both the high distribution of the strictly hierarchical structured network components as well as their compartmentalization into single disjunct domains (e.g. control sub-systems of an aircraft). Therefore, a central point for extraction and analysis, which not coercively scans all but only relevant portions of network traffic and communication behavior, is not given. In order to not have to integrate additional components into concerned networks due to that reason and therefore further advance the evolutions described above, the innovative approach of a decentralized feature handling process on unused resources of the already available network components is pursued.

Thus, in this master thesis we develop FHS-DNA, a **F**eature **H**andling **S**ystem on **D**ecentralized **N**etwork **A**gents. So, the main functionality consists of the decentralized extraction of network communication features by utilizing unused resources of available network components, further processing them afterwards and finally store them appropriately, so that various models have the option of subsequent use for anomaly detection. Special characteristics and therefore focus of FHS-DNA are besides the option for general connection to various such detection models also a broad, inclusively offered variety of advanced analysis options of extracted features, an easy extensibility concerning processed features of multiple levels of abstraction as well as high configurability with respect to involved network components, their interfaces and extracted features. Hence, the developed solution is based on an in-depth analysis of these requirements, already existing, similar (partial) approaches, concepts and technologies and finally combines The Bro Network Security Monitor, the Bro Analysis Tools as well as the Parquet file format to a fine-tuned synthesis.

Zusammenfassung

Heutige IT-Trends wie das Internet der Dinge und Big Data induzieren eine Reihe von Auswirkungen auf beteiligte Computernetzwerke. Diese beinhalten das stetige Wachstum der Netze in der Anzahl ihrer Komponenten, die daraus resultierenden steigenden Abhängigkeiten zwischen den Netzwerkkomponenten untereinander als auch generell mehr Verkehr und Schnittstellen nach außen. Zusammenfassend führt dies zu komplexeren Netzwerken und damit einhergehenden Nebeneffekte wie vermehrten Angriffen und Eindringen. Die Ausgereiftheit dieser macht es schwierig, sie rechtzeitig mit herkömmlichen Gegenmaßnahmen zu erkennen, zu untersuchen und abzuwehren. Klassische Firewalls sind z.B. nicht in der Lage, schädlichen Netzwerkverkehr tiefgründig zu inspizieren. Da aufgrund der zuvor beschriebenen Entwicklungen aber besonders die Sicherheit und funktionale Integrität dieser Netze in kritischer Infrastruktur von Bedeutung sind, benötigt man ein modernes Network Intrusion Detection System, um Abweichungen vom gewohnten Verhalten und Verkehr sowohl frühzeitig als auch verlässlich zu identifizieren.

Die Anwendung eines solchen Systems benötigt insbesondere einen Mechanismus zur Extraktion und Auswertung von Netzwerkkommunikationseigenschaften (sog. Features), was sich in sicherheitskritischen Netzen jedoch oftmals äußerst schwierig gestaltet. Dies liegt vor allem an den charakteristischen Eigenschaften dieser Netzwerke, wie sie beispielsweise in Autos und Flugzeugen vorkommen. Von besonderer Bedeutung sind dabei sowohl die große Dispersion der streng hierarchisch strukturierten Netzwerkkomponenten als auch deren Kompartimentierung in einzelne disjunkte Bereiche (z.B. Steuerungssubsysteme eines Flugzeugs). Ein zentraler Extraktions- und Analyseknoten, der nicht zwingend den gesamten, sondern lediglich relevante Teilbereiche des Netzwerkverkehrs und Kommunikationsverhaltens untersucht, ist somit nicht gegeben. Um aus diesem Grund nicht noch weitere Komponenten in betroffene Netzwerke integrieren zu müssen und damit die oben beschriebenen Entwicklungen weiter voran zu treiben, wird der innovative Ansatz eines dezentralen Featureverarbeitungsprozesses auf den ungenutzten Ressourcen der bereits vorhandenen Netzwerkkomponenten verfolgt.

In dieser Masterarbeit entwickeln wir daher FHS-DNA, ein **F**eature **H**andling **S**ystem on **D**ecentralized **N**etwork **A**gents. Die Hauptfunktionalität besteht also daraus, Netzwerkkommunikationsfeatures dezentral und mit Hilfe ungenutzter Ressourcen der vorhandenen Netzkomponenten zu extrahieren, anschließend entsprechend weiter zu verarbeiten und schließlich in geeigneter Form abzuspeichern, sodass verschiedene Modelle die Möglichkeit der Weiterverwendung zur Anomaliedetektion haben. Besondere Eigenschaften und somit Fokus von FHS-DNA sind neben der Möglichkeit der generischen Anbindung solcher verschiedener Detektionsmodelle auch eine große, integrativ angebotene Vielfalt an weiterführenden Analysemöglichkeiten der extrahierten Features, eine einfache Erweiterbarkeit bezüglich verarbeiteter Features über mehrere Abstraktionsebenen, sowie hohe Konfigurierbarkeit im Hinblick auf beteiligte Netzwerkkomponenten, deren Schnittstellen und erhobene Features. Die entwickelte Lösung basiert daher auf einer gründlichen Vorabanalyse dieser Anforderungen, bereits existierender ähnlicher (Teil-)Ansätze, Konzepte und Technologien und kombiniert letztlich The Bro Network Security Monitor, die Bro Analysis Tools sowie das Parquet Dateiformat zu einer wohl abgestimmten Synthese.

Acknowledgments

First of all, I would like to thank my advisors M.Sc. Marcel von Maltitz, M.Sc. Stefan Liebald, M.Sc. Simon Bauer and Dr. rer. nat. Holger Kinkelin for their valuable and always helpful feedback throughout this thesis. I appreciate your spent time.

In addition, I would like to thank Prof. Dr.-Ing. Georg Carle for supervising me and providing such an interesting, important and instructive topic.

Furthermore, I want to thank my family for always supporting and therefore affording me to successfully finish my studies.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Questions and Goals . . . 2
1.3 Outline . . . 3

2 Background **5**
2.1 General Anomaly Management Process . . . 5
2.2 Characteristics of On-Board Networks . . . 6
2.2.1 Hierarchic, Decentralized and Compartmentalized Topology . . . 6
2.2.2 Static and Predefined Communication Traffic Patterns . . . 7
2.3 Aspects of Network Communication Features . . . 7
2.3.1 Feature Definition . . . 7
2.3.2 Feature Levels of Abstraction . . . 8
2.3.3 Various Feature Traffic Datasets . . . 9
2.3.4 Feature Relevance . . . 9
2.3.5 Effects of Feature Reduction . . . 10
2.4 Options for Traffic Monitoring and Feature Extraction . . . 11
2.4.1 Traffic Capturing and Basic Packet Analysis . . . 11
2.4.2 Traffic Manipulation and Replay . . . 12
2.4.3 Network Intrusion Detection Systems . . . 13
2.5 Technologies for Feature Processing and Storage . . . 18
2.5.1 Relational Database Management Systems . . . 18
2.5.2 Non-Relational Database Management Systems . . . 19
2.5.3 Search and Analysis Engines . . . 20
2.5.4 Processing and Storing with Generic Data Formats . . . 21

3 Analysis **23**
3.1 Problem Statement . . . 23
3.2 High-Level Requirements . . . 26

3.2.1 Functional Requirements 26
3.2.2 Non-Functional Requirements 27
3.3 Design Choices 29
3.3.1 Feature Extraction 29
3.3.2 Feature Processing 30
3.3.3 Feature Storage 32
3.3.4 Cohesive Processing and Storage Concept 34

4 Related Work **35**
4.1 Anomaly Detection for SOME/IP 35
4.2 VAST 36
4.3 Collective Intelligence Framework 37
4.4 Security Information and Event Management Tools 37
4.5 Approaches Partly Similar to FHS-DNA 38
4.5.1 Two-Modules Approach 38
4.5.2 Two-Layers Approach 39

5 Design **41**
5.1 Design Decisions 41
5.1.1 Feature Extraction 41
5.1.2 Feature Processing 43
5.1.3 Feature Storage 45
5.1.4 Cohesive Processing and Storage Concept 46
5.2 System Architecture 47
5.2.1 Internal Components 47
5.2.2 External Components 49
5.3 Important External Interfaces 51
5.3.1 External Linkage Interface 51
5.3.2 Logs Synchronization Interface 51
5.3.3 (Re-)Configuration Interface 52
5.3.4 Detection Model Interface 55
5.4 Intermediate Log Files 56
5.4.1 General Logs 57
5.4.2 Interface-Specific Sliding Window Logs 58
5.5 Final Feature Storage 59
5.5.1 Final Storage Format 59
5.5.2 Final Storage Scheme 60

6 Implementation **63**

6.1 Feature Extraction . . . 63
6.1.1 Bro as Tool for Traffic Monitoring and Feature Extraction . . . 63
6.1.2 Bro ”Normal” Cluster for Decentralized On-Agent Monitoring and Feature Extraction . . . 64
6.1.3 Adaption of Bro ”Normal” Cluster to Internal Components . . . 65
6.1.4 Manually Implemented and Extracted Features . . . 66
6.1.5 Intermediate Log Files . . . 72
6.2 Feature Processing and Storage . . . 74
6.2.1 BAT as Tools for Feature Processing and Storage Preparation . . . 74
6.2.2 Parquet as Final Feature Storage Format . . . 75
6.2.3 Intermediate Bro Logs’ Transformation to Parquet Format . . . 76
6.2.4 Final Storage Scheme . . . 77
6.2.5 Final Storage Update . . . 79
6.3 Interface Interaction . . . 80
6.3.1 Automated Synchronization of Intermediate Log Files . . . 80
6.3.2 Dynamic Reconfiguration . . . 81
6.4 Summarizing Overview . . . 84

7 Testing and Evaluation **89**
7.1 Test Setup Deployment . . . 89
7.1.1 Virtualization . . . 89
7.1.2 Required Prerequisites of a Bro ”Normal” Cluster . . . 91
7.1.3 Installations . . . 92
7.1.4 Configurations . . . 93
7.1.5 Summarizing Overview . . . 95
7.2 Evaluation . . . 97
7.2.1 Common Methodology . . . 98
7.2.2 Results and Implications . . . 100

8 Requirements Assessment **111**
8.1 Functional Requirements . . . 111
8.2 Non-Functional Requirements . . . 112
8.2.1 General Claims . . . 112
8.2.2 Feature Extraction . . . 114
8.2.3 Feature Storage . . . 114

9 Conclusion **117**
9.1 Main Results and Contributions . . . 117
9.2 Future Work . . . 118

A Appendix **121**

A.1 Implementation 121

A.1.1 Overview of Manually Constructed Intermediate Log Files 121

A.1.2 Extracts of Exemplary Intermediate Log Files 126

A.2 Testing and Evaluation 128

A.2.1 Statistics of exercise-traffic.pcap 128

B List of acronyms **131**

Bibliography **135**

List of Figures

2.1 A general, three-step anomaly management process 5
2.2 A typical, hierarchic topology with decentralized and compartmentalized components of an on-board network from [6] 6

3.1 A typical, hierarchic topology with differently marked components of an on-board network based on [6] 24

5.1 Architectural system design overview 47
5.2 Exemplary extract of the abstract and conceptual structure of a statically and manually to define *monitoring layout* 53
5.3 Exemplary extract of the abstract and conceptual structure of a statically and manually to define *configuration mode* 54

6.1 Pseudo code fragment for extraction of feature *synack* 69
6.2 Pseudo code fragment for extraction of feature *land* 70
6.3 Pseudo code fragment for extraction of feature *ct_dst_sport_ltm* 71
6.4 Abstract implementation of the hierarchic directory structure of the final on-disk feature storage scheme 77
6.5 Pseudo code fragment for updating the final feature storage on the external *Feature Processor and Storage Component* 80
6.6 Sequence diagram for the automated synchronization of intermediate *Bro* log files 81
6.7 Sequence diagram for the dynamic, on-demand reconfiguration of FHS-DNA 83
6.8 System implementation overview 88

7.1 Code snippet of *Vagrantfile* for the exemplary multi-machine test setup 93
7.2 System demonstration and test setup deployment overview 96
7.3 Median extraction times for selected feature combinations in growing network traffic 104

7.4 Median transformation times for all features' generated *Bro* log files of selected feature combinations in growing network traffic 105
7.5 Median extraction times for all measured feature combinations for specific network traffic 108
7.6 Median transformation times for all features' generated *Bro* log files of all measured feature combinations for specific network traffic 108
7.7 Median sizes of all summed up features' generated *Bro* log files and respectively transformed *Parquet* files and the median fraction of them for most comprehensive feature combination *18* 110

A.1 Exemplary extract of an intermediate *General Log*, generated by script *feature-extraction-UNSWNB15-34.bro* for feature *synack* 126
A.2 Exemplary extract of an intermediate *Interface-Specific Sliding Window Log*, generated by script *feature-extraction-UNSWNB15-46.bro* for feature *ct_dst_sport_ltm* 127

List of Tables

2.1 Eleven highest ranked features from datasets UNSW-NB15 and NSLKDD according to a particular algorithm as specified in [12] 10

6.1 Manually implemented features for potential extraction by FHS-DNA . 68
6.2 Filename prefixes whitelisting respective intermediate *Bro* logs for potential transformation to *Parquet* format 77

7.1 All evaluated feature extraction combinations 101

A.1 Information logged for feature *land* 121
A.2 Information logged for feature *synack* 122
A.3 Information logged for feature *ct_src_ltm* 122
A.4 Information logged for feature *ct_srv_dst* 122
A.5 Information logged for feature *ct_dst_sport_ltm* 123
A.6 Information logged for feature *count* 123
A.7 Information logged for feature *raw_layer2_packet_header* 123
A.8 Information logged for feature *raw_IPv4_packet_header* 124
A.9 Information logged for feature *raw_IPv6_packet_header* 124
A.10 Information logged for feature *raw_TCP_packet_header* 124
A.11 Information logged for feature *raw_UDP_packet_header* 125
A.12 Information logged for feature *raw_ICMP_packet_header* 125
A.13 Protocol hierarchy of *exercise-traffic.pcap* 128
A.14 Conversations of *exercise-traffic.pcap* 129
A.15 Endpoints of *exercise-traffic.pcap* 129
A.16 Packet Lengths of *exercise-traffic.pcap* 129

Chapter 1

Introduction

Our work discusses the need and aspects of proper feature handling in the allover context of a forensic center for anomaly detection in networks.
Context of this work is the joint research project DecADe (for **Dec**entralized **A**nomaly **De**tection) of Technische Universität München and various of its partners like Airbus Group Innovations and Audi Electronics Venture GmbH [1]. The project puts special focus on on-board networks of cars and aircraft.

In the following, we give a short motivation for our work, state the research questions as well as related goals and conclude with an outline explaining the structure for the rest of this thesis.

1.1 Motivation

Nowadays computing trends like Internet of Things, big data and ubiquitous computing both foster and gain on an increasing amount of networking devices. That causes growing connectivity, more network traffic [2] [3] [4], more complexity and therefore more dependencies [1] between those devices. Hence, these aspects affect both safety and security of those networks, as common side-effects of those developments are an increasing amount of intrusions and sophisticated attacks. However, those advanced persistent threats often stay undetected by common security infrastructure like firewalls, as they do not inspect network traffic in depth [3]. Thus, actual circumstances call for modern network intrusion detection systems.

However, on the one hand, networks of most critical infrastructure such as cars and aircraft are strictly hierarchical and therefore consist of different clusters of decentral-

ized [5] and compartmentalized devices [6]. That makes it very hard to monitor these networks and their communication traffic for intrusions or attacks, as there is not a central point to capture traffic from.
Though, on the other hand, those networks are also characterized by well- and predefined communication patterns, as they are static and closed. Hence, those patterns in principle perfectly support the detection of anomalies, as every deviation of the predefined model may be considered as an anomaly.

This is, where our work comes into play. We pursue an innovative approach suggested by the DecADe project [1], which manages decentralized feature handling directly on networking devices under surveillance. Hence, we aim to develop the groundwork for later anomaly detection and incidence response, by properly extracting, processing and storing features from network communication traffic.

1.2 Research Questions and Goals

The research questions of this thesis are:

1. Which network communication features can be useful for anomaly detection?
2. How can these features be extracted and collected from decentralized and distributed networks?
3. How can this feature extraction and collection mechanism be extended and configured?
4. How can distributively collected features be further processed and stored for subsequent anomaly detection purposes?

By elaborating these research questions, we aim to develop a **F**eature **H**andling **S**ystem on **D**ecentralized **N**etwork **A**gents, named FHS-DNA. The system's goals are:

1. Directly extract and collect network communication features on decentralized networking devices under surveillance.
2. Allow reconfiguration and extensibility for the feature extraction and collection process.
3. Process and store those features for subsequent anomaly detection purposes.

1.3 Outline

The rest of this thesis is structured as follows:

Chapter 2 introduces a general anomaly management process, characteristics of on-board networks and various aspects of network communication features. Furthermore, different options for network traffic monitoring and feature extraction are explored, as well as several technologies for further feature processing and storage are explained.

Chapter 3 analyses the underlying problematic of our work in-depth before deducing most important functional and non-functional requirements of FHS-DNA. Moreover, it explores different design choices for feature extraction, processing and storage and a cohesive concept resulting from those high-level requirements.

Chapter 4 describes diversified related work to various facets of our target system.

Chapter 5 picks up design choices from Chapter 3 and argues our design decisions resulting in the allover system's architecture. As this architecture is separated into internal (i.e. within the (sub-)network(s) under surveillance) and external components, also important interfaces of this architecture are explained. Furthermore, different types of intermediate log files are explored.

Chapter 6 explicates a proof of concept implementation for several selected aspects of FHS-DNA, such as feature extraction, processing and storage as well as interface interactions (e.g. dynamic reconfiguration of our system), before giving a summarizing overview of its allover workflow.

Chapter 7 is categorized into a section describing our system's test setup deployment and a second one for its evaluation. The former one explores various aspects concerning virtualization, prerequisites, installations and configurations before summarizing with an overview of the test setup. The latter one presents the methodology, results and implications for the evaluation of most important aspects of FHS-DNA.

Chapter 8 assesses the high-level requirements stated in Chapter 3.

Chapter 9 concludes this thesis by summarizing our main contributions and giving an outlook on possible future work.

Chapter 2

Background

This chapter contains basic background knowledge for better understanding of subsequent chapters. It starts by introducing a typical, overall anomaly management process in order to better understand the intended use of FHS-DNA and our problem domain (see Section 2.1), followed by an explanation of on-board networks' characteristics (see Section 2.2). Afterwards, general aspects of features are explored in Section 2.3, before different options for feature extraction (see Section 2.4) as well as various technologies for feature processing and storage are elaborated (see Section 2.5).

2.1 General Anomaly Management Process

Figure 2.1 shows an overall, generic three-step workflow of an abstract anomaly management process as e.g. in a FC.

In the first step, features are extracted from network traffic, collected and prepared for storage. In the second step, (anomaly) detection models ((A)DMs) access those features and run some (machine learning (ML)) algorithms, which compare the actual network communication pattern to a predefined, static one. Whenever those two patterns differ, an anomaly in network communication is identified and a potential incidence response

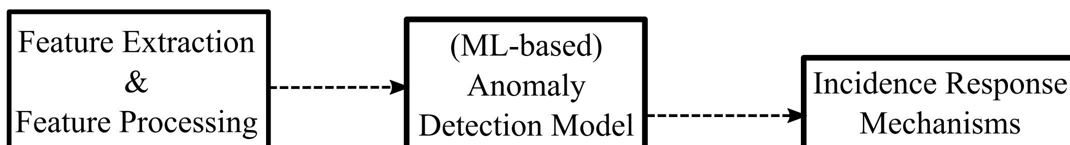

Figure 2.1: A general, three-step anomaly management process

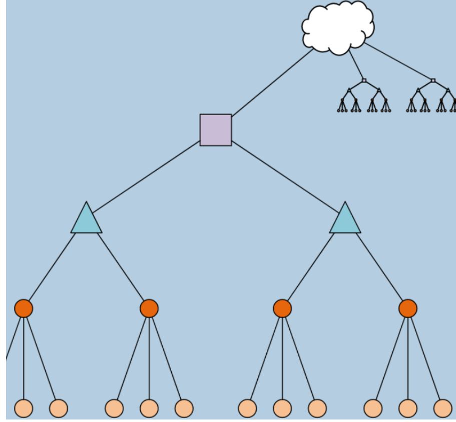

Figure 2.2: A typical, hierarchic topology with decentralized and compartmentalized components of an on-board network from [6]

may be selected and triggered in step three.
Hence, FHS-DNA's operational purpose is to implement step one. Potential fields of application for FHS-DNA are in particular on-board networks (OBNs) of cars and aircraft, which require special considerations regarding the development of FHS-DNA due to their characteristics.

2.2 Characteristics of On-Board Networks

Beside the fact that cars and aircraft hold strict limitations to the hardware of OBNs (e.g. size, weight, space and performance), their most crucial characteristics with respect to feature handling (FH) are explained in the following two subsections.

2.2.1 Hierarchic, Decentralized and Compartmentalized Topology

The first and most important characteristic is the hierarchic network topology with multiple clusters of compartmentalized devices within those (sub-)networks.
Within an aircraft for example, different (sub-)systems are integrated into the overall aircraft system. Such (sub-)systems are e.g. the on-board lightning, the passenger entertainment and the flight assistant system. They all are of different criticality and therefore need to be compartmentalized within the overall aircraft system.
Consequently, as Figure 2.2 from [6] indicates, the abstract compartmentalization of such an OBN induces clusters of networking devices, sensors and actuators within various sub-networks. Furthermore, especially in aircraft, these (sub-)networks are decentralized [5].

2.2.2 Static and Predefined Communication Traffic Patterns

Another key characteristic of those OBNs are well-defined communication patterns between the networking agents. This is, because different devices of e.g. a flight control system fulfill special and very well predefined tasks only. In addition, the number of agents of an OBN is static and does not change, as those networks are not publicly accessible. Hence, the topology and its communication behavior are known and clearly determined ex ante. That makes network traffic anomaly detection in step two from Figure 2.1 especially suitable in those nets, because each deviation from the intended network communication may be considered as an anomaly with potential influence on the car's or aircraft's safety and / or security.

2.3 Aspects of Network Communication Features

In the next few subsections, we explicate some important aspects of network communication features. First of all, we give a definition for feature (see Subsection 2.3.1). In close relation to the introduced definition, different levels of abstraction for feature categorization considered in our work are explored in Subsection 2.3.2. Next, various commonly well-known traffic datasets are introduced, from which many features are deduced (see Subsection 2.3.3). Furthermore, we underline the relevance of features for the subsequent steps after extracting and processing them (see Subsection 2.3.4). Lastly, some background research regarding the effects of reducing the number of features for subsequent anomaly detection is explored (see Subsection 2.3.5).

2.3.1 Feature Definition

A common definition for **feature** is ”specification of an attribute and its value” [7]. However, it is often used as a synonym for attribute, e.g. ”in feature-subset selection” [7]. Due to this definition, we also want to define the term attribute in the context of our work.
An **attribute** is a ”quantity describing an instance. An attribute has a domain defined by the attribute type, which denotes the values that can be taken by an attribute” [7]. In the following, different feature levels of abstraction considered in our work are explained, which may also be accounted for the different feature domains according to the aforementioned definition of an attribute.

2.3.2 Feature Levels of Abstraction

In accordance to the definitions given above, one possible domain for the features being handled in this work is abstraction level. This domain can have three categorical types, namely:

- Packet level (PL),
- Connection level (CL),
- Network level (NL).

According to [7], such a categorical type definition is one of the two most common ones and determines a ”finite number of discrete values” [7]. The goal is to handle features of all three levels of abstraction in our final system.

Packet Level

PL features are related to a single network packet only. By conducting a unique instance of such a feature for further analysis only allows statements about this particular related packet. However, combining and analyzing more than just one instance of a feature of this abstraction level can allow more complex or significant conclusions related to more than just one network packet (e.g. connection related statements).
An example for a PL feature is all TCP header information of a single packet.

Connection Level

CL features are also often called flow level features. They induce conclusions about single connections or network flows, as each instance of such a feature is related to one connection or flow in the network. Hence, combining multiple PL features may - but not necessarily does - have the same significance as a single CL feature. In some cases, e.g. by making use of all packets of a connection, explanatory power may be equal to only considering one CL feature regarding this connection. However, by combining only two or three random packets of a long connection, a CL feature related to this connection may have more expressiveness than the combined PL features always depending on what exactly wants to be expressed. Again, combining more than one CL feature leads to higher level insights. Under the premise that those CL features are related to the same network, their combined analysis may lead to network wide information.
An example for a CL feature is the number of data bytes transferred from a source to a destination within a single connection or flow.

Network Level

NL features are the third and highest abstraction level considered in our work. They provide information about the entire network. Each single instance of these features by itself allows for statements about the entire, related network. Thus, it is the same with one NL feature and multiple CL features (of the same network) as it was with one CL feature and multiple PL features (of the same connection). Combining multiple CL features may - but not necessarily does - have the same explanatory power as a single NL feature, although it depends on what exactly is analyzed or expressed.
An example for a NL feature is the number of connections to the same destination address within the last 100 connections.

2.3.3 Various Feature Traffic Datasets

Besides this vertical feature categorization, a more common one is horizontal. Typically, features deduced from some traffic datasets by running intrusion detection algorithms on them are categorized horizontally. Such classes can include, but are not limited to, content features, time features and basic or statistically aggregated features.
Anyway, different network traffic datasets may therefore serve on the one hand as an option to deduce features and on the other hand as one possible input for the evaluation of a (network) intrusion detection system ((N)IDS). Well-known datasets are for example the DARPA98, KDD99, NSLKDD and others, which have been in use over the last decades [3] [8] [9] [10]. Though, numerous studies argue that those do not reflect nowadays low-footprint intrusions, are biased and corrupt [3] [4] [11]. Hence, this work is essentially based on results for today's most valid dataset UNSW-NB15, which tackles major flaws of the previous ones [3] [8] [10] [11].
Depending on the specific traffic dataset, its synthetically included attacks and the applied algorithms, even different features may be deduced and further turn out differently valuable for the subsequent detection process [3] [9] [12]. Therefore, we put major focus on features proven most valuable and deduced from UNSW-NB15. The significance of considered features directly leads to the next aspect.

2.3.4 Feature Relevance

The relevance of extracted features and their validity fundamentally contributes to the quality of an IDS [12] [13] [14]. This is what makes the first of the three steps from Figure 2.1 so important. The (A)DMs and final results of their algorithms heavily rely on the features and their contribution to the detection of the appropriate attack

or intrusion. Therefore, even the selection and triggering of an incidence response in step three transitively depends on the handled features' relevance. Hence, this is why we consider different features' validity before extracting and processing them in our FHS-DNA's PoC implementation.

2.3.5 Effects of Feature Reduction

Feature reduction is another aspect closely related to the quality of the overall detection process. Research indicates that reducing the number of features to only some valuable ones with respect to traffic classification by adjacent algorithms has been proven to increase performance and validity of results [12] [15] [16]. For that reason, we aim to identify, extract and further process only one possible subset of features for our PoC implementation of FHS-DNA. Nevertheless, arbitrary (other) subsets are in principle also possible.

Suggestions for valuable feature subsets are given in [9] [12] [17]. [9, p. 5] presents considerable examples, where most frequently appearing features according to [8] are listed. Additionally, in [9, p. 6], a very significant subset of features according to [8] is listed.

Further, Table 2.1 based on [12, p. 8] shows the eleven highest ranked features from UNSW-NB15 and NSLKDD according to a specific algorithm as proposed in [12]. Hence, a couple of those features are included into our dedicated subset for the PoC implementation and will be extracted from network traffic as explained in Chapter 6. These features are marked green in Table 2.1.
Selection criteria include considerations to extract features from every vertical level of abstraction as described above and with different complexities. Furthermore, as the UNSW-NB15 dataset is a more recent, reliable and valid one, we decided to include twice as much features from it than from the NSLKDD dataset. An overview of all manually implemented features part of the selected subset is given in Table 6.1.

Table 2.1: Eleven highest ranked features from datasets UNSW-NB15 and NSLKDD according to a particular algorithm as specified in [12]

UNSW-NB15 Features	**NSLKDD Features**
state	dst_bytes
dttl	dst_host_srv_diff_host_rate

UNSW-NB15 Features	NSLKDD Features
synack[1]	srv_diff_host_rate
swin	land
dwin	dst_host_same_src_port_rate
ct_state_ttl	count
ct_src_ltm	src_bytes
ct_srv_dst	logged_in
sttl	protocol_type
ct_dst_sport_ltm	num_root
djit	srv_rerror_rate

2.4 Options for Traffic Monitoring and Feature Extraction

Before feature extraction (FE) can actually happen, network traffic needs to be monitored. A comprehensive overview of possible options to do so is given in [18]. In order to explain a valuable subset of tools from this huge variety, we classified them into three general, overall categories. For each category, we present examples most representative for it, which are also well-known and commonly used for respective purposes. In general, all categories can somehow monitor network traffic, which is why in principle all of them may be utilized to extract features from the traffic in some way. However, some tools are more suitable for FE than others, often depending on their specific main purpose.

2.4.1 Traffic Capturing and Basic Packet Analysis

This first category aims at simply monitor network traffic, maybe save it in files and do some basic packet analysis. Moreover, this packet analysis often is not even done automatically, but requires manual inspection. Hence, it is a less advanced category. Common examples are presented below.

[1] Green features are considered and manually implemented for possible extraction by FHS-DNA as explained in Chapter 6

Tcpdump and Libpcap

Tcpdump[1] is a command line packet analyzer. It utilizes Libpcap[2], a library for network packet capture. Tcpdump can be used in multiple ways, e.g. save captured network traffic in files of Libpcap's format (pcap[3]) or read from previously saved pcap-files instead of the actual traffic. It further prints out descriptions on network packets that matched a pre-specified Boolean expression [19].

Wireshark

Wireshark[4] is probably the most commonly known network packet analyzer. Its primary intention is to capture packets from a wire and display them on the screen as detailed as possible. For that purpose, an interface needs to be specified, from which the network traffic is captured. Hence, Wireshark may especially be used for deep packet inspection. However, that needs to be done manually with a set of options provided. These include e.g. adapting filters to search for specific packets only, creating plenty of statistics and providing a user-optimized graphical user interface with colorized packets. Furthermore, it is possible to capture live data as well as read in pcap-files from disk or many other capture programs. Additionally, exporting captured packets is possible, too [20].
Hence, Wireshark originally is not a packet manipulation system nor an IDS [20]. Such systems are presented below.

2.4.2 Traffic Manipulation and Replay

This second category may be seen as slightly more powerful in contrast to the previous one. Its overall purpose is to not only monitor network traffic, but also manipulate it in some way and replay it afterwards. Hence, presented examples are common options for e.g. testing or training detection tasks with previously captured and intentionally forged traffic.

---

[1] http://www.tcpdump.org/

[2] http://www.tcpdump.org/

[3] http://www.tcpdump.org/manpages/pcap.3pcap.html

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

Tcpreplay

A first example for those purposes is Tcpreplay[1]. Previously captured network traffic within pcap-files serves as input to Tcpreplay. It ”allows you to classify traffic as client or server, rewrite Layer 2, 3 and 4 headers and finally replay the traffic back onto the network and through other devices such as switches, routers, firewalls, NIDS” [21].

Scapy

Scapy[2] is a more powerful packet manipulation system. It can be used to capture, decode, forge and resend packets of a wire. Regarding the manipulation of traffic, it is important to mention its outstanding capabilities of being totally independent of packets' standard structures. That means, it is not only possible to tamper with e.g. some values of predefined header fields, but allows to e.g. craft a totally untypical new structure of and values for packet fields. Furthermore, a number of additional tasks like tracerouting, scanning, probing and attack detections are possible. Although all these tasks are supported, they are not implemented automatically. A very important aspect and also kind of the intention of Scapy is to just decode, but not interpret the data before presenting it to the user. Hence, the user has to manually decide on what the decoded data means to him or her [22].

2.4.3 Network Intrusion Detection Systems

NIDSs constitute the third category. Their approaches aim to detect malicious traffic in a network more automatically and can be distinguished into signature- (also called misuse-) and anomaly-based ones [10]. The signature-based ones work on patterns, which explicitly mark instances to be identified as intrusions, whereas anomaly-based approaches work on estimations of ”normal” behavior. Specifying how ”normal” behavior looks like allows anomaly-based IDSs to detect zero-day intrusions [2] [3] [8]. This is, why they have gained more focus in research and development over the past few years [2] and are recommended [3]. However, especially at the early stages of their implementation, they often produce higher false-alarm-rates [2] [8] [12].
Anyway, by our system extracted features may be used as input to either kind of NIDSs. Furthermore, our system can be seen as partly fulfilling some tasks of a NIDS itself,

[1] http://tcpreplay.synfin.net/

[2] https://scapy.net/

which are e.g. monitoring network traffic and extracting features from it. Common examples for NIDSs are presented in the following.

Snort

Snort[1] is an intrusion detection and prevention system. It offers three general modes, in which to run the system. The first mode is called *Sniffer* mode, which simply reads and displays the network traffic packets [23, section 1.2]. The second mode is called *Packet Logger* mode. In comparison to the first mode, it additionally logs the packets to the disk [23, section 1.3]. The most advanced mode is called *Network Intrusion Detection System* mode, which also is the most comprehensive and configurable one. Within this mode, network traffic can be detected and analyzed in more detail [23, section 1.4]. In general, this is done by specifying some rules. Once packets matching these rules are detected, pre-specified alert mechanisms are invoked. Hence, Snort primarily works signature-based, not anomaly-based [24] [25, p. 6]. That means, based on some predefined whitelists and blacklists, further handling of the network traffic is specified. In addition, commonly known attack patterns are checked by default, too [26]. For whitelists in Snort there are provided two different meanings. They are *unblack* and *trust*. The first one explicitly allows IP-addresses, which also are on blacklists. That means, the whitelist has higher priority than blacklists. The other option *trust* means that corresponding packets get bypassed and not further detected by Snort [27].

Hence, in comparison to Wireshark, Snort is able to passively monitor and post-process network traffic. The main similarity is that *Sniffer* mode in Snort basically fulfills the same purpose as using Wireshark. In addition to Wireshark, Snort also supports comprehensive logging, which allows post-processing analysis. Furthermore, handling network packets in *Network Intrusion Detection System* mode of Snort fundamentally differs from what Wireshark provides.

In comparison to Scapy, Snort's intention is not to be a packet manipulation system, but a NIDS or network intrusion prevention system. Though, the user may manipulate packets, resend and finally inspect them in Scapy on a very low-level basis. In Snort, the user captures packets as they come and specifies some rules on how to handle them in order to detect or prevent network intrusions.

[1] https://www.snort.org/

The Bro Network Security Monitor (Bro)

Another passive network traffic analyzer is Bro[1]. It primarily "inspects all traffic on a link in depth for signs of suspicious activity" [28]. A detailed and comprehensive description of it can be found in [29]. Although it can also be used both as a real-time as well as an offline network traffic analyzer and NIDS comparable to Snort, it fundamentally differs from it in a couple of aspects.

First of all, Bro's internal architecture is a key difference to other known NIDSs. In principal, it is built out of two major components as described in [28]. The first one is the event engine (or core). Its purpose is the reduction of network packet streams into some high-level events. They characterize the packet stream from which they were created by the event engine. However, this characterization is policy-neutral, which means it is not classifying nor interpreting the packet stream at this point. So, this interpretation is done by the second component - the policy script interpreter. Therefore, the events get passed on to this component, which then launches a set of scripts - called event handlers. These handle the incoming traffic, derive statistics and find correlations by being capable of maintaining state over time [28]. Hence, this event-based architecture with its event engine and the policy script interpreter is a first main difference when compared to all tools elaborated above.

Considering the policy script interpreter, the second main difference of Bro is its domain-specific and Turing-complete scripting language for implementing any analysis tasks [28] [30]. That makes Bro very flexible, highly extensible and customizable as well as powerful with respect to traffic analysis tasks even beyond the security domain. Furthermore, this scripting policy makes Bro not a typical signature-based IDS (like e.g. Snort). Although Bro supports this kind of traffic analysis, it even provides a "much broader spectrum of very different approaches to finding malicious activity, including semantic misuse detection, anomaly detection, and behavioral analysis" [28]. A comprehensive overview of Bro's features is given in [28].

Another characteristic main difference of Bro is its substantial set of high-level log files. These go beyond low-level logging and already identify and correlate information across connections and even various application-layer transcripts [28].

Lastly, one of the outstanding possibilities of Bro is the deployment of a distributed setup of various single but coordinated Bro instances across larger systems compounded of separated, physical machines. This is called a cluster. For operators controlling such a

[1] https://www.bro.org/index.html

cluster, a central management framework - BroControl - is provided by Bro per default [28]. Moreover, there are actually two different types of clusters, which again differ in a couple of aspects. Nevertheless, for both types of clusters some special prerequisites need to be considered before usage. For those, we want to refer to appropriate documentation provided by Bro (e.g. see [31] and [32]).

("Normal") Cluster: An overview of this first type and its generic architecture is given in [33]. It is "a set of systems jointly analyzing the traffic of a network link in a coordinated fashion" [34]. In contrast, the standalone mode of BroControl handles Bro running on a single system, i.e. device. Hence, the ("normal") cluster mode of BroControl can basically be compared to an automated, coordinated and decentralized composition of multiple standalone modes on a couple of different networking devices. However, a ("normal") cluster is not just the composition of many standalone modes. Rather it is an amount of Bro instances - each one working as a single entity - taking over a specific part of the overall cluster's functionality. Nevertheless, it is possible that multiple ("normal") cluster's Bro instances run on the same physical host [34].

According to [33], Bro is not multithreaded, which explains the composition of single, cohesive Bro instances and their coordination in a ("normal") cluster setup across multiple cores or even physical devices for purposes of spanning larger systems. This way, instances in a ("normal") cluster work together as different types. For each ("normal") cluster, at least one *manager*, one *proxy* and one or more *worker* nodes need to be defined.

Manager Hence, a *manager* node is a mandatory one. It has primary two jobs, which are to:

1. Receive logs;
2. Handle notices.

Regarding the first task, logs from all the *worker* nodes are collected on the *manager*, which leads to a single log per subject (e.g. extracted feature) aggregated on the *manager* instead of multiple ones on each *worker* [33]. Though, whenever a *logger* node is defined in the ("normal") cluster setup as well, this first task of collecting logs is taken care of by the *logger* and not the *manager* [33].
Nevertheless, independent of a *logger* node being configured or not, the second task of handling notices is always implemented by the *manager* node. A notice may be raised

by a *worker* whenever something special - that was defined by the operator's scripts - is identified in the traffic.

Worker The *worker* nodes are where the actual traffic sniffing and protocol analysis is performed. Hence, at least one node of this type is mandatory. Moreover, the most affordable resources with respect to memory and CPU should be provided to this type of nodes [33]. Therefore, the "maximum recommended number of workers to run on a machine should be one or two less than the number of CPU cores available on that machine" [34]. Nevertheless, although most of the work is performed on the *worker* nodes, requirements with respect to disk space are very limited, because all logs are collected at the *manager* or a *logger* node [33]. Further specifications and suggestions on provided resources for *worker* nodes can be found in [33].

Proxy A *proxy* node in a ("normal") cluster is described as "a Bro process that manages synchronized state. Variables can be synchronized across connected Bro processes automatically. Proxies help the workers by alleviating the need for all of the workers to connect directly to each other" [33]. Further, the number of *proxy* nodes may grow higher than one, but at least one is required in a "normal" cluster. However, most often one *proxy* (at least for smaller clusters) should be enough, as they do neither need that much CPU nor memory. This may also be a reason, why many operators often run a *proxy* on the same physical host as the *manager* [33].

Logger The *logger* node is optional in contrast to all the other types. "If a logger is defined [...], then it will receive logs instead of the manager process" [34]. Hence, by having a *logger* in a ("normal") cluster setup, resources on the *manager* node are preserved.

Deep Cluster: The second type of Bro clusters is a so-called deep cluster. It "provides one administrative interface for several conventional clusters and/or standalone Bro nodes at once" [31]. In other words, a deep cluster can even be a cluster of distributed "normal" clusters and standalone Bro nodes. Furthermore, it fosters direct information exchange between different, originally not directly connected Bro (cluster) instances by enabling the communication with each other via the underlying and additionally integrated publish-subscribe system [31]. Hence, a deep cluster can be considered as an overlay peer-to-peer network, whose goal it is to "setup large numbers of Bro instances that might be deployed in different parts of the network (or in different networks)" [31].

Consequently, the communication between different (deep cluster or standalone) nodes fundamentally differs from the basic, routable communication in ”normal” clusters. An additional difference to a Bro ”normal” cluster is that a deep cluster’s nodes have no types anymore (e.g. *manager*, *proxy*, *worker*). Instead, the nodes can accept different roles on-the-fly. Even more than one role at a time is possible and there are more roles available in a deep cluster than types in a ”normal” cluster [31].

2.5 Technologies for Feature Processing and Storage

Feature processing (FP) can be considered as post- or pre-processing. Pre-processing means managing features before they are input to subsequent (A)DMs, whereas post-processing means managing features after they were extracted from network traffic. Hence, both terms mean the same in the context of our work, but refer to different points in the allover FH process.
Regarding feature storage (FS), that especially means the final storage of previously extracted and processed features, from which subsequent (A)DMs can query them in order to do the actual anomaly detection.
Moreover, processing and storage are considered together in the following subsections, as both tasks are closely related to each other. For instance, FS preparation is considered a part of FP throughout this work.

2.5.1 Relational Database Management Systems

For final FS, a general option is to utilize a database management system (DBMS). Such a first type are relational database management systems (RDBMSs). They and their concept function with data structured in and across predefined tables in a relational model. They organize their data in rows and can reference entries across tables with primary and foreign keys describing columns (i.e. attributes of row-based entries). Two well-known RDBMSs are shortly explored in the following.

PostgreSQL Database

PostgreSQL[1] is open-source with a plug-in available for Bro as described in [35] and [36]. PostgreSQL’s main goals are to be compliant to standards and very extensible. By

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

being programmable it offers a broad number of third-party tools and libraries support, which makes working with PostgreSQL very simple and powerful. Furthermore, it offers all highly required and fundamental RDBMSs' functionality such as full support for atomicity, consistency, isolation and durability (ACID). In addition, it is highly efficient and supports multi-version concurrency control, which ensures the ACID compliance [37].
Benefits of PostgreSQL include, but are not limited to, being an open-source, standard compliant RDBMS with a broad community and documentation. Further, it is extensible and objective, whereas on the downside it is less performant for read intensive queries [37].
Hence, whenever high speed with respect to querying and a simple setup is required, this database is not appropriate to use. However, if complex queries in combination with high data integrity is in demand, PostgreSQL may be a good solution [37].

SQLite Database

SQLite[1] also is an open-source, RDBMS library with an available plug-in for Bro [38]. According to [39], it is the worldwide most used database engine. As a self-contained, file-based database, it provides a huge set of tools in order to process almost any data with little constraints [37]. Using SQLite is very fast and efficient, as it works with "functional and direct calls made to a file holding the data [...] instead of communicating through an interface of sorts" [37].
Being a file-based database is a big advantage of SQLite with respect to portability. However, performance tuning is almost not possible. That is a con especially with respect to always only allowing a single write operation at a time [37].
Hence, applications with high write volumes should consider another DBMS, whereas SQLite in principle is a good solution for applications that directly need to read from and write to disk [37].

2.5.2 Non-Relational Database Management Systems

In contrast to RDBMSs, another concept of DBMSs are non-relational database management systems (NRDBMSs). NRDBMSs (also often referred to as NoSQL databases) are missing a predefined, relational model (with all their constraints) in contrast to RDBMSs [40]. In the last years, more and more NRDBMSs were applied, due to reasons

[1] https://www.sqlite.org/index.html

such as the rise of big data containing less structured information [40], as NRDBMSs allow for a unique and not predefined scheme definition that suits the data in use [40]. Thus, data does not have to be structured in rows and columns, but maybe in documents or graphs.

Summarizing, for both aforementioned concepts of DBMSs, processing mainly refers to analysis after structuring, storing and querying information in some scheme (predefined or manually defined). In case of RDBMSs, that could for example include further analytics or statistical aggregations by combining (e.g. joining) queries' results.

2.5.3 Search and Analysis Engines

An additional, but completely different, concept means a cohesive kind of processing and storage option. It stores huge amounts of data, which can either be first queried and results subsequently analyzed or allow implementation of analyzing routines ex ante querying, if respective APIs are provided. Hence, for that concept processing primarily refers to querying and analyzing results ex post storage. Nevertheless, if APIs are available, operators have the options to also manually implement their own analytics ex ante querying and then afterwards query for only their results (as e.g. patterns not typically stored as such but hiding within the stored data).

Elasticsearch

One example for such a concept is ElasticSearch[1]. It is a search and analysis engine based on representational state transfer (REST), which centrally saves all data [41]. It is capable of handling tones of data, but also scaling very well both on distributed indexing and searching [42]. Moreover, it is built upon Apache's Lucene[2] project, it provides various APIs (e.g. Java API and RESTful API), is capable of (almost) real-time search and fulfills ACID consistency [43]. Furthermore, it is also open-source and offers a plug-in to Bro [36] [42]. A detailed description on a possible integration to Bro is given in [44].
Summarizing, Elasticsearch is a good solution for effectively, quickly and dynamically searching data in a huge amount of information. Hence, it is especially useful for search intensive queries. However, if the operator wants to retrieve and save data out of

[1] https://www.elastic.co/de/products/elasticsearch

[2] https://lucene.apache.org/core/

queries, it is not the best choice [45], as exporting search results is not part of its core functionality.

2.5.4 Processing and Storing with Generic Data Formats

Another, general and abstract as well as cohesive processing and storage option is to transform intermediate log files, which contain the previously extracted features, into a generic data format for final FS. Those transformed log files can then be stored in some manually structured way and serve as input to various subsequent (A)DMs, which can easily utilize them due to their general data format. Thus, storage is not necessarily a typical kind of database, but rather may also be the logically and manually organized storage of transformed features' log files. These could for example be directly stored in the file system. Hence, processing for this concept is twofold. First, it especially means transformation of some features' log files into a generic data format, so that subsequent (A)DMs are able to easily access them and then run further detection or misuse algorithms on them. Second, complex analysis tasks (e.g. statistical aggregations) may also be implemented before transformation directly on the original feature log files, which could even result in some new additionally generated logs which finally also get transformed for final storage.

Bro Analysis Tools (BAT)

As an example for that abstract concept, we want to introduce BAT. Included tools are - as already indicated by the name - especially suited for processing and analyzing Bro logs (which contain the extracted features) [46]. Hence, focus in this concept is much more on processing and analysis than on storage by default. So, BAT's main purpose is to read in log files provided by Bro and do post-logging processing and analysis on them [46]. Running complex statistical evaluations and aggregations, applying ML or anomaly detection exploration is all possible with BAT by offering bridges to make those log files accessible for subsequent (A)DMs and ML-components like Spark[1] and others. For that purpose, BAT provide methods for easy transformation of Bro log files to various generic data formats, e.g. Parquet[2], Python dictionaries or Pandas dataframes[3] [46]. Consequently, these various, generic data formats finally also allow compressed and

[1] https://spark.apache.org/

[2] https://parquet.apache.org/

[3] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

efficient storage (e.g. on disk), as well as easy access for further analysis tools due to their generic format. Nevertheless, structuring and organization of this storage solution still remains to be developed manually.
Summing up, BAT provide comprehensive analysis and processing options perfectly compatible to Bro log files [46]. It is not a typical storage concept such as a RDBMS or a NRDBMS, but offers transformation to generic data formats, so that multiple subsequent tools or analysis components can easily access those data appropriately stored on disk. So, BAT may be seen as a storage preparator or assistant, whereas the structure and organization are manually developed.

Parquet as Generic Data Format

One of those generic data formats that are supported by e.g. BAT is Parquet. It is a columnar, self-describing and language-independent storage format [47], which further supports compression [48]. For a detailed overview of its characteristics we want to refer to its official documentation provided in [48].

Chapter 3

Analysis

This chapter describes and analyzes the underlying problem statement of our work in Section 3.1. Outgoing from the problem statement, important high-level requirements that need to be respected in our solution are categorized into different classes and explained in Section 3.2. Closing this chapter, different design choices regarding single aspects of our final solution are presented in Section 3.3.
Furthermore, we declare the (sub-)network(s) under surveillance as N(s)US for easy reference throughout this thesis.

3.1 Problem Statement

Correctly behaving and communicating network devices are crucial for the safety and security of their spanning networks. If those networks are themselves part of critical infrastructure such as cars or aircraft, the necessity for appropriately detecting anomalies in network communication is even more important. For that purpose, an allover anomaly detection process as depicted in Figure 2.1 is necessary.
The groundwork with essential influence on the quality of such an allover detection process is the determination, extraction, collection and appropriate processing of features from network traffic. However, networks in e.g. cars and aircraft are especially characterized by strictly hierarchic topologies as described in Section 2.2. Furthermore, due to their hierarchy those OBNs are compartmentalized into different clusters of networking devices within different decentralized or distributed (sub-)networks [5].
In general, extracting, collecting and processing network traffic in centralized networks is not a problem. At a central point in the net, the traffic is captured by e.g. installing

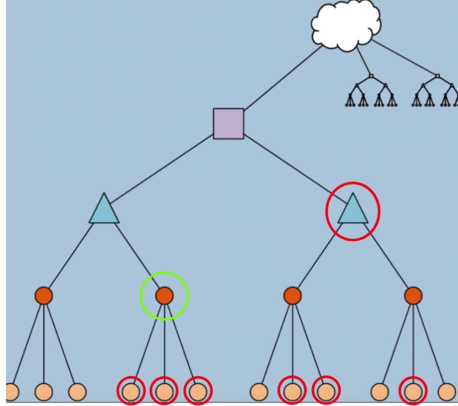

FIGURE 3.1: A typical, hierarchic topology with differently marked components of an on-board network based on [6]

a switch with a mirrored port. That way, features can be extracted from the interesting network traffic very easily. However, in hierarchic networks with many compartmentalized and decentralized devices, such a central gateway / point is not available for capturing only the relevant traffic.

Figure 3.1 based on [6] illustrates this problematic. A typical, hierarchic OBN's topology is depicted, with orange circles representing clusters of decentralized and compartmentalized devices in various (sub-)networks. For further reference, we specify three (types of) components.

- Each orange device within a red circle is referenced as a component of type A.
- Device within green circle is referenced as component B.
- Blue triangle device within red circle is referenced as component C.
- Each orange device without a red circle is referenced as a component of type D.

As an example, those devices of type A and their communication traffic should be monitored. Hence, there is no central point in the network, which allows for capturing the communication traffic of all - but only - those devices of type A. Of course, one could implement a switch at the device B, because all underlying network agents are of type A and their traffic needs to be monitored. Though, one would only capture the traffic from those three devices of type A below component B and not from the other three agents of type A under the component C. Furthermore, it is not possible to utilize component C as a central gateway, because some of its inner devices should not be monitored, as they are of type D.
Thus, we need to propose a decentralized approach for extracting, collecting and pro-

cessing network communication traffic features in such hierarchic, compartmentalized networks. Following this idea, there are three potential ways to do so.
First, we could simply monitor all the network traffic of all devices - regardless of their type - and afterwards discard all the information related to components which are not considered relevant. However, in our example we are generally only interested in traffic of components of type A. Thus, this approach would first of all induce an enormous overhead on monitored traffic, which increases necessary effort in various aspects. For example, handling more traffic takes more time and disk space for respective feature extraction, collection and processing. Moreover, those OBNs and their agents are in principle and per assumption constrained in their resources, which would further amplify these negative overhead affects. Additionally, after FE it may become (almost) impossible to discard unwanted information from irrelevant devices, as extracted features may combine and build around this data.
A second option is to simply integrate additional monitoring and maybe analyzing components into each NUS logically next to each device under surveillance. However, considering specific requirements of especially the avionics and car industries, this approach has some disadvantages. Integrating additional components leads to more costs, higher network complexity and requires more physical space and weight, which would negatively influence the competitive capability.
The third way is to utilize already available, free and so far unused resources on those devices of type A, although they might be somehow limited. That implies, devices whose traffic should be monitored become monitoring agents. This approach is the innovative idea of DecADe [1].

Although going that third way is in accordance with the industries' conditions, some additional problems need to be considered and properly handled. The first one is to facilitate each respective networking agent with the abilities to monitor and analyze its passing network communication traffic. Resulting from that, some form of coordination among those devices is required. Agents only analyzing the traffic directly passing themselves and not bringing their extracted features into correlation with others limits conclusions about characteristics e.g. concerning the entire NsUS. Third, in close relation with the need for coordination among those devices is the need for different types of them, each indicating some specific tasks and abilities. As already hinted in the aspect mentioned before, comprehensive and encompassing statements about the network only arise from the combination of extracted features of multiple monitoring devices. Therefore, some agents need to send their results to some or at least one other device, where the extracted features are gathered and (at least intermediately) stored. Outgoing from this challenge, another one follows. An appropriate storage format and

concept for extracted (and maybe further processed) features needs to be easily accessible and efficiently queriable by various subsequent (A)DMs. Last but not least, flexible reconfiguration and extensibility needs to be ensured. On the one side, arbitrary additional features should be able to be handled. On the other side, flexible definition of the agents being part of the monitoring process should be possible. Furthermore, flexible determination which features are handled by which agents should be possible, too.

Summarizing, extraction, collection and processing network communication features from strictly hierarchical, decentralized and compartmentalized networks is challenging as explained above.

3.2 High-Level Requirements

Based on the problem statement above, some (partly already hinted) additional high-level requirements are deduced in the following. Those are categorized into functional (see Subsection 3.2.1) and non-functional requirements (see Subsection 3.2.2) with some further, fine-grained classes. Moreover, as the overall system's intention is to serve as a decentralized FH system, the aspects of decentralization and necessary coordination among those decentralized components characterize the generic functional layout.

3.2.1 Functional Requirements

Functional requirements (FRs) describe, which basic functionality the final system FHS-DNA should have.

FR.1: Feature Extraction from Network Communication Traffic: Extracting features of multiple levels of abstraction as described in Subsection 2.3.2 from network communication traffic states the groundwork for FHS-DNA and must be possible. Hierarchic, compartmentalized networks demand this task to work in a decentralized way.

FR.2: Processing of Previously Extracted Features: Features should not only be extracted, but also further processed. Hence, FP generally needs to be allowed by FHS-DNA. It could include for example the creation of new features out of the already existing ones (e.g. statistical aggregations), execution of various analysis options on features or the transformation of extracted features' log files into an appropriate storage format. In any case, it refers to the post-processing after extraction and the pre-processing before import of features by subsequent (A)DMs.

FR.3: Storage of Previously Extracted and Processed Features: Features are not pushed out to subsequent (A)DMs immediately. Hence, querying and importing features is initiated by the subsequent (A)DMs, which is why they need to be stored in an appropriate format and scheme in the meantime.

3.2.2 Non-Functional Requirements

Non-functional requirements (NFRs) describe characteristics of FHS-DNA, that are not directly related to its functionality. Hence, they determine the constraints of the system that need to be fulfilled. They are further categorized into general claims, NFRs for FE and NFRs for FS.

General Claims

In the following, NFRs considering more than only one aspect or task of the overall system are explained.

NFR.1: Ease of Integration onto Devices Inside the NsUS: As described in Section 3.1, the innovative idea of DecADe is to do decentralized FH directly on the network agents under surveillance by utilizing free resources. Hence, our system FHS-DNA should allow easy integration of such FH capabilities onto the already existing agents of the NsUS.

NFR.2: Explicit Applicability on Network Communication Data: Focus in our work is on network communication traffic and not application data. Hence, FHS-DNA explicitly needs to be capable of FH of network communication features. Nevertheless, our approach may also be generalizable to network application data, although this is not explicitly required.

NFR.3: Generality of Interfaces in Their Utilized Technologies: With respect to their utilized technologies, interfaces should be as generic as possible. That in particular means not to introduce any unnecessary compatibility restrictions. In specific, the storage format must be usable by common subsequent (A)DMs.

NFR.4: Applicability on General Networks, Topologies and Industries: Furthermore, FHS-DNA should not be restricted to any general types of networks, topologies or industries, although specific characteristics of OBNs in the car and aviation industries are respected.

NFR.5: Automation of Entire Feature Handling Process from Extraction To Storage: After an initial setup, FHS-DNA is supposed to work totally autonomous and without any mandatory interference with the operators. Explicitly excluded from this requirement is the reconfigurability from NFR.6, which naturally functions on demand.

Feature Extraction

NFRs concerning only certain aspects of FE are presented in the following.

NFR.6: Reconfigurability of Monitoring Agents and Their Handled Features: Reconfigurability refers to FE in two aspects. First, the system should allow to adapt the selection of devices and their interfaces being monitored. Second, the operators should be able to explicitly and dynamically order, which features are handled by which devices and therefore are extracted from which interfaces.

NFR.7: Extensibility of Features Handled: Network communication features originally not considered by our proof of concept (PoC) implementation of FHS-DNA should be able to be manually implemented and added.

Feature Storage

Lastly, some NFRs for FS are explained.

NFR.8: Ability to Query and Combine Stored Features by Subsequent Detection Models: Stored features must be easily queriable by (and therefore accessible to) subsequent (A)DMs, because they are not pushed to but rather pulled by them. In addition, storage format should allow them meaningful combination of different queries' results, for instance with a join.

NFR.9: Storage Format Efficiency in Disk Space and Query Performance: Storage format should support compression, in order to save disk space with respect to the large amount of network data. Further, the storage format should allow fast read times by subsequent (A)DMs, especially for large-scale queries.

3.3 Design Choices

Based on the problem statement and the identified requirements, a system for decentralized feature handling is developed. As there are different design choices for single aspects of the final system, we shortly want to explore them in the following. For each aspect of these design choices below, we present two different approaches. Certainly, further ones are possible, too. Nevertheless, the explained ones are always contrary on a rather high level and therefore in principle form the basis for other ones, which often result from a balanced and synthesized combination of the explored ones. This can for example be seen in Section 5.1 for our final design decisions.
Moreover, criteria on which all approaches of the different design aspects are compared to each other mainly refer to the constraints of our final system and the underlying NsUS. Hence, these constraints include differently available resources and requirements (e.g. of different components) especially with respect to CPU, disk space, memory and storage, bandwidth within the NsUS and to external components and information loss.

3.3.1 Feature Extraction

For FE, the requirements already adumbrate how it is intended to work. Nevertheless, it is not pre-specified whether traffic capturing and feature extraction are done together or as separate steps.

Capturing and Subsequently Reading Traffic for Feature Extraction

A first option for extracting features with our final system is to distributively capture network traffic, save it in e.g. pcap-files and then send those to other components, which finally extract features from the reread traffic files.
Overall, this seems to be a good way of task sharing and results in a clear separation of concerns and responsibilities. Moreover, the captured and saved traffic may be further manipulated or changed (i.e. for example filtering some traffic or tampering with packets) according to specific (e.g. testing) needs with an intermediate manipulation component before finally being sent to and reread by the FE components. However, capturing - maybe even filtered - network traffic and then resending it actually induces more load than necessary. First, saving that traffic in e.g. pcap-files requires huge amounts of memory and disk space, as those files typically can grow very large. Second, sending those pcap-files to other components for subsequent feature extraction requires huge bandwidth depending on their size. Furthermore, this also requires some CPU. Finally, the other components further read the pcap-files and therefore need to process

contained traffic again to extract desired features. Finally, as another consequence, FE from live traffic is never possible, which prevents FHS-DNA to be operated within an actual and active network without having recorded traffic available.

Monitoring and Feature Extraction on Live Traffic

In contrast to the first option, the second one aims to monitor traffic and directly extract features from it. Consequently, network traffic is not needed to be saved in e.g. pcap-files, does not have to be resent to and read by additional components.
Hence, monitoring traffic and directly extracting features preserves much more resources. First, no traffic needs to be captured resulting in huge files, which preserves memory and disk space. Second, those pcap-files do not have to be sent to other components, so no bandwidth is lost. Third, potentially saved traffic does not need to be processed a second time, which preserves CPU. This is especially true, because handling traffic per se makes up a major portion of processing time compared with FE as we show in our evaluation in Section 7.2. So, even though FE requires some additional CPU on those devices, that approach overall preserves CPU compared to the first one. This is further affirmed, as our requirements of DecADe and from Section 3.2 demand the FE to directly take place on the free resources of the devices of the NsUS, which per assumption are resource-constrained. Hence, this prevents the first approach to outsource the FE to substantially more powerful components. However, for FE from live traffic the respective monitoring devices have to fulfill both tasks for monitoring and FE. That means, each one has to be facilitated with appropriate capabilities. Moreover, as traffic is not saved in files it subsequently cannot be manipulated for other (e.g. testing) needs. Thus, the entire traffic always is analyzed as it is.

3.3.2 Feature Processing

The specific meaning of FP depends on the technologies and concepts used (see Section 2.5) and the particular design decision for FP (see Subsection 5.1.2).
Furthermore, we state for better clarification that *external* in the following refers to components that are not part of the NsUS and therefore are not constrained in their resources (or at least not to the same extend). On the contrary, *internal* refers to more resource-constrained (per assumption) components inside NsUS.

On Decentralized Monitoring Devices

The first option is to do FP on these internal monitoring devices, which also do FE. As a result, not only FE, but also FP would work decentralized.
Hence, for this approach to function, it is a mandatory prerequisite that those agents are equipped with capabilities both for FE and FP. However, it is important to keep in mind that the available resources on those utilized network devices are limited. Consequently, one major disadvantage of this approach is that extensive FE and FP at the same time mutually limits each other. Hence, less comprehensive FP (e.g. analysis of extracted features) would be feasible when handling traffic peaks, as FE also needs to be done and consumes some CPU, memory and disk space. Of course, depending on the specific definition of processing, some subtasks are not necessarily executed simultaneously to FE (e.g. the potential transformation of extracted features' log files to another storage format). Nevertheless, other subtasks of FP (e.g. the creation of some certain, statistical aggregated features) would need to be done simultaneously to FE. This is especially true for features relying on some mandatory, contextual information, which is not contained in the basic ones. Nevertheless, although this approach mutually limits FE and FP (with respect to CPU, memory and disk space), one major advantage is that in principle multiple - even different - FP tasks may be executed at the same time and even on different feature subsets. This is possible, as each decentralized monitoring device could work as an autonomous instance and do its own FP. This can e.g. include the FP subtask of storage format transformation, which would result in many, but small storage data.

On Single, External Component

A second option is to do central FP on an external component in contrast to decentralized FE on the internal monitoring devices. Thus, no processing capabilities need to be deployed onto the network agents, whereas an external FP component can easily be installed.
A first major disadvantage of this approach is that distributively extracted features need to be sent to this external component. Depending on the sizes of the extracted features' log files, this consumes some additional bandwidth, as those files are not yet compressed, as compression is considered as a FP subtask. Furthermore, another additional, resource-unconstrained component is required, of course. On the contrary, the missing option of internal FP (as provided e.g. in the first approach) also yields a major advantage. This is, that all available resources (memory, CPU and disk space) of those internal agents can be utilized for FE only. On the one side, this allows more complex and comprehensive FE when handling traffic peaks. Thus, less information may get lost, as more features can explicitly and immediately be extracted from live traffic. On

the other side, also more comprehensive and powerful FP (e.g. analysis) is possible on a non-restricted external FP component. As a consequence, a trade-off between FE and FP is not needed anymore, as it would be with the first approach. Hence, this one introduces another degree of freedom for the operators to choose whether to extract some complex features directly from traffic or construct them by processing basic ones. As a result, sometimes complex FE is not needed anymore, as those complex features may also be gained by processing original ones, which would further preserve resources on the internal devices and therefore allow extracting more basic features for higher traffic peaks. However, depending on the specific features of interest, it is important to keep in mind that some kind of information might still get lost when not explicitly and immediately extracted from live traffic.

3.3.3 Feature Storage

FS means not the intermediate saving of extracted features' log files, but rather the final storage of features in an appropriate format, from which the external (A)DMs can import features. As already mentioned above, the transformation of intermediate feature log files into the final storage format is considered as a subtask of FP. Therefore, it is important to understand that each device supposed to do feature transformation needs to have FP capabilities. So, this characterizes the close relation between FP and FS throughout this work.
Besides the common assessment criteria of available resources, the evaluation for FS design choices is also particular based on related requirements regarding the comfortable access, querying and combination of stored features (especially with respect to NFR.8). Furthermore, the self-initiated traffic is another important criterion, as this also costs bandwidth and further may falsify the FE.

On Decentralized Monitoring Devices

First, a decentralized FS approach within the NsUS is imaginable. Each device extracting features could also finally store its features. This requires each device to either transform its features into a compressed storage format (according to NFR.9) on its own or send them to a transformer (which basically is a FP component then) and receive them back for storage.
For both cases there are some disadvantages. In the first case, each device needs transformation capabilities in addition to FE capabilities. However, resources on each device are limited, so available disk space and CPU may be restricted. That leads to similar negative effects as described in Subsection 3.3.2. In the second case, transformation

would be done centrally, whereas final storage is still intended to be done decentralized. Consequently, a lot more self-initiated traffic is generated and bandwidth is consumed, as features would be sent across the NsUS to the central transformation component and then received back from it after transformation. Moreover and independent of the two cases, the access to decentralized stored features by subsequent (A)DMs is more complicated to handle than it would be for a single, central storage. In close relation to this, comprehensive statements concerning e.g. the entire NsUS also become much harder. That is, as external (A)DMs must access multiple storages of various devices and correlate their (transformed) features, instead of accessing just a single storage.
Nevertheless, there are also some advantages. First, as each FS component only stores its own extracted features, the storage per device is much smaller than a single, central storage. That increases clarity and simplifies organization and structure of the final storage, as features of specific devices or interfaces are known to be found in the respective agent's storage. Second, having transformed the features to a compressed storage format already inside the NsUS saves bandwidth per query for transferring them to the outside of the NsUS. However, having the storage inside the NsUS requires transfer of features and therefore some bandwidth for each query by the (A)DMs. Hence, depending on the query frequency, this may eventually cost even more bandwidth in total than having the features stored externally.

On Single, External Component

Therefore, a single, external storage option is also possible. Hence, all intermediate feature log files of all devices get transferred to an external component, which then transforms those into a compressed storage format and stores them. Hence, transformation capabilities do not need to be deployed onto the network agents themselves.
A first disadvantage is, that another additional, resource-unconstrained component is required. Further, a central, comprehensive storage grows bigger than many decentralized ones. Although the total storage size basically would be the same, this may make it a little harder to simply, clearly and intuitively organize the storage, as features of all devices and interfaces are then stored within a single storage rather than on each respective device directly. Hence, the storage structure is more complex. Moreover, as the external storage gets the uncompressed and untransformed intermediate feature log files transferred, the storage either is not always absolutely up to date (between two transfers the external storage will be the same), or must get updated with transferred features on a very frequent basis. However, the smaller this time interval gets, the more bandwidth again is consumed for feature transfer and self-initiated traffic is increased, but the external FS would get more up to date. Hence, this requires some trade-off per

default.
However, there are also a couple of advantages. First of all, an external, resource-unconstrained component can easily be integrated and storage growing big on there is in principle not a problem. Further, in contrast to the second case of the first approach, intermediate feature log files only need to be transferred one-way from the internal devices to the external FS (i.e. they do not need to be sent back to them). In contrast to the first choice, this preserves some bandwidth and limits self-initiated traffic, as not for every query from the external (A)DMs features need to be transferred from the device-specific internal storages. Hence, the total required bandwidth over all (A)DMs' queries do not require any bandwidth nor induce additional traffic inside the NsUS, because the storage would be external as well as the querying (A)DMs. Outside of the NsUS, bandwidth is not a bottleneck anymore and traffic is not monitored. However, the updating feature transfers do induce additional traffic and consume bandwidth. That is especially true the smaller this update interval is chosen. Hence, a reasonable trade-off has to be found. Another advantage is, that feature access for subsequent (A)DMs is much simpler, as they only need to have access to this single, external storage and not multiple, internal ones. Furthermore, comprehensive statements are possible to be made by this external component even before final storage, as all feature log files will be available on it after transfer and before transformation. Last but not least, all available resources of those internal agents can be utilized for FE only, which further saves memory, disk space and also CPU (as no transformation is done).

3.3.4 Cohesive Processing and Storage Concept

Independent of the specific design decisions finally made, there has to be some harmonized interaction between processing component(s) and storage component(s), as they are closely related with each other as already indicated above. Hence, a cohesive concept between FP and FS is indispensable, as e.g. the transformation into an appropriate storage format is considered as part of the processing. Thus, there are multiple potential options. These generally result from the different combinations of design decisions for FP and FS. Identification of a final, best solution has to be made once the underlying single design decisions are clear.

Chapter 4

Related Work

This chapter explores some related work by classifying it into a couple of different categories. Each category is explained in its own section.

4.1 Anomaly Detection for SOME/IP

In close relation to NIDSs and with special focus on the automotive and aerospace industries, Herold et al. suggest in [49] an ”anomaly detection system for SOME/IP” [49]. SOME/IP is a ”standardized automotive middleware protocol” [49], which is used for control messages [50]. In the proposed solution, domain specific rules are applied to SOME/IP packets. This way, attacks and protocol violations can be detected [49]. This is an especially useful extension for anomaly detection in specific packets, as other common NIDSs - such as Bro and Snort - lack in adaptability to SOME/IP [49].

However, the presented approach mainly focuses on packets and overall is an anomaly detection approach. In contrast, we aim to develop a system that is not supposed to do anomaly detection, but rather enable it by doing the groundwork regarding FE, FP and FS up-front.

4.2 VAST

VAST[1] stands for "Visibility Across Space and Time" [51]. It is "a distributed platform for high-performance network forensics and incident response that provides both continuous ingestion of voluminous event streams and interactive query performance" [52]. Outgoing from that, VAST can also be considered as kind of a search and analysis engine as explained in Section 2.5, because it is supposed to provide "*(i)* an expressive data model to capture descriptions of various forms of activity; *(ii)* the capability to use a single, declarative query language to drive both post-facto analyses and detection of future activity; and *(iii)* the scalability to support archiving and querying of not just log files, but a network's *entire activity*, from high-level IDS alerts to raw packets from the wire" [52]. Hence, VAST may be seen as an intelligent database concept that processes logs from network traffic in addition to enabling network forensics and incidence response mechanisms. Furthermore, that processing is intended to be very extensive, scalable and conclusive. Moreover, it is supposed to allow the import of log files from various different systems, such as Bro, firewalls, embedded devices and IDSs in general [51]. For the sinks on the other side of the architecture, it already supports exporting information to various formats [52], e.g. ASCII[2], JSON[3], Bro, pcap and Kafka[4]. A high-level architecture of VAST can be found in [53].

In the context of our target system, VAST may be considered as a compounded feature database and analyzing component. Hence, VAST can be compared to our described design choice of a cohesive FP and FS component in Section 3.3, which allows analysis and other processing of stored information. So, VAST could possibly be integrated into our target system as such a component. Regarding the general technology or concept applied for the potential implementation of such a design choice, VAST can be seen as a search and analysis engine as explained in Section 2.5. However, right now, both no actual release nor an official documentation is available. Thus, VAST is heavily under development at the moment and not yet offers the full functionality it is proposed to do. This fact is underlined as only very limited analysis options for processing are available so far [54]. Therefore, depending on our concrete intention to already provide more

[1] http://vast.io/

[2] http://www.theasciicode.com.ar/

[3] https://www.json.org/

[4] https://kafka.apache.org/

potential feature processing and analysis tasks than VAST currently offers, it finally is not yet utilized by FHS-DNA.

4.3 Collective Intelligence Framework

The Collective Intelligence Framework (CIF) is an intelligence management framework for cyber threats, which allows the operators ”to combine known malicious threat information from many sources and use that information for identification (incident response), detection (IDS) and mitigation (null route)” [55]. Hence, it can be compared to a server fetching intelligence data from multiple sources, whereas a NIDS like Snort or Bro may be the client to query and (post-)process CIF's output data. Further, ”CIF is used mainly in two ways: either to query for data stored about an IP address, a domain or a url, or to produce feeds based on the stored data sets” [56]. Those feeds can then serve as input to tools like Snort or Bro as mentioned before.

However, as our system especially considers static, strictly hierarchical and closed networks in cars and aircraft - even though FHS-DNA is not limited to them in any way - CIF may not be considered necessary for gathering external intelligence data. In contrast, predefined models of well-known communication patterns inside those NsUS can be used as a benchmark for subsequent anomaly detection.

4.4 Security Information and Event Management Tools

Another category of related work are security information and event management (SIEM) tools. By conducting distributed FE and anomaly detection, SIEM tools seem to be similar to the context of our work. However, they detect aplenty events of network infrastructure components (e.g. servers, firewalls and antivirus filters) on a more high-level view [57] [58]. Events raised, logged, analyzed and evaluated are not low-level per-packet or per-connection information, but already rather aggregated by default instead (e.g. the number of failed logins on a component per time). Additionally, they incorporate high-level contextual information about all types of assets [58]. Another difference is, that our system FHS-DNA will not classify feature instances nor react to them. It will only extract, process and store them, so that another application may build upon them, maybe implement pattern-mining and finally trigger incidence response mechanisms. In contrast, a main excellence of SIEM tools is to do exactly that and detect patterns in the logs through correlation [58] [59] and react appropriately. Therefore, FHS-DNA may be compared to a partial, low-level application layer within an entire, common

SIEM tool or NIDS. Nevertheless, a similarity may be the necessity to log some kind of events and process them in order to make them easy to use in a meaningful way for an adjacent inspection model [59].

4.5 Approaches Partly Similar to FHS-DNA

Another key facet of our work is the decentralization of FH. Approaches similar to FHS-DNA - at least in some aspects - is [60] and [61].

4.5.1 Two-Modules Approach

[60] proposes an IDS consisting of two modules. Traffic from various networks is captured via a central switch with a mirrored port and then sent to the first module - a central traffic collector outside the NsUS. This component extracts the packets from the traffic and even more the headers of the packets and redistributes those to the second module via a so-called master node serving as a gateway between the two modules. The second module is a Spark cluster consisting of multiple worker nodes, each of them having a Hadoop Distributed File System[1] deployed on them for storage and further processing. Their task is to extract features from the previously extracted packet headers and send those features via the gateway master node back to the first module. Finally, this central collector and monitor analyzes the features, trains the IDS's algorithm and updates the cluster [60].

By comparing this approach with our target system, three major differences are evident. First, the presented approach utilizes a central switch with a mirrored port in order to capture traffic. That was exactly the case which is not possible for strictly hierarchical networks as described in Section 3.1. Instead, the innovative idea of the DecADe project is to develop a decentralized FE approach on the agents themselves of the NsUS.

That directly leads to the second difference. In the presented approach from [60], the actual FE is done outside the NsUS. Worker nodes of an external Spark cluster extract features from redistributed packet headers. In contrast, FHS-DNA will deploy lightweight traffic monitoring and FE capabilities on the agents of the NsUS themselves and make use of so far underused resources.

[1] http://hadoop.apache.org/

Third, the approach in [60] only considers features that are extracted from packet headers. As explained in Subsection 2.3.2, we consider features of different vertical abstraction levels. So, features regarding packets, connections and the entire network will be handled - not only from packets or even only packets' headers. Hence, FHS-DNA will be more comprehensive with respect to information and features under consideration.

Regarding potential design choices as described in Section 3.3, the presented approach from [60] decided to first capture and save traffic in files, before redistributing it for actual FE. Hence, this approach implements FE similar to the option as explained in Subsection 3.3.1. Furthermore, regarding FP and FS, it implements a cohesive processing and storage concept similar to the one described in Subsection 3.3.4. FP (in that case especially further feature analysis) is on an external, central component and therefore can be compared to the option of Subsection 3.3.2, whereas FS somehow can be described as a composition of options from Subsections 3.3.3 and 3.3.3, as the features on the one hand are stored outside the NsUS, but decentralized in a Hadoop Distributed File System on the other hand.

4.5.2 Two-Layers Approach

Another approach, which now actually takes place inside the NUS itself, is described in [61]. It suggests a two-layer IDS. Agents on the lower host-layer capture network traffic and extract features with special focus on identified connections. Furthermore, they already analyze those extracted features and their respective connections and in addition apply some background classification on them. These classifications are assessments, whether the identified instances are considered as anomalies or not. Hence, they send instances of concern to the upper classification-layer hosts, which will further take care of these concerns and check the provided classification suggestion with a misuse detection algorithm. This has the purpose to identify the appropriate attack types of those pre-classified anomaly suggestions, before retraining and updating the database and propagating their classifications in order to probably determine appropriate responses [61]. Although this approach is applied on the network agents under surveillance themselves, it differs from our target system in a few major aspects.

First of all, the FH components of FHS-DNA will not make use nor include some particular classification algorithms in order to do anomaly detection. Their purpose is, to extract, process and format some features and finally store them appropriately. Those features may then be used as input to various (ML-based) (A)DMs. Thus, the anomaly classification is not part of our work. Furthermore, our system may not just consider

features characterizing connections, but rather all three levels of feature abstraction as explained in Subsection 2.3.2.

Regarding potential design choices as explained in Section 3.3, the presented approach from [61] decided to implement FE from live traffic directly on the decentralized, monitored devices and therefore can be compared to the option of Subsection 3.3.1. It also uses a cohesive FP and FS concept similar to the one of Subsection 3.3.4. FP in this approach particularly means to distributively analyze connection instances. However, this FP (i.e. analysis) is not completely done on the monitored devices, but rather finally handled by the upper classification-layer hosts, which are not inside the NsUS. So, this FP implementation can be seen as a combination of the options described in Subsection 3.3.2. In contrast, FS better matches a single option described in our design choices, as it uses a single, external database similar to the option described in Subsection 3.3.3.

Chapter 5

Design

In this chapter we introduce the high-level design of our final **F**eature **H**andling **S**ystem on **D**ecentralized **N**etwork **A**gents (FHS-DNA). It is supposed to be independent from any specific implementations, making it generally reusable and deployable to any domain, network, topology or hardware with respective resources.

In Section 5.1, we motivate our design decisions for various partial aspects of the overall system. Further, in Section 5.2 those partial design solutions are compounded to result in a big picture of the allover high-level system architecture. Moreover, this is further explored in detail within different subsections, describing the components within the (sub-)networks under surveillance (NsUS), referenced as *internal* (see Subsection 5.2.1) and outside the NsUS, referenced as *external* (see Subsection 5.2.2).
Finally, important interfaces between those internal and external components are explained.

5.1 Design Decisions

Outgoing from the compared design choices as elaborated in Section 3.3, in this section we argue our partial decisions within respective subsections in the same order.

5.1.1 Feature Extraction

The first design decision to make concerns FE. Two different choices are presented in Subsection 3.3.1.
On the one hand, we considered an option to share tasks of traffic capturing and FE

among different components. Hence, the actual FE would not be on live traffic, but rather on captured traffic saved in files (e.g. pcap-files) sent to other components for reading those files and finally extracting features from it.
On the other hand, we proposed an option which monitors and directly extracts features from live traffic. Thus, traffic is not saved, resent and reread in files, but handled immediately as it is instead.
We finally decided for the second option, where FE is directly done on monitored live traffic. A couple of reasons are stated below for arguing that decision.

First, there is no need to manipulate or tamper with the monitored traffic. This possibility is explicitly given by the first choice, but not the second one. Thus, the first choice would be especially useful for purposes like pre-filtering some traffic or manipulate some packets before extracting features from it, because traffic saved in e.g. pcap-files can be changed or tampered with. Hence, such an approach would also help a system to learn and detect e.g. different intrusion patterns or attacks by training it with differently changed traffic (maybe from an underlying, basic pcap-file). Hence, a system implementing such an approach would turn out to evolve to kind of an anomaly detection system. However, this is not the intention of FHS-DNA. It should provide the groundwork for later anomaly detection (as explained in Section 2.1) and therefore extract (and further process and store) features from traffic as it is. Consequently, those intermediary traffic manipulation capabilities are not needed, would make our system's design overcomplicated and miss the actual purpose. In addition, FHS-DNA should not miss the option to handle live traffic, as this would prevent FHS-DNA to be operated within an actual and active network without having recorded traffic (i.e. for example pcap-files) available.

Second, based on the absence of that intermediate manipulation need, additionally saving, resending and rereading traffic would consume limited available resources, especially disk space due to large files of captured and saved traffic as well as bandwidth for resending them. In addition, also CPU consumption is increased, as the traffic files need to be read again (even though the original traffic may be limited by pre-filtering). This is especially true, as a major portion of processing time is consumed by handling the traffic and not the actual FE as underlined by our evaluation in Section 7.2. By eliminating those unnecessary steps and directly extracting features from traffic instead, we therefore preserve important resources.

Moreover, with the selected option our system is actually capable of handling live traffic, which is not possible with the other choice. That way, we gain an additional time advantage for the overall performance of our system only by design.

Due to these main reasons, a design decision for monitoring and directly extracting features from live traffic makes much more sense and improves various aspects (especially with respect to the constrained resources) of our final system by design.

5.1.2 Feature Processing

A second design decision concerns FP. Again, two different choices are presented in Subsection 3.3.2.
A first option is, to facilitate those internal, decentralized monitoring and FE components of the network with additional capabilities to also do FP. Hence, each network agent as part of the extraction cluster would not only do its own FE, but also its own FP.
In contrast, the second option is to only do central feature processing on a single, external component. That way, FP would be offloaded from the resource-constrained network agents, which therefore would not need to have FP capabilities deployed on them.
As both of these approaches have reasonable advantages but also disadvantages, we decided for kind of an efficient and flexible compromise between central, external and decentralized, internal FP. This compromise aims to combine advantages and eliminate disadvantages of both choices.

Overall, FHS-DNA should be flexible and adaptable with respect to extracted features. Nevertheless, for each specific feature and in general we put strong focus on not to miss any (meta) information that we could get coming along with it. That means, for each feature we also want to extract all information that we can get, although it may not be part of the actual feature. An example for that could be to also log the source port of each connection, even if the actual feature only is about the source address. Hence, it is important to allow comprehensive feature (and information) extraction even for traffic peaks by best utilizing limited available resources. Thus, offloading extensive FP in principle for performance reasons to an external component not restricted in resources per assumption is very helpful. Hence, offloading FP subtasks e.g. includes various, comprehensive analysis options for extracted features and the transformation of features' log files into an appropriate storage format. So, internal agents save e.g. CPU, which can be used for more comprehensive FE then.

Although these subtasks of FP are decided to be offloaded from the agents within the NsUS, we still can do some form (i.e. subtask) of FP directly on the internal agents, which is the creation of statistically aggregated features.
As already mentioned before, a result from offloading intensive FP (e.g. analysis options) is the growing ability of extensive FE on the internal agents. Hence, also more complex

features (e.g. statistically aggregated ones) can directly be extracted from live traffic. That way, these complex features already may be extracted and therefore do not need to be constructed anymore by subsequent FP.
In other words, the direct extraction of more complex features actually is a task of FE, but can also be considered as a subtask of FP. Therefore, extracting complex features has direct, positive effects on FP, as these complex features would in principle also be able to be generated (i.e. not being extracted but constructed later on) out of more basic ones (i.e. which explicitly have to be extracted) during FP.

Although FP on an external component is per assumption not restricted in resources, this further yields two major advantages.
First, by directly extracting (i.e. not generating by FP) those complex features, the above mentioned additional (meta) information can also be gained from traffic. Some (meta) information would not be generated when constructing the complex features during external FP, as it maybe simply cannot be deduced from already available features or information. As reduction of information loss to a reasonable minimum is intended to be possible with FHS-DNA, this is an important aspect for our decision. Hence, the crucial argument for this kind of compromise is not only the available resources, but also the minimization of information loss.
Second, having the additional option to directly extract more complex features also allows the potential extraction of features, which may not be constructed with provided analysis capabilities during FP. Thus, this is closely related to the first reason of not missing any relevant information (or even features) from live traffic.
Nevertheless, of course it is still possible to only extract some basic features and later on further process (e.g. statistically aggregate or analyze) them on the external FP component. Hence, this design induces great flexibility and another degree of freedom for the operators. If the respective FP capabilities for a specific feature are available on the external FP component, the operators can decide whether to directly extract or generate (i.e. resulting from FP) it.

Summarizing, no specific additional FP capabilities are deployed on the internal monitoring devices. Instead, an external component provides specific FP tasks and capabilities, like e.g. the transformation of log files into a compressed storage format and various analysis options. This preserves resources on the internal agents, which allows more complex and comprehensive FE, which furthermore reduces the need of subsequent FP, as comprehensive features are already extracted. Hence, this closes the circle of synergy effects resulting from that solution.

5.1.3 Feature Storage

The third design decision concerns final FS. Final storage means the storage of all transformed and therefore compressed features in an appropriate format, from which the external (A)DMs can query them. Once again, two different approaches are presented in Subsection 3.3.3.
A first choice is that all internal monitoring and extraction devices directly save their own extracted features. That means, each monitoring agent would not only do FE, but also partial (i.e. its own) final FS. So, as extracted features are not compressed and in the appropriate storage format by default, this choice requires either each FE component to do local transformation (i.e. a subtask of FP) on its own or to send the extracted features to a central transformation component and receive them back.
In contrast, the second choice suggests that FS is done on a single, external component. Therefore, no finally transformed features would need to be sent back to the internal agents.
Similar to the above design decision regarding FP, we also decided for kind of a compromise solution here. Thus, its concept is twofold and again tries to combine advantages and eliminate disadvantages of direct FS inside the NsUS and an additional, external FS component.

We already explained above, that feature transformation is a subtask of FP. Hence, in the first case of the first choice, each internal FE agent which should also store its own features would further need FP capabilities. However, we already argued in Subsection 5.1.2 why we decided to offload extensive FP (which includes feature transformation) from internal agents to a single, external component. Consequently, the first case of the first choice can directly be neglected according to the same reasons.
Hence, from the first choice it remains the second case. Though, as the limited resources like the available bandwidth and the self-initiated traffic are major factors in our decision, neither the merely second case of the first choice nor the merely second choice is perfectly suited as argued in Subsection 3.3.3.

Therefore, our compromise combines the idea of a single, external, final FS and a single, intermediate FS on a dedicated agent inside the NsUS. Within the internal, intermediary storage, only untransformed (and therefore uncompressed) features are saved, whereas the final FS will contain the finally transformed and compressed features and therefore form the interface to the subsequent (A)DMs' queries. Moreover, the final FS will be updated once in a reasonable time interval, whereas the internal, intermediate FS will always be absolutely up to date.

The first main reason for that compromise is that it saves bandwidth and reduces self-initiated traffic inside the NsUS. That is, because all features extracted from all decentralized FE devices are directly and originally logged solely on that aforementioned dedicated device and therefore do not need to be sent across the NsUS. However, as this dedicated, internal device is also resource-constrained, it does not have to do FE itself but solely should collect the log files. Thus, it is clear that for this internal, intermediate storage the device of the NsUS with the most available disk space, but probably with the less required CPU should be chosen. In contrast, all FE agents do not need to have any disk space for FS and can utilize all their resources solely for FE.

A second main reason for that solution is, that the reasonable time interval for the synchronization to and update of the final, external FS further limits consumed bandwidth and self-initiated traffic to a reasonable minimum, as features are not sent out permanently but still keeps the final storage relatively up to date (depending on the specific update frequency). Each pre-specified time interval those internally gathered log files are synchronized from within the NsUS to an external, not resource-constrained transformation component, which results from the solution identified in Subsection 5.1.2, which then further initiates the update of the final FS.

Third, access for subsequent (A)DMs is much simpler, as they only need to have access to this single, external storage instead of to each internal one. That way, comprehensive feature correlations are also possible to be made by this external component even before final storage.

Summarizing, this compromise solution once again underlines the close interrelation of FS and FP. Therefore, the following last design decision identifies a cohesive concept as shortly explained below.

5.1.4 Cohesive Processing and Storage Concept

A fourth, last design decision is to be made on the cohesive processing and storage concept.
As already hinted by the aforementioned design decisions, they offer to combine the external transformation and the final storage component. Thus, it has to be decided, whether to combine those external FP and FS components or not.

Per assumption, external components are not restricted in resources as it holds for the networking devices. Hence, we decided to compound external components for both FP and FS into a single one. This especially makes sense, as final storage requires transformation, which is considered as one aspect of FP. Hence, once transformed, the

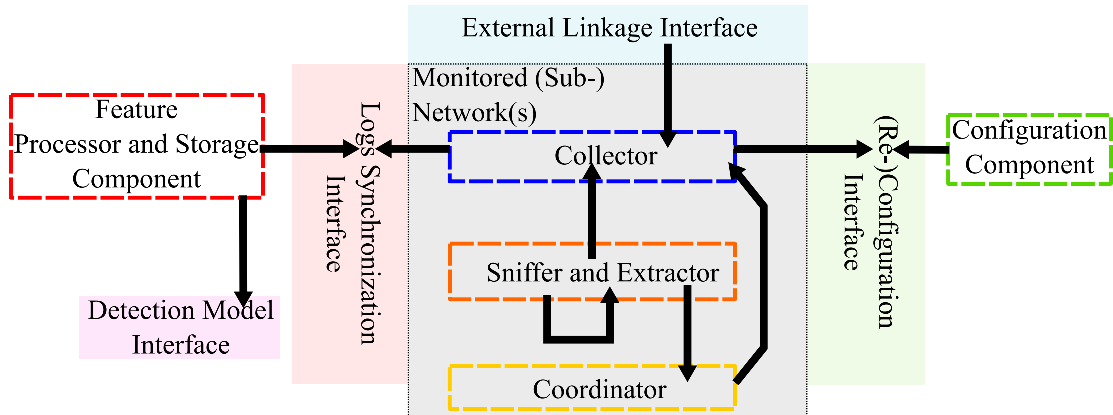

Figure 5.1: Architectural system design overview

same component can directly store the features.
Moreover, this design decision keeps the number of components and interfaces to a reasonable minimum, which further simplifies the overall system architecture. This architecture is now explained below in more detail.

5.2 System Architecture

Based on those design decisions explained in Section 5.1, we come up with an overall architecture for our system. An overview of it is depicted in Figure 5.1. It is explored in detail in the following by first describing internal components (see Subsection 5.2.1) and then external ones (see Subsection 5.2.2).
Moreover, besides all components and their important main tasks also all interfaces are explained in the respective (sub-)sections below.

5.2.1 Internal Components

Again, *internal* refers to components within the NsUS. Thus, these are not additional components, but rather those monitored devices, on which our system FHS-DNA is deployed on to handle features as explained in Section 3.1. Hence, all those components are limited in their available resources per assumption.
Furthermore, as already hinted by the specific design decisions in Section 5.1, there is the need of having different types of internal components taking over various tasks. It

is important to know, that all those types and their tasks are disjunct per assumption. That means, each type fulfills specific tasks, which are explicitly not taken care of by other types. Consequently, a type explicitly defines some tasks and resource requirements for each device of that type. Moreover, all these types mandatorily need to be present in a monitoring cluster of FHS-DNA, which basically means that at least one device must be of each type. In principle, also one device can be of multiple types at the same time. Furthermore, it is also possible and allowed that between two runs of FHS-DNA's monitoring cluster devices change types. Hence, this increases the flexibility of FHS-DNA, as each device in principle can take over each type (as long as they fulfill the respective resource requirements of that type).

Sniffer and Extractor

A first type is named *Sniffer and Extractor*. Its tasks are to monitor the passing network communication traffic and extract features from it as specified by the operator beforehand. According to the design decisions as stated in Section 5.1, this type does not care about further FP (e.g. transformation or analysis) nor FS.
Nevertheless, it logs all its extracted features directly and originally onto the *Collector* typed device for intermediate, internal FS as explained in Subsection 5.1.3. Hence, this type requires less disk space than a *Collector* as it does not log any extracted features, but some CPU resources for sniffing the network traffic and corresponding FE. Furthermore, that way bandwidth inside the NsUS is preserved and self-initiated traffic reduced, as no additional transmitting of extracted features' log files from the *Sniffer and Extractor* agents to the dedicated internal, intermediate storage agent (i.e. the *Collector*) is necessary.
In general, within the monitoring cluster of FHS-DNA there can be multiple devices of this type at the same time.

Collector

The second type of internal devices for our system is named *Collector*. In contrast to the above type, FHS-DNA's monitoring cluster always only contains exactly one device of this type at a time. Its task is to save all decentralized extracted features of all *Sniffer and Extractor* typed devices on its disk. As already explained before, this is done by allowing the *Sniffer and Extractor* typed devices to directly and originally log onto the *Collector*, as this preserves bandwidth for not needing to send the extracted features across the NsUS. Hence, this typed device serves as the intermediate, internal FS, from where the log files (containing the extracted features) are synchronized once in a rea-

sonable time interval to the external, cohesive FP and final FS component (resulting from design decision in Subsection 5.1.4). Hence, the *Collector* needs more disk space than any *Sniffer and Extractor*, as it saves all their extracted features, but requires less CPU, as it does not monitor traffic nor extract features itself. Thus, as this feature gathering type already by design plays a central role among those internal devices, it further serves as a gateway to the outside for different purposes.
First, as the intermediate, internal FS component it provides an interface to the compounded external FP and final FS component outside of the NsUS.
Second, the *Collector* provides another interface for (dynamic re-)configuration of FHS-DNA's monitoring cluster. That means, via the *Collector* it is possible to specify, which devices of the NsUS are part of the monitoring cluster and of which type. Even further, the operators can specify, which of those *Sniffer and Extractor* typed devices extract which features.
Third, also starting and stopping FHS-DNA's monitoring process is initiated on the *Collector*, which is especially needed when a new configuration of the monitoring cluster should be loaded.

Coordinator

The third and last type for internal components is named *Coordinator*. Again, within a monitoring cluster of FHS-DNA, it is in principle possible to have more than one *Coordinator* at a time. Its task is to keep state across different devices of type *Sniffer and Extractor*. This is especially necessary, if multiple, decentralized devices of type *Sniffer and Extractor* extract and log features to the same feature log file on the *Collector*. As all logs are intermediary gathered at a central, internal point (i.e. the *Collector*), it is reasonable to have only one common log file for each feature whenever possible, which is written by multiple *Sniffer and Extractor* devices.
However, there are also features which corresponding logs are not able to be aggregated across *Sniffer and Extractor* devices (and respective interfaces) as described in Subsection 5.4.2 below.

5.2.2 External Components

In contrast to internal components, *external* means outside the NsUS. It can even be, that they are physically not located next to each other or the NsUS. With respect to the car and aircraft, that could mean they do not even have to be on board. Consequently, those components do not have to be restricted in their resources per assumption.

Feature Processor and Storage Component

The first external component is the *Feature Processor and Storage Component*.
As explained in Subsection 5.1.4, it is reasonable to combine the external FP and final FS component into a single one. Hence, its main task is to poll out the intermediately and internally collected feature log files from the internal *Collector* typed device. This is done once in a predefined time interval by synchronizing the respective directories.

Although this means that the final, external FS is not always absolutely up to date with respect to actual extracted features' log files, it is important to understand that related design decisions (as explained in Section 5.1) do not only preserve limited available resources (especially bandwidth), but even further does not loose in any way the fundamental advantage of FHS-DNA to handle live traffic and extract features from it. In case we would have decided that handling live traffic is not necessary for FHS-DNA, it would never be possible to directly operate FHS-DNA in an active network without having previously recorded traffic (e.g. saved in pcap-files) available. Hence, that would generally restrict FHS-DNA to be applied to any use case where no recorded traffic is available.

Further, once the feature log files are synchronized to the external *Feature Processor and Storage Component*, it has multiple opportunities of locally processing them. This includes comprehensive analysis options, the creation of new features out of already available ones (e.g. statistical aggregations) and the transformation of the intermediate feature log files into the final, compressed and generic storage format. Once this has been done, the final storage can be updated by the *Feature Processor and Storage Component*, before it finally does some cleanup, which includes locally removing the uncompressed, intermediate feature log files, as they are no longer needed.
Moreover, the storage has to allow queries and their results' combination by external (A)DMs (see requirement NFR.8), which need to import some compressed stored features for subsequent anomaly detection. Thus, the *Feature Processor and Storage Component* also serves as a gateway to those (A)DMs via another interface.
The so far intended number of such components is limited to one at a time. As suggested in Section 9.2, development of ensuring mechanisms may be conducted as future work.

Configuration Component

A second external component is called *Configuration Component*. Its main task is to initiate dynamic reconfiguration of FHS-DNA's monitoring cluster on demand of the operators. This reconfiguration includes the determination of which *Sniffer and Ex-*

tractor extracts which features. In order to allow that dynamic reconfiguration, the *Configuration Component* needs to connect to the internal *Collector* in order to locally initiate the reconfiguration from there.
The reason for this component to be an additional and especially external one therefore is to keep the design as realistic as possible. In a real-world scenario, authorized operators probably would (maybe even remotely) connect to the *Collector* in order to initiate a reconfiguration.
Due to that, our system in principle also allows multiple external *Configuration Component* typed devices, each representing an authorized operator's computer.

5.3 Important External Interfaces

Due to the design decisions as stated in Section 5.1 and the existence of external components (see Subsection 5.2.2), our system requires interfaces as depicted in Figure 5.1 to the outside of the NsUS. Those are explored in detail in the subsections below.

5.3.1 External Linkage Interface

This interface has the purpose to centrally start and stop FHS-DNA's monitoring cluster. As the *Collector* serves as a central gateway between the internal and external components, this interface is to the internal *Collector* typed device.

5.3.2 Logs Synchronization Interface

Further, the *Logs Synchronization Interface* has the purpose to allow polling of collected and intermediately saved features' log files from the *Collector* to the external *Feature Processor and Storage Component*. Thus, it also is adjacent to the internal *Collector* component. Furthermore, this interface utilizes generic and widely-used technologies as explained in Subsection 6.3.1, e.g. for transferring those features' log files via synchronization. That is done once in a predefined time interval as explained in Subsection 5.1.3 and totally autonomous in accordance to NFR.5 (from Subsection 3.2.2). Moreover, this synchronization is done no matter whether FHS-DNA is currently monitoring traffic or not.

5.3.3 (Re-)Configuration Interface

The purpose of this interface is to allow (dynamic re-)configuration of FHS-DNA's monitoring cluster. Reconfiguration is initiated on demand by the operators, each represented by an external *Configuration Component*. Hence, this interface is between the internal *Collector* component and that external *Configuration Component* typed device(s). Thus, in order to enable easy appending of additional authorized components of type *Configuration Component* to initiate reconfiguration on the *Collector*, this interface also utilizes generic and widely-used technologies as explained in Subsection 6.3.2, e.g. for remotely connecting from such a *Configuration Component* to the internal *Collector*.

Furthermore, we want to specify on a rather high level, how such a (dynamic re-)configuration works in general. For that, it has to be differentiated between a static configuration (e.g. for the initial startup of FHS-DNA's monitoring process) and a dynamic reconfiguration. Intuitively it is already clear, that a (static) configuration is the basis for a potential subsequent (dynamic) reconfiguration. Moreover, dynamic means that switching from one configuration into another only needs to be initiated by a respective script call, but not manually edited (i.e. implemented). Furthermore, it means that it can even be initiated independent of whether FHS-DNA's monitoring process is actually running or not, whereas a static (manually implemented) configuration must only be edited or changed when FHS-DNA is currently not monitoring traffic.

(Static) Configuration

A particular configuration is actually fourfold and necessarily defines:

1. Which internal devices of the NsUS are part of the monitoring cluster.
2. What type each agent of the monitoring cluster has.
3. Which interfaces each agent particularly monitors.
4. Which features each *Sniffer and Extractor* as part of the monitoring cluster extracts.

The design for such a configuration requires to statically and manually define (only while FHS-DNA is not monitoring) point one, two and three within a single file which is referenced and utilized every time FHS-DNA launches another monitoring run and called *monitoring layout* from now on. Conceptually, such a *monitoring layout* is supposed to have the structure as depicted in Figure 5.2. Of course, it is not lower nor upper bounded to four agents. Within the brackets, we can set a random name for each agent part of

```
[name-monitoring-agent-1]
type=
host=

[name-monitoring-agent-2]
type=
host=

[name-monitoring-agent-3]
type=
host=
interface=

[name-monitoring-agent-4]
type=
host=
interface=
```

Figure 5.2: Exemplary extract of the abstract and conceptual structure of a statically and manually to define *monitoring layout*

the monitoring cluster within the NsUS. The *host* section must contain its IP-address. The *type* section specifies the type and therefore the tasks of that agent. In addition, for each *Sniffer and Extractor* we further have to specify the *interface* section, which has to be the exact name of the interface to extract features from. As an additional note, all the defined IP-addresses need to be part of networks, which are further communicated to FHS-DNA in another file by simply listing them in there.

In contrast to the other points, aforementioned point four is defined in another, separate file. The definition of such a file specifying which agent extracts which particular features is called a *configuration mode* in the following. Hence, as an abstract, conceptual *configuration mode* can be seen in Figure 5.3, loading various scripts for FE to either all monitoring agents of type *Sniffer and Extractor* (line 3 - 5) or to only specific ones (line 9 - 20) is possible. Even further, in case the monitoring cluster only consists of one standalone agent fulfilling all tasks of type *Sniffer and Extractor*, *Collector* and *Coordinator* altogether, it is further possible to load only a specific subset of additional scripts for FE (line 22 - 25) besides the default ones (line 3 - 5) to that standalone agent. Consequently, the directly loaded scripts *cluster-main-3-script*, *cluster-main-1-script* and *standalone-main-1-script* transitively load the respective actual scripts for FE. However, the most important line of such a *configuration mode* is the first one, although it is a comment. It explicitly specifies the *configuration mode* and always has to be of the format *# Mode X*, with *X* being the value to identify and reference the *configuration mode*.

```
1  # Mode X
2  # Default, independent script loading.
3  @load ./testing/feature-extraction/feature-extraction-script-1
4  @load ./testing/feature-extraction/feature-extraction-script-2
5  @load ./testing/feature-extraction/feature-extraction-script-3
6
7
8  # Additional, conditional script loading
9  @if (Cluster::is_enabled())
10   # Matches on name-monitoring-agent-3.
11   @if (/name-monitoring-agent-3/ == Cluster::node)
12     # Load internal specific scripts here
13     @load decade/cluster-main-3-script
14   @endif
15   # Matches on name-monitoring-agent-4.
16   @if (/name-monitoring-agent-4/ == Cluster::node)
17     # Load internal specific scripts here
18     @load decade/cluster-main-1-script
19   @endif
20 @endif
21
22 @if (!(Cluster::is_enabled()))
23   # Load scripts for standalone mode.
24   @load decade/standalone-main-1-script
25 @endif
```

Figure 5.3: Exemplary extract of the abstract and conceptual structure of a statically and manually to define *configuration mode*

Summarizing, although all *configuration modes* as well as the *monitoring layout* need to be statically and manually defined before being used, a crucial difference between them is that it is possible to have more than one operationally available *configuration mode*, but only one combinable *monitoring layout* at a time.

(Dynamic) Reconfiguration

Hence, by further implementing and adding such transitively loading scripts and combining them in different manually and statically defined *configuration modes*, we can dynamically switch between those modes as described in the following.
Hence, dynamic reconfiguration of FHS-DNA's monitoring process means to switch between various manually and statically defined *configuration modes* by always keeping the same manually and statically defined *monitoring layout*. Moreover, dynamic means that this switching process can even be initiated while the FHS-DNA's monitoring process is running. Nevertheless, it of course is also possible if FHS-DNA is actually not monitoring any traffic. In the latter case, the *configuration mode* is just switched and remembered for the next run. In addition, *configuration modes* can of course be switched more than once (even if monitoring is currently not done), as always only the last *configuration mode* is remembered. However, if monitoring is currently done and the operators temporarily switch *configuration modes* multiple times, FHS-DNA handles each switch separately and therefore it may take some time until the final *configuration mode* is on. However, it is possible to switch directly from any *configuration mode* into every other one, so temporarily switching *configuration modes* multiple times is not a typical use case anyway (but still possible).

Moreover, this conceptual design further contributes to the required extensibility and flexibility as specified in NFR.6 and NFR.7. Furthermore, a more detailed and therefore low level description of the concrete implementation of the dynamic reconfiguration workflow is explained in Subsection 6.3.2.

5.3.4 Detection Model Interface

This last explicitly listed interface of FHS-DNA serves the purpose to allow access to the compressed, appropriately transformed and stored features in the final feature storage on the external *Feature Processor and Storage Component* (as argued in Subsections 5.1.3 and 5.1.4). Subsequent (A)DMs need to be able to query and easily combine features from there (see NFR.8 from Subsection 3.2.2) for following anomaly detection. Hence, this interface is located between that external *Feature Processor and Storage Component* and potentially many external (A)DMs.

5.4 Intermediate Log Files

Intermediate log files in general always mean the respective extracted features from their initial logging on the internal *Collector* until the transformation into the final, compressed FS format by the external *Feature Processor and Storage Component* (from that moment on it is referenced as a feature in its final, compressed FS format). Hence, an intermediate log is the file an extracted feature is initially saved in as its content. Moreover, we further differentiate between two different types of these intermediate log files as explained in the following Subsections 5.4.1 and 5.4.2. Differences certainly also affect the content to include depending on the specific feature as well as the overall format. Latter characterizes one of the two types and also depends on the specific feature to extract and in particular its complexity.

Nevertheless, a common similarity for all intermediate log files - regardless of which type they are - is their general high-level organization scheme to store feature information.
With respect to NFR.8 and NFR.9, finally stored features are required to be (efficiently) queriable. Furthermore, easy combination and analysis (e.g. during FP) is desirable. Hence, although no queries of subsequent (A)DMs will ever be done on the intermediate log files ((A)DMs will always query the finally compressed, stored features from the external *Feature Processor and Storage Component*), having the same principal organization scheme as the finally stored and queriable features allows much easier and faster transformation into that final, compressed FS format than additional reorganization would be necessary.
So, outgoing from a tabular organization scheme with rows and columns, a columnar FS format is preferable over a row-based one. That especially is a consequence resulting from our feature context, as even single columns of a feature's intermediate log file can in principle be understood as further and more basic (sub-)features, too. For better clarification what we exactly mean by this, an example is given in the following.

Considering for instance the intermediate log file which represents the feature (we call it *conn-count* in the following) for counting the number of connections from each identified source address. This log file could have structured that information as its content in two columns. The first column contains all the identified source addresses, whereas the second one contains the respective count. Hence, whenever referencing feature *conn-count* (before its transformation), we mean the appropriate intermediate log file which contains this particular feature's information as its content. However, even those two columns (in particular the first one with all the source addresses) can further be seen as basic (sub-)features themselves. For example, we could call the first column the feature

src-adr, which therefore does not need an own intermediate log file, because it is a more basic one and already included in another feature's intermediate log file.

Hence, using such a tabular organization scheme for already the intermediate log files further reduces the transformation to simply converting each one into an adequate and also tabular, but moreover compressed, columnar final FS format. In addition, querying for basic (sub-)features (i.e. columns) in such a columnar format is much easier and faster, as not plenty of distinct rows have to be (eventually selectively) queried in order to gain all entries for a specific, basic (sub-)feature. Moreover, rows in principle can grow very large (depending on the number of extracted basic (sub-)features, i.e. columns), which would further downgrade features' querying, combination and analysis performance.

5.4.1 General Logs

The first intermediate logging type is the *General Logs*. Logs of this type are mainly characterized by two aspects.
First, each specific feature - no matter whether it is extracted just from a single interface of a single *Sniffer and Extractor* device or even from multiple interfaces of various *Sniffer and Extractor* devices at the same time - always is logged in a unique and therefore common intermediate log file on the *Collector*. Hence, for each feature of type *General Logs* we get exactly one corresponding intermediate log file, which contains raw feature information extracted and combined across all participating (i.e. contributing to the generation of the common log file by extracting the appropriate feature) *Sniffer and Extractor* devices.
The second characteristic aspect refers to the raw feature information, which is contained in these *General Logs*. Raw feature information explicitly means that these logs do not contain sliding windows of the respective feature's exact evolution over time. Nevertheless, each additional extracted information (which is related to and therefore should be incorporated into this feature's log file) over time is appended in a new row at the end of the log file with the respective values of the more basic (sub-)features (which make up the actual, superior feature and its respective log file) in the appropriate columns, according to the overall tabular organization scheme as explained above. Hence, whenever the log file of this type is referenced or utilized, its content represents a snapshot of the feature as it is at the moment without focusing on the exact, previous step-by-step evolution.

Furthermore, it is important to understand that extracting, potentially combining and finally logging information from all over across the participating *Sniffer and Extractor*

devices of FHS-DNA's monitoring cluster, is totally independent of the level of abstraction of the respective feature. In other words, it does not mean that each feature which is extracted from multiple interfaces of various *Sniffer and Extractor* devices at the same time always is e.g. a NL feature. Quite the contrary, as already mentioned above the type is mainly dependent on the specific feature (but not its level of abstraction) and in particular its complexity.
Hence, regarding the complexity it is in principle the case for FHS-DNA that less complex features are of this type *General Logs*, because distributively extracted, raw feature information can more easily be combined within a single, common log file per feature on the *Collector*.

5.4.2 Interface-Specific Sliding Window Logs

On the contrary, more complex features - again in principle independent of the respective level of abstraction - typically are of the second type, which is *Interface-Specific Sliding Window Logs*. Again, logs of this type can be characterized by a few aspects.
First of all, the more complex features often already incorporate for example some statistics. Hence, in order to also be able to directly extract such features from network traffic instead of generating them during subsequent FP (e.g. Subsection 5.1.2 explains why that can be meaningful), some additional calculation (e.g. some statistical aggregation) during extraction is necessary. So, in contrast to the *General Logs*, operators could not only be interested in a snapshot of the respective, complex feature's statistic, but also in its exact evolution over time. Thus, this is another facet further increasing the complexity of the logged information. *General Logs* are per design not representing this information, as they are aggregated across all participating *Sniffer and Extractor* devices and logged within a single, common file on the *Collector* not taking care of the exact evolution of its entries as explained above. That means, the order in which each new row is appended is per design not necessarily the order, in which the information was extracted. Already that argues why a second, intermediate log type with sliding windows representing the exact evolution of the respective feature's statistic should also be supported by our system FHS-DNA. Therefore, a design solution for not only continually expanding the log randomly (that refers to the order, not the content), but rather doing that in the exact order is necessary. Conceptually designing such a solution has the advantage of being independent of a particular (available) implementation. Nevertheless, it is clear that some kind of history of that feature's statistic needs to be stored, continually expanded and logged as one sliding window (with a separating row between each two sliding windows) for every change propagated by one participating *Sniffer and Extractor*. However, in case more than one *Sniffer and Extractor* propagate

a change on that feature's history at the very same time, the history could escalate in more than it is supposed to do within one step, which could easily exceed the meaningful composition of one statistic's step and therefore compromise the original intention of these logs as described above. That means, the history would again not represent an accurate step-by-step evolution, which argues to restrict the extraction of the same feature generating a log of type *Interface-Specific Sliding Window Logs* to one interface (and therefore one *Sniffer and Extractor*) for each monitoring run of FHS-DNA. Hence, extracting the same interface-specific feature from more than one interface at the same time must be prevented in order to produce meaningful *Interface-Specific Sliding Window Logs*. This decision is even further reasoned by the implementation limitations as explained in Subsection 6.1.5. Moreover, this is also the reason why this type is called interface-specific. Nevertheless, different features which each generates a log of this type are allowed to be extracted (from the same or different interfaces) at the same time. Further, all logs of that type are still logged on the *Collector* device, but in an optimal way with additional meta information (especially from which interface they were extracted) in their log file's name. Capturing that meta information also contributes to the intention of FHS-DNA to not miss any information that could easily and additionally be saved as explained before (e.g. in Section 5.1). How this additional meta information is than used to further implement a reasonable organization for the final FS is explained in Subsection 6.2.4.

5.5 Final Feature Storage

In contrast to the intermediately stored log files on the internal *Collector*, the final FS on the external *Feature Processor and Storage Component* contains the extracted features in a transformed, compressed and generic storage format (accordingly to NFR. 8 and especially NFR.9) as described in Subsection 5.5.1. Moreover, the corresponding storage scheme is explained in Subsection 5.5.2.

5.5.1 Final Storage Format

Although all options of various technologies for final FS from Subsection 2.5 are open-source, fundamental differences appear.
PostgreSQL and SQLite are RDBMSs. Those require some predefined schema, which maps a reasonable distribution of the logging data across tables and rows [37]. So, they typically work row-based. However, basic features can typically be compared to columns of tables (as described in Section 5.4). Hence, in the context of extracted network fea-

tures, those row-based types of database systems make it hard to define a model that allows easy and fast large-scale queries concerning some basic features. Querying for specific columns (i.e. basic features) instead of rows (i.e. an entry with many different features, whereas only one basic feature / column is relevant) is much simpler. Moreover, one query for that column over all entries (e.g. for more comprehensive analysis) is also much more efficient and faster than querying over all entries and retrieving all their columns (which are not relevant except the only interesting one). Beyond that, a simple setup as well as fast read times are not really supported by PostgreSQL [37], but are required for the approach within our work.
Another crucial disadvantage of both RDBMSs and NRDBMSs is that those are just storage options, which do not offer **integrated** analysis or further processing.
Regarding Elasticsearch it almost holds the same, as it does not provide all functionality needed (e.g. transformation to a compressed format). Nevertheless, it contains integrated analysis options. However, querying and probably saving those results for further analysis may be a possibly relevant aspect and therefore should not be restricted at early stages.

Hence, the most proper solution is to transform the intermediate log files into a compressed data format. In order to further contrast that solution's possibilities to the other options disadvantages, the final FS data format also has to be generic, so that various (A)DMs can properly utilize it. Moreover, it would be best, if that storage format further is a columnar one, so that easy and efficient querying on basic features (i.e. columns) is supported as explained above. In that case, NFR.8 and NFR.9 can be perfectly complied with. Nevertheless, storing those transformed logs (and their contained features) in such a data format requires to manually define an appropriate storage scheme as explained below and in Subsection 6.2.4.

5.5.2 Final Storage Scheme

Conceptually, our storage is a hierarchic directory structure directly in the file system of the *Feature Processor and Storage Component*. At the lowest level of that hierarchy it stores the transformed, generic, compressed and columnar feature files. Below, we state the main reasons, why we decided for such a final FS scheme.

First, the overall goal is to have an appropriate storage scheme, from which various (A)DMs should be able to query and combine results (according to NFR.8 and NFR.9). So, with a hierarchic directory structure we already pre-cluster the transformed files according to their contained features. That way, (A)DMs already get a hint, which files may typically be combined for more comprehensive analysis. So, regarding easy and

efficient querying and combination of results, this structure yields particular benefits.
First of all, a join of multiple files (maybe even across different levels) is in principle not just always possible, e.g. with Apache Spark SQL [62] [63] [64], as long as an index for a join is available in the respective files, but also even cheaper for very likely or typical correlations. That is, as especially files within a certain, common categorization (i.e. directory) are more likely to be combined for analysis, as they contain similar information. Consequently, additional queries for preselecting various transformed files before actually combining them for analysis are not needed, as our structure allows to do that with only one query on the respective superior directory to receive all contained files.
Moreover, in case of less typical correlations, no additional inefficiencies are coercively induced, as selecting only single files from various levels and directories is still possible and maybe even accelerated due to easier navigation in a well-structured hierarchy.
So, this structure is in no way restrictive to any case of querying or meaningful combination of transformed files by (A)DMs. Furthermore, it already provides a reasonable categorization on certain levels, which allows cheaper and more efficient associations in most cases. That especially contributes to compliance with NFR.8 and NFR.9 and is further especially related with the third reason below.

Second, our hierarchic structure is meant to be relatively stable. The peak of it in principle is much less volatile (and broad) than the basis, which perfectly fulfills our intention of allowing easy navigation (i.e. access) for (A)DMs. The top level always is the single directory containing the entire FS. The second level always contains maximal two directories - one associated with the last (i.e. maybe current) live monitoring run and one for historic data. The more we descend in this pyramid, the more growth is possible. For instance, depending on the traffic load, the number of additional interfaces (which could e.g. be another level in the pyramid) is probably hardly growing, whereas the size of the transformed files on the bottom level definitely increases as well as their number with each additional monitoring run does.

Third, the increasing number of smaller files on the bottom level of this pyramid also is another reason for that storage scheme. Putting all information and all features into e.g. a single file would indeed eliminate the need for joins, but make subsequent anomaly detection nevertheless much less efficient.
First, the size of the file to import is much bigger than only importing a relevant subset of features in respective files already pre-categorized in our pyramid.
Second, the number of rows in a single file would rapidly grow, whereas many columns for each row would stay empty, as not every row necessarily contains an entry for each column. Hence, for efficient handling of those logged features, subsequent (A)DMs are better off with our storage structure of **more smaller**, but **less bigger** transformed,

compressed files. However, the size dimension only refers to rows. With respect to columns, our files contain as much reasonable data as possible to reduce information loss. Thus, reasonable means non-empty entries, which again closes the circle in comparison to a single file with many empty columns for each row. So, making (efficient) joins on much smaller (subset) data is still supposed to be way less expensive than always having to import one very large file.

Fourth, by already pre-clustering the files in a hierarchic directory structure on disk, the (A)DMs also get a better overview on which data is available and therefore possible to query or combine. Hence, that makes the storage both more clearly arranged as well as deducible with respect to missing features maybe being extracted in future for generating specifically desired insights by combination with already existing files.

Fifth, our pyramid can clearly separate between live data and historic data. Live data always gets handled with special care, as new live data requires the old one to be deleted beforehand. Hence, old "live" data gets not directly inserted to the historic branch (on the second level as mentioned above) of the storage, as it may still grow or change. Therefore, this data is not considered as historic before the actual monitoring run is finished. Therefore, handling FS updates requires special care (see Subsection 6.2.5).

Finally, besides the mere, final FS in a structured way as explained above, this design moreover contributes to our intention to reduce information loss to a reasonable minimum whenever possible as already mentioned before. This is achieved, as the levels of the hierarchic directory structure may also incorporate (and therefore log) some additional meta information (e.g. related levels of abstraction, interfaces, dates and timestamps of files) not directly related to the originally extracted features themselves. How this is actually implemented is explained in detail in Subsection 6.2.4.

Chapter 6

Implementation

In this chapter we explain our implementation of the previously introduced high-level design from Chapter 5. The implementation presented serves as a proof of concept (PoC) and is available on [65].
It is important to note that only exemplary extracts of FHS-DNA are described in detail in the following. Hence, for a more comprehensive review of code and its considered implementation concepts we refer to appropriate scripts in the codebase in [65]. For their implementation languages, our scripts are mainly implemented in the *Bro scripting language*[1] for FE and *Bash*[2] as well as *Python*[3] scripts for everything else.

6.1 Feature Extraction

The first section is about live traffic monitoring and extracting features from it.

6.1.1 Bro as Tool for Traffic Monitoring and Feature Extraction

After having compared a huge variety of potential options such as the ones described in Section 2.4, we decided to use *The Bro Network Security Monitor* due to following main reasons.

---

[1] https://www.bro.org/sphinx/script-reference/index.html

[2] https://www.gnu.org/software/bash/

[3] https://www.python.org/

First of all, *Bro* by default provides a monitoring cluster layout composed of multiple *Bro* instances being coordinated [33] [34]. As the FH approach needs to be a decentralized one as monitoring capabilities need to directly be deployed on the agents of the NsUS, such a *Bro* monitoring cluster perfectly suits that intention. Supporting that option natively (e.g. *Bro's node.cfg* file exactly specifies such a *monitoring layout* as presented in Figure 5.2) and additionally providing mechanisms for coordination and central logging [33] prevents otherwise most likely occurring issues regarding instantiation and coordination of such a cluster. Moreover, even a central cluster management interface - named *BroControl* - for comfortable control and access of this cluster is provided by Bro by default [28]. Moreover, in *Bro* cluster mode *Bro* and *BroControl* only need to be installed on the *manager* [66], which additionally preserves disk space on the other nodes. In contrast, tools like Snort and others do not natively provide these options.

Moreover, a central logging mechanism is not only already integrated into the cluster mode, but also abstracts a little bit and therefore is rather high-level [28]. In addition, the central logging architecture supports the fact of "data centric architectures being incorporated into modern aircraft" [67].

Another crucial reason for *Bro* is its Turing-complete and domain-specific scripting language [28]. This opens up much broader implementation opportunities for FE than for instance just defining rules for signature-based detections as possible in Snort, because "arbitrary analysis tasks" [28] for monitored traffic can be implemented. That particularly means that *Bro* is not restricted to work signature-based, but rather also supports anomaly-based analysis and extraction [28]. Hence, that better fits our work's context, which primarily is network anomaly detection. As this especially requires passive network monitoring for FE and not deep-packet inspection nor packet manipulation, that makes *Bro* also more appropriate than e.g. Wireshark and Scapy.

6.1.2 Bro "Normal" Cluster for Decentralized On-Agent Monitoring and Feature Extraction

As explained in Subsection 2.4.3, there are basically two different cluster modes of *Bro* provided by default. Below we argue the main reasons why we decided to use the *"normal" cluster* instead of the deep cluster. Those include, but are not limited to:

1. The deep cluster concept is still under development. At the current state, implementation and adaption of some functionality already integrated and usable in *"normal" cluster* is not yet finished for a deep cluster setup (e.g. the sumstats framework) [31].

2. Focus of a deep cluster is on very big setups. The intention is to "bring monitoring from the edge of the monitored network into its depth" [31]. Even a group of cars or aircraft interconnected with each other is too small in comparison to such an intended field of application as described in [31].

3. A deep cluster is based on fundamental and more comprehensive requirements as for instance, which components need to be connected. In detail, the communication system requires each deep cluster node being able to communicate with each other. Therefore, the created peer-to-peer overlay network and the publish-subscribe system lead to a bidirectional communication system for all nodes. One reason for such an overlay network is, that all *worker* nodes and sub-clusters within the deep cluster can be fed with rules, information and reactions from the entire network. In contrast, the *"normal" cluster* requires much less communication interfaces, as e.g. not every *worker* needs to communicate with all other *worker* nodes. Hence, the *"normal" cluster* is totally fine for FHS-DNA's purpose, as it does not require such a high interconnectivity.

4. Moreover, such a peer-to-peer overlay network with its publish-subscribe communication system is not really appropriate for our system. Besides the fact that it would be very hard to ensure this kind of connectivity between e.g. cars, they even should not be connected that way. In contrast, a centralized approach, where e.g. all cars write to a central logging hub (e.g. a *logger*) after FE and FP, is more realistic and perfectly matched by a *"normal" cluster*.

Hence, this is why we decided for the *"normal" cluster* of *Bro* and against the deep cluster concept. An even more detailed comparison is given in [31].

6.1.3 Adaption of Bro "Normal" Cluster to Internal Components

Hence, the final implementation of a coordinated, decentralized, on-agent monitoring cluster is done with a *Bro "normal" cluster*, which in the rest of this chapter is simply referenced as *Bro cluster*, if not explicitly stated otherwise. Within this subsection, we give an overview on how it is adapted and mapped to our system's architecture and its internal components' types and tasks.

Sniffer and Extractor

Each device with monitoring type *Sniffer and Extractor* is of type *worker* in the integrated *Bro cluster* of our system. This is possible, as the *worker* typically fulfills

the same main tasks (see Paragraph 2.4.3) as required by a *Sniffer and Extractor* (see Subsection 5.2.1).

Collector

The internal *Collector* typed device is represented by the *manager* node of a *Bro cluster*. Outgoing from its main purpose of gathering all the logs of the distributively extracted features, this type could also be simulated by a *logger* node of a *Bro cluster* at a first glance. However, as one node of the internal monitoring cluster also needs to serve as a gateway for a couple of interfaces as defined in Section 5.3, this type must be implemented as a *manager* node. Hence, starting and stopping the *Bro cluster* is initiated on the *manager* via the *External Linkage Interface*, which is implemented by the default *BroControl* interface.

Coordinator

Finally, each device with monitoring type *Coordinator* is of type *proxy* in the integrated *Bro cluster* of our system. Again, this is possible as the *proxy* typically fulfills the same main tasks (see Paragraph 2.4.3) as required by a *Coordinator* (see Subsection 5.2.1).

6.1.4 Manually Implemented and Extracted Features

In order to keep information loss while FE at a reasonable minimum, FHS-DNA is supposed to extract features and as much relevant additional meta information as possible from multiple levels of abstraction (PL, CL, NL). For that, in addition to the features extracted by *Bro* by default (see [68]), Table 6.1 gives an overview of all comprehensive, additionally and manually implemented features that are handled in the PoC implementation of FHS-DNA. So, it is important to understand that those comprehensive features may even contain more basic ones as additionally logged meta information. Consequently, all together make up one possible and reasonable feature subset.
As already explained in Section 2.3, features originally deduced from different datasets differ in their relevance depending on the specific aspects of their deduction and corresponding valuation. In order to consider as actual and meaningful insights as possible on those aspects, we took Table 2.1 based on [12, p. 8] - showing the eleven highest ranked features from UNSW-NB15 and NSLKDD according to a specific algorithm as proposed in [12] - as a foundation for our reduced subset composition. Selection criteria include considerations to extract features from every vertical level of abstraction (which is why we additionally also personally constructed some PL features not respected in

Table 2.1) and with different complexities. Furthermore, as the UNSW-NB15 dataset is a more recent, reliable and valid one, we decided to include (i.e. marked green in Table 2.1) twice as much features from it than from the NSLKDD dataset.

In the first column from Table 6.1, we see the common name of each feature. Below, we give a short description on what each feature stands for:

- *land*: Indicates a land-attack. If a connection's IP-addresses and port numbers from source and destination are the same, then the connection related entry has value 1, otherwise 0 (based on [69]).
- *synack*: Calculates the TCP connection setup time. This is the time between the SYN packet from the originator and the respective SYN_ACK packet from the responder in a TCP handshake (based on [3]).
- *ct_src_ltm*: For each of the 100 last connections anytime it counts the number of connections that contain the same source address (based on [3]).
- *ct_srv_dst*: For each of the 100 last connections anytime it counts the number of connections that contain the same service and destination address (based on [3]).
- *ct_dst_sport_ltm*: For each of the 100 last connections anytime it counts the number of connections that contain the same destination address and source port (based on [3]).
- *count*: Whenever a connection is identified, it counts the number of connections to the same destination host as the current connection in the next two seconds (based on [69]).
- *raw_layer2_packet_header*: All available layer two header information of a packet.
- *raw_IPv4_packet_header*: All available IPv4 header information of a packet.
- *raw_IPv6_packet_header*: All available IPv6 header information of a packet.
- *raw_TCP_packet_header*: All available TCP header information of a packet.
- *raw_UDP_packet_header*: All available UDP header information of a packet.
- *raw_ICMP_packet_header*: All available ICMP header information of a packet.

The second column lists the script implementing the appropriate feature. All scripts are implemented in the *Bro scripting language* [30]. In the third column, we either list the dataset from which the respective feature is deduced or at least referenced in or - in case the feature was personally constructed - we write *Personal*. The fourth column associates the respective feature with the vertical level of abstraction (PL, CL, NL) as

described in Subsection 2.3.2.
Furthermore, the general layout of and contained information in the respectively generated intermediate *Bro* logs can be seen in the appendix in Subsection A.1.1.

Table 6.1: Manually implemented features for potential extraction by FHS-DNA

Feature Name	**Script for Extraction**	**Origin**	**Level**
land	feature-extraction-NSLKDD-7.bro	NSLKDD	CL
synack	feature-extraction-UNSWNB15-34.bro	UNSW-NB15	CL
ct_src_ltm	feature-extraction-UNSWNB15-44.bro[1]	UNSW-NB15	NL
ct_srv_dst	feature-extraction-UNSWNB15-42.bro	UNSW-NB15	NL
ct_dst_sport_ltm	feature-extraction-UNSWNB15-46.bro	UNSW-NB15	NL
count	feature-extraction-NSLKDD-23.bro	NSLKDD	NL
raw_layer2_packet_header	feature-extraction-P-1.bro	Personal	PL
raw_IPv4_packet_header	feature-extraction-P-2.bro	Personal	PL
raw_IPv6_packet_header	feature-extraction-P-3.bro	Personal	PL
raw_TCP_packet_header	feature-extraction-P-4.bro	Personal	PL
raw_UDP_packet_header	feature-extraction-P-5.bro	Personal	PL
raw_ICMP_packet_header	feature-extraction-P-6.bro	Personal	PL

Packet Level Features

Table 6.1 lists six manually implemented FE scripts for PL features. They all extract different packet header information. As an example, we shortly constitute some implementation details of feature *raw_layer2_packet_header*, which logs layer two[2] header information. The structural idea is as follows.
Whenever *Bro's* event engine throws an event named *raw_packet*, a packet with a valid link layer header is seen by *Bro*. Within the event handling it first of all is checked which data link layer header fields are available for that packet. This is done by declaring a vector of Boolean variables, each representing the availability of various - in principle optional - layer two header fields. So, this vector is initialized with value *false* for all header fields at the beginning. Each header field identified available later on sets the respective Boolean entry of the vector to *true*. Next, all available header fields of layer two that could be found are extracted and logged. Afterwards, this packet is worked

[1] Orange marked scripts should only be loaded to one specific interface per monitoring run!

[2] Refers to the data link layer in the ISO/OSI model

```
1  event connection_established (c:connection) {
2      if (c.source.sent_SYN_packet() == true) {
3          log("synack", c.source.addr, c.setup_time);
4      }
5  }
```

Figure 6.1: Pseudo code fragment for extraction of feature *synack*

off and the next is ready to be handled whenever the *Bro* event engine sees a new raw packet. On overview of which data link layer header fields are potentially extracted and logged is given in the appendix in Table A.7.

In contrast to that feature's extraction, all other manually implemented PL features from Table 6.1 only extract and log mandatory header fields as listed in the respective tables in the appendix (see Subsection A.1.1).

Connection Level Features

Regarding CL features, Table 6.1 lists two different ones. Both their extraction logic is shortly explained in the following with respective pseudo code extracts.
The first one is feature *synack*. It extracts and logs a TCP connection's source host and setup time for every connection, for which a SYN_ACK packet from the responder is seen during the TCP handshake. A corresponding pseudo code extract is depicted in Figure 6.1 and referenced in the following.
Every *Bro worker* node throws an event named *connection_established* (line 1) whenever it sees a SYN_ACK packet from the responder in a TCP handshake. In order this connection was indeed correctly established, a SYN packet must have already been sent by the connection's source host (line 2). Hence, only if that is the case, the connection's source host and TCP setup time for feature *synack* is extracted and logged to the corresponding intermediate *Bro* log file (line 3).

The second manually implemented CL feature is *land*. It extracts and logs every TCP, UDP and ICMP connection's start time, source host and port, destination host and port and the value (*1* or *0* as specified above) for the *land* feature. A corresponding pseudo code extract is depicted in Figure 6.2 and referenced in the following.
Each *Bro worker* node throws an event named *new_connection* (line 1) whenever it sees a first packet of a so far unknown TCP, UDP or ICMP flow. If its source and destination hosts as well as the ports are equal (line 2), it extracts and logs all the aforementioned information to the corresponding intermediate *Bro* log file by specifying the value for the *land* feature with *1* (line 3). Otherwise, also everything is extracted and logged, but *land* has value *0* instead (line 6).

```
1  event new_connection (c:connection) {
2     if ((c.source.addr == c.dest.addr) && (c.source.port == c.dest.port)) {
3        log("land", c.start_time, c.source.addr, c.dest.addr, c.source.port, c.dest.port, 1);
4     }
5     else {
6        log("land", c.start_time, c.source.addr, c.dest.addr, c.source.port, c.dest.port, 0);
7     }
8  }
```

Figure 6.2: Pseudo code fragment for extraction of feature *land*

Network Level Features

For the third level of abstraction, Table 6.1 contains four NL features.
As an example, we present some implementation details of feature *ct_dst_sport_ltm*, which logs for each of the 100 last connections (at any time) the statistical step-by-step evolution of the count of connections with the same source port and the same destination address. As its FE script already directly extracts, calculates and logs some statistical aggregations as described above, further subsequent statistical analysis (i.e. as part of FP) for the creation of this feature is not needed any more. Hence, that feature is a good example for a more comprehensive (with respect to its calculation during direct FE) NL feature, as its general, abstract implementation logic moreover is also comparable to NL features *ct_src_ltm* and *ct_srv_dst*. A corresponding pseudo code extract is depicted in Figure 6.3 and referenced in the following.
First of all, three global variables are declared (line 1 - 3), which are fundamental for the subsequent implementation of feature *ct_dst_sport_ltm*. These variables are set globally, as they have to persist throughout one entire monitoring process.
The first one is named *count_table* (line 1). It is a map with its key being the pair of the current connection's destination address and source port and its value being the count of connections with the same key among the last 100 ones. Hence, this map always contains the actual status (i.e. snapshot) of this statistical feature and therefore represents one sliding window of the corresponding *Bro* log file at any time. Moreover, it is important to understand that it is filled up based on the content of the second variable, which is *pair_table* (line 2). It is an array of the last 100 connections' pairs of destination address and source port. Hence, the number of same pairs in *pair_table* is the corresponding count value in *count_table*. Limiting *pair_table* to a maximal size of 100 entries is ensured by appropriate handling of its index for circular fill-up, which is the third global variable named *index_pair_table* initially set to *0* (line 3).
Whenever a *Bro worker* raises an event *log_conn* (line 5), this *index_pair_table* is checked at the very beginning (line 6). That means, it should never exceed value *99*. Then the *new_pair* for the actually handled connection is extracted (line 7). In case it is

```
1  global count_table = [pair(destAddr, srcPort); count];
2  global pair_table = [pair(destAddr, srcPort)];
3  global index_pair_table = 0;
4
5  event log_conn (c:Conn::Info) {
6     check_index(index_pair_table);
7     new_pair = (c.dest.addr; c.src.port);
8     if (pair_table.len() < 100) {
9        pair_table.update(new_pair);
10       count_table.update(new_pair);
11    }
12    else {
13       old_pair = pair_table[index_pair_table];
14       pair_table.update(new_pair);
15       if (new_pair != old_pair) {
16          count_table.update(new_pair);
17          count_table.cleanup(old_pair);
18       }
19    }
20    for (pair in count_table) {
21       log("ct_dst_sport_ltm", pair.destAddr, pair.srcPort, count_table[pair]);
22    }
23    log("ct_dst_sport_ltm", , , );
24 }
```

Figure 6.3: Pseudo code fragment for extraction of feature *ct_dst_sport_ltm*

under the first 100 handled ones (line 8 - 11), *pair_table* gets updated (line 9) by simply inserting the *new_pair* to *pair_table* at the current index, before *index_pair_table* is incremented. Furthermore, the *count_table* gets also updated (line 10) based on the new *pair_table*, which means incrementing the count of *new_pair* (eventually *new_pair* was even needed to be inserted to *count_table* before). In case *pair_table* already has been completely filled up (line 12 - 19), we intermediately save the *old_pair* (line 13), as this one will be overwritten by inserting the *new_pair* in the update of *pair_table* (line 14), but will still be needed subsequently for also updating *count_table*. However, only in case *new_pair* is not equal to *old_pair* (line 15 - 18), we further need to update the statistics of our feature saved in *count_table*. That includes increasing the count for key *new_pair* (and potentially even inserting it first) (line 16), as well as the decreasing the count for *old_pair* (and potentially even deleting it) (line 17).
Finally, this new sliding window (i.e. statistical evolution's snapshot) is logged by writing each key and its corresponding count value in the *count_table* to the appropriate, intermediate *Bro* log file (line 20 - 22), before a separating line is logged, too (line 23).

6.1.5 Intermediate Log Files

In the following, implementation details of different types of log files as described in Section 5.4 are explained. As we use the default logging mechanism of *Bro*, all log files in principle follow the layout of *Bro*. An overview of what information is explicitly logged in manually constructed log files is given in the appendix in Subsection A.1.1. Besides those manually constructed and implemented ones, default intermediate log files are also considered by FHS-DNA. For a detailed description of those we refer to [68].

General Logs

General intermediate logs only represent direct feature (and maybe some meta) information as it currently is without representing the exact evolution (i.e. no sliding windows) of it. That makes the implementation of such a snapshot log less complicated than it is for the interface-specific sliding window ones.
The default logging mechanism of *Bro* is utilized and we only have to specify, which columns the respective log file should have and which information is inserted appropriately, as it already handles coordination among the participating *Bro workers* in order to aggregate all information into a single log file. Depending on the specific feature, the information to insert requires some additional calculations up-front (e.g. calculating some duration values), before finally being logged. Logging for that type of intermediate log files then means to append all the information in one row (with the respective values in the appropriate columns) whenever a participating *Bro worker* extracted (and maybe further calculated) the values as described in Subsection 5.4.1. Hence, also no sliding windows' separating lines need to be logged. An example for a feature generating such *General Logs* in our PoC implementation is *synack* from Table 6.1. An exemplary fragment of such a *General Log* for that feature is represented by Figure A.1 in the appendix in Subsection A.1.2.

Interface-Specific Sliding Window Logs

In contrast, features generating sliding window logs in my PoC implementation are *ct_srv_dst*, *ct_src_ltm* and *ct_dst_sport_ltm* (see Table 6.1). As described in Subsection 6.1.4, these features' log files are filled up based on some global variables initialized and handled in *Bro* events. The variables contain some statistics, which are calculated directly during FE. Each single change of that statistic represents one sliding window in the *Bro* log. That makes those features and their logs much more comprehensive and complicated to implement and also is the reason, why each such feature should only be extracted from one network interface of a specific host in one monitoring run

of FHS-DNA. In addition to the high-level argumentation as explained in Subsection 5.4.2, implementation restrictions underlining the need of this constraint are presented in the following.

Outgoing from the intention to eliminate the aforementioned constraint, we identify two theoretical ways to do so below. However, for each of those options, we further argue some limitations within *Bro*, which finally explain their corresponding option not to work in practice.

1. Control access on those global variables (of each feature) and / or the feature's unique, common log file.
2. Generate more than one log file for the same feature (e.g. one for each network interface that same feature has been extracted from).

Regarding the first option, although synchronizing variables in *Bro* is possible, it is still vulnerable to race conditions. "We implemented synchronized tables by propagating changes to the data in terms of descriptions of the operations to perform on the data rather than the full (and probably mostly unmodified) data itself [...]. This can in some circumstances however lead to race conditions. Avoiding them would require mutually-exclusive data operations [...], but this would violate Bro's real-time processing constraints due to having to wait for access before performing an operation" [70]. However, in order to represent the exact stepwise evolution of the statistics, race conditions are required to be excluded for sure. Otherwise, statistics may end up falsified. Regarding the second option, that seems to be possible at a first glance. However, we would also further need the three global variables for each log file (of the same feature) in order to finally write the logs based on those variables' content as described in Subsection 6.1.4. However, FHS-DNA should for instance be (dynamically re-)configurable and extensible with respect to extracted features (see NFR.6 and NFR.7 from Subsection 3.2.2). That means, in general different features should be allowed to be extracted from different interfaces. Hence, that would explicitly require us to dynamically (i.e. within the event handling) declare (e.g. depending on the names of the *Bro* nodes and their interfaces), initialize and fill those global variables for each interface. However, exactly this is not allowed in *Bro scripting language*, as declarations "cannot occur within a function, hook, or event handler" [71].
Concluding, the only reasonable solution is to extract the same feature generating this type of log file on only one specific interface for each monitoring run. Nevertheless, our system is robust enough to cope with violations of this constraint, although generated logs may be falsified in those cases.

Extracted features generating these *Interface-Specific Sliding Window Logs* in our PoC implementation only are *ct_srv_dst*, *ct_src_ltm* and *ct_dst_sport_ltm* from Table 6.1. All others generate *General Logs*. An example for such a manually constructed *Interface-Specific Sliding Window Log* in our PoC implementation refers to feature *ct_dst_sport_ltm* and is represented by Figure A.2 in the appendix in Subsection A.1.2. The sliding windows can clearly be identified and exactly illustrate the stepwise evolution of that feature's statistic as desired.

6.2 Feature Processing and Storage

As stated in Subsections 5.1.2 and 5.1.3, we decided for compromised design solutions regarding FP and FS. Nevertheless, they work together on an external, not resource-constrained and compounded *Feature Processor and Storage Component* as explained in Subsections 5.1.4 and 5.2.2. Crucial implementation details regarding this component are explained in the following.

6.2.1 BAT as Tools for Feature Processing and Storage Preparation

For FP, we decided to use *BAT* and integrate it into our system FHS-DNA. Main reasons for that decision include that *BAT* according to [46]:

1. is compatible to and directly works on *Bro* logs,
2. enables processing of real-time data,
3. offers a comprehensive set of integrated analysis and processing options,
4. enables potentially offloading analysis and processing tasks from the NsUS,
5. supports various generic data formats as transformation output (e.g. *Parquet*, python dictionaries and Panda data frames),
6. offers support for combination with various ML-based anomaly detection and storage models (e.g. Spark),
7. provides a stable release with documentation.

BAT is only installed on the external *Feature Processor and Storage Component*. Automatic installation is implemented in the Bash script *BAT-Installation.sh*. Moreover, as that component is a compounded one, *BAT* may be utilized both for FP of intermediate *Bro* log files (containing the features) and for preparation of final FS by transforming

them into an appropriate, compressed and generic storage format as described in Subsection 5.5.1.

6.2.2 Parquet as Final Feature Storage Format

Based on the decision to use a generic, compressed and columnar data format for final FS (see Subsection 5.5.1) in combination with utilizing *BAT* for FP and final FS preparation, we decided to use *Parquet* as our final storage format. Only most important reasons pro *Parquet* are listed below.

1. *Parquet* is a columnar storage format [47] [72], which is especially suitable for our feature context. Querying on columns (i.e. features) instead of single rows is beneficial, as this way basic features can e.g. easily and more efficiently be analyzed, combined and queried than selectively querying many single rows for conclusions about one feature (in compliance with NFR.8 and NFR.9). Further reasoning for a columnar FS format is also given in Section 5.4 and Subsection 5.5.1.
2. *Parquet* supports compression [48] [73], which helps to save resources with respect to disk space (in compliance to NFR.9).
3. *Parquet* is a self-describing and language-independent storage format [47].
4. *Parquet* supports high "extensibility of storing multiple types of data in column data" [74]. This is important, as different features' data has different types.
5. *Parquet* allows generic utilization by various subsequent (A)DMs, such as e.g. Spark or Presto[1] (in compliance to NFR.3 and NFR.8). Moreover, *Parquet* format is "available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language" [72].
6. *Parquet* is very performant in multiple aspects (in compliance to NFR.9). It is very efficient for large-scale queries on huge amounts of data as well as for performing aggregation operations (e.g. average) on multiple columns [74]. So, it optimizes query performance [47] [75]. All aspects are very relevant for our work, as our system may extract huge amounts of network data on the one hand and allow some statistical aggregations on them on the other hand.

[1] https://prestodb.io/

7. *Parquet* minimizes I/O [47] [75] and allows fast read times of subsets of data (in compliance to NFR.9) as datasets are partitioned horizontally and vertically [75].
8. *Parquet* files can easily be queried and combined in many meaningful ways with e.g. Apache Spark SQL[1] [62] [63] [64] and therefore supports compliance to NFR.8.
9. *BAT* supports transformation of *Bro* logs to *Parquet* format [46] [76].

On the contrary, *Parquet* for instance is write intensive [73] [75]. However, as our external *Feature Processor and Storage Component* is per assumption not resource-constrained, this is negligible. Summarizing, *Parquet* is fully compliant to many related requirements (see Section 3.2) and basic design decisions (see Subsection 5.5.1). Even further, it provides a lot of important advantages in addition as listed above.

6.2.3 Intermediate Bro Logs' Transformation to Parquet Format

Thus, our external *Feature Processor and Storage Component* needs to transform intermediate *Bro* logs into *Parquet* format as part of its processing and final storage preparation tasks. The entire transformation is automatically executed (in compliance to NFR.5) every time directly after synchronization of intermediate *Bro* logs from the internal *manager* to the external *Feature Processor and Storage Component*. This happens once in a predefined time interval (every five minutes in the PoC implementation) by a *cron*[2] job specified in the *crontab*[3] file on that component. Nevertheless, FHS-DNA also allows anytime manual synchronization and transformation. Latter is done with *BAT* and making use of its function *log_to_parquet*. A summarizing description of the implementation is given in the following. However, for a more detailed review, we want to refer to *Bro-To-Parquet.py* in the codebase in [65].

First of all, synchronized intermediate *Bro* logs are checked for their filename matching a regular expression containing one of the prefixes listed in Table 6.2. Only files with their name matching one of those prefixes get further processed, as those *Bro* logs typically are expected to have a format matching the specific requirements of the subsequently applied method *log_to_parquet* of *BAT*. Nevertheless, in order to make FHS-DNA robust against unforeseen deviations from that expected format, it is additionally and

[1] https://spark.apache.org/sql/

[2] http://man7.org/linux/man-pages/man8/cron.8.html

[3] http://man7.org/linux/man-pages/man5/crontab.5.html

explicitly checked for each of those files with a matching name, whether it contains a line that starts with *#fields* or not. Only in case it does, it finally gets transformed.

Table 6.2: Filename prefixes whitelisting respective intermediate *Bro* logs for potential transformation to *Parquet* format

File Label		
capture_loss	http	smtp
communication	intel	software
conn	known_hosts	ssh
dhcp	known_services	ssl
dns	notice	traceroute
feature-extraction	packet_filter	weird
files	reporter	x509

6.2.4 Final Storage Scheme

Once, all the logs are transformed to *Parquet* format, the final FS can be updated. However, in order to understand how its update works, we want to describe its abstract implementation of the underlying scheme described in Subsection 5.5.2 first.

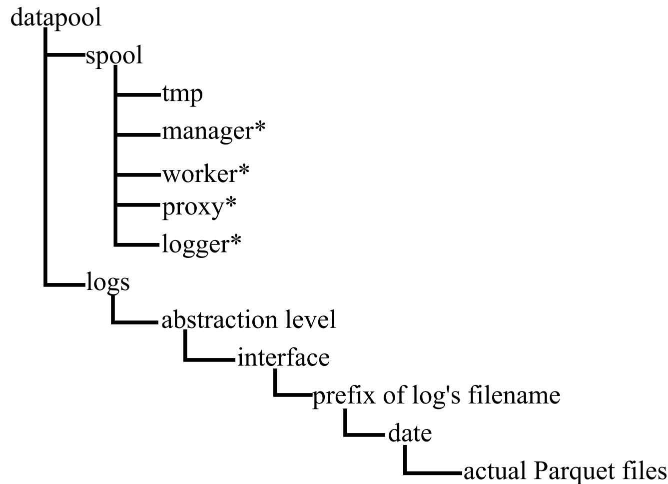

Figure 6.4: Abstract implementation of the hierarchic directory structure of the final on-disk feature storage scheme

Conceptually, our storage is a hierarchic directory structure directly in the file system of the *Feature Processor and Storage Component* (as described in Subsection 5.5.2). It consists of six layers or levels. These are depicted in Figure 6.4 and explained in more detail below.

The top layer is the overall storage's directory named *datapool*. It contains the entire final FS with all its *Parquet* files in its structured subdirectories.
On the second level, we have two branches named *logs* and *spool*. The *logs* branch contains all *Parquet* files in respective subdirectories representing historic features. Historic in that context means, that it is not related to the lastly (maybe currently) live monitored traffic. In contrast, the *spool* directory contains all so far synchronized and transformed *Parquet* files containing features extracted from the lastly (maybe currently) monitored live traffic.
On the third level, the *spool* directory can have multiple branches, which depend on the exact layout of the *Bro cluster* and its types as specified in the *node.cfg* file. Each of those branches is the deepest level in the *spool* directory.
More interesting is the sub-structure of the historic data in the *logs* branch. Therefore, in the following only the *logs* branch is further explained.
On the third level, *Parquet* files are categorized according to their features' vertical abstraction level (PL, CL, NL). Hence, this level is threefold, namely *packet*, *connection* and *network*.
On the fourth level, the *Parquet* files are categorized according to the respective network interfaces, from which their features where extracted. Thus, for *Interface-Specific Sliding Window Logs* the name of the interface is the name of the respective subdirectory (e.g. *h3-eth0*). However, for *General Logs* containing aggregated feature information across all *worker* nodes, the directory on this layer is named *all*.
On layer five, each of the directories from level four is compartmentalized into many different branches according to the *Parquet* files' names' prefixes (see Table 6.2). Hence, level five directories indicate, which *Parquet* files are contained in which directory (e.g. all *http*.parquet* files in directory *http*). So, possible directory names are respectively listed in Table 6.2.
On the last - the sixth - level, each of the directories from layer five is divided into different dates, indicating on which the corresponding, intermediate *Bro* logs were generated. Hence, this level specifies, when the respective features where extracted (e.g. *2018-07-19*).
Finally, the actual *Parquet* files are located within these differently categorized folders. In addition, their name contains a timestamp, indicating from which specific monitoring run these features result.

Concluding, the implementation of the hierarchic on-disk FS additionally logs some meta information not directly contained in the features themselves as explained in Subsection 5.5.2. Examples for that are the concrete and dynamic instantiations of the statically

defined levels three to seven of the pyramid (e.g. related levels of abstraction, interfaces, dates and timestamps).

6.2.5 Final Storage Update

As explained in Subsections 5.5.2 and 6.2.4, the storage scheme is a relatively stable pyramid. Updating it is mainly implemented in *Update-Datapool.py* script and handled with special care as argued in Subsection 5.5.2. An update is automatically executed (in compliance with NFR.5) by a *cron* job of the *crontab* file on the *Feature Processor and Storage Component* every time directly after transformation of *Bro* logs into the final FS *Parquet* format (i.e. every five minutes in our PoC implementation). However, FHS-DNA also allows anytime manual storage update.

Before the actual storage update, the directory */opt/bro/spool/* containing "old" live data is removed, so that subsequently only the actual live data is inserted to that directory of the storage. This is implemented in file *Synchronize-Transform-Update.sh.* Afterwards, the actual update begins and works as described below referring to Figure 6.5, which represents an extract of the implementation in pseudo code. Each synchronized directory with transformed, intermediate logs (pre-classified by *Bro* into historic and live data) is handled separately by calling the method as defined in line 1. For each file of the respective folder it is checked, whether it still may underlie changes by the monitoring run (i.e. is live data) (line 15 - 17) or not (i.e. is historic data) (line 3 - 14). In the first case, the file is directly copied into the respective directory of the storage pyramid (line 16). In the latter case, a local array of strings is declared, with the intention of each string representing the concrete instantiation of a level in the storage pyramid (line 4). The already set value *logs* on the last position indicates the superior branch as the historic one, as the respective directory on the second level of the hierarchic storage scheme is named *logs* as explained in Subsection 6.2.4. Afterwards, the other strings are gradually extracted for each currently handled file and saved in the respective array, before finally inserting the *Parquet* file (depending on the values in *levels*) into the appropriate subdirectory of the final storage (line 13). In case no file has been of that category yet, the subdirectory needs to be created beforehand. However, special care has to be taken on the extraction of the string representing the value for the *interface* level. As *General Logs* are aggregated across all monitored interfaces (see Section 5.4), the string has value *all* in that case (line 10). In the other case (i.e. the original *Bro* log is of type *Interface-Specific Sliding Window Log*), the exact interface's name is taken (line 7).

```
1   method write_Parquet_files_to_storage (folder:Folder) {
2       for (file in folder) {
3           if (file.is_historic()) {
4               local levels = ["", "", "", "logs"];
5               levels[2] = file.get_name_prefix();
6               if (file.is_interface_specific_sliding_window()) {
7                   level[1] = file.get_interface();
8               }
9               else {
10                  level[1] = "all";
11              }
12              level[0] = file.get_abstraction();
13              copy_to_historic_branch(levels);
14          }
15          else {
16              copy_to_live_data_branch(file);
17          }
18      }
19  }
```

Figure 6.5: Pseudo code fragment for updating the final feature storage on the external *Feature Processor and Storage Component*

6.3 Interface Interaction

In order to e.g. allow the *Feature Processor and Storage Component* to transform *Bro* logs to *Parquet* format and update the storage as described in Section 6.2, the extracted features (in their respective, intermediate logs) need to be synchronized from the *manager* of the internal monitoring cluster to it beforehand. How that synchronization from the NsUS to the external *Feature Processor and Storage Component* is implemented is described in the following Subsection 6.3.1. Afterwards, we explain the implementation for a dynamic reconfiguration of the internal monitoring cluster in Subsection 6.3.2.

6.3.1 Automated Synchronization of Intermediate Log Files

Again, synchronization is automatically executed (in compliance with NFR.5) once in a predefined time interval (every five minutes in our PoC implementation) as specified by a *cron* job of the *crontab* file on the *Feature Processor and Storage Component*. Nevertheless, FHS-DNA also allows anytime manual synchronization. The respective interface for this task is the *Logs Synchronization Interface* as explained in Section 5.3. The implementation is primarily done in file *Synchronize-Transform-Update.sh.*

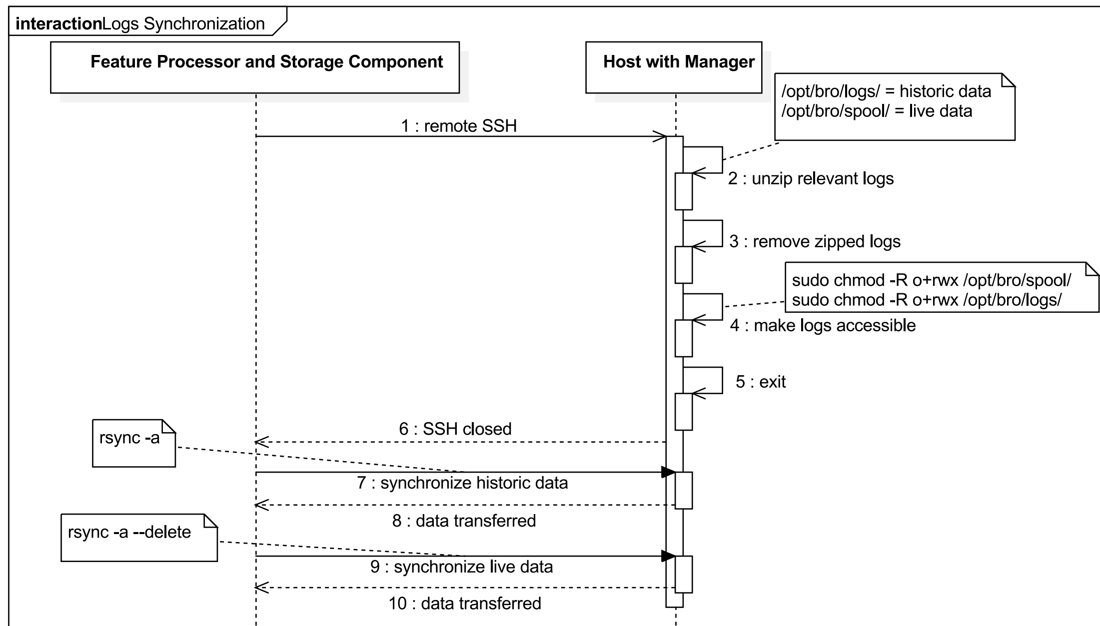

Figure 6.6: Sequence diagram for the automated synchronization of intermediate *Bro* log files

Detailed explanations in the following refer to the single steps of the respective sequence diagram in Figure 6.6.

First, the *Feature Processor and Storage Component* remotely accesses the *manager* of the *Bro cluster* via *SSH* (step 1). Within this remote connection, intermediate *Bro* logs on the *manager* are unzipped (step 2), zipped files are removed (step 3) and access rights to those logs' directories for subsequent synchronization with *rsync* are adapted (step 4). These directories to synchronize are */opt/bro/logs/* containing historic data and */opt/bro/spool/* with live data. Afterwards, the remote *SSH* connection is exited (step 5).
Finally, via *rsync* the directories are synchronized, so that they are locally available on the *Feature Processor and Storage Component* (step 7 and 8). With synchronization configured accordingly, it is guaranteed that old "live" data on the *Feature Processor and Storage Component* is locally removed, so that only the actual live data is saved in directory */opt/bro/spool/* (step 8).

6.3.2 Dynamic Reconfiguration

In contrast to synchronization, transformation and storage update, dynamic reconfiguration as explained in Subsection 5.3.3 and required by NFR.6 is not automated. It is explicitly required to function on demand as specified in NFR.5 (see Subsection 3.2.2).

Moreover, this task is executed via another interface, which is the *(Re-)Configuration Interface* as explained in Section 5.3.

Thus, reconfiguration is initiated on demand by the external *Configuration Component* representing an authorized operator's computer as explained in Subsection 5.2.2. Implementation and respective code can be found in file *Dynamic-Bro-Reconfiguration.sh* as well as in various additional files (*Reconfigure-1.sh*, *Reconfigure-2.sh*, *Reconfigure-3.sh*, *Reconfigure-4.sh* and *Reconfigure-5.sh*), each one defining a respective *configuration mode* as described in Subsection 5.3.3 and abstractly and conceptually depicted in Figure 5.3. Hence, we implement and provide five different *configuration modes* by default in our PoC implementation. The modes generally differ with respect to loading different scripts for FE onto different monitoring agents and their interfaces. More explanations on that are already given in Subsection 5.3.3. Nevertheless, additional modes can be implemented and added accordingly.

In detail, the basic implementation for dynamic reconfiguration works as depicted in sequence diagram in Figure 6.7. Further explanations in the following refer to the single steps of this diagram.
First, the external *Configuration Component* remotely connects to the host with the *manager* of the integrated *Bro* monitoring process on it via *SSH* (step 1).
Second, on this *manager* node, it initiates reconfiguration by launching the *Dynamic-Bro-Reconfiguration.sh* script with an additional passed parameter (e.g. *1*, *2*, *3*, *4* or *5*) representing the *configuration mode* to configure to (step 2). As for dynamic reconfiguration of our system's integrated *Bro* monitoring process a redeployment of it is mandatory in order to load the new *configuration mode* (due to *Bro* by default), this redeployment can only be executed by the cluster's *manager* node. Therefore, it is important to know that connecting to this *manager* host is unavoidable and can only be initiated on this host itself.
Next in process, the launched script checks, whether the passed parameter is valid or not (step 3). Valid parameters are only integers in the range [1;5] in our PoC implementation, each number representing one possible *configuration mode* to configure to. Hence, outgoing from Figure 5.3, the passed parameter has to be the concrete value of X (line 1). If the parameter is not valid, no reconfiguration is initiated. However, if it is valid the script further checks, whether the new mode differs from the old one (step 5). Only if so, it further calls another script (either *Reconfigure-1.sh*, *Reconfigure-2.sh*, *Reconfigure-3.sh*, *Reconfigure-4.sh* or *Reconfigure-5.sh*) accordingly to the passed, valid parameter (step 7).
This second script then checks, whether FHS-DNA's actually is monitoring network traffic or not (step 8). In case it indeed is, the monitoring process is stopped (step 10)

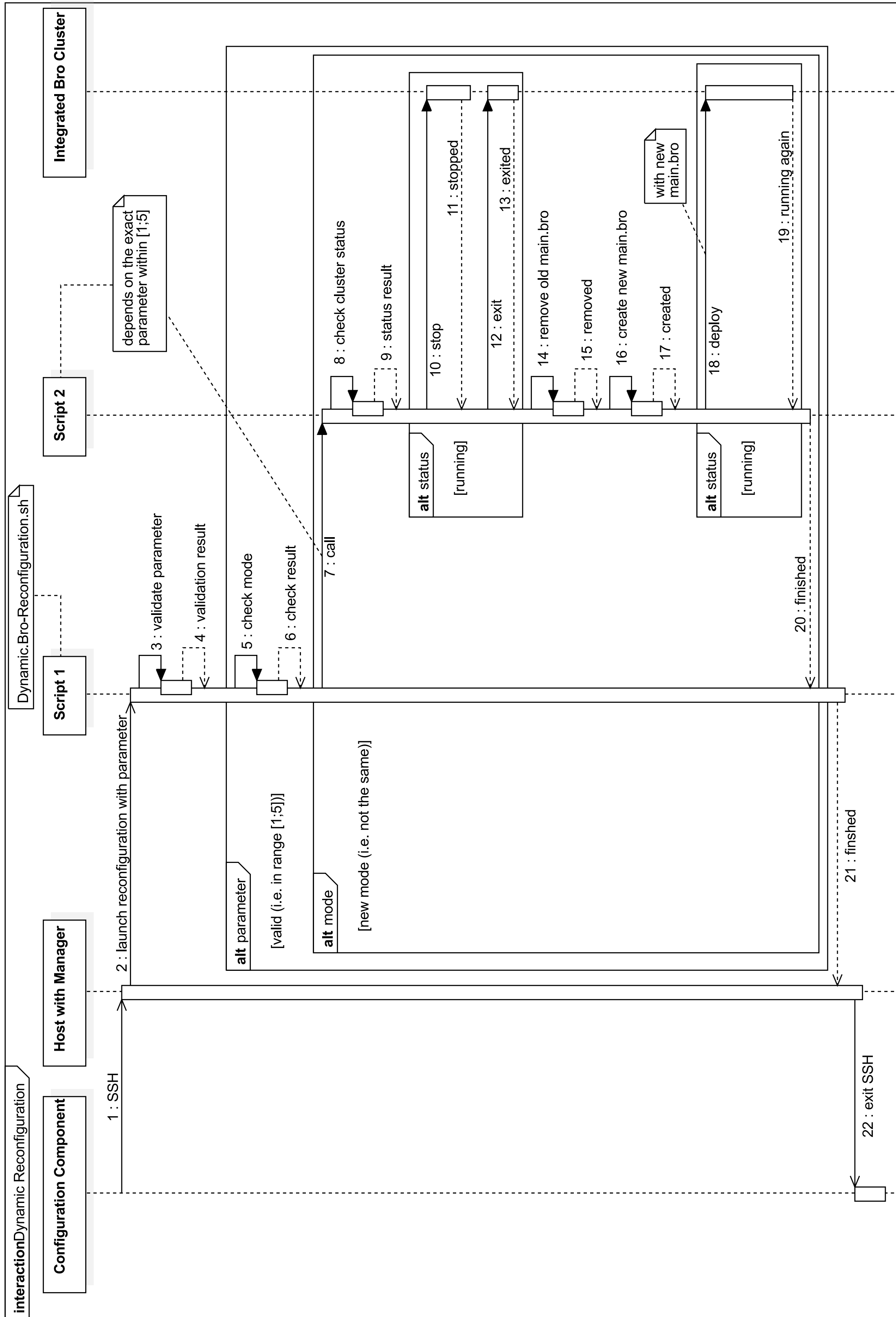

Figure 6.7: Sequence diagram for the dynamic, on-demand reconfiguration of FHS-DNA

and exited (step 12). Following, the old monitoring *configuration mode* file *main.bro* is removed (step 14) and a new one depending on the passed parameter is created (step 16). This is done independent whether FHS-DNA was monitoring before or not. Nevertheless, this can only be done if monitoring was stopped beforehand in case it was running. Consequently, that means that file *main.bro* is the concrete implementation of a *configuration mode* as abstractly and conceptually depicted in Figure 5.3. Afterwards, the newly configured *Bro* monitoring process is deployed again by now loading the new configuration files including the *main.bro* file (step 18). This may take a while. If FHS-DNA was not monitoring before, step 18 is not executed, as monitoring is not just started due to reconfiguration. Hence, in that case the new *configuration mode* is just saved (in file *main.bro*) for the next regular startup.
Typically, the remote *SSH* connection is finally closed again, when the operator knows that everything worked just fine (step 22). However, it is important to know that one could also exit the remote *SSH* connection directly after step 2, as the rest of the reconfiguration (steps 3 - 21) does not require any more interoperability by the operator once the script *Dynamic-Bro-Reconfiguration.sh* is called.

Concluding, it can clearly be seen that the most important file for this dynamic reconfiguration process is *main.bro*, which is the concrete instantiation of a *configuration mode* and therefore follows the conceptual and abstract layout of Figure 5.3. It can be found in the codebase in [65] besides all the other concrete instantiations of scripts further mentioned in Subsection 5.3.3 for transitive, conditional loading of the actual FE scripts. Their names in our PoC implementation are *cluster-main-1.bro*, *cluster-main-2.bro*, *cluster-main-3.bro*, *standalone-main-1.bro* and *standalone-main-2.bro*. Though, further files may also be added in order to allow the required extensibility and flexibility.

6.4 Summarizing Overview

Summarizing, we present an overview of all architectural components, their interaction and tasks they fulfill, in Figure 6.8. Each number stands for a task implemented in FHS-DNA and executed by its corresponding components. For better understanding, each one is shortly explained in the following.

- 1: Via *BroControl* serving as the *External Linkage Interface*, FHS-DNA's integrated *Bro cluster* is started for traffic monitoring and FE.
- 2: The *manager* of the integrated *Bro cluster* loads all configuration files, i.e. in specific also the current *main.bro* file.

- 3 and 4: The default *Bro* process of distributing this actual configuration to the respective *worker* and *proxy* nodes is applied.
- 5: The *manager* is started first [33].
- 6: Afterwards, the *proxy* is launched.
- 7: Finally, the *worker* nodes are started.
- 8: A *proxy* keeps the state of multiple *worker* nodes by the default process of *Bro*.
- 9: This step represents the *worker* nodes sniffing traffic as specified by the current *configuration mode* in *main.bro*.
- 10: FE generates (intermediate) *Bro* logs, which are directly logged onto the *manager*.
- 11: The *manager* collects and intermediately saves those *Bro* logs.
- 12: The external *Feature Processor and Storage Component* remotely connects via *SSH* to the internal *manager* for preparation of subsequent synchronization of *Bro* logs.
- 13: This synchronization is locally executed with *rsync* and represented by this step.
- 14: Once all the relevant (intermediately saved) *Bro* logs are locally available on the external *Feature Processor and Storage Component*, storage preparation is done by transforming the *Bro* logs into *Parquet* format.
- 15: Having the logs transformed to *Parquet*, they get inserted into the final storage. Hence, the local directory *datapool/* gets updated.
- 16: Finally, the *Feature Processor and Storage Component* does some local cleanup.
- 17: This step is associated with the dynamic reconfiguration of FHS-DNA. Thus, initially the external *Configuration Component* remotely connects to the internal *manager* via *SSH*.
- 18: On this *manager*, the new (re-)configuration is initiated by calling the respective script *Dynamic-Bro-Reconfiguration.sh* with an appropriate parameter.
- 19: Then, this script checks the passed parameter representing the new *configuration mode* to configure to.
- 20: If the passed parameter is a valid one and actually represents a new *configuration mode* (i.e. not the same as the system is currently running in), the monitoring

process is stopped and exited via *BroControl*, if it has been running so far. If the monitoring process is currently not running, then this step and the following ones (21, 22, 23) are not executed.

- 21: In detail, this stopping process is implemented by *Bro* by default and first stops the *manager*.
- 22: Afterwards, the *proxy* is stopped.
- 23: Finally, the *worker* nodes are stopped.
- 24: Once all the nodes are stopped, the old configuration (i.e. in particular the *main.bro* file) is removed.
- 25: The new configuration file (i.e. in particular the *main.bro* file) is written to disk of the *manager*.
- 26: Finally, if the monitoring process was running before initiating the reconfiguration, the monitoring process is started again via the *BroControl deploy* command with the new *configuration mode*. If FHS-DNA was not monitoring before the reconfiguration, then this step is not executed.
- 27: This step is totally independent of the other ones. It is not a sequential step as it represents the import and querying of (multiple) *Parquet* files by various external (A)DMs from the *datapool/* directory on the *Feature Processor and Storage Component*.

Finally, we want to mention a few more important aspects of the overall diagram in Figure 6.8 concerning multiple groups of its steps explained above.

- As declared in the diagram itself, all steps with a prefixed * are only executed, if the passed parameter for dynamic reconfiguration represents a valid, new *configuration mode*, i.e. it is not the same as the monitoring process is currently running in.
- Steps eight to eleven are not sequential steps, but are continuously and automatically executed as long as the monitoring process of FHS-DNA is running.
- Further, also steps twelve to 16 are not sequential steps with respect to the other ones, but are continuously and automatically executed every predefined time interval (five minutes in our PoC implementation). This is specified in a *crontab* file of the *Feature Processor and Storage Component*. Nevertheless, FHS-DNA also allows manual execution of these steps at any time.

- Once again, also steps 17 to 25 are not sequential steps with respect to the other ones. Dynamic reconfiguration may be initiated at any time on demand of the *Configuration Component*. However, in contrast to steps twelve to 16, these steps always appear as a block and are implemented sequentially within this block.

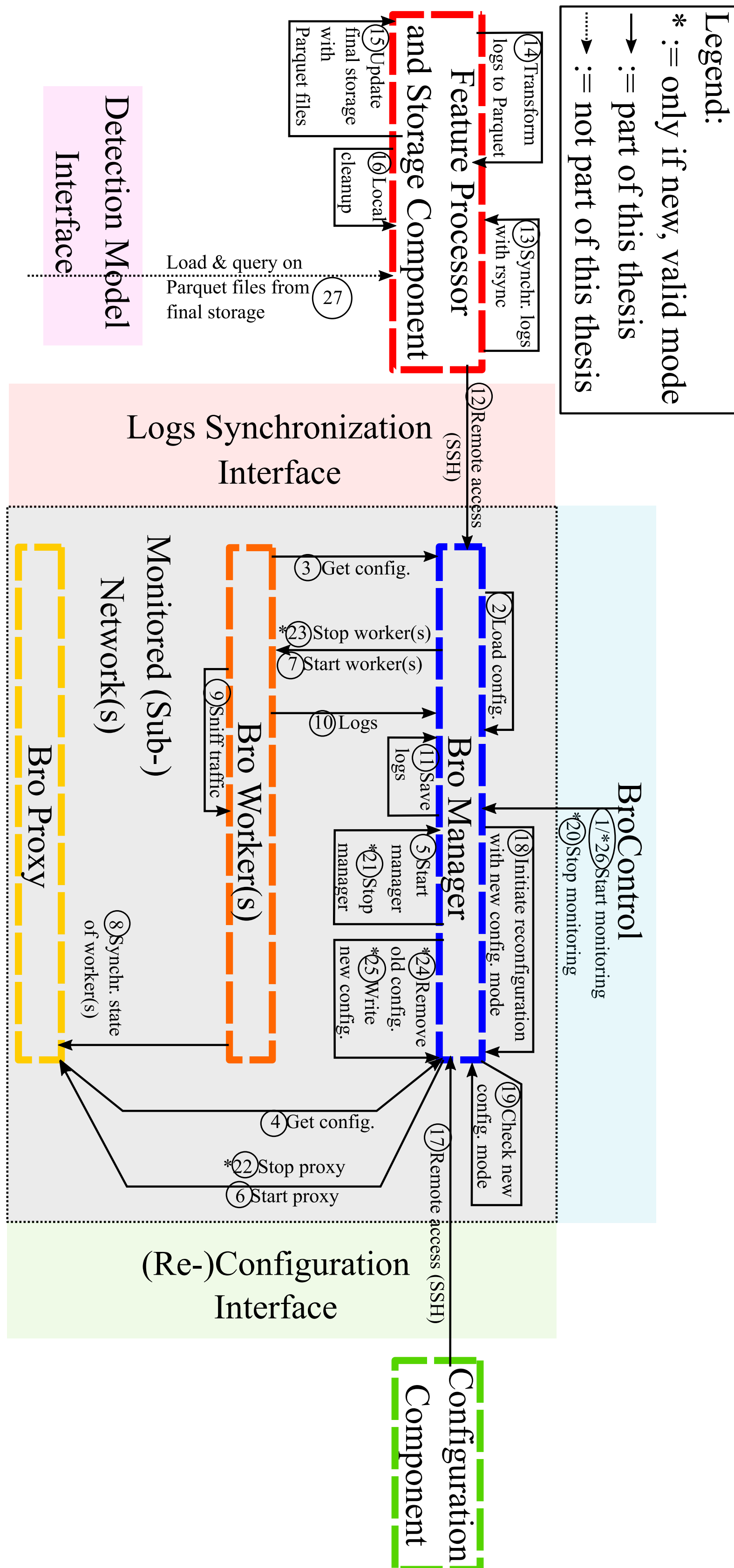

Figure 6.8: System implementation overview

Chapter 7

Testing and Evaluation

In the previous Chapters 5 and 6, the design and concrete PoC implementation for our system FHS-DNA is explained. Within this chapter, a test setup meeting required prerequisites for running our system in a virtualized network is described (see Section 7.1). Moreover, utilizing this test setup deployment we further evaluate our FHS-DNA with concrete measurements and stating their implications (see Section 7.2).

7.1 Test Setup Deployment

For testing, demonstration and evaluation purposes, we need a comfortable playground to easily and efficiently set up our system.

7.1.1 Virtualization

Therefore, some virtualization programs are used in order to simulate FHS-DNA and their single components' interactions. Within this subsection, those virtualization programs are shortly explored.

VirtualBox

The first of these programs is *VirtualBox*[1]. It is a "powerful x86 and AMD64/Intel64 virtualization product" [77] across platforms. In our setup, it is utilized as a provider

[1] https://www.virtualbox.org/

for running virtual machines (VMs) created by another virtualization program named *Vagrant*[1].

Vagrant

Hence, *Vagrant* is chosen for creating those VMs. It is a ”tool for building and managing virtual machine environments in a single workflow” [78] and is designed ”as the easiest and fastest way to create a virtualized environment” [78]. In close relation to that aspect, another very helpful capability of *Vagrant* is that it allows the creation and control of more than one VM via a single file named *Vagrantfile* [79]. That is called a multi-machine setup [79]. In Subsection 7.1.4 below, the configuration of this *Vagrantfile* for our multi-machine setup is explained in more detail.

Mininet as Network Emulation Tool

As network emulation tool we use *Mininet*[2]. It ”is a network emulator which creates a network of virtual hosts, switches, controllers, and links” [80]. Furthermore, hosts in *Mininet* are able to ”run standard Linux network software” [80], which allows e.g. running a *Bro ”normal” cluster* on those emulated hosts. Hence, with this tool the NsUS (e.g. OBNs) are emulated on a single VM built with *Vagrant* and ran in *VirtualBox*. Main reasons for choosing *Mininet* include, but are not limited to:

1. Support for standard network topologies by default [80].
2. Providing an extensible Python API for creating manually defined network topologies [80]. Together with the reason above, this allows us to test our system FHS-DNA on arbitrary network topologies (according to NFR.4). Nevertheless, the special characteristics of hierarchic, compartmentalized networks can also be considered.
3. Capability of running real Linux programs on the hosts [81]. This is especially needed, as FHS-DNA will run the *Bro ”normal” cluster* on emulated hosts for testing and demonstration purposes.
4. Emulated networks serve as a simple and cheap testbed for up to hundreds of machines on a single operating system kernel due to process-based instead of full system virtualization [80]. Hence, there is no need to wire up a physical network.

---

[1] https://www.vagrantup.com/

[2] http://mininet.org/

5. *Mininet* allows easy configuration and adaption of many networking options, e.g. simulate resource-constrained hosts, configure port-mirroring on switches, etc. [81]. This allows us to simulate e.g. resource constraints of some internal (i.e. inside the NsUS) components of FHS-DNA in the *Bro "normal" cluster*.
6. *Mininet* allows easy deployment to real hardware with minimal changes [80].
7. Fast (re-)booting, quick (re-)configuration, testing and analysis of entire network and its parameters is possible [80].
8. Good, comprehensive documentation is available [82].

However, it is important to know - especially for testing our system FHS-DNA - that by default all *Mininet* hosts "share the host file system and PID space" [81]. Hence, our system's *Bro "normal" cluster* nodes will all share the same underlying file system. Though, this is not an issue, as e.g. logging from *worker* nodes to the *manager* is explicitly handled within separated directories clearly specifying which device generated which logs.
Our implemented testing topology for the NsUS is described in more detail in Subsection 7.1.4 below.

7.1.2 Required Prerequisites of a Bro "Normal" Cluster

Before the actual test environment is completely set up with all installations (see Subsection 7.1.3) and configurations (see Subsection 7.1.4) made, we shortly want to refer to some prerequisites of a *Bro "normal" cluster*. In the following, the most fundamental ones are listed, whereas a detailed description can be found in [32].

- All devices of the NsUS, which should be part of the *Bro "normal" cluster's*, need to run with the exact same version of the same operating system [32].
- The program *rsync*[1] must be installed on every node of the cluster [32].
- During operation, every node in the cluster (except the *manager*) need *sshd*[2] installed and running. In contrast, the *manager* needs *ssh*[3] installed [32].

[1] https://rsync.samba.org/?

[2] https://man.openbsd.org/sshd.8

[3] https://man.openbsd.org/ssh.1

- *ssh* to connect from the *manager* to all other hosts of the cluster needs to work without any prompt [32].
- In cluster mode, *Bro* and *BroControl* only need to be installed on the *manager* [32].
- At least one *proxy*, one *worker* and a *manager* need to be defined [32].
- Nodes of type *proxy* and *worker* need to be able to connect to the cluster's *manager* [32].
- Every *worker* needs to be connected to one *proxy* [32].
- Having a *logger* in the cluster, every other host of the cluster needs to connect to this *logger* [32].
- Finally, the *manager* must be able to connect to each of the other cluster's hosts on TCP port 22 [32].

Hence, those prerequisites on the one hand need to be respected in the definition of our emulated test network and on the other hand require specific installation setups as explained in the following.

7.1.3 Installations

1. *VirtualBox* and *Vagrant* need to be installed on the host system.
2. Ansible[1] needs to be installed on the host system for simple and automated configuration management of multiple inter-correlating VMs.
3. Installation and configuration of all further systems (e.g. *Mininet* and *Bro*) and packages (e.g. *python*) for the automated setup of the various VMs must be ensured as described in Subsection 7.1.4.
4. Once all the VMs and the *Mininet* hosts are ready, further *Bro* installations can be done. These for example include the execution of the scripts *Broccoli-and-Python-Bindings-Installation.sh* for the installation of *Broccoli*[2] and the respective *Python Bindings*[3] on the *manager* node and *BAT-Installation.sh* for installing *BAT* on the *preprocessor* machine.

[1] https://www.ansible.com/

[2] https://www.bro.org/sphinx/components/broccoli/broccoli-manual.html

[3] https://www.bro.org/sphinx/components/broccoli-python/README.html

```
21  config.vm.define "networksimulation" do |box|
22    box.vm.network "private_network", ip: "192.168.33.10"
23  end
24  config.vm.define "preprocessor" do |box|
25    box.vm.network "private_network", ip: "192.168.33.11"
26  end
27  config.vm.define "controlunit" do |box|
28    box.vm.network "private_network", ip: "192.168.33.12"
29  end
```

Figure 7.1: Code snippet of *Vagrantfile* for the exemplary multi-machine test setup

7.1.4 Configurations

As already induced above, configurations and installations are closely related. In addition to this subsection, we want to refer to files *README-networksimulation.txt*, *README-preprocessor.txt* and *README-controlunit.txt*, which give very detailed configuration (and installation) instructions on how to set up FHS-DNA for testing, evaluation and demonstration purposes.

Vagrant

For the virtualization of different components of FHS-DNA, we define a *Vagrant* multi-machine setup. It consists of three VMs. Definition is done in the *Vagrantfile* as depicted in Figure 7.1, by giving each VM a name and an IP-address.

In order to limit coordination and configuration overhead as well as used resources of our host machine, we decided to emulate the NsUS on a single VM instead of defining one VM per internal component. So, machine *networksimulation* (line 21 - 23) runs the entire emulated and monitored network. Consequently, we install *Mininet* upon this VM and define a test network with various hosts as described in Subsection 7.1.4. Moreover, FHS-DNA's internal components as part of the *Bro "normal" cluster* then finally run upon those defined *Mininet* hosts.
However, in contrast to the NsUS, each external component is separately simulated by a single VM, as they fulfill different tasks and use different interfaces. This setup is also more realistic to a real-world deployment. Thus, the *preprocessor* VM (line 24 - 26) represents the external *Feature Processor and Storage Component*. Furthermore, the *controlunit* machine defined in the *Vagrantfile* (line 27 - 29) represents a computer of an operator, who is authorized to initiate dynamic reconfiguration of FHS-DNA on the *manager*. Hence, this VM simulates an external *Configuration Component*.
Besides those key configurations, even further ones can be made in the *Vagrantfile*, such

as for example installing package *python3* and customizing the memory of each VM to *2048* MB.

Ansible

Once all three VMs are defined in the *Vagrantfile*, *Ansible* is used to provision these VMs with the appropriate systems, further packages and manual implemented scripts automatically during startup. This is done by defining some roles (*common*, *bro*, *mininet*, *preprocessor* and *controlunit*), which get associated to a VM as defined in file *playbook.yml*. Moreover, each defined role has a directory structure, in which both a file named *main.yml* and the manual implemented scripts to load are located. Thus, in each of those role-specific *main.yml* files, respective packages (e.g. *python-dev*) and scripts are defined to automatically load and install on their associated VM. This is done by executing the commands *vagrant up* and *vagrant provision* on the respective machines (i.e. VMs).

Mininet

For a detailed description of the definition and configuration of an appropriately emulated test network, we refer to script *sshd-topology-1.py* in the following.
Overall, in method *myFirstNetwork()*, we define two switches, two controllers and four resource-constrained hosts with IP-addresses *20.0.0.1/8*, *20.0.0.2/8*, *172.16.0.1/12* and *172.16.0.2/12*, equally spread across those two subnets. Corresponding interfaces to monitor traffic on are named *h1-eth0*, *h2-eth0*, *h3-eth0* and *h4-eth0*. In close conjunction with the prerequisites of the *Bro "normal" cluster* as described in Subsection 7.1.2, method *sshd(...)* runs *sshd* on all emulated hosts. Moreover, the prerequisites regarding routable connections between different typed devices of the *Bro "normal" cluster* are also considered in the defined network's topology.
Automated startup of this network is implemented in script *Start-sshd-topology-1.sh*.

The Bro Network Security Monitor

Once the emulated network is up and running, *Bro* is potentially able to run on its hosts. However, in order to deploy our system and its integrated *Bro "normal" cluster* correctly, some further configurations need to be adapted to and synchronized with the underlying emulated network and its hosts.

1. File *broctl.cfg* managed by *Ansible* is specified with global *BroControl* configurations.

2. File *networks.cfg* is expanded with our local networks *20.0.0.0/8* and *172.16.0.0/12* as emulated in *Mininet*. Hence, this file is the additional one containing all the networks' IP-addresses as described in Subsection 5.3.3.

3. File *node.cfg* specifies the exact layout of our system's integrated *Bro "normal" cluster*. In specific, host *20.0.0.1* is of type *manager* and *proxy*, whereas nodes *172.16.0.1* and *172.16.0.2* are of type *worker*. Hence, the only emulated host not part of the *Bro "normal" cluster* is host *20.0.0.2*. So, this file specifies a *monitoring layout* as described in Subsection 5.3.3.

4. In order to allow communication without prompts between nodes of the *Bro "normal" cluster*, script *Bro-Nodes-SSH-Key-Generation.sh* creates an *SSH* key pair without a passphrase. It is important to understand that only one key for all *Bro* nodes is sufficient, as they all share the same file system of the emulated *Mininet* network hosts as already mentioned above.

5. Very similar to the previous one, script *VM-SSH-Setup.sh* prepares setup of passwordless *SSH* as user *vagrant* from the *preprocessor* machine to the *manager* host of the cluster. The same is implemented in script *VM-SSH-Setup-2.sh* for the *controlunit* machine.

More general information on the configuration of the files for the first three points is given in [34]. In addition, how this testing and demonstration setup of FHS-DNA can now be deployed to and executed on any host system fulfilling the specified requirements (e.g. installations) is described in detail in files *README-networksimulation.txt*, *README-preprocessor.txt* and *README-controlunit.txt*.

7.1.5 Summarizing Overview

This subsection summarizes all key installations and exemplary configurations as they are in our FHS-DNA's testing and demonstration setup and depicts them in Figure 7.2. Nevertheless, this diagram only contains the most important information and therefore has not the intention to be a fully complete deployment abstraction.

However, there are three major differences from this testing deployment to a potential real-world scenario, which are all indicated by a big red exclamation mark in Figure 7.2. Hence, they are shortly explained in the following in the order of their red numbers next to the respective exclamation mark.

For *1*, we see that the generated directories (*/logs* and */spool*) containing *Bro's* intermediately saved log files are located in the file system of the *networksimulation* VM

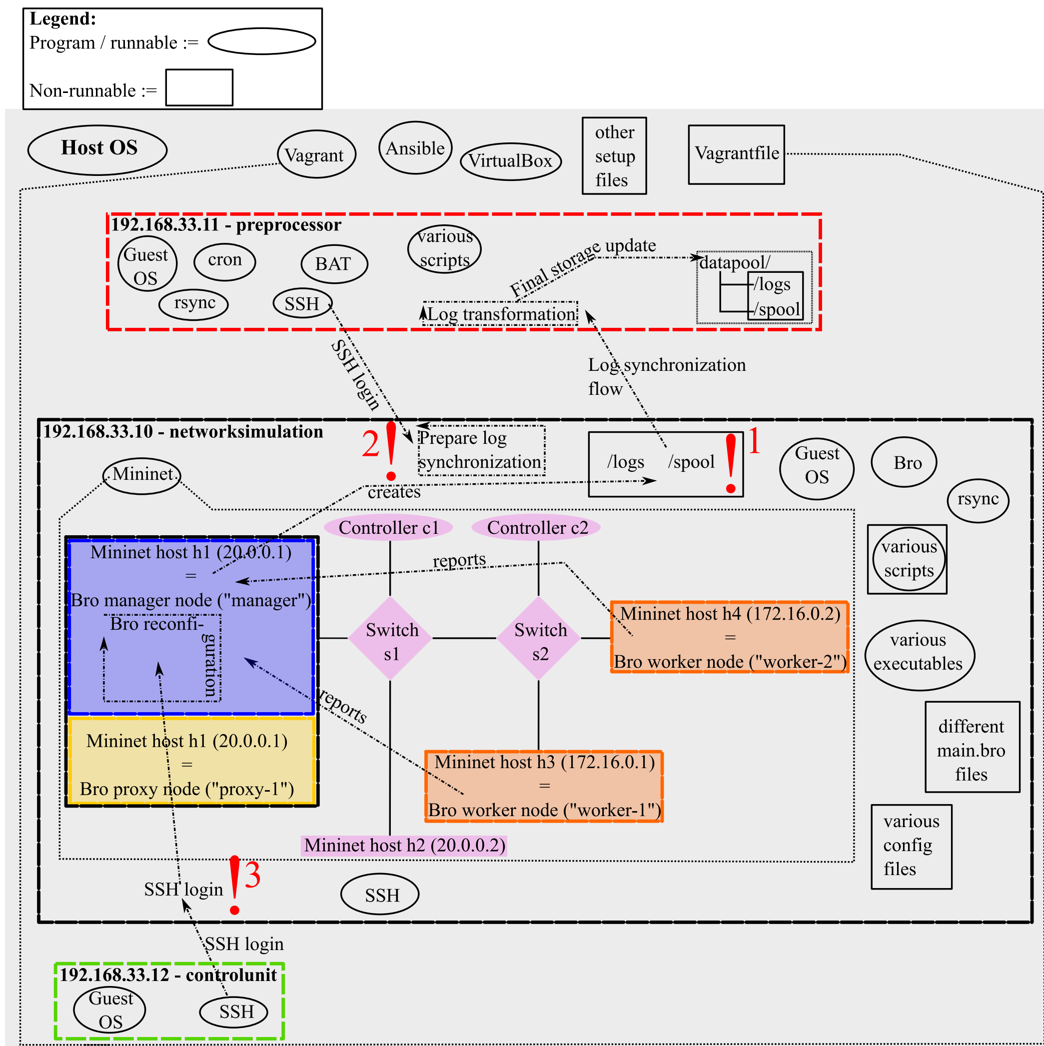

Figure 7.2: System demonstration and test setup deployment overview

instead of the collecting *manager* node directly. This is, as *Mininet* is installed on the *networksimulation* VM and all emulated hosts (including the one on which the *manager* is running on) share the same file system of the underlying *networksimulation* machine. In contrast, in a real-world scenario these directories would be generated in the file system of the *manager* host directly.

Closely related to this first difference is the second one. For *2*, the appropriate *SSH* login from the *preprocessor* VM (representing the external *Feature Processor and Storage Component*) to the *networksimulation* VM would need to be made to the *manager* host instead, as the logging directories in a real-world scenario are located in its file system directly as explained above. Consequently, preparation of subsequent synchronization would also be done on the *manager* node instead of the *networksimulation* machine.

The same holds for *3*. As depicted in Figure 6.7, the external *Configuration Component* (simulated by the *controlunit* VM) remotely connects via *SSH* to the integrated *manager* host. Though, in this test setup deployment, the *controlunit* machine first remotely connects via *SSH* to the *networksimulation* VM and then the *networksimulation* VM subsequently connects via *SSH* to the emulated *manager* host in order to initiate dynamic reconfiguration on it (because dynamic reconfiguration can only be executed by the *manager* host itself as explained in the respective Subsection 6.3.2). However, as the *controlunit* VM does principally not know the *manager* host emulated on another VM, this detour has to be taken. Hence, in contrast to this test setup deployment, in a real-world scenario the external *Configuration Component* (simulated by the *controlunit* VM) remotely connects via *SSH* directly to the integrated *manager* host as depicted in Figure 6.7.

Summarizing, it is important to understand how our explicit test setup deployment differs from a potential real-world scenario. However, a real-world deployment of our system is in principle not a problem, although a couple of aspects have to be considered as e.g. explained above.

7.2 Evaluation

Based on the test and demonstration setup deployment previously described, we evaluate our system with respect to different aspects. These are time measurements for traffic handling in combination with FE and file transformation as well as measuring of disk space consumption by *Bro* logs and *Parquet* files. These aspects are considered most important for the evaluation, as first of all resources such as CPU and disk space are limited within the NsUS per assumption. Hence, respective prerequisites of FHS-DNA

definitely need to be evaluated. Furthermore, by the evaluation of these aspects we cover all main functionality of FHS-DNA and the respective requirements from Section 3.2. This includes the basic traffic handling (i.e. monitoring) in combination with FE, the transformation (which is considered as (partial) FP) of intermediate features' *Bro* log files as well as the final FS and its savings in disk space due to the compressed *Parquet* format.
Hence, for each of those aspects the (common) methodology is explained, results are presented and implications stated.

7.2.1 Common Methodology

In general, all measurements are conducted on an Apple MacBook Air from 2017. The respective processor is an 2,2 GHz Intel Core i7. Furthermore, it has an available memory of eight GB and a 256 GB SSD hard disk. All aforementioned aspects of FHS-DNA are measured by the script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh*. Hence, the general and common methodology is the execution of this script.

However, for simplicity, validation and precision reasons, this measuring is performed on a single *standalone Bro* instance directly on the *networksimulation* VM, i.e. not on one or more emulated *Mininet* hosts as part of a *Bro "normal" cluster*. So, avoiding another layer of virtualization makes our measurements more valid and accurate. Even further, this way of measuring also serves as a stress test of FHS-DNA, as all traffic is monitored on a single interface all at once. Consequently, in order to directly transform the generated *Bro* log files into *Parquet* format on the *standalone Bro* instance, we further need to install *BAT* on this *networksimulation* VM with the script *BAT-Installation.sh*. However, it is important to know that this installation is only for testing purposes and usually not required for normal operation of FHS-DNA. The same also holds for the transformation script *transformation-measurements.py*. It locally (i.e. directly on the *networksimulation* VM and not as usually on the *preprocessor* VM) transforms all intermediate *Bro* log files to *Parquet* format. Hence, the *Parquet* files will also be locally available on the *networksimulation* VM, where their sizes are summed up for disk space measurements, before deleting them again.
Hence, only the *networksimulation* VM of the three VMs defined in the *Vagrantfile* is up running and has four GB memory from the overall available eight GB. Consequently, together with the host system (i.e. the Apple Macbook Air), two machines are running at the same time. So, in contrast to the demonstration and testing setup (where each VM is running and has two GB memory), these are modifications, as very large traffic

files and therefore large *Bro* log files are processed. However, this is only necessary for our particular measurements as defined in script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh*.

In detail, script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh* generates three different, manual (i.e. not *Bro* related) log files with plenty of information for each specific execution run. Again, we take special care of not to lose any information and therefore log as much as possible.
The first generated log file has the prefix *times-TH-FE-TF-*, followed by the starting timestamp of that script's run. Besides some meta information relevant for later visualization and evaluation, it contains both different timings for the traffic handling and FE as well as for the file transformation. For both aspects, these time values are separated into three different categories each, which are *real* (i.e. the wall clock), *user* (i.e. the CPU in user mode) and *sys* (i.e. CPU in kernel mode).
The second generated log file has the prefix *sizes-Logs-Parquets-*, followed by the same starting timestamp. Again, besides some meta information, it contains the name and the size of each *Bro* log and *Parquet* file ever generated by FHS-DNA during that script's execution.
The third and last generated log file has the prefix *duration-*, once again followed by the same starting timestamp of that script's run. It is the smallest log file and contains the duration of the execution of the script in minutes.
Regarding the content of this script, 14 different traffic pcap-files, ranging from 100,000 up to 3,000,000 packets, are monitored. However, it is important to know that all those pcap-files are crafted out of the default *exercise-traffic.pcap*[1] distributed by *Bro*. This file contains exactly 100,000 packets. Detailed statistics and descriptions of this file are given in the appendix in Section A.2. Hence, the other 13 ones are constructed out of this one by concatenating it respectively often. This gives us high determinism of the monitored traffic and allows for conclusions concerning e.g. the evaluation of FHS-DNA as a function of traffic growth.
Besides the 14 different pcap-files this script also iterates over 19 different feature combinations to extract (as listed in Table 7.1), where combination *0* is the one without any manually constructed features (i.e. non from Table 6.1 being extracted). It only generates the default *Bro* log files, which are always generated (i.e. also by all other combinations) and therefore apply as baseline for subsequent evaluations.
Furthermore, in order to get meaningful and comprehensive results for later visualiza-

[1] Distributed on `https://www.bro.org/static/traces/exercise-traffic.pcap`

tion and evaluation, each single permutation of tested pcap-file and feature combination is iterated 20 times in a third, nested loop.

Summarizing, the most important script for evaluation is *measurements-for-traffic-handling-feature-extraction-file-transformation.sh*. It calculates the actual time values for traffic monitoring, FE and transformation of intermediate *Bro* log files as well as the required disk space of all ever-generated files of each format (*Bro* and *Parquet*).
Visualized results of this script's execution on a single *standalone Bro* instance directly on the *networksimulation* VM and corresponding implications are stated below. All visualizations are generated by executing file *FHS-DNA.ipynb*, which is a *jupyter*[1] notebook.

7.2.2 Results and Implications

Before exploring the results and stating implications, we want to shortly explain all feature combinations as listed in Table 7.1, which are considered most relevant and therefore measured by script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh* as described above.
The relevance of these feature combinations is argued based on a couple of decisions.

1. Sequentially build up more comprehensive (which also means more resource intensive) combinations outgoing from some baseline up to finally reaching a combination of extracting all features available in the PoC implementation of FHS-DNA. That way, we gain conclusions about effects on resources of more and more features being handled. For that, combination *0* only extracts (and therefore logs) standard features, which are handled by *Bro* by default. An overview of those potentially extracted default features and respective logs is given in [68]. In contrast, combination *18* additionally extracts all features considered in our PoC implementation of FHS-DNA so far and therefore is the most comprehensive combination.

2. Explicitly extract each manually implemented feature (from Table 6.1) on its own (of course, together with the default ones) in combinations *1* to *12*. That way, we respect another relevance criterion, which is to allow conclusions about which level of abstraction (PL, CL, NL) basically consumes the most resources (i.e. especially processing time and disk space) for the particular traffic under surveillance.

[1] https://jupyter.org/

3. Consider feature combinations, which equally represent features of all levels of abstraction. Such combinations are for example *13* and *16*, which contain (besides the default ones) exactly one and respectively two features of PL, CL and NL. Furthermore, in order to measure the exemplary evolution from less to more comprehensive feature handling, combinations *14* and *15* are stepwise building up combination *16* outgoing from *13*.

4. Always combine the most comprehensive and intensive features (with respect to necessary calculations directly during FE and resulting logs' sizes) of each level of abstraction, so that these measurements can serve as an exemplary upper bound.

Summarizing, due to the high number of manually implemented features and therefore potential combinations, we had to limit their number to a reasonable minimum with respect to execution time of script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh*. Considering more combinations always stands in a potential trade-off to e.g. less repetitions or less (or smaller) traffic (files). However, in order to also gain conclusions about FHS-DNA behaving in growing network traffic and making these conclusions as reliable and stable as possible, we also want to have a relatively high number of repetitions and different (and big) traffic files. Hence, all the results below are based on 20 repetitive runs for each permutation of the 14 different traffic files and the 19 feature combinations as measured by script *measurements-for-traffic-handling-feature-extraction-file-transformation.sh* and explained above.

Table 7.1: All evaluated feature extraction combinations

Combination No.	**Combination's Extracted Features**
0	Default FE by Bro only[1]
1	0, raw_layer2_packet_header
2	0, raw_IPv4_packet_header
3	0, raw_IPv6_packet_header
4	0, raw_TCP_packet_header
5	0, raw_UDP_packet_header
6	0, raw_ICMP_packet_header
7	0, land
8	0, count
9	0, synack
10	0, ct_srv_dst

[1] On overview of potentially generated default log files is given in [68]

Combination No.	Combination's Extracted Features
11	0, ct_src_ltm
12	0, ct_dst_sport_ltm
13	0, 1, 9, 11
14	0, 1, 2, 9, 11
15	0, 1, 2, 7, 9, 11
16	0, 1, 2, 7, 8, 9, 11
17	0, 1, 2, 4, 5, 7, 8, 9, 10, 11, 12
18	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Processing Time

The first questions we want to evaluate are, how different feature combinations behave in time over growing network traffic with respect to:

1. Traffic monitoring and FE;
2. Transformation (i.e. transforming the original extracted features' *Bro* logs into final FS *Parquet* format)

Regarding feature combinations, different on the one side can mean differently comprehensive. Hence, it is particular interesting, whether more features being extracted together take more time than less ones from the same amount of traffic. On the other side, different can also mean equally comprehensive combinations (as e.g. all combinations from *1* to *12*), but containing different features (e.g. from different levels of abstraction). That way, we gain insights on which feature takes how much time, that means is more resource-intensive to extract / transform than another one for the same amount of traffic. Measuring that for various features of the same abstraction level also yields conclusions not only about a single feature, but categorized for the respective abstraction levels. In addition, as we also evaluate all these aspects for various amounts of traffic, we even gain additional conclusions about whether these insights for the same amount of traffic change with increasing traffic or are stable over the number of packets being monitored.

Our expectations for these aspects regarding **traffic monitoring and FE** are:

- For the same amount of traffic, more / less comprehensive FE requires more / less time. Extracting more features should intuitively require FHS-DNA to take more processing time than extracting less features, as each feature is handled separately and synergy effects therefore are not expected. Thus, we expect the

baseline combination *0* to require least time and in contrast combination *18* the most.

- For the same amount of traffic, PL features require more time than features of the other two levels of abstraction (CL and NL). Intuitively that should be the case, as PL features handle each single packet of the traffic, whereas e.g. CL features only handle (in most cases less) connections.
- For the same amount of traffic, CL features require less time than NL features, as latter ones need to correlate information over e.g. all connections. Typically, these correlations require some additional calculations besides default logging, which intuitively require more time than only logging information.
- In growing network traffic, all previously expected observations will increase linearly in the number of packets. This is expected, as we explicitly crafted all measured traffic files by concatenating the original 100,000 packets from *exercise-traffic.pcap*. Hence, monitoring and FE should always require the same time per the same 100,000 packets, as synergy effects are not intuitive.

Moreover, our expectations for above aspects regarding **transformation** are:

- For the same amount of traffic, handling more features requires more time. Having more features being extracted clearly results in more *Bro* log files (as for each feature we have one file) and therefore more files need to be transformed.
- For the same amount of traffic, PL features typically tend to require more time than features of the other two levels of abstraction (CL and NL). Intuitively that should be the case, as PL features handle each single packet of the traffic and therefore typically tend to result in bigger *Bro* log files, whereas e.g. CL features only handle (in most cases less) connections and therefore typically tend to generate smaller *Bro* log files. Hence, the smaller a *Bro* log file is, the less time for transformation is anticipated.
- For the same amount of traffic, general assumptions on whether CL or NL features require more time are not possible. That is, because transformation time mainly is expected to be correlated to the file's content only. Hence, it may be the case that each type of feature sometimes grows bigger than the other type (and vice versa), which intuitively prevents general conclusions.
- In growing network traffic, all previously expected observations will increase linearly in the number of packets. This is expected, as we explicitly crafted all measured traffic files by concatenating the original 100,000 packets from *exercise-*

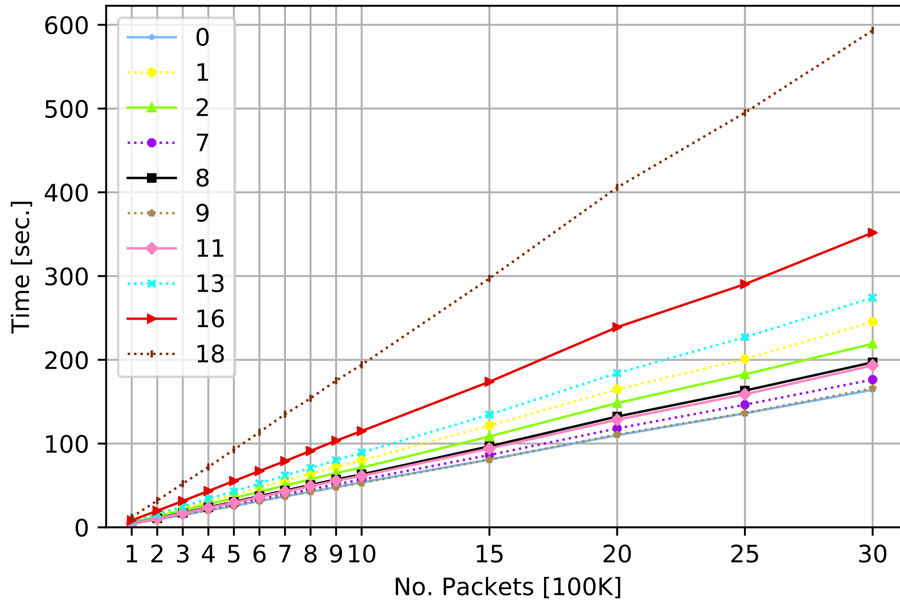

Figure 7.3: Median extraction times for selected feature combinations in growing network traffic

traffic.pcap. Hence, transformation should always require the same time per the same 100,000 packets, as all features' log files are also expected to grow linearly and therefore, synergy effects are not intuitive.

In order to explore our expectations, Figures 7.3 and 7.4 depict respective line plots of selected feature combinations. This selected subset is the same in both figures and contains combination *0* as a baseline, so that times for traffic handling and FE as well as for transformation can be benchmarked to the integrated default process (and logs) of FHS-DNA (resulting from *Bro*). In contrast, the most comprehensive combination *18* is also plotted, in order to have an upper bound. In between, especially combination *13* and *16* are respected, as well as all their included combinations with features of all three levels of abstraction. Hence, we have a minimal and therefore clearly represented subset, which considers all general relevance criteria as initially explained at the beginning of this section. Moreover, both figures show the amount of traffic in 100,000 packets on the x-axis and the appropriate median time values (over all 20 runs) in seconds on the y-axis.

In Figure 7.3, we see that all our respective expectations are met. The *0* line needs the least processing time and indicates, that monitoring traffic per se already consumes a major part of it. Slightly more time than for only processing the default *Bro* features

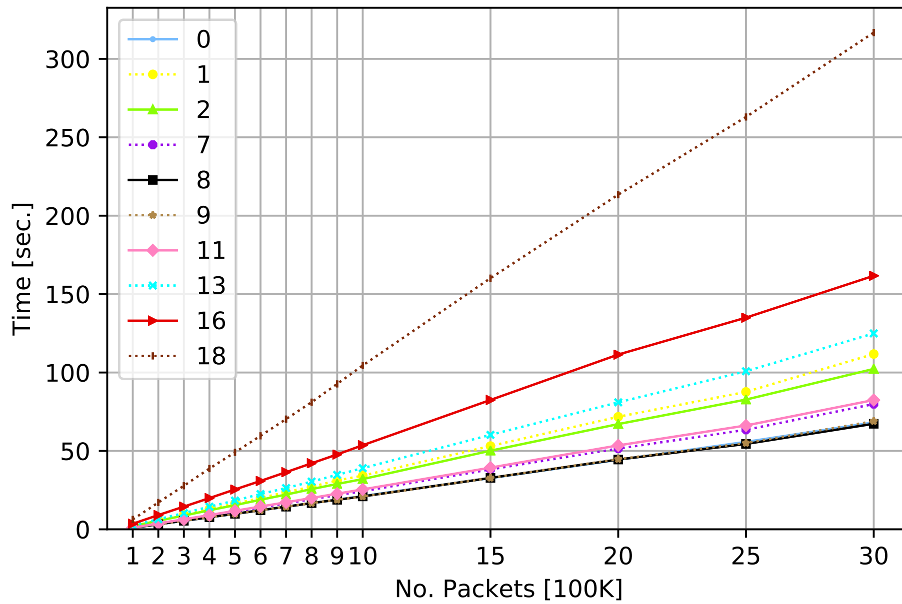

Figure 7.4: Median transformation times for all features' generated *Bro* log files of selected feature combinations in growing network traffic

is taken by additional CL features (combinations *7* and *9*), followed by the plots of additional NL features (combinations *8* and *11*) and the plots for additional PL features (combinations *1* and *2*). Furthermore, more comprehensive FE requires even more processing time (combinations *13*, *16* and *18*). In addition, median processing time grows almost perfectly linearly in the amount of monitored packets.

For the transformation time diagram, Figure 7.4 shows that also all our appropriate expectations are fulfilled. The more features we extract, the more transformation time is needed. However, this only holds if the features being extracted actually could be identified in the traffic, as they otherwise yield in empty *Bro* logs and therefore require (almost) no transformation time. Hence, this can be seen by the plots for combinations *8* (one additional NL feature is extracted) and *9* (one additional CL feature is extracted), as they indicate almost the same time as the baseline for combination *0*. Hence, general conclusions for NL and CL features cannot be made in this context. Nevertheless, represented PL features tend to result in the biggest files (as considering the most information, i.e. all packets) and therefore need the most transformation times. So, the transformation of additional (i.e. besides the default ones) PL features' logs (combination *1* and *2*) take more time than all the other depicted, equally compre-

hensive combinations (*7*, *8*, *9* and *11*). Finally, the median transformation time again grows almost perfectly linearly in the amount of monitored packets.

The next questions we want to evaluate are, whether handling more features at the same time (i.e. within one feature combination) yields synergy effects with respect to processing times compared to handling features (and their respective log files) more separately for:

1. Traffic monitoring and FE;
2. Transformation (i.e. transforming the original extracted features' *Bro* logs into final FS *Parquet* format)

In other words, the questions are whether time values for separately handled (i.e. within different runs) features (and log files) do sum up to the same time values as if they would have been extracted together or not. Answering these questions is especially interesting with respect to the high configurability, flexibility and extensibility for FE of FHS-DNA. Once implications are known, the operators are able to best utilize the limited available resources due to potential synergy effects while extracting more features at a time than doing it separately. As we further want to know, whether potential synergy effects for these two questions also depend on the amount of traffic being monitored, we decided to evaluate aforementioned questions for three different amounts of traffic. These are one, two and three million packets respectively.

Our expectations for these aspects regarding **traffic monitoring and FE** are:

- Extracting more features within the same run (i.e. in one feature combination) yields synergy effects, as simply monitoring the traffic (even without any FE) already makes up a major part of the processing time and additionally also always the default features are extracted for each run.
- The more comprehensive one feature combination is, the bigger are the synergy effects, as splitting up huge, comprehensive combinations would result in even more single features and therefore require even more runs for monitoring the (same) traffic for each feature's extraction. Additionally, also always the default features are extracted in each run and therefore additional processing time is required.
- The aforementioned expected observations are intuitively considered to be independent of the amount of traffic.

Moreover, our expectations for the above aspects regarding **transformation** are:

- Transforming all log files from a more comprehensive FE run also intuitively yields synergy effects, as for each run always the default features' log files need to be

transformed, too. Hence, unnecessary overhead induced by multiple, less comprehensive runs can be calculated by the number of runs minus one and then multiply the result with the transformation times of combination *0*.

- The more comprehensive one feature combination is, the bigger are the synergy effects, as splitting up huge, comprehensive combinations would result in even more runs with single features and log files and therefore require even more transformation time. That is expected, as always the default features' log files would need to be transformed for every run causing additional time.
- The aforementioned expected observations are intuitively considered to be independent of the amount of traffic.

Hence, for both cases the expected major overhead results from the every time (i.e. for every run) extracted and therefore transformed features (as well as the monitored traffic for FE). In order to explore our expectations, Figures 7.5 and 7.6 depict respective bar plots of all feature combinations from Table 7.1 for the selected traffic files with one, two and three million packets. Moreover, both figures show the different feature combinations on the x-axis and the appropriate median time values (over all 20 runs) in seconds on the y-axis.

In both Figures 7.5 and 7.6, we see that all our expectations are met. Independent of the considered amount of traffic, processing times for e.g. combination *13* is much less than it would be by summing up times for contained combinations *0*, *1*, *9* and *11*. Furthermore, for even more comprehensive combinations (e.g. *16* and *18*), the synergy effects compared to summing up all the contained combinations grow even bigger.

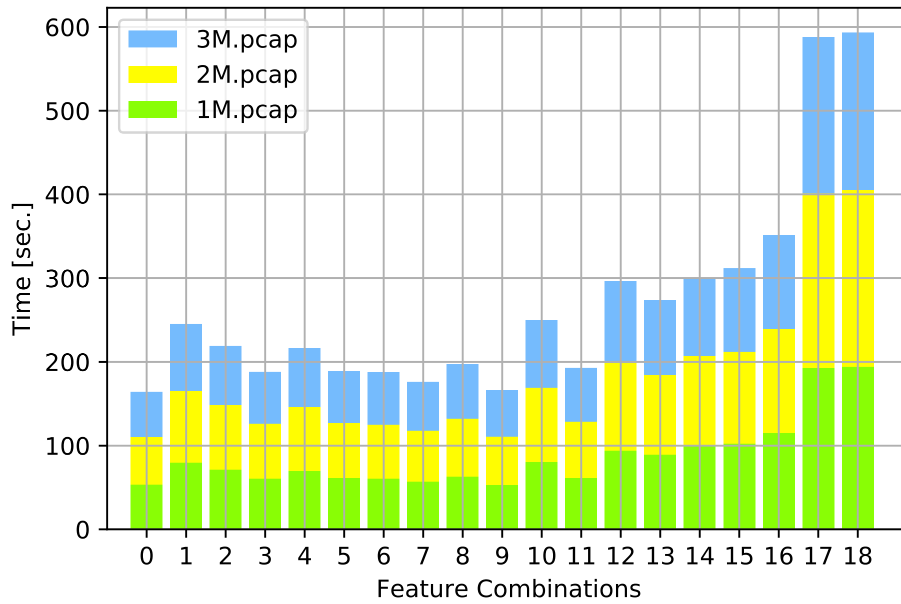

Figure 7.5: Median extraction times for all measured feature combinations for specific network traffic

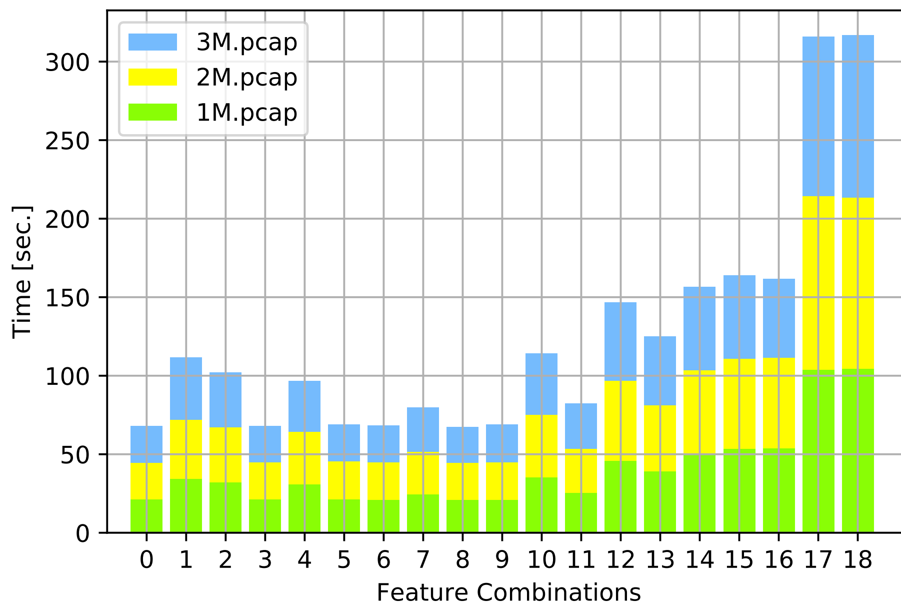

Figure 7.6: Median transformation times for all features' generated *Bro* log files of all measured feature combinations for specific network traffic

Concluding this subsection, we can infer from our measurements shown and explained above that FHS-DNA behaves as expected, intended and required and does not induce any unclear or hardly to explain surprises with respect to processing time.

Disk Space

In order to evaluate our system FHS-DNA also with respect to disk space and related requirements (e.g. NFR.9), we further explore the question how much disk space the final FS component would require for storing the original *Bro* log files compared to the actual *Parquet* files. This is especially interesting, as we can conclude from that how much disk space is saved by our decision to utilize the compressed *Parquet* format.

Hence, our expectations regarding this aspect are:

- Limited available disk space is preserved, as the *Bro* log files are not yet compressed and therefore require more disk space compared to the respective *Parquet* files.
- The bigger the *Bro* log files grow, the more disk space is preserved. That is, as each *Parquet* file requires some structural space, but for the same number of log files the more content is compressed, the more savings are gained in the end.
- Hence, for the same number of log files, the bigger the log files grow, the smaller is the respective fraction of appropriate *Parquet* files divided by the *Bro* log files.

Thus, Figure 7.7 shows the growing network traffic in 100,000 packets on the x-axis and the median sizes of all summed up files in GB on the y-axis. Also, on the y-axis we find the percentage for the median fraction of all summed up *Parquet* files divided by all summed up *Bro* logs. So, Figure 7.7 depicts the bar plots for the median sizes of all summed up *Bro* logs and the respective *Parquet* files, as well as the line plot for the evolution of the median fraction. Everything is for the feature combination *18* only, because this is the most comprehensive one and therefore serves as an upper bound. That means, no other feature combination yields more or bigger files (both *Bro* logs and *Parquet* files).

Thus, it can clearly be seen that all our expectations are met again. First of all, *Parquet* files preserve a lot of disk space due to their compression in comparison to *Bro* logs. Second, the more packets are monitored, the bigger grow the *Bro* logs. However, the respective *Parquet* files grow significantly slower, so that even more disk space is preserved. That is especially true, as for each amount of traffic, we always have the same number of generated *Bro* logs and *Parquet* files, as the feature combination always is number *18*. Hence, no additional *Parquet* files' overhead is needed. Hence,

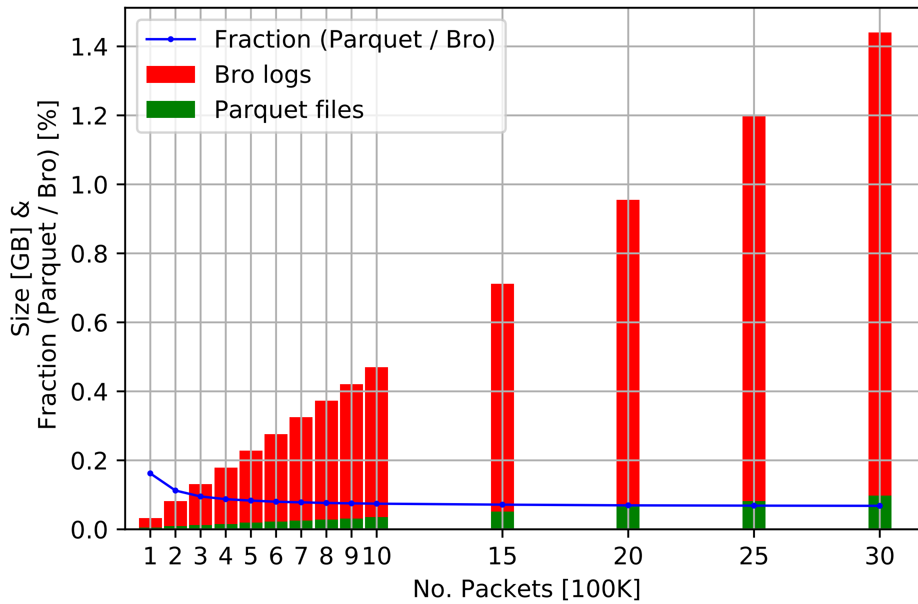

Figure 7.7: Median sizes of all summed up features' generated *Bro* log files and respectively transformed *Parquet* files and the median fraction of them for most comprehensive feature combination *18*

the median fraction of all summed up *Parquet* files divided by all summed up *Bro* logs continually decreases asymptoticly. Thus, the behavior generally is as expected, whereas the asymptotic part of that fractional observation is suspected to be based on the specific organization of *Parquet* files.

Concluding this subsection, we overall can infer from our measurements shown and explained above that our system FHS-DNA in general behaves as expected, intended and required (especially with respect to NFR.9).

Chapter 8

Requirements Assessment

Within this chapter, implementation of high-level requirements introduced in Section 3.2 are valuated. For that purpose, this chapter is organized in the same structure. In Section 8.1, FRs are evaluated, whereas in Section 8.2 NFRs of different subcategories are judged.
Generally, integrating a *Bro "normal" cluster* into FHS-DNA (as described in Section 6.1) ensures the actual FE approach working in a decentralized way. By explicitly specifying which network components and interfaces should be part of the monitoring cluster in *Bro's node.cfg* file, only the relevant network traffic is monitored. Moreover, the *Bro "normal" cluster* guarantees coordination among its clustered components as well as their easy management (e.g. starting and stopping via the *BroControl* interface) by default. Hence, the overall functional layout of our system FHS-DNA works in a coordinated, manageable and decentralized way.

8.1 Functional Requirements

FRs represent the basic functionality that our system should have.

FR.1: Feature Extraction from Network Communication Traffic: Integrating a *Bro "normal" cluster* into FHS-DNA allows decentralized FE as already explained above. Besides the default logs typically generated by *Bro*, which already contain a couple of basic features and even some further information, extraction of arbitrary features (and additional meta information) being implementable with the Turing-complete *Bro scripting language* [28] [30] is possible. That especially refers to all three vertical levels

of abstraction (PL, CL, NL) as described in Subsection 2.3.2. Examples for manually implemented features from all those levels of abstraction being extracted in our PoC implementation are listed in Table 6.1 and explained in Subsection 6.1.4.

FR.2: Processing of Previously Extracted Features: In our system's architecture we decided to include an external *Feature Processor and Storage Component* (see Subsection 5.2.2), implementing the cohesive processing and storage concept as argued in Subsections 3.3.4 and 5.1.4. Facilitating this external and therefore not resource-constrained component with *BAT* as explained in Section 6.2 allows the transformation of intermediate *Bro* log files into the final, generic storage format *Parquet* as described in Subsection 6.2.3. In addition, *BAT* also provides comprehensive analysis and statistical aggregation opportunities on those log files (see [46] and [76]), which are therefore also possible in our system.

FR.3: Storage of Previously Extracted and Processed Features: The appropriateness of *Parquet* as storage format is explained in Subsection 6.2.2. After the transformation of *Bro's* intermediate log files to *Parquet* format, the resulting *Parquet* files containing all the features and additionally logged meta information are stored directly in the file system of the single, external *Feature Processor and Storage Component* in a hierarchic folder structure as explained in Subsection 6.2.4. Updating this storage is implemented as described in Subsection 6.2.5.

8.2 Non-Functional Requirements

As NFRs do not specify functionality but rather describe the constraints of our system, their assessment is sporadically closely related to the evaluation as explored in Section 7.2.

8.2.1 General Claims

NFRs within this class affect more than just one single aspect of our entire system.

NFR.1: Ease of Integration onto Devices Inside the NsUS: Hosts and interfaces of the NsUS, which should e.g. monitor traffic and extract features from it, are defined in the *Bro* file *node.cfg*. Having that done, no additional devices need to be integrated into the network, nor further installations (except the initial setup of course) need to be made.

As long as those defined devices fulfill the prerequisites for operating in *Bro "normal" cluster* mode as described in [32], the traffic monitoring and FE process works just fine on the networking devices themselves.

NFR.2: Explicit Applicability on Network Communication Data: Integrating *Bro* as a network traffic monitoring and FE tool into our system guarantees capabilities of handling communication data as proven in our PoC implementation with the manually extracted features listed in Table 6.1 and explained in Subsection 6.1.4.

NFR.3: Generality of Interfaces in Their Utilized Technologies: In our system, both interfaces for intermediate log files' synchronization as well as dynamic reconfiguration are realized with *SSH* as described in Figures 6.6 and 6.7. *SSH* is a broadly accepted, generic and well-established concept for remote access. That way, in principle any component can e.g. be added or removed as an external *Configuration Component* in case it is trusted or not. For the interface to manage the monitoring cluster, the default central management framework of *Bro* - *BroControl* - is utilized. Furthermore, the generality of the implemented storage format is guaranteed, as *Parquet* is a generic and compressed data format as explained in Subsection 6.2.2, which therefore in principle allows easy utilization by various subsequent (A)DMs.

NFR.4: Applicability on General Networks, Topologies and Industries: Another requirement is to allow applicability of FHS-DNA to general types of networks, topologies and industries. This is ensured, as monitoring NsUS as defined in *Bro's node.cfg* file is in no way restricted to any of those means. Summarizing, our system FHS-DNA especially considers characteristics of OBNs in cars and aircraft as described in Section 2.2, but is not restricted in its application to OBNs, specific industries or underlying topologies.

NFR.5: Automation of Entire Feature Handling Process from Extraction To Storage: Once the system is correctly set up and all prerequisites are fulfilled, FHS-DNA works completely autonomous. The integrated *Bro "normal" cluster* monitors whenever it sees traffic on the respective devices' interfaces and extracts, collects and logs features as explained in Sections 5.1 and 6.1. The external *Feature Processor and Storage Component* synchronizes those centrally collected log files in a pre-specified time interval (e.g. every

five minutes) with a *cron*[1] job as described in Subsections 5.2.2 and 6.3.1. Afterwards, it automatically processes the features as specified (see Subsection 6.2.3) and updates the storage (see Subsection 6.2.5). No interference with an operator is necessary except for the dynamic reconfiguration, which explicitly functions on demand as explained in Subsection 6.3.2.

8.2.2 Feature Extraction

In the following, NFRs regarding FE only are evaluated.

NFR.6: Reconfigurability of Monitoring Agents and Their Handled Features: Reconfigurability of our system spans two different aspects. First, the devices and their interfaces being monitored in the network should be configurable. This is possible statically (i.e. when FHS-DNA is not running) via the *node.cfg* file of *Bro* as described in Subsection 5.3.3. Second, dynamic reconfiguration of different features being extracted from various agents is designed and implemented as explained in Subsections 5.2.2, 5.3.3 and 6.3.2.

NFR.7: Extensibility of Features Handled: As we integrated *Bro* as the traffic monitoring and FE tool, all features that can be implemented with the Turing-complete *Bro scripting language* are in principle extractable [28], as long the traffic actually contains those features. The manually implemented scripts for additional FE only need to be loaded during the next startup of FHS-DNA or via (previously statically prepared) dynamic reconfiguration of the integrated monitoring process as explained in Subsections 5.2.2, 5.3.3 and 6.3.2.

8.2.3 Feature Storage

The last two evaluated NFRs refer to FS only.

NFR.8: Ability to Query and Combine Stored Features by Subsequent Detection Models: Regarding the final FS concept, we decided for a cohesive *Feature Processor and Storage Component* (see Subsections 5.1.4 and 5.2.2), saving features in generic *Parquet* files directly in the file system of that component (see Subsection 6.2.4). As explained

[1] https://wiki.ubuntuusers.de/Cron/

and referenced in Subsections 6.2.2 and 6.2.4, those hierarchic organized *Parquet* files on disk containing the features can then in general easily be queried, imported and combined (e.g. via various joins) in meaningful ways by different (A)DMs.

NFR.9: Storage Format Efficiency in Disk Space and Query Performance: Regarding the first aspect of the storage format supporting compression and therefore being disk space efficient, we have seen that finally storing network traffic features in compressed *Parquet* format (see Subsection 6.2.2) provides enormous disk space savings as explored in Subsection 7.2.2. Regarding the second aspect, *Parquet* format is known to offer query efficiency per default [47] [75] [83], especially for large-scale queries [74] (further see Subsection 6.2.2). Moreover, even the fact that *Parquet* is a columnar storage format suits our feature context very well and contributes to efficient query performance (as single columns may also be considered as basic features and the ability to query on them instead of rows yields efficiency advantages as explained in Section 5.4). This is further supported (in close relation to NFR.8) by the hierarchic feature storage concept directly in the file system of an external component, from where (A)DMs can easily query and combine results (as explained in Subsections 6.2.2 and 6.2.4).

Chapter 9

Conclusion

Within this chapter, we summarize the results and contributions of our work in Section 9.1. Finally concluding this thesis, we state possible future work in Section 9.2.

9.1 Main Results and Contributions

Outgoing from the underlying problematic that a central point for traffic monitoring and feature extraction is not available in strictly hierarchical, decentralized and compartmentalized networks, we design and implement a system called FHS-DNA (for **F**eature **H**andling **S**ystem on **D**ecentralized **N**etwork **A**gents). General main purpose of that system is to provide a flexibly extensible and configurable framework for extraction, processing and storage of valuable subsets of features. Although those nets are characteristic to on-board networks in cars or aircraft (and therefore typically also closed and static), our system is thoroughly deployable to other networks, topologies and industries as well.

By not only extracting features, but also relevant meta data, FHS-DNA reduces loss of information while traffic monitoring to a reasonable minimum whenever possible. Additionally, it provides subsequent processing of extracted features. This for instance includes the ability to create new features out of the already extracted ones, utilizing analysis functions such as statistical aggregations, transformation of extracted features' log files into a compressed format and updating the corresponding storage. By offloading all that feature processing from the monitored devices, we further preserve resources for on-agent handling of traffic peaks. That is especially helpful, as FHS-DNA's monitoring capabilities are required to be deployed directly onto the devices of the network(s) under

surveillance and therefore resources are limited. These also play an important role for the third aspect of our system - appropriate feature storage. In order to let subsequent anomaly detection models comfortably query and combine previously extracted (and processed) features, storage format and concept need to be efficient. Thus, features are stored in a compressed, but large-scale query efficient, column-based format suitable to the means of subsequent, feature-based anomaly detection.

Altogether, FHS-DNA offers an outstanding set of combined functionality and properties compared to related work as described in Chapter 4. As partially already induced above, our work's main contributions include:

1. Developing a decentralized feature handling approach, which extracts features (e.g. from live traffic) directly on the agents of the monitored (sub-)network(s) by utilizing their unused resources.
2. Considering features of three different levels of abstraction (packet, connection and network).
3. Inclusively providing further processing options for extracted features.
4. Allowing high reconfigurability - especially with respect to dynamically changeable feature extraction - of FHS-DNA's monitoring process.
5. Offering great extensibility with respect to extracted features and participating monitoring agents.
6. Respecting leading-edge insights, like e.g. the depending valuation of feature subsets for subsequent detection models and effects of reduction.
7. Integrating and combining different existing tools (*The Bro Network Security Monitor* and *Bro Analysis Tools*) to a fine-tuned synthesis.
8. Providing high generality with respect to the implementation of interface communication, stored features' usability by subsequent anomaly detection models and operational networking aspects (e.g. underlying topologies).

Hence, all research goals and related questions as stated in Section 1.2 are met and answered by the developed system together with this thesis.

9.2 Future Work

Concluding, there are a couple of options for future work. That further underlines the actual relevance of a system like FHS-DNA.

1. More features may be implemented to directly extract from monitored network traffic.

2. Construction of additional features (e.g. statistical aggregations) may be conducted with provided processing options instead of directly extracting them from traffic.

3. Some configuration rules may be integrated. For instance, it should always be automatically guaranteed by the system that each feature generating *Interface-Specific Sliding Window Logs* is not extracted from more than one interface at a time.

4. Mechanisms for automatically limiting the number to one cohesive, external *Feature Processor and Storage Component* may be developed.

5. Contextual relevant IT security guidelines and data privacy aspects may be incorporated.

Chapter A

Appendix

A.1 Implementation

A.1.1 Overview of Manually Constructed Intermediate Log Files

Table A.1: Information logged for feature *land*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A timestamp (in order to enable transformation of log file to Parquet format)	time	M
srcHost	The source host IP address of a new connection	addr	M
destHost	The destination host IP address of a new connection	addr	M
srcPort	The source port of a new connection	port	M
destPort	The destination port of a new connection	port	M
flag	The flag for the "land" feature. If source and destination IP addresses and port numbers of a connection are the same, then this variable takes value 1, else 0	count	M

Table A.2: Information logged for feature *synack*

Field	**Description**	**Type**	**Manda-tory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	O
originHost	The origin host, for which the TCP syn-synack-time is calculated	addr	M
synSynAckInSecs	The TCP syn-synack-time of the respective host in secs	double	M

Table A.3: Information logged for feature *ct_src_ltm*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	O
srcHost	The source host	addr	O
number	The count of connections from the same source host within the last 100 connections	count	O

Table A.4: Information logged for feature *ct_srv_dst*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	O
destHost	The destination host	addr	O
sry	The service of the connections to the destination host, for which the count is calculated	string	O
number	The count of connections to the same destination host and with the same service within the last 100 connections	count	O

TABLE A.5: Information logged for feature *ct_dst_sport_ltm*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	O
destHost	The destination host	addr	O
srcPort	The source port	port	O
number	The count of connections to the same destination host and with the same source port within the last 100 connections	count	O

TABLE A.6: Information logged for feature *count*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	The timestamp for the calculated count	time	M
destHost	The destination host, for which the count is calculated	addr	M
number	The count of connections to the same destination host as the current connection's within the next two seconds	double	M

TABLE A.7: Information logged for feature *raw_layer2_packet_header*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
encap	L2 link encapsulation	link_encap	M
len	Total frame length on wire	count	M
cap_len	Captured length	count	M
src	L2 source (if Ethernet)	string	O
dst	L2 destination (if Ethernet)	string	O
vlan	Outermost VLAN tag if any (and Ethernet)	count	O
inner_vlan	Innermost VLAN tag if any (and Ethernet)	count	O
eth_type	Innermost Ethertype (if Ethernet)	count	O
proto	L3 protocol	layer3_proto	M

Table A.8: Information logged for feature *raw_IPv4_packet_header*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
hl	Header length in bytes	count	M
tos	Type of service	count	M
len	Total length	count	M
id	Identification	count	M
ttl	Time to live	count	M
p	Protocol	count	M
src	Source address	addr	M
dst	Destination address	addr	M

Table A.9: Information logged for feature *raw_IPv6_packet_header*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
class	Traffic class	count	M
flow	Flow label	count	M
len	Payload length	count	M
nxt	Protocol number of the next header (RFC 1700 et seq., IANA assigned number) e.g. IPPROTO_ICMP	count	M
hlim	Hop limit	count	M
src	Source address	addr	M
dst	Destination address	addr	M

Table A.10: Information logged for feature *raw_TCP_packet_header*

Field	**Description**	**Type**	**Mandatory / Optional**
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
sport	Source port	port	M

Field	Description	Type	Mandatory / Optional
dport	Destination port	port	M
seq	Sequence number	count	M
ack	Acknowledgment number	count	M
hl	Header length (in bytes)	count	M
dl	Data length (xxx: not in original tcphdr!)	count	M
flags	Flags	count	M
win	Window	count	M

Table A.11: Information logged for feature *raw_UDP_packet_header*

Field	Description	Type	Mandatory / Optional
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
sport	Source port	port	M
dport	Destination port	port	M
ulen	Udp length	count	M

Table A.12: Information logged for feature *raw_ICMP_packet_header*

Field	Description	Type	Mandatory / Optional
ts	A pseudo timestamp (in order to enable transformation of log file to Parquet format)	time	M
icmp_type	Type of message	count	M

A.1.2 Extracts of Exemplary Intermediate Log Files

```
#separator \x09
#set_separator  ,
#empty_field    (empty)
#unset_field    -
#path   feature-extraction-UNSWNB15-34
#open   2018-07-19-13-45-47
#fields ts      originHost      synSynAckInSecs
#types  time    addr    double
-       192.168.1.104   0.023531
-       192.168.1.104   0.092778
-       192.168.1.104   0.023544
-       192.168.1.104   0.023542
-       192.168.1.104   0.023541
-       192.168.1.104   0.092998
-       192.168.1.104   0.02328
-       192.168.1.102   0.176683
-       192.168.1.102   0.168706
-       192.168.1.102   0.185182
-       192.168.1.102   0.174183
-       192.168.1.102   0.168941
-       192.168.1.102   0.170695
-       192.168.1.102   0.174941
-       192.168.1.103   0.167958
-       192.168.1.103   0.168189
-       192.168.1.104   0.167452
-       192.168.1.104   0.296356
-       192.168.1.104   0.175943
-       192.168.1.104   0.212203
-       192.168.1.104   0.170191
-       192.168.1.104   0.193171
-       192.168.1.104   0.167957
-       192.168.1.103   0.172935
-       192.168.1.103   0.172429
-       192.168.1.103   0.170707
-       192.168.1.103   0.168195
-       192.168.1.103   0.166956
-       192.168.1.103   0.171448
-       192.168.1.103   0.198417
-       192.168.1.102   0.174204
-       192.168.1.102   0.352581
```

Figure A.1: Exemplary extract of an intermediate *General Log*, generated by script *feature-extraction-UNSWNB15-34.bro* for feature *synack*

```
#separator \x09
#set_separator  ,
#empty_field    (empty)
#unset_field    -
#path   feature-extraction-UNSWNB15-46
#open   2018-07-19-13-56-16
#fields ts      destHost        srcPort number
#types  time    addr    port    count
-       192.168.1.1     68      1
-       -       -       -
-       192.168.1.1     68      1
-       192.168.1.255   137     1
-       -       -       -
-       192.168.1.1     68      1
-       192.168.1.255   137     2
-       -       -       -
-       192.168.1.255   138     1
-       192.168.1.1     68      1
-       192.168.1.255   137     2
-       -       -       -
-       192.168.1.255   138     2
-       192.168.1.1     68      1
-       192.168.1.255   137     2
-       -       -       -
-       192.168.1.255   138     2
-       192.168.1.1     68      1
-       192.168.1.255   137     3
-       -       -       -
-       192.168.1.255   138     3
-       192.168.1.1     68      1
-       192.168.1.255   137     3
-       -       -       -
-       192.168.1.255   138     3
-       192.168.1.1     68      2
-       192.168.1.255   137     3
-       -       -       -
-       192.168.1.255   138     4
-       192.168.1.1     68      2
-       192.168.1.255   137     3
-       -       -       -
```

Figure A.2: Exemplary extract of an intermediate *Interface-Specific Sliding Window Log*, generated by script *feature-extraction-UNSWNB15-46.bro* for feature *ct_dst_sport_ltm*

A.2 Testing and Evaluation

A.2.1 Statistics of exercise-traffic.pcap

Table A.13: Protocol hierarchy of *exercise-traffic.pcap*

Protocol	**Packet Portion in %**	**Count of Packets**
Ethernet	100	100,000
Link Layer Discovery Protocol	1.2	1,235
Internet Protocol Version 6	0.2	223
User Datagram Protocol	0.2	200
Multicast Domain Name System	0.1	58
Link-local Multicast Name Resolution	0.0	16
DHCPv6	0.1	116
Data	0.0	10
Internet Control Message Protocol v6	0.0	23
Internet Protocol Version 4	98.0	98,002
User Datagram Protocol	5.7	5,692
Simple Service Discovery Protocol	0.0	40
Network Time Protocol	0.0	1
NetBIOS Name Service	1.2	1,157
NetBIOS Datagram Service	0.4	442
SMB (Server Message Block Protocol)	0.4	442
SMB MailSlot Protocol	0.4	442
Microsoft Windows Browser Protocol	0.4	442
Multicast Domain Name System	0.1	58
Link-local Multicast Name Resolution	0.0	12
Domain Name System	2.6	2,596
Data	1.2	1,246
Bootstrap Protocol	0.1	140
Transmission Control Protocol	92.3	92,310
Simple Mail Transfer Protocol	0.1	65
Secure Sockets Layer	0.9	889
Malformed Packet	0.0	5
Hypertext Transfer Protocol	5.3	5,277
Text item	0.0	6
Portable Network Graphics	0.1	122
Online Certificate Status Protocol	0.0	6

Protocol	Packet Portion in %	Count of Packets
MSN Messenger Service	0.0	4
MIME Multipart Media Encapsulation	0.0	8
Media Type	0.2	212
Line-based text data	0.8	763
JPEG File Interchange Format	0.3	322
JavaScript Object Notation	0.0	13
HTML Form URL Encoded	0.0	20
eXtensible Markup Language	0.2	172
Compuserve GIF	0.5	459
Data	0.0	2
Address Resolution Protocol	0.5	540

Table A.14: Conversations of *exercise-traffic.pcap*

Protocol	Count of Conversations
Ethernet	20
IPv4	419
IPv6	6
TCP	1,278
UDP	1,311

Table A.15: Endpoints of *exercise-traffic.pcap*

Protocol	Count of Endpoints
Ethernet	18
IPv4	315
IPv6	8
TCP	1,590
UDP	1,325

Table A.16: Packet Lengths of *exercise-traffic.pcap*

Range	Count of Packets	Average Length	Min. Value	Max. Value
0 - 19	0	-	-	-
20 - 29	0	-	-	-
40 - 79	33,092	63.84	42	79
80 - 159	6,737	119.51	80	159

Range	**Count of Packets**	**Average Length**	**Min. Value**	**Max. Value**
160 - 319	2,716	236.82	160	319
320 - 639	3,078	447.63	320	639
640 - 1,279	2,359	921.56	640	1,279
1,280 - 2,559	52,018	1,433.20	1,280	1,518
2,560 - 5,119	0	-	-	-
>= 5,120	0	-	-	-

Chapter B

List of acronyms

ACID	Atomicity, Consistency, Isolation, Durability.
(A)DM	(Anomaly) Detection Model.
AMD	Advanced Micro Devices.
API	Application Programming Interface.
ASCII	American Standard Code for Information Interchange.
BAT	Bro Analysis Tools.
CIF	Collective Intelligence Framework.
CL	Connection Level. The middle of three considered vertical feature abstraction levels.
CPU	Central Processing Unit.
DecADe	Decentralized Anomaly Detection. The contextual joint research project of this master thesis.
DHCPv6	Dynamic Host Configuration Protocol Version 6.
FE	Feature Extraction. A task for capturing characteristics from network traffic.
FH	Feature Handling. Multiple single tasks summarized regarding characteristics of network traffic.
FHS-DNA	Feature Handling System on Decentralized Network Agents. The final system developed by the authors.
FP	Feature Processing. A task for treating characteristics of network traffic.
FS	Feature Storage. A task for saving characteristics of network traffic.

GB	Gigabyte.
GHz	Gigahertz.
GIF	Graphics Interchange Format.
GmbH	Gesellschaft mit beschränkter Haftung.
HTML	Hypertext Markup Language.
ICMP	Internet Control Message Protocol.
I/O	Input / Output.
IP	Internet Protocol.
IPv4	Internet Protocol Version 4.
IPv6	Internet Protocol Version 6.
ISO	International Organization for Standardization.
IT	Information Technology.
JPEG	Joint Photographic Experts Group.
JSON	JavaScript Object Notation.
MIME	Multipurpose Internet Mail Extensions.
ML	Machine Learning.
MSN	Microsoft Network.
NetBIOS	Network Basic Input / Ouput System.
(N)FR	(Non-)Functional Requirement.
(N)IDS	(Network) Intrusion Detection System. A computer system to detect (network) intrusions.
NL	Network Level. The highest of three considered vertical feature abstraction levels.
NRDBMS	Non-Relational Database Management System.
NUS	(Sub-)Network Under Surveillance.
OBN	On-Board Network. A computer network on board of e.g. an aircraft or car.
OSI	Open Systems Interconnection. Reference model for layered network architectures by the OSI.
pcap	Packet Capture. A file format to store network traffic.
PL	Packet Level. The lowest of three considered vertical feature abstraction levels.
PoC	Proof of Concept.

(R)DBMS	(Relational) Database Management System.
REST	Representational State Transfer.
SIEM	Security Information and Event Management.
SMB	Server Message Block.
SOME/IP	Scalable Service-Oriented Middleware over IP.
SSD	Solid State Drive.
SSH	Secure Shell.
SYN	Synchronization.
SYN_ACK	Synchronization_Acknowledgment.
TCP	Transmission Control Protocol. Stream-oriented, reliable, transport layer protocol.
UDP	User Datagram Protocol. Datagram-oriented, unreliable transport layer protocol.
URL	Uniform Resource Locator.
VAST	Visibility Across Space and Time.
VM	Virtual Machine.

Bibliography

[1] TUM and DecADe Consortium. *DecADe BMBF Project. Dezentrale Anomalieerkennung.* [Online]. 2016. URL: https://www.net.in.tum.de/sites/decade/# (visited on 04/12/2018).

[2] P. García-Teodoro et al. "Anomaly-based network intrusion detection: Techniques, systems and challenges". In: *Computers & Security* 28.1-2 (Feb. 2009), pp. 18–28. DOI: 10.1016/j.cose.2008.08.003.

[3] Nour Moustafa and Jill Slay. "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)". In: *2015 Military Communications and Information Systems Conference (MilCIS)*. IEEE, Nov. 2015. DOI: 10.1109/milcis.2015.7348942.

[4] Dipali G. Mogal, Sheshnaryan R. Ghungrad, and Bapusaheb B. Bhusare. "A Review on High Ranked Features based NIDS". In: *IJARCCE* 6.3 (Mar. 2017), pp. 349–353. DOI: 10.17148/ijarcce.2017.6380.

[5] M. Segvic, K. Krajcek, and E. Ivanjko. "Technologies for distributed flight control systems: A review". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, May 2015. DOI: 10.1109/mipro.2015.7160432.

[6] DecADe Consortium. *Decentralized Anomaly Detection. Dezentrale und autonome Anomalieerkennung durch ungenutzte Rechenkapazitäten.* [Online]. Feb. 2017. URL: https://www.net.in.tum.de/sites/decade/img/project_poster_feb17.pdf (visited on 04/12/2018).

[7] "Glossary of Terms". In: *Mach. Learn.* 30.2-3 (Feb. 1998), pp. 271–274. ISSN: 0885-6125. URL: http://dl.acm.org/citation.cfm?id=288808.288815.

[8] Nour Moustafa and Jill Slay. "The Significant Features of the UNSW-NB15 and the KDD99 Data Sets for Network Intrusion Detection Systems". In: *2015 4th International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, Nov. 2015. DOI: 10.1109/badgers.2015.014.

[9] Tharmini Janarthanan and Shahrzad Zargari. “Feature selection in UNSW-NB15 and KDDCUP’99 datasets”. In: *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*. IEEE, June 2017. DOI: 10.1109/isie.2017.8001537.

[10] Nour Moustafa and Jill Slay. “The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set”. In: *Information Security Journal: A Global Perspective* 25.1-3 (Jan. 2016), pp. 18–31. DOI: 10.1080/19393555.2015.1125974.

[11] W. Haider et al. “Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling”. In: *Journal of Network and Computer Applications* 87 (June 2017), pp. 185–192. DOI: 10.1016/j.jnca.2017.03.018.

[12] Nour Moustafa and Jill Slay. “A hybrid feature selection for network intrusion detection systems: Central points”. In: (2015). DOI: 10.4225/75/57a84d4fbefbb.

[13] Dai Hong and Li Haibo. “A Lightweight Network Intrusion Detection Model Based on Feature Selection”. In: *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, Nov. 2009. DOI: 10.1109/prdc.2009.34.

[14] Khalil El-Khatib. “Impact of Feature Reduction on the Efficiency of Wireless Intrusion Detection Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.8 (Aug. 2010), pp. 1143–1149. DOI: 10.1109/tpds.2009.142.

[15] Dipali Gangadhar Mogal, Sheshnarayan R. Ghungrad, and Bapusaheb B. Bhusare. “NIDS using Machine Learning Classifiers on UNSW-NB15 and KDDCUP99 Datasets”. In: *IJARCCE* 6.4 (Apr. 2017), pp. 533–537. DOI: 10.17148/ijarcce.2017.64102.

[16] Herve Nkiama, Syed Zainudeen, and Muhammad Saidu. “A Subset Feature Elimination Mechanism for Intrusion Detection System”. In: *International Journal of Advanced Computer Science and Applications* 7.4 (2016). DOI: 10.14569/ijacsa.2016.070419.

[17] Nour Moustafa and Jill Slay. “Creating Novel Features to Anomaly Network Detection Using DARPA-2009 Data set”. en. In: (2015). DOI: 10.13140/rg.2.1.1993.7765.

[18] Xiaming Chen. *GitHub - caesar0301/awesome-pcaptools: A collection of tools developed by other researchers in the Computer Science area to process network traces. All the right reserved for the original authors.* [Online]. 2018. URL: https://github.com/caesar0301/awesome-pcaptools (visited on 07/02/2018).

[19] Van Jacobson Craig Leres and Steven McCanne. *Manpage of TCPDUMP*. [Online]. 2018. URL: http://www.tcpdump.org/manpages/tcpdump.1.html (visited on 07/02/2018).

[20] Wireshark Foundation. *Chapter 1. Introduction*. [Online]. URL: `https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html#` (visited on 06/14/2018).

[21] Syn Fin dot Net. *Tcpreplay*. [Online]. 2013. URL: `http://tcpreplay.synfin.net/` (visited on 07/02/2018).

[22] Philippe Biondi and the Scapy community. *Scapy*. [Online]. 2018. URL: `https://scapy.net/` (visited on 06/14/2018).

[23] Martin Roesch et al. *SNORT Users Manual 2.9.11. The Snort Project*. [Online]. 2014. URL: `http://manual-snort-org.s3-website-us-east-1.amazonaws.com/` (visited on 06/14/2018).

[24] M.Ali Aydin, A Zaim, and K Gökhan Ceylan. "A hybrid intrusion detection system design for computer network security". In: 35 (May 2009), pp. 517–526.

[25] Matthias Vallentin. *Network Intrusion Detection & Forensics. with Bro*. [Online]. Mar. 2016. URL: `https://matthias.vallentin.net/slides/berke1337-bro-intro.pdf` (visited on 06/05/2018).

[26] Thomas Joos. *Intrusion-Detection-System Snort: Netzwerke wirkungsvoll absichern - TecChannel Workshop*. [Online]. 2015. URL: `https://www.tecchannel.de/a/netzwerke-wirkungsvoll-absichern,3277633#s1` (visited on 06/14/2018).

[27] Cisco and/or its affiliates. *README.reputation*. [Online]. 2018. URL: `https://www.snort.org/faq/readme-reputation` (visited on 06/14/2018).

[28] The Bro Project. *Introduction - Bro 2.5.4 documentation*. [Online]. 2016. URL: `https://www.bro.org/sphinx/intro/index.html` (visited on 06/14/2018).

[29] Vern Paxson. "Bro: a system for detecting network intruders in real-time". In: *Computer Networks* 31.23-24 (Dec. 1999), pp. 2435–2463. DOI: `10.1016/s1389-1286(99)00112-7`.

[30] The Bro Project. *Script Reference - Bro 2.5.4 documentation*. [Online]. 2016. URL: `https://www.bro.org/sphinx/script-reference/index.html` (visited on 07/10/2018).

[31] The Bro Project. [Online]. 2014. URL: `https://www.bro.org/development/projects/deep-cluster.html` (visited on 06/18/2018).

[32] The Bro Project. *BroControl - Bro 2.5.4 documentation*. [Online]. 2016. URL: `https://www.bro.org/sphinx/components/broctl/README.html` (visited on 06/18/2018).

[33] The Bro Project. *Bro Cluster Architecture - Bro 2.5.4 documentation*. [Online]. 2016. URL: `https://www.bro.org/sphinx/cluster/index.html` (visited on 06/14/2018).

[34] The Bro Project. *Cluster Configuration - Bro 2.5-668 documentation.* [Online]. 2016. URL: https://www.bro.org/sphinx-git/configuration/index.html (visited on 06/18/2018).

[35] Johanna Amann. *GitHub - 0xxon/bro-postgresql: Postgresql reader / writer plugin for Bro.* [Online]. 2018. URL: https://github.com/0xxon/bro-postgresql (visited on 06/14/2018).

[36] The Bro Project. *A Collection of Plugins for Bro - Bro 2.5.4 documentation.* [Online]. 2016. URL: https://www.bro.org/sphinx/components/bro-plugins/README.html (visited on 06/14/2018).

[37] O.S. Tezer and DigitalOcean™ Inc. *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems | DigitalOcean.* [Online]. 2014. URL: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems (visited on 06/14/2018).

[38] The Bro Project. *Logging To and Reading From SQLite Databases - Bro 2.5-660 documentation.* [Online]. 2016. URL: https://www.bro.org/sphinx-git/frameworks/logging-input-sqlite.html (visited on 06/14/2018).

[39] SQLite Consortium. *SQLite Home Page.* [Online]. URL: https://www.sqlite.org/index.html (visited on 06/14/2018).

[40] Inc. MongoDB. *What Is A Non Relational Database | MongoDB.* [Online]. 2018. URL: https://www.mongodb.com/scale/what-is-a-non-relational-database (visited on 07/02/2018).

[41] Elasticsearch. *Elasticsearch: RESTful, Verteilte Suche und Analyse | Elastic.* [Online]. 2018. URL: https://www.elastic.co/de/products/elasticsearch (visited on 06/14/2018).

[42] The Bro Project. *Indexed Logging Output with ElasticSearch - Bro 2.5.4 documentation.* [Online]. 2016. URL: https://www.bro.org/sphinx/components/bro-plugins/elasticsearch/README.html (visited on 06/14/2018).

[43] elastic. *GitHub - elastic/elasticsearch: Open Source, Distributed, RESTful Search Engine.* [Online]. 2018. URL: https://github.com/elastic/elasticsearch (visited on 06/14/2018).

[44] Travis Smith and Elasticsearch. *Integrating Bro IDS with the Elastic Stack | Elastic.* [Online]. 2016. URL: https://www.elastic.co/blog/bro-ids-elastic-stack (visited on 06/14/2018).

[45] LLC. Morpheus Data. *When Is Elasticsearch is the right tool for your Job | Morpheus.* [Online]. 2018. URL: https://www.morpheusdata.com/blog/2015-01-30-is-elasticsearch-the-right-solution-for-you (visited on 06/14/2018).

[46] Kitware Inc. *Bro Analysis Tools (BAT) - bat 0.3.4 documentation*. [Online]. 2017. URL: https://bat-tools.readthedocs.io/en/latest/ (visited on 07/10/2018).

[47] The Apache Software Foundation. *Parquet Format - Apache Drill*. [Online]. 2012. URL: https://drill.apache.org/docs/parquet-format/ (visited on 07/10/2018).

[48] Apache Software Foundation. *Apache Parquet*. [Online]. 2018. URL: http://parquet.apache.org/documentation/latest/ (visited on 07/10/2018).

[49] Nadine Herold et al. "Anomaly detection for SOME/IP using complex event processing". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 2016. DOI: 10.1109/noms.2016.7502991.

[50] Lars Völker. *Scalable service-Oriented MiddlewarE over IP (SOME/IP)*. [Online]. 2012. URL: http://some-ip.com/index.shtml (visited on 06/21/2018).

[51] Matthias Vallentin. "VAST: Network Visibility Across Space and Time". Master Thesis. Technische Universität München, Jan. 15, 2009. URL: https://matthias.vallentin.net/papers/thesis-msc.pdf (visited on 04/12/2018).

[52] Matthias Vallentin, Vern Paxson, and Robin Sommer. "VAST: A Unified Platform for Interactive Network Forensics". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 345–362. ISBN: 978-1-931971-29-4. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/vallentin.

[53] Matthias Vallentin. *VAST: Interactive Network Forensics*. [Online]. Aug. 2015. URL: https://www.bro.org/brocon2015/slides/vallentin_vast.pdf (visited on 04/12/2018).

[54] Matthias Vallentin. *VAST - Home*. [Online]. 2018. URL: http://vast.io/ (visited on 04/12/2018).

[55] CSIRT Gadgets. *Introduction · csirtgadgets/bearded-avenger-deploymentkit Wiki · GitHub*. [Online]. 2018. URL: https://github.com/csirtgadgets/bearded-avenger-deploymentkit/wiki/Introduction (visited on 06/21/2018).

[56] Ismael Valenzuela. *Open Security Research: Identifying Malware Traffic with Bro and the Collective Intelligence Framework (CIF)*. [Online]. 2014. URL: http://blog.opensecurityresearch.com/2014/03/identifying-malware-traffic-with-bro.html (visited on 06/21/2018).

[57] Luigi Coppolino et al. "Enhancing SIEM Technology to Protect Critical Infrastructures". In: *Critical Information Infrastructures Security*. Springer Berlin Heidelberg, 2013, pp. 10–21. DOI: 10.1007/978-3-642-41485-5_2.

[58] Sandeep Bhatt, Pratyusa K. Manadhata, and Loai Zomlot. "The Operational Role of Security Information and Event Management Systems". In: *IEEE Security & Privacy* 12.5 (Sept. 2014), pp. 35–41. DOI: 10.1109/msp.2014.103.

[59] Moukafih Nabil et al. "SIEM selection criteria for an efficient contextual security". In: *2017 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, May 2017. DOI: 10.1109/isncc.2017.8072035.

[60] Ahmad M Karimi et al. "Distributed network traffic feature extraction for a real-time IDS". In: *2016 IEEE International Conference on Electro Information Technology (EIT)*. IEEE, May 2016. DOI: 10.1109/eit.2016.7535295.

[61] Zongxing Xie et al. "A Distributed Agent-Based Approach to Intrusion Detection Using the Lightweight PCC Anomaly Detection Classifier". In: *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06)*. IEEE. DOI: 10.1109/sutc.2006.1636211.

[62] The Apache Software Foundation. *Spark SQL & DataFrames | Apache Spark*. [Online]. 2018. URL: https://spark.apache.org/sql/ (visited on 08/10/2018).

[63] The Apache Software Foundation. *Spark 2.3.1 ScalaDoc*. [Online]. URL: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset (visited on 08/10/2018).

[64] The Apache Software Foundation. *Spark SQL and DataFrames - Spark 2.3.1 Documentation*. [Online]. URL: http://spark.apache.org/docs/latest/sql-programming-guide.html (visited on 08/10/2018).

[65] Manuel Ehler and Marcel von Maltitz. *tumi8-theses / ehler_ma_distributedanomalydetection · GitLab*. [Online]. URL: https://gitlab.lrz.de/tumi8-theses/ehler_ma_distributedanomalydetection.git (visited on 08/24/2018).

[66] The Bro Project. *BroControl - Bro 2.5-660 documentation*. [Online]. 2016. URL: https://www.bro.org/sphinx-git/components/broctl/README.html (visited on 06/14/2018).

[67] Veronica Magan Woodrow Bellamy III and LLC Access Intelligence. *The Connected Aircraft: Beyond Passenger Entertainment and Into Flight Operations*. [Online]. 2014. URL: http://interactive.avionicstoday.com/the-connected-aircraft/ (visited on 06/14/2018).

[68] The Bro Project. *Log Files - Bro 2.5.4 documentation*. [Online]. 2016. URL: https://www.bro.org/sphinx/script-reference/log-files.html (visited on 08/14/2018).

[69] L. Dhanabal and Dr. S. P. Shantharajah. "A Study on NSL-KDD Dataset for Intrusion Detection System Based on Classification Algorithms". In: 2015.

[70] R. Sommer and V. Paxson. "Exploiting Independent State For Network Intrusion Detection". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE. DOI: 10.1109/csac.2005.24.

[71] The Bro Project. *Declarations and Statements — Bro 2.5.5 documentation*. [Online]. 2016. URL: https://www.bro.org/sphinx/script-reference/statements.html (visited on 08/27/2018).

[72] Apache Software Foundation. *Apache Parquet*. [Online]. 2018. URL: https://parquet.apache.org/ (visited on 07/10/2018).

[73] Silvia Oliveros and Silicon Valley Data Science LLC. *How to Choose a Data Format — Silicon Valley Date Science*. [Online]. 2016. URL: https://svds.com/how-to-choose-a-data-format/ (visited on 04/12/2018).

[74] Silvia Oliveros, Stephen O'Sullivan, and Andrew Ray. *Data Formats*. [Online]. URL: http://www.svds.com/dataformats/ (visited on 04/12/2018).

[75] Rajesh Dangi. *Hadoop File Formats, when and what to use? | Rajesh Dangi | Pulse | LinkedIn*. [Online]. June 2017. URL: https://www.linkedin.com/pulse/hadoop-file-formats-when-what-use-rajesh-dangi-pmp (visited on 04/12/2018).

[76] Brian Wylie and Python Software Foundation. *bat · PyPI*. [Online]. 2018. URL: https://pypi.python.org/pypi/bat (visited on 07/10/2018).

[77] Oracle and/or its affiliates. *Oracle VM VirtualBox*. [Online]. URL: https://www.virtualbox.org/ (visited on 07/11/2018).

[78] HashiCorp. *Introduction - Vagrant by HashiCorp*. [Online]. URL: https://www.vagrantup.com/intro/index.html (visited on 07/11/2018).

[79] HashiCorp. *Multi-Machine - Vagrant by HashiCorp*. [Online]. URL: https://www.vagrantup.com/docs/multi-machine/ (visited on 07/11/2018).

[80] Mininet Team. *Mininet Overview - Mininet*. [Online]. 2018. URL: http://mininet.org/overview/ (visited on 07/12/2018).

[81] Bob Lantz et al. *Introduction to Mininet · mininet/mininet Wiki · GitHub*. [Online]. 2018. URL: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet (visited on 07/12/2018).

[82] Mininet. *Documentation · mininet/mininet Wiki · GitHub*. [Online]. 2018. URL: https://github.com/mininet/mininet/wiki/Documentation (visited on 07/12/2018).

[83] Julien Le Dem. *How to use Parquet as a basis for ETL and analytics*. [Online]. Feb. 2015. URL: https://de.slideshare.net/julienledem/how-to-use-parquet (visited on 07/10/2018).