

Implementation of a framework for generating attack traces in Open RAN systems

BACHELOR THESIS

Bachelor of Science (B.Sc.) Computer Engineering
at the Faculty of Information, Media and Electrical Engineering
of the Cologne University of Applied Sciences.

Author: Johannes Franz Müller
Matriculation number: 11127581
Address: Steinackerstraße. 41
53840 Troisdorf
johannes_franz.mueller@smail.th-koeln.de

1st Examiner: Prof. Dr. Andreas Grebe
2nd Examiner: Arn Jonas Dieterich

Cologne, January 15, 2024

Abstract

The introduction of 5G and Open RAN opens up new attack vectors on telecommunications infrastructure for attackers. To prevent the exploitation of potential attack vectors, it is essential to analyse 5G and Open RAN systems for vulnerabilities in the context of prevention, detection, and treatment. This bachelor thesis aims to develop a framework for generating attack traces in Open RAN systems. It will be carried out as part of the 5G-FORAN project in collaboration with TH Köln and Procyde GmbH. The purpose of the framework is to simplify and automate the process of generating traces, so the attack traces can be analysed, to mitigate attack vectors.

Keywords: 5G, Open RAN, Kubernetes, Attack Simulation

Kurzfassung

Die Einführung von 5G und Open RAN eröffnet Angreifern neue Angriffsvektoren auf Telekommunikationsinfrastrukturen. Um die Ausnutzung dieser Angriffsvektoren zu verhindern, müssen 5G und Open RAN im Kontext von Prävention, Erkennung und Behandlung auf Schwachstellen untersucht werden. Diese Bachelorarbeit beschäftigt sich mit der Implementierung eines Frameworks zur Generierung von Angriffsvektoren in Open RAN-Systemen und wird im Rahmen des Projektes 5G-FORAN in Kooperation zwischen der TH Köln und der Procyde GmbH durchgeführt. Ziel des Frameworks ist es, die Generierung von Angriffsspuren zu vereinfachen und automatisieren, so dass diese anschließend analysiert werden können. Auf dieser Basis können Angriffsvektoren behoben oder weitere Maßnahmen ergriffen werden.

Stichwörter: 5G, Open RAN, Kubernetes, Angriffssimulation

Contents

Kurzfassung	I
Kurzfassung	II
List of Figures	VI
List of Listings	VIII
List of Tables	X
1. Introduction	1
1.1. Initial situation	1
1.2. Objective	2
1.2.1. Objective 5G-FORAN	2
1.2.2. Objective of the Bachelor's thesis within 5G-FORAN	2
1.3. Personal Motivation	3
1.4. Thesis Structure	4
1.5. Hints to Syntax	4
1.5.1. Module and Class Names	4
1.5.2. Methods, Variables, Files, Folder	5
2. Technical Background	6
2.1. General	6
2.1.1. 5G	6
2.1.2. Open RAN	7
2.1.3. Kubernetes	9
2.1.4. 5G-FORAN Lab Environment	10
2.1.5. MITRE ATT&CK Framework	11
2.1.6. Kubernetes ATT&CK Matrix	12
2.1.7. Common Vulnerabilities and Exposures	13
2.1.8. Attacker Perspectives	13

2.2.	CF-Framework	14
2.2.1.	MongoDB	14
2.2.2.	Ansible	15
2.2.3.	Prompt Toolkit	15
2.2.4.	Kubehunter	15
2.2.5.	Kdigger	16
2.2.6.	RedKube	16
3.	Solution Concepts	17
3.1.	Preparatory work	17
3.1.1.	Evaluation Architectural Approaches	17
3.1.2.	Evaluation Programming language	18
3.1.3.	Evaluation Metadata	19
3.1.4.	Evaluation of CF-Framework Features	20
3.1.5.	Evaluation of Automation Framework	21
3.2.	Framework Architecture	22
3.2.1.	General Architecture	22
3.2.2.	Controller	24
3.2.3.	Tool	25
3.2.4.	Menu	27
3.2.5.	Database	29
3.2.6.	State	29
3.2.7.	Environment	30
3.2.8.	Features	31
3.3.	Database Architecture	32
3.4.	Automation	33
3.5.	Test strategy	34
4.	Implementation	35
4.1.	Framework Architektur	35
4.1.1.	Controller	35
4.1.2.	Tool	38
4.1.3.	Menu	52
4.1.4.	Database	54
4.1.5.	State	56
4.1.6.	Environment	57
4.1.7.	Features	58
4.2.	Database Architecture	62

4.3. Automation	63
4.3.1. Deployment CF-Framework	63
4.3.2. Database	64
4.4. User Manual	67
5. Testing	69
5.1. Functional Testing CF-Framework	69
5.2. Functional Testing Tool	72
5.3. Features Testing	78
5.3.1. Campaign	78
5.3.2. Template	80
5.4. Testing Phase Categorisation	82
6. Summary and Outlook	83
6.1. Summary	83
6.1.1. Objectives	83
6.1.2. Personal Motivation	84
6.2. Outlook	84
6.2.1. Tools and Features	84
6.2.2. Attacker perspectives	85
6.2.3. Testing with DFIR tools	85
Appendix	86
A. Detailed CF-Framework Architecture	86
B. Detailed CF-Framework Architecture	88
Glossary	88
Acronyms	91
Bibliography	94
Selbstständigkeitserklärung	97

List of Figures

2.1. General Architecture 5GS [9]	6
2.2. Logical O-RAN Architecture [10]	7
2.3. O-RAN Structure in Kubernetes [Source: 5G-FORAN TH-Köln]	9
2.4. 5G-FORAN Lab Environment [Source: 5G-FORAN TH-Köln]	10
2.5. Kubernetes ATT&CK Matrix [17]	12
2.6. Hierarchy Attack Perspectives	13
3.1. Framework general Architecture	22
3.2. Structure of the controller module in conjunction with the external module	24
3.3. Structure of the tool module	25
3.4. Structure of the menu module	27
3.5. Example menu run	28
3.6. Structure of the database module	29
3.7. Structure of the state module	29
3.8. Structure of the environment module	30
3.9. Structure of the feature modules automation and campaign	31
3.10. Database Architecture	32
3.11. Framework and Database Automation with Ansible	33
4.1. Controller module - Files	35
4.2. Tool module - Files	39
4.3. Menu module - Files	52
4.4. DEV Collections	62
4.5. CF-Framework User manual	67
5.1. Example Usage of Commands: back, end, exit	70
5.2. Pre-parameterised methods Kubehunter	71
5.3. Pre-parameterised methods Kubehunter	72
5.4. Default Configuration of Kubehunter specific Parameter	73
5.5. Set Kubehunter specific Attack Parameter and launch Attack pod	73
5.6. Vulnerability analysis - Nodes	74

- 5.7. Vulnerability analysis - Services 75
- 5.8. Vulnerability analysis - Vulnerabilities 76
- 5.9. Database Entry for Vulnerability analysis 77
- 5.10. Feature - Campaign Usage 79
- 5.11. Database Entry in Collection artifacts-campaign 79
- 5.12. Campaign ID Field in Collection artifacts-raw 80
- 5.13. Feature - Template Usage 80
- 5.14. Demonstrator menu Entries 81
- 5.15. Set Kdigger specific Attack Parameter and launch Test Attack 82
- 5.16. Database Entry for Kdigger Test Attack 82

List of Listings

4.1. PromptInput class	36
4.2. MenuStateMachine class - Variables class	37
4.3. MenuStateMachine class - Run method	37
4.4. LocalEnv Base class	40
4.5. KubehunterEnv subclass - Parameter	40
4.6. Wrapper Base class class	41
4.7. KubehunterWrapper subclass - Variables and pre-parameterised methods	42
4.8. KubehunterWrapper subclass - Build command	43
4.9. Tool Base class - Variables	44
4.10. Tool Base class - Handle user input	45
4.11. Tool Base class - Insert into database	46
4.12. Tool Base class - Abstract Methods	47
4.13. Kubehunter subclass - Variables	48
4.14. Kubehunter subclass - Methods	49
4.15. KubehunterParser class	50
4.16. Menu Base class	52
4.17. Start subclass	53
4.18. DatabaseWrapper class - Establishing connection	54
4.19. DatabaseWrapper class - Write data	55
4.20. MenuState class	56
4.21. Environment class - Metadata variables	57
4.22. GlobalVariables class	58
4.23. Campaign class	58
4.24. Template class	60
4.25. AttackGenerator class	61
4.26. Ansible - Cloning the CF-Framework	63
4.27. AttackGenerator class	63
4.28. Ansible - Install MongoDB packages	64
4.29. Ansible - Add foran user to MongoDB	65

4.30. Ansible - Create Authentication Certificate for foran user	65
5.1. Template Bash Script	81

List of Tables

3.1. Comparison of CLI and GUI Features	17
3.2. Comparison of Programming Languages for CLI Tools	18
3.3. Key Differences between Chef, Puppet, and Ansible	21

1. Introduction

The following section provides a brief explanation of the motivation, objectives, and structure of the Bachelor's thesis.

1.1. Initial situation

In recent years, cyber attacks on critical infrastructures have continuously increased [1]. The security of critical infrastructures against cyber attacks is therefore of crucial importance, as they guarantee essential services and functions in society.

With this in mind, it is particularly important to test new technologies that are to be installed in critical infrastructure for vulnerabilities and to thoroughly analyse the resulting attack vectors in order to be able to guarantee protective measures against cyber attacks.

One of the new technologies of recent years is 5th generation mobile networks (5G). 5G offers a multitude of new use cases that are particularly promising in the area of basic communication infrastructures such as energy and water supply, logistics and transport [2, page 6]. For this reason, 5G networks must be analysed for potential risks and protective measures taken before they are used in critical infrastructure.

A key component in 5G networks is the radio access network (RAN) [3]. In order to analyse the risks to the RAN, the company Procyde has launched the 5G-FORAN project in cooperation with the Cologne University of Applied Sciences (TH Köln). The project analyses a concrete implementation proposal for a 5G RAN, namely Open RAN (O-RAN), specified by the O-RAN ALLIANCE, founded 2018 by AT&T, China Mobile, Deutsche Telekom, NTT DOCOMO and Orange [4], for potential attack vectors.

1.2. Objective

1.2.1. Objective 5G-FORAN

The concrete objective of the project 5G-FORAN (Forensic in O-RAN) is the development, design and practical simulation of a method for analysing, handling and resolving IT security incidents in the area of O-RAN. The basis for this development are traceable attack traces on the components, which are provided by an attack simulation as part of the overall project. The overall project is divided into two sub-projects: "Active attack simulation (offensive attack) on O-RAN components" and "Digital Forensics and Incident Response (DFIR) in O-RAN" [5]. Since Procyde, as a consulting and service company in the field of cyber security with a focus on the detection and defence of cyber attacks, has greater experience in the field of DFIR [6], it is responsible for the DFIR part of the project, while the attack simulation for trace generation is carried out by the Technical University of Cologne.

To realise the goal, the lab environment uses the implementations of the O-RAN software community, which focused on the development of open-source software for RAN components [7], in combination with Kubernetes to orchestrate the various O-RAN components. As the infrastructure of this implementation, Kubernetes offers a further attack vector that needs to be investigated. Since the O-RAN Alliance was founded 2018 and the development of the components is still an ongoing process, only a few practical attack vectors are known for O-RAN, the focus is therefore mainly on possible attack vectors in Kubernetes. However, if vulnerabilities for O-RAN become known, the focus will be expanded to include these vulnerabilities.

1.2.2. Objective of the Bachelor's thesis within 5G-FORAN

The following bachelor thesis is part of the active attack simulation sub-project. The objective of the bachelor thesis is to implement a framework that can be used to generate attack traces in O-RAN systems in a simplified manner. The framework is named ClusterForce and in the following will be referred to as CF-Framework.

To generate attack traces in a Kubernetes environment, the CF-Framework should provide existing open-source attack tools for attack simulations on Kubernetes by implementing wrappers of these tools. This allows users to generate traces using multiple attack tools via the CF-Framework.

The requirements for the CF-Framework are that the architecture is implemented in a dynamic and scalable way. This means that without significant code changes, the CF-Framework can be extended with additional attack tools. Furthermore, the it

should be effectively, user friendly and easy to use.

To achieve this, the wrappers should provide pre-parameterised methods, allowing users to execute attacks without having to deal with the usage of the attack tool directly. However, users should also be able to use the attack tools with all the associated functionality through the CF-Framework.

In addition, to allow for a traceable assessment of attacks, it is necessary that all attack results are recorded in a database for comparison with logs. Prior to being written to the database, entries should be categorised according to Mitre, Kubernetes ATT&CK Matrix and Common Vulnerabilities and Exposures (CVE).

On the one hand, this should be possible by parsing the result for possible categories, and on the other hand, tools should also be presorted into phases based on the mitre tactics. If a user executes a tool in a certain phase, this is also used for categorisation. When using a tool, the CF-Framework must also track metadata in addition to the attack result. This includes information such as the timestamp of the attack. A more detailed description of all metadata that will be collected, is presented in the Section 3.1.3.

Also, the ability to save executed attacks to an template and access them through the CF-Framework should be possible. This will allow for the generation of specific traces, such as for testing detection rules.

Furthermore, it should be possible to logically group attacks and assign them to a campaign. For example a user wants to generate traces on a specific component, he can start a campaign and all attacks that are executed gets assigned to this campaign. This assignment to a campaign should also be saved in the database.

Finally, in order to make the deployment of the CF-Framework and the database user-friendly, a way to automate the deployment has to be provided as well.

1.3. Personal Motivation

As I am interested in cyber security within computer science, specifically in attacking systems to find vulnerabilities, the bachelor thesis topic provides an opportunity for further development in this field. The project also involves Kubernetes technology, widely used in many areas, and O-RAN, a promising technology in mobile communications. Therefore, it is worthwhile and highly interesting to explore these technologies as well.

1.4. Thesis Structure

The thesis is divided into six chapters. The first chapter provides a brief introduction to the thesis, explaining the initial situation and discussing the topic. The objective of the project, the thesis, and the personal motivation are then presented. After this introduction, a short outline of the thesis is given to introduce the central topic.

The second chapter deals with the technical background of the thesis. First, a general description of all technologies involved in 5G-FORAN is given. Secondly, all relevant technologies related to the CF framework or the database are described.

In the third chapter, the solution concepts are explained. First, the preparatory work is discussed, followed by the solution approach for the architecture and automation of the CF-Framework and the database. After that, the test strategies for testing the CF-Framework are discussed.

The chapter four presents the implementation of the solution concepts. This includes the architecture of the CF-Framework with all its associated components, the features implemented within the CF-Framework, like creating a template or campaign, and the automation of the deployment of the CF-Framework and its associated database. Also the architecture of the database will be described.

Chapter five covers the testing of the general functions and attack tools of the CF-Framework. Additionally, the implemented features and phase categorisation of attacks are also tested for functionality.

Finally, the chapter Summary and Outlook, the results of the thesis will be summarized. To addition, the future of the CF-Framework in the project, and examples for further features are given.

1.5. Hints to Syntax

In this section, essential hints to explain the syntax used in the context of this thesis is defined. The Syntax is used to easily distinguish them from other elements in the code.

1.5.1. Module and Class Names

Module and class names are presented in **bold** font. Module names will always be lowercase, while class Names will always be uppercase. For example: This is a **example** module and **Example** class.

1.5.2. Methods, Variables, Files, Folder

Methods, variables, files and folder names are presented in *italic* font and will always be lowercase. For example: This is a *method*, this is a *variable*, this is a *file* and this is a *folder*

2. Technical Background

The following section describes the general and CF-Framework specific technologies used in the bachelor's thesis.

2.1. General

2.1.1. 5G

5G refers to the fifth generation of mobile network and is a new global wireless standard. It is characterised in particular by its higher transmission speeds and lower latency times. As a result, 5G offers new application possibilities, not only in areas of critical infrastructure, but also in many areas of everyday life. [8]

These range from faster internet and download times, improved network stability and real-time control of smart home technology, to near real-time monitoring of continuous data such as pulse and blood pressure of patients in hospitals.

The overall structure of a 5G system (5GS) can be seen in figure 2.1 below.



Figure 2.1.: General Architecture 5GS [9]

In general, 5GS uses the same components as previous generations. These include both user equipment (UE) and the two main components of the 5G system, the next generation radio access network (NG-RAN) and the core network (5GC). As the brain of the 5G network, the 5GC combines both the Access and Mobility Management Function (AMF) and the User Plane Function (UPF). While the UPF manages user data, the AMF provides access to the UE and RAN.

On the other side the 5G RAN is the link between the UE and the 5GC in a mobile network. It consists of the Next-Generation Node B(gNB), which controls communication via the New Radio interface (NR-Uu). UEs can also connect to the 5G RAN via this interface [3].

2.1.2. Open RAN

Conventional RAN solutions often come from a few large manufacturers such as Ericsson, Nokia or Huawei. The standards and interfaces of these RAN solutions are proprietary because these providers often only offer their own hardware and software, which results in less flexibility. With each new generation of mobile communications, network operators often have to replace the entire technology, which is associated with considerable costs.

To counteract this, the O-RAN Alliance was founded with the goal of open O-RAN implementation. This is intended to promote interoperability by enabling the integration of components from different manufacturers through the use of open interfaces. The resulting flexibility will allow network operators to buy equipment from different vendors and build a customised RAN environment, which will lead to healthy competition and likely lower costs for RAN equipment [2, page 8-9].

The following figure 2.2 shows the logical O-RAN architecture with its functions and interfaces.

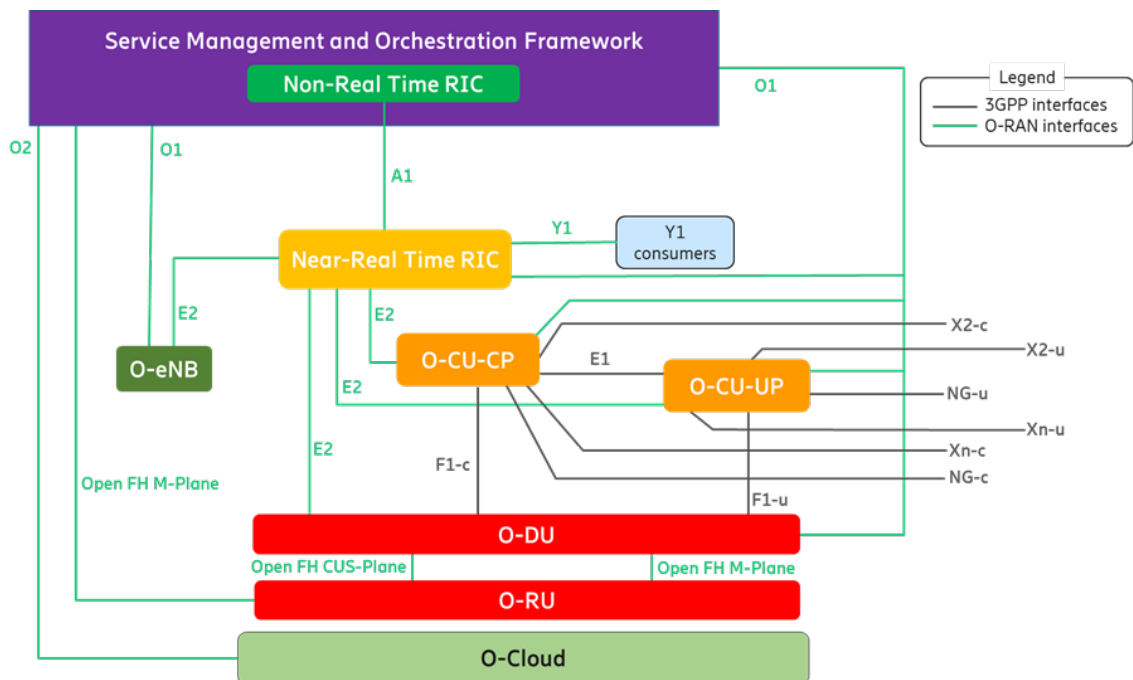


Figure 2.2.: Logical O-RAN Architecture [10]

The key components defined in the figure are:

- Service Management and Orchestration (SMO), which monitors orchestration, management, and automation of RAN elements.
- Non-Realtime (Non-RT) RAN Intelligent Controller (RIC), which is responsible for optimizing RAN resources and the use of RAN elements.
- Near-Realtime (Near-RT) RIC, which is responsible for data collection and control.
- O-Cloud provides the physical infrastructure for RAN network functions.

Furthermore, figure 2.2 displays the nodes defined by the 3rd Generation Partnership Project (3GPP), a collaboration of telecommunications standardisation organisations:

- O-RAN Radio Unit (O-RU): responsible for the radio transmission.
- O-RAN Distributed Unit (O-DU): responsible for distributed data processing.
- O-RAN Central Unit (O-CU): coordinates centralised control.

The matching interfaces for the nodes that will enable efficient data transmission and communication in the 5G network are marked in black and green [11].

2.1.3. Kubernetes

Kubernetes is an open source system for automating the deployment, scaling and management of containerised applications. Containers are grouped into logical units to ensure easy management and discovery [12].

In the O-RAN environment, the management of the containers containing the various components of the RAN plays a critical role in ensuring smooth operation. To effectively implement O-RAN, an orchestration tool such as Kubernetes is required to divide and manage the containers into logical groups called pods. These pods and containers are then run on a Kubernetes cluster, which provides the necessary compute resources for execution [13].

Figure 2.3 below shows the structure of an O-RAN environment in Kubernetes.

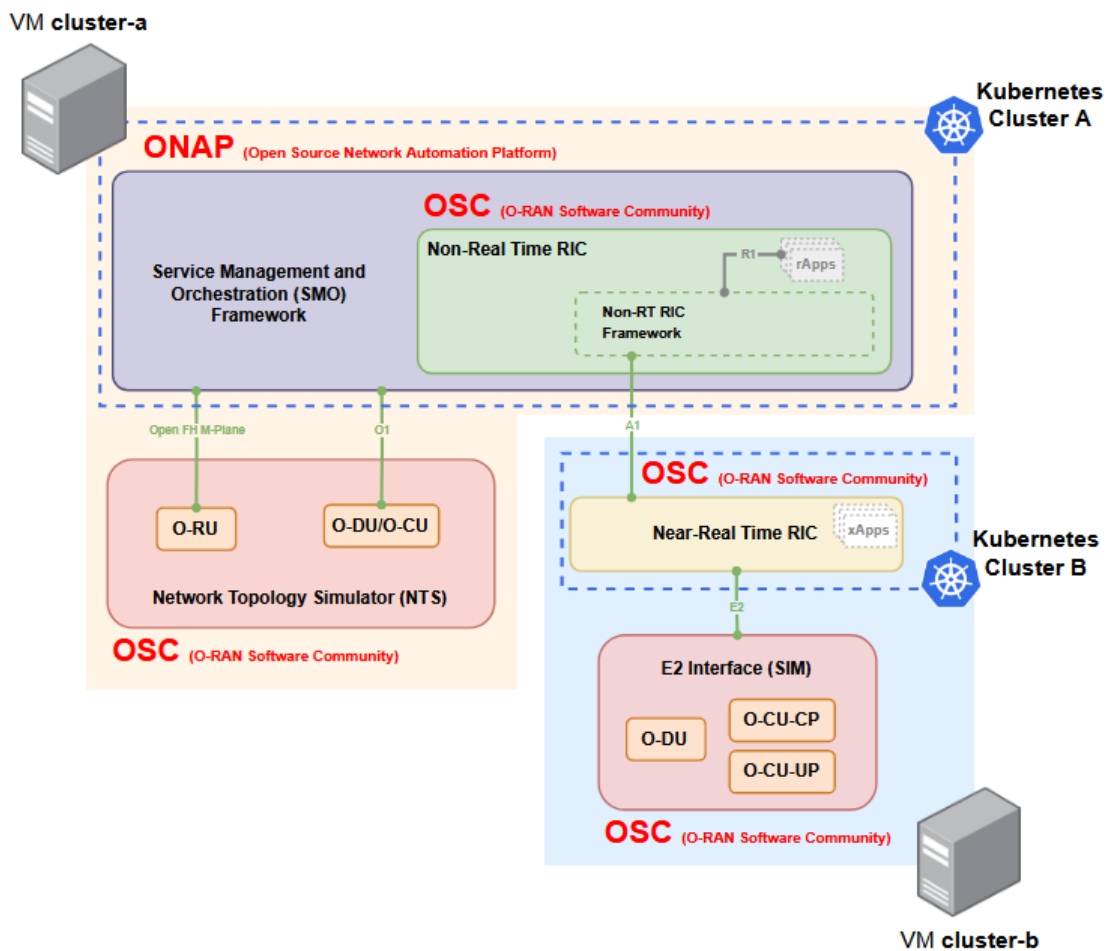


Figure 2.3.: O-RAN Structure in Kubernetes [Source: 5G-FORAN TH-Köln]

The O-RAN components are deployed across two Kubernetes clusters. As shown in figure, Cluster A comprises the O-RAN components SMO and Non-RT RIC, while Cluster B contains the Near-RT RIC. Cluster A is deployed on a virtual machine (VM) that also hosts the Network Topology Simulator (NTS), which facilitates network function simulation. Cluster B and the E2 Interface Simulator, which facilitates communication between O-DU and O-CU, are also hosted on a separate VM.

2.1.4. 5G-FORAN Lab Environment

The overall project lab environment is illustrated in figure 2.4.

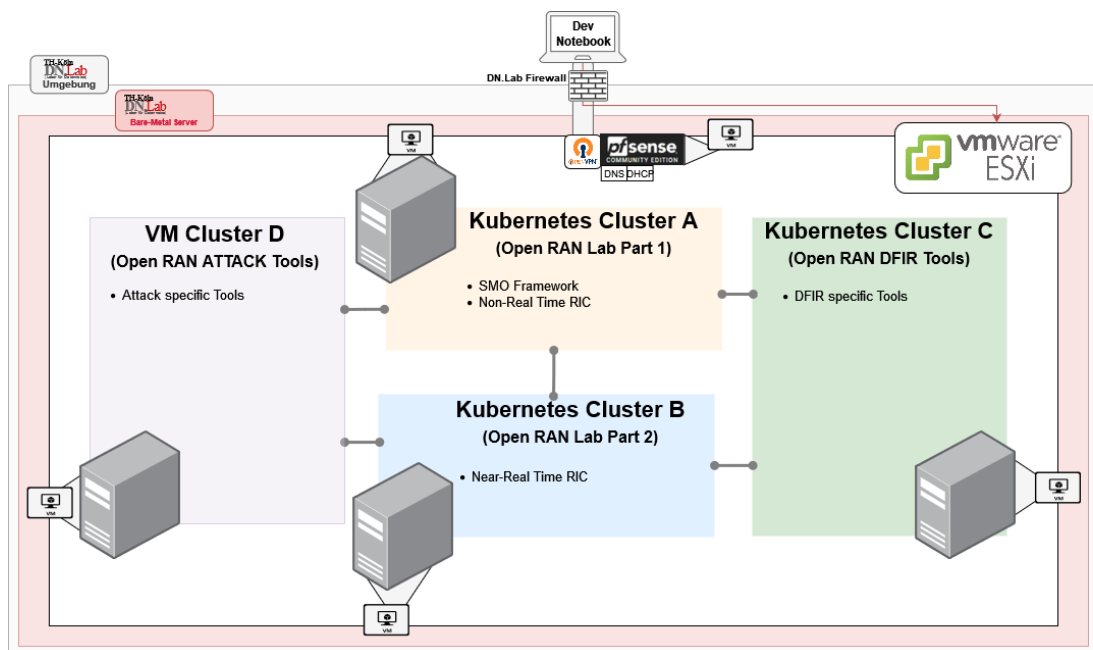


Figure 2.4.: 5G-FORAN Lab Environment [Source: 5G-FORAN TH-Köln]

The structure of Kubernetes Cluster A and B corresponds to the diagram in figure 2.3. Additionally, there is a separate Kubernetes Cluster C for DFIR specific tools and a VM Cluster D for attack specific tools. In this thesis, the term Cluster A, Cluster B, Cluster C, and Cluster D refers to the clusters of the 5G-FORAN Lab environment displayed in this figure.

2.1.5. MITRE ATT&CK Framework

The MITRE ATT&CK is a publicly accessible database that documents real attack methods and techniques. It serves as a basis for developing customised threat models and methods. Thanks to the wide acceptance of MITRE, the use of MITRE ATT&CK has proven to be extremely valuable in the cyber security industry as well as in the public and private sectors. This is because it helps organisations and professionals respond to current and emerging threats and optimise their security strategies [14].

To categorise attacks, MITRE uses ATT&CK tactics and techniques. The CF-Framework used the following tactics to categorise attacks [15]:

- Reconnaissance: Collect information about the system to be attacked
- Initial Access: Gain access to the network
- Execution: Execution of malicious code
- Persistence: Maintain access to systems across interruptions
- Privilege Escalation: Obtain higher authorisation on the system
- Defence Evasion: Avoidance of detection
- Credential Access: Theft of usernames and passwords
- Discovery: Exploration of the environment of the attacked system
- Lateral Movement: Control of remote systems in the network
- Collection: Collecting critical data
- Command and Control: Control and communication of compromised systems
- Exfiltration: Stealing data from the network
- Impact: Manipulation, disruption or destruction of systems

2.1.6. Kubernetes ATT&CK Matrix

The Kubernetes ATT&CK Matrix, created by Microsoft Azure Security Center, is based on the ATT&CK framework and focuses on the security aspects of Kubernetes and containerised environments. It contains various attack techniques and tactics that are relevant for the security of container orchestration systems, especially for Kubernetes [16].

Figure 2.5 displays the tactics and their associated techniques.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Impact
Using cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List Kubernetes secrets	Access Kubernetes API server	Access cloud resources	Images from a private registry	Data destruction
Compromised image In registry	Bash or cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete Kubernetes events	Mount service principal	Access Kubelet API	Container service account	Collecting data from pod	Resource hijacking
Kubeconfig file	New container	Kubernetes CronJob	Writable hostPath mount	Pod or container name similarity	Container service account	Network mapping	Cluster internal networking		Denial of service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from proxy server	Application credentials in configuration files	Exposed sensitive interfaces	Application credentials in configuration files		
Exposed sensitive interfaces	SSH server running inside container	Container service account			Access Managed Identity credentials	Instance Metadata API	Writable hostPath mount		
	Sidecar injection	Static pods			Malicious admission controller		CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Figure 2.5.: Kubernetes ATT&CK Matrix [17]

A comparison of MITRE’s tactics with those in the illustration shows that the Kubernetes ATT&CK Matrix does not have all the tactics. The following tactics from the MITRE ATT&CK framework are relevant for the Kubernetes ATT&CK matrix:

- Initial Access
- Execution
- Persistence
- Privilege Escalation
- Defense Evasion
- Credential Access
- Discovery
- Lateral Movement
- Collection

- Impact

In addition to referencing the MITRE tactics and techniques, Microsoft also employs its own IDs for each tactic and technique. These IDs are used by the CF-Framework for categorisation.

2.1.7. Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) was founded in 1999 and is led by the National Cybersecurity Federally Funded Research and Development Centre (FFRDC), which is operated by MITRE. It is funded by the US federal government, including contributions from the Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA) and is freely accessible to everyone.

The CVE database contains publicly known information security vulnerabilities. Each vulnerability is assigned a unique CVE number to facilitate the exchange of information about cybersecurity vulnerabilities. This creates a standardised approach for a variety of stakeholders, including vendors, companies and researchers, to help them plan vulnerability management measures [18]. This leads to wide acceptance and standardisation of CVE through the CVE number. Because of this the CF-Framework uses CVE to categorise traces of attacks.

2.1.8. Attacker Perspectives

To ensure a comprehensive attack simulation, it is important to consider various attacker perspectives, which are specified by the Federal Office for Information Security (BSI) [2, page 39-41]. Figure 2.6 illustrates the potential attacker perspectives and their corresponding requirements.

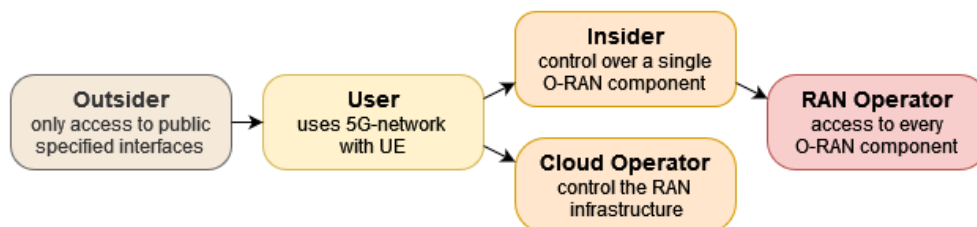


Figure 2.6.: Hierarchy Attack Perspectives

The figure should be understood as a hierarchy from left to right. The outsider, who can only access publicly accessible interfaces, has the least opportunity to cause serious damage. The next most dangerous level is a user who is connected to the network via a user device, followed by the insider and cloud operator. The term 'insider' in this context refers to an individual who has access to a RAN component, such as an employee of the RAN operator. It is important to note that the RAN operator holds the most powerful attacker role, as it has access to all RAN components. However, it is also possible for an outsider or user to discover vulnerabilities that grant them extended rights and greater access. For instance, an outsider could initially take over an end device and subsequently infiltrate RAN components from that point.

The attack simulations in this project are conducted from the viewpoint of an O-RAN operator. As a result, all attacks which are displayed in chapter 5 are restricted to this particular attacker perspective.

2.2. CF-Framework

2.2.1. MongoDB

During an attack, it is a requirement to collect metadata which should be saved in a database. The CF-Framework therefore stores all captured data in a MongoDB instance. MongoDB is an open source NoSQL database that specialises in managing large volumes of structured and unstructured data. In contrast to relational databases, NoSQL database like MongoDB are characterised by a high degree of flexibility, as no predefined schemas are required. Data is stored in documents with key-value pairs that are structured in collections, which enables flexible data modelling. The database also uses the BSON format (Binary JSON) to store documents [19].

This type of data modelling is advantageous for the CF-Framework because the data collected varies greatly when using different open source tools, meaning that it is not possible to store the data in SQL format. Instead, a NoSQL format must be used to save the data as a document.

X. 509 Client Certificate Authentication

X.509 client certificate authentication is a secure method for verifying the identity of a client during communication. The X.509 standard is used, in which clients receive digital certificates from a trusted certification authority (CA). During authentication, the client presents its X.509 certificate to the server so that the server can verify the

client's identity using the information in the certificate. This method ensures mutual authentication and increases general communication security. One advantage of using MongoDB is that clients do not need to transmit sensitive login information, such as passwords, during authentication, which increases security. MongoDB supports X.509 client certificate authentication as one of its authentication mechanisms [20] and is therefore used to ensure a secure connection between the CF-Framework and the database.

2.2.2. Ansible

For automated deployment of the CF-Framework on various clusters, the open source software Ansible is used. Ansible is written in Python, specially developed for IT automation and enables the configuration of systems, the provision of software and the orchestration of complex workflows for application deployments and system updates. Ansible emphasises simplicity, user-friendliness and security and an easy-to-understand language, with extensive documentation, which makes it possible to use the software without much prior knowledge [21].

2.2.3. Prompt Toolkit

Prompt Toolkit is a library for creating powerful interactive command lines and terminal applications in Python, which runs on both Windows and Linux systems. Because the library offers many functions ranging from auto-suggestion and auto-compilation to a history of previously entered commands [22], the library gets used within the CF-Framework to improve efficiency and the usability.

2.2.4. Kubehunter

Kubehunter is an open-source attack tool for identifying security vulnerabilities in Kubernetes clusters. The aim is to increase security in Kubernetes environments by improving awareness and visibility of potential vulnerabilities.

In addition, Kubehunter supports the Kubernetes ATT&CK matrix format, which enables more precise detection of vulnerabilities. Although Kubehunter has defined its own vulnerabilities, these are still transferred to the Kubernetes ATT&CK matrix to enable a standardised categorisation of attacks. Because Kubehunter can be used both outside a cluster to scan vulnerabilities from the outside and inside a cluster either directly on a host system or as a pod with elevated rights to scan vulnerabilities from the inside, Kubehunter offers various application options for the CF-Framework to generate traces.

In addition to scanning, Kubehunter offers the possibility of "active hunting", which means not only scanning vulnerabilities but also exploiting them to find further vulnerabilities, which makes the tool attractive for trace generation [23], and therefore is implemented inside the CF-Framework.

2.2.5. Kdigger

Kdigger is an open-source attack tool for security research and penetration testing. It focuses on detecting and analysing Kubernetes systems, enabling the identification of vulnerabilities and potential security gaps in Kubernetes clusters. Kdigger can analyse resources such as pods, services and permissions to uncover potential vulnerabilities in the configuration by collecting information about resources, permissions and configurations in a Kubernetes cluster through automated queries. The tool provides a comprehensive view of the security posture of a Kubernetes cluster and thus offers a solid basis for security-related decisions. With its support in detecting configuration errors, Kdigger helps to reduce the attack surface and minimise the risk of security breaches [24]. Because of this, Kdigger is made available for use inside the CF-Framework.

2.2.6. RedKube

-TODO what are kubectl commands RedKube is an open-source attack tool that uses kubectl commands, TODO, to assess the security of Kubernetes clusters from a hacker's perspective. The commands can either passively collect data and disclose information or perform active actions that can influence the cluster and change the state of the cluster. In addition, RedKube provides the collection of kubectl commands, already categorised according to MITRE ATT&CK tactics, to be used in active or passive mode. The active mode in particular sets RedKube apart from other tools because to the aggressive attack mode additional tracks can be generated [25]. This is why the Usage of RedKube is also possible via the CF-Framework.

3. Solution Concepts

The following section explains the solution concepts that form the foundation for implementing the CF-Framework.

3.1. Preparatory work

3.1.1. Evaluation Architectural Approaches

Two approaches are available for implementing the CF-Framework architecture, which are shown below.

Feature	CLI (Command Line Interface)	GUI (Graphical User Interface)
Operating Mode	Commands in the terminal.	Visual elements (icons, buttons, windows).
Performance	More efficient and faster.	Requires system resources, may reduce performance.
Precision	Higher precision, granular control.	Lower precision compared to CLI.
Ease of Use	Complex, requires technical knowledge.	Easy to use, suitable for beginners.

Table 3.1.: Comparison of CLI and GUI Features

The first option for implementation is as a command line interface (CLI) tool. This implementation has the advantage that the tools to be integrated are also implemented as a CLI tool, making integration into the CF-Framework easier. Additionally, implementation as a CLI tool offers the most flexibility as it can be executed on any system with a command line and requires fewer resources. One disadvantage is that visualisation cannot be implemented without a graphical component. Additionally, the use of the application without a graphical component is only possible via the command line, which can make it more complicated.

The second option is to provide access to the CF-Framework over a Grafical User Interfac (GUI). The advantage of a GUI is that it offers users an understandable user interface and visual representation through an interactive application but this requires more memory and processing power, making the GUI slower than CLI. Also an GUI provides lower precision, functionality, and granular control of the OS [26].

Overall, the flexibility and integration benefits of a CLI tool outweigh those of a GUI. Despite its higher level of complexity, the CLI tool approach offers the programmer the greatest potential for implementation of all CF-Framework features, thus enabling users to generate tracks more effectively. Because of this the CF-Framework architecture is implemented as CLI tool.

3.1.2. Evaluation Programming language

As the architecture of the tool is implemented as a CLI tool, it is necessary to consider possible programming languages.

The table 3.2 compares four programming languages: C, C++, Java, and Python. They are compared, taking into account whether they require a compiler, whether they are object-oriented, their platform dependency, and their library support [27].

	Compiler	Style	Platform	Library
C	Compiled Language	Not object oriented	Platform specific	only small number of Libraries
C++	Compiled Programming language	Object oriented	Platform dependent	only small number of Libraries
Java	Compiled Programming language	Object oriented	Platform-unaffected	good number of Libraries
Python	Interpreted Programming Language	Object oriented	Platform independent	extensive number of Libraries

Table 3.2.: Comparison of Programming Languages for CLI Tools

C is a compiled language, which means it needs to be translated into machine code before execution, resulting in faster execution times compared to interpreted languages. However, being non-object-oriented, developers need to manually manage data structures and function interactions. This manual handling can make it more challenging to implement dynamic architectures that easily adapt to changes. For

instance, adding new features or modifying existing functionalities in a non-object-oriented language might involve more extensive modifications throughout the code, potentially leading to increased complexity and error-proneness. Additionally, the need for manual memory handling in C adds another layer of complexity, as developers must explicitly allocate and deallocate memory, increasing the risk of memory-related errors. Moreover, the limited number of libraries in C might require more effort in building certain functionalities from scratch, contributing to the overall complexity of the development process [28].

C++ inherits many characteristics from C but introduces object-oriented programming (OOP) concepts. Similar to C, C++ is platform-dependent, potentially limiting its portability. The availability of libraries is also somewhat limited compared to languages like Java and Python [29].

Java is a compiled, object-oriented language that is platform unaffected. The only requirements is, that the java is installed on the system. However, it has a smaller number of libraries compared to Python [30].

Python, being an interpreted and object-oriented language, stands out for CLI tool development. The big advantage, apart from the simple syntax, is the extensive standard and third party libraries. These libraries cover a variety of use cases and allow developers to access proven solutions without having to implement everything from scratch [31].

In conclusion, the CF-Framework is implemented using the Python programming language. One of the Python libraries used in the CF-Framework is the prompt toolkit, which supports fundamental CLI Tool functions.

3.1.3. Evaluation Metadata

The metadata consists of all the data that the CF framework writes to the database, together with the results that are based on the attacks. These are listed below::

- Timestamp of the attack
- output of the attack
- tool version, tool name
- command used for the attack
- ip used in the attack
- host name
- categorization, for mitre, cve, kubernetes attack matrix, phase

3.1.4. Evaluation of CF-Framework Features

In addition to the possibility of generating traces via the CF-Framework, there are other features that are to be integrated into the CF-Framework to make the tool more efficient and user-friendly, which are listed below:

- Database
- Template
- Parser
- Campaign

On the one hand, it is important to store attack data in a database so as not to lose the knowledge gained. Additionally, it should be possible that, executed attacks can be saved in a template. This means that when creating a template, all previous attacks are stored in a bash script that can be accessed from the CF-Framework menu. On that way a sequence of attacks can be repeated at will to generate traces quickly, or a particular sequence of attacks can be executed multiple times without having to execute each attack individually. This technique can also be useful when testing DFIR tools, as it allows for the creation of test traces without the need for specific ones. For instance, it can be used to verify the log behaviour of the DFIR tools.

The results of each attack within the CF-Framework should be filtered for both CVE entries and the MITRE tactics listed in section 2.1.5. Additionally, attacks should be mapped to the entries of the Kubernetes ATT&CK matrix if possible. All tactics and CVEs found are stored as a separate field in the database to provide additional information on the attack.

In addition to this, the output of the tools have to be analysed to determine whether it is feasible to implement a parser, based on the output. In the scope of this thesis, one parser has to be implemented for the tool KubeHunter.

Furthermore, , it should be feasible to merge the listed attacks into a campaign, which must be stored in a distinct collection within the database. The corresponding campaign ID for an attack is also saved as a separate field along with the metadata.

3.1.5. Evaluation of Automation Framework

To automate the CF-Framework and database deployment the three automation frameworks Chef, Puppet and Ansible get compared below, for key difference:

Criteria	Chef	Puppet	Ansible
Complexity	Moderate	Moderate	Low
Setup and Installation	Agent-based	Agent-based	Agentless
Scalability	Good	Good	Good

Table 3.3.: Key Differences between Chef, Puppet, and Ansible

Firstly, Chef stands out for its notable flexibility in configuration management and good scalability. However the framework is considered more complex than Ansible for example. This is primarily due to its agent-based setup, which requires the installation of software agents on managed nodes to enable continuous communication with a central server for configuration management.

Secondly, Puppet shares similarities with Chef, boasting scalability and flexibility. It also comes with a agent-based setup, which adds additional complexity.

Lastly, Ansible shares scalability and flexibility with the other two frameworks. However, it distinguishes itself from them by its simplicity, which is based on the agent-less setup. With the agent-based setup, systems can be managed without requiring the installation of software agents on the target nodes, communicating directly through SSH [32].

In conclusion, since all frameworks provide scalability and flexibility, but Ansible stands out, due its lower complexity. This means that less time needs to be spent learning the frameworks and more time can be spent on implementing the automation.

3.2. Framework Architecture

3.2.1. General Architecture

To implement the CF-Framework, it is necessary to plan the architecture carefully, considering all the functions and features that need to be implemented. The architecture should be designed with scalability in mind, allowing for the integration of further open source tools, and with code efficiency as a priority, avoiding redundant code as much as possible. Figure 3.1 illustrates the basic architecture of the CF-Framework. The diagram displays only the modules, which are logical entity's, that contain python files, which perform a specific task, and their inter-dependencies. The detailed Framework Architecture with the modules and classes in combination can be seen in appendix A.

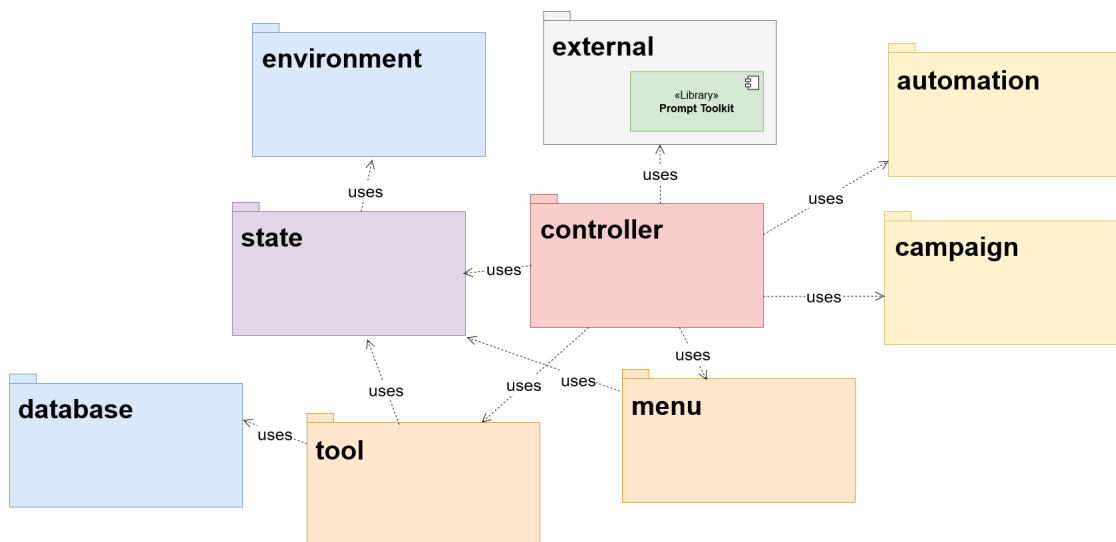


Figure 3.1.: Framework general Architecture

The architecture's central control is the **controller** module, highlighted in red at the figure's centre and it manages the prompt and user input. The Prompt Toolkit library inside the **external** module, shown in grey and green in the figure, is used to manage the prompt. If the input is a general command, such as exiting the CF-Framework, the **controller** module handles the processing. However, if the input is related to menu navigation within the CF-Framework, it is transferred to the **menu**. The **menu** module utilizes the input and returns a status. Similarly, for tool-specific inputs, the user input is transferred to the **tool** module, utilized, and a status is returned to the **controller** module. So the state can be either of type Tool or Menu. The corresponding module **state**, contains all possible states and is highlighted in purple in the figure. To make this more clear, if the last input was

specific to a tool, the **controller** module receives the next state, from the last tool used. With this new state it knows where the new input has to be transferred to be handled.

To realise the features of template and campaign, the **controller** module uses the modules **automation** and **campaign**, shown in yellow in the figure.

Additionally, both the **controller** module and the **tool** module use the **environment** module, shown in blue in the figure, to manage the metadata generated during use of the CF-Framework. Here, the focus of the '**controller**' module is primarily on reading the metadata, while the **tool** module both stores metadata in the **environment** module and reads metadata in order to write it to the database after a successful attack. This is realised by using the **database** module, also shown in blue in the figure. More precisely, the **database** module provides a connection to the database and controls the interaction between the CF-Framework and the database.

3.2.2. Controller

The structure of the **controller** module in conjunction with the **external** module is illustrated in figure 3.2.

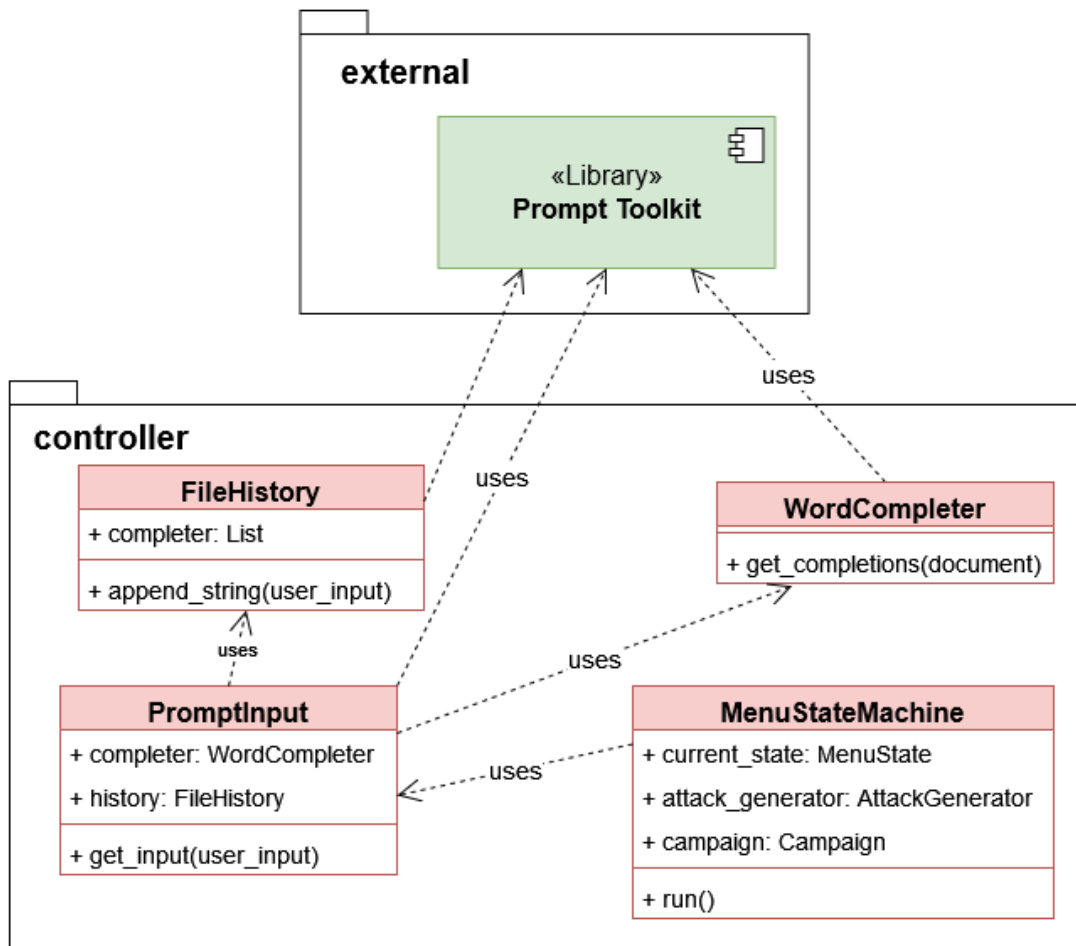


Figure 3.2.: Structure of the controller module in conjunction with the external module

The **controller** module consists of four classes. The actual control unit of the module is the **MenuStateMachine** class, which contains both the current state of the CF-Framework and also an instances of the **AttackGenerator** class, from the **automation** module, and **Campaign** class, from the **campaign** module. The **PromptInput** class is used to manage the prompt. This is based on the Prompt Toolkit and is intended to provide both an input history for user-friendliness and the option of automatically completing the content that has already been entered. For this purpose, the **PromptInput** class uses both the **FileHistory** class and the **WordCompleter** class, which are also based the Prompt Toolkit library.

3.2.3. Tool

The structure of the **tool** module is shown in figure 3.3.

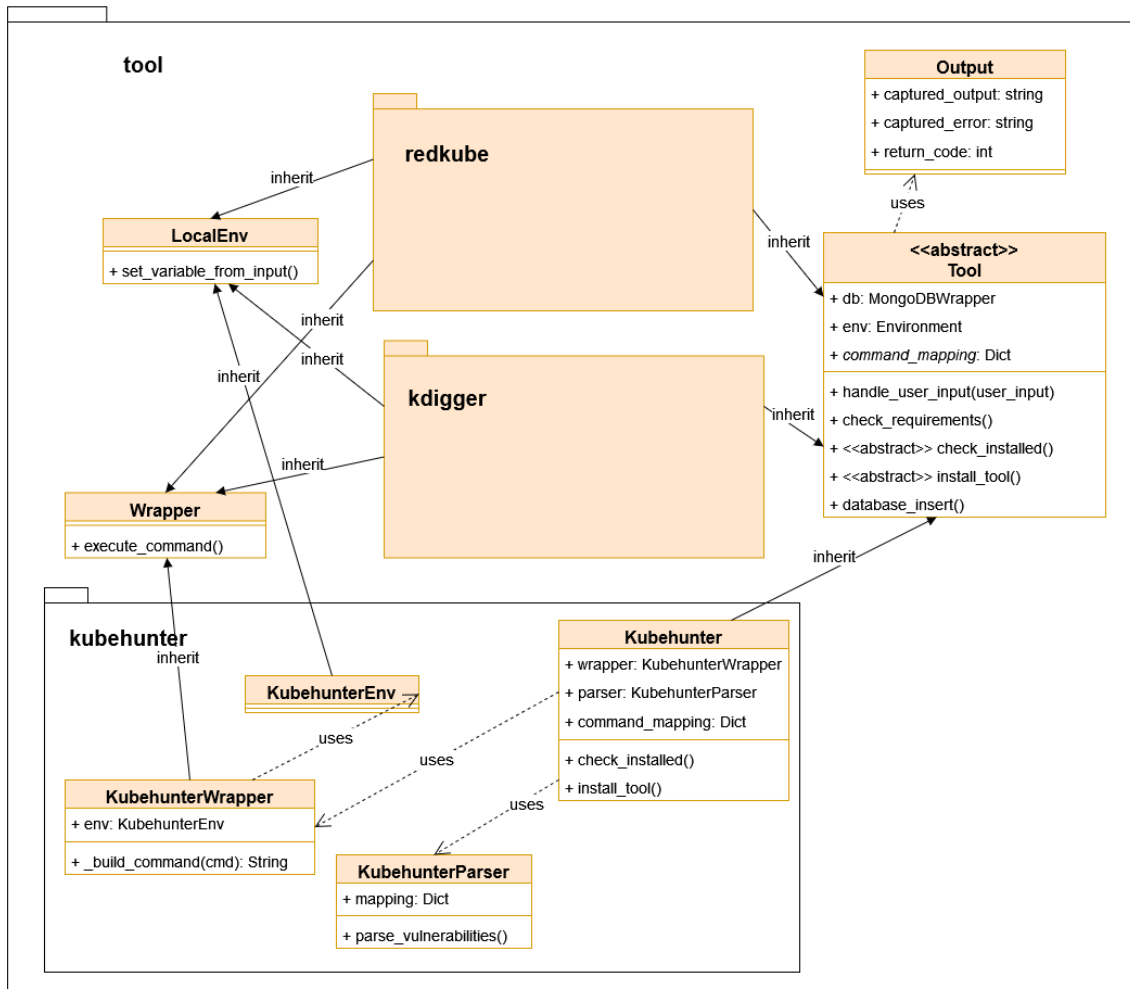


Figure 3.3.: Structure of the tool module

The **tool** module passes user input to the abstract base class **Tool**, which declares and implements all variables and methods that span across tools. Additionally, it defines abstract variables and methods that are implemented by inheriting tool Subclasses. For instance, each tool must implement a method to verify if it has already been installed or if it still needs installation. The *handle_user_input()* method utilizes the received user input.

In the module there are separate modules for each implemented tool, consisting of a Subclass, a wrapper class, an environment class for tool-specific parameter, and, if possible, a parser class. The Subclass inherits from the Base class and handles user input through the wrapper class. The Wrapper class implements the corresponding tool using the tool-specific parameters stored in the Environment class.

Both the wrapper classes and the environment classes, from each tool, inherit from the base class **Wrapper** and **LocalEnv**, which implement overarching commonalities within these classes.

During the course of the thesis, there are three tools that have got implemented, which are listed below.

- Kubehunter
- Kdigger
- RedKube

The structure of these tools differs only in terms of naming, which is why the on the figure only the Kubehunter tool is illustrated fully.

The class **Kubehunter** is the subclass of the base class **Tool**, which uses an instance of the class **KubehunterWrapper** and **KubehunterEnv**, to handle the user input and execute an attack if necessary.

3.2.4. Menu

Figure 3.4 is an illustration of the **menu** module structure.

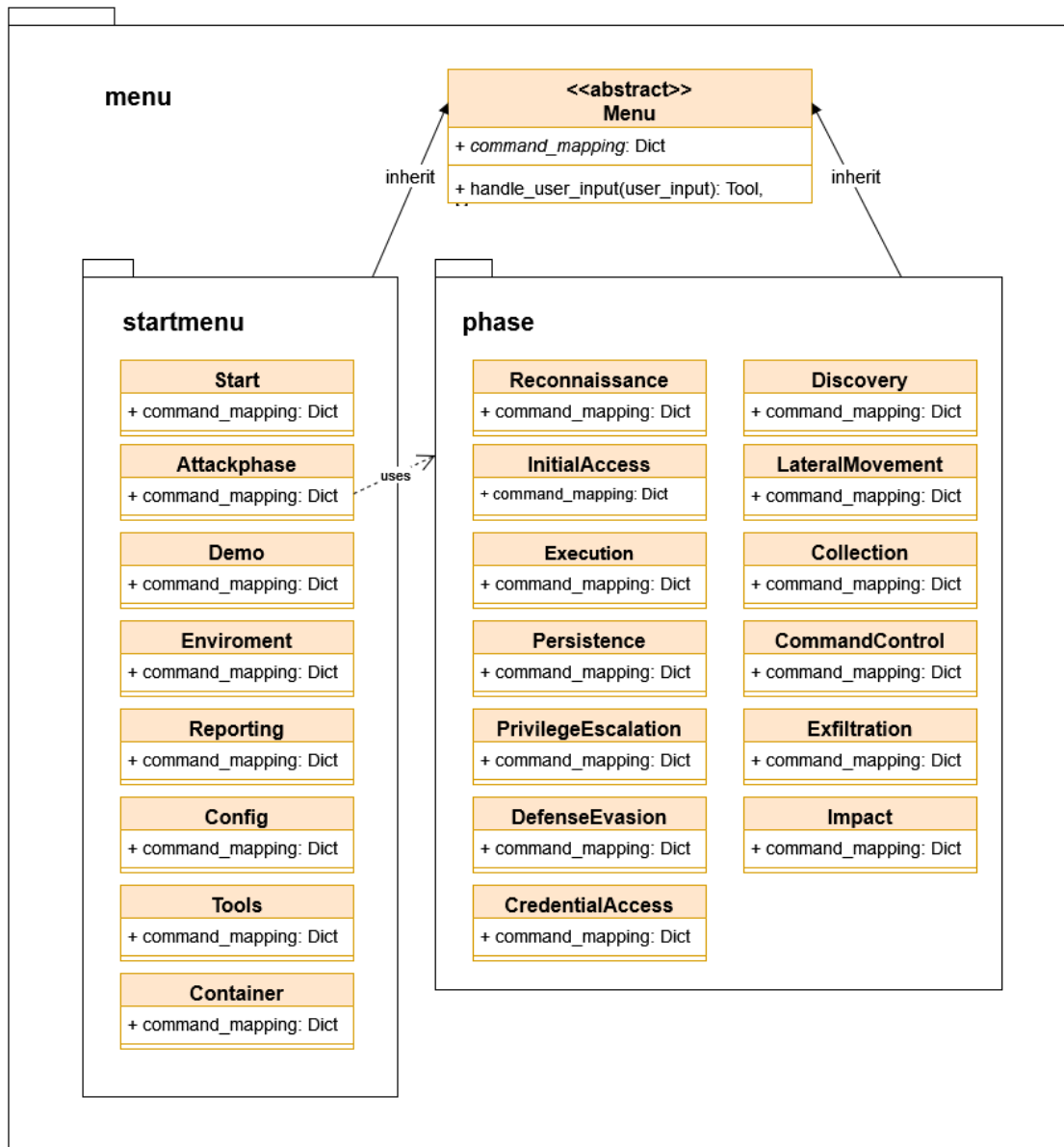


Figure 3.4.: Structure of the menu module

As with the **tool** module, the **menu** module defines methods and variables for each of the menu classes in an abstract base class. There are two menu submodules in total: **startmenu** and **phase**, which are logically separated. The **startmenu** submodule contains all the classes that the CF-Framework can accept after it has been started. The **phase** submodule contains all the classes that need to be implemented for the attack phase state.

When the CF-Framework is started, the subclass **Start** is used to handle the user

input. The , which menu for the **Start** subclass is displayed on the console. From here, the user has the following selection options:

- Attackphase
- Tools
- Demo
- Environment
- Reporting
- Config
- Container

Once an option is selected, the user is directed to the next menu and the state is set based on their input. If **Tools** is chosen, an overview of all available tools will be presented. If **Demo** is selected, all available templates to run an one click Attack, will be displayed. If **Attackphase** is selected, the user will be taken to the next menu phase. Here, one of the thirteen available MITRE phases can be chosen, which can be seen on the right in the **phase** submodule. After selecting a phase, a list of tools for that phase will be displayed. The following figure 3.5 illustrates an example Menu cycle.

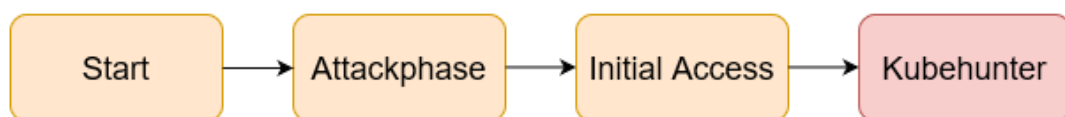


Figure 3.5.: Example menu run

The menu starts in the Start state. To enter the Attack phase, the user selects the Attack phase option from the menu, which takes them to the Attack phase state. Here, they can choose the MITRE Phase Initial Access and view all the tools available for Initial Access. The user selects the Kubehunter tool and moves from the menu state to the Kubehunter state, which is a tool-specific state. The CF-Framework remains in this final state until the user is finished with the tool.

3.2.5. Database

The **database** module contains the **MongoDBWrapper** class, which can be seen in the following figure 3.6

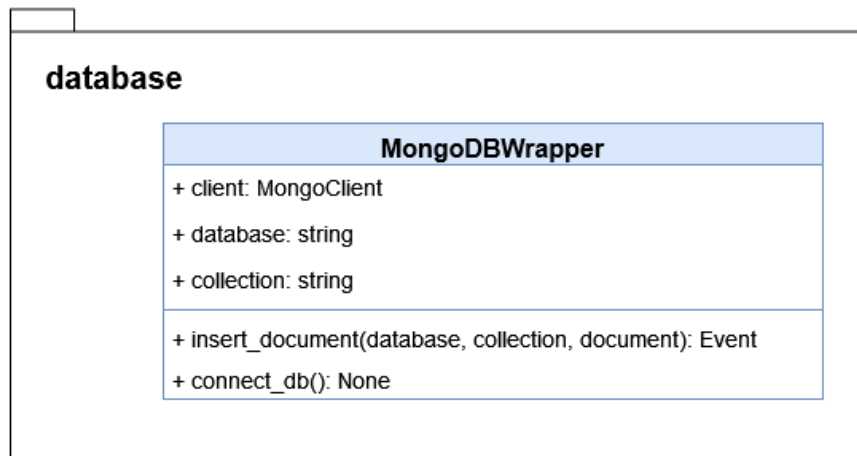


Figure 3.6.: Structure of the database module

The **MongoDBWrapper** class provides all the necessary methods for interacting with the database. This means that both the connection setup and reading and writing from and to the database are implemented in this class.

3.2.6. State

Figure 3.7 below shows the **state** module, which contains the menu state class.

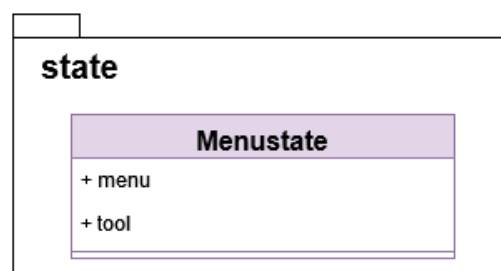


Figure 3.7.: Structure of the state module

All possible states should be defined within this class. This means that each tool and menu class has its own defined state, which is recorded in **Menustate** class and which the CF-Framework can use to determine which state it is currently in.

3.2.7. Environment

The **environment** module comprises of two classes: the **Environment** class, where all metadata are defined, and the **GlobalVariables** class, as illustrated in figure 3.8.

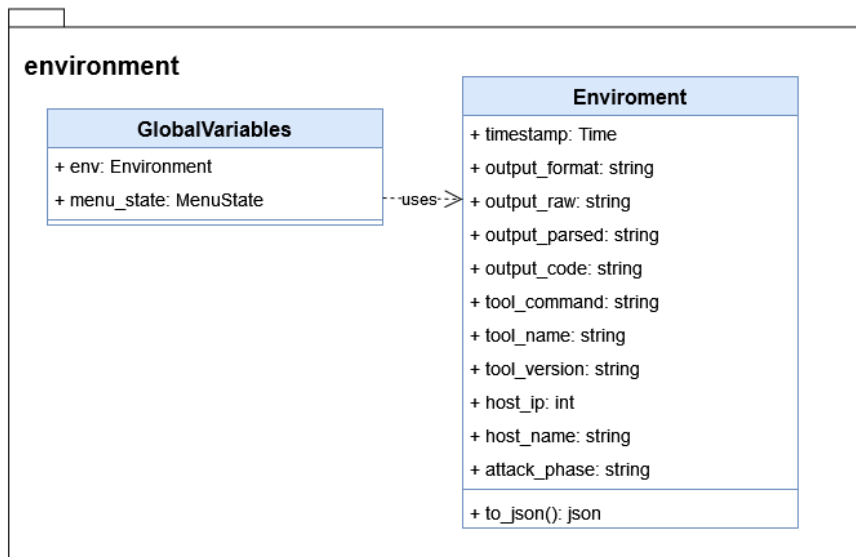


Figure 3.8.: Structure of the environment module

The **GlobalVariables** class stores the fundamental global variables of the CF-Framework, including the current state and metadata. Through the **GlobalVariables** class all modules that require information about the state or metadata can gain access to the information.

The **Environment** class defines all metadata, which are defined in 3.1.3. These can be seen in below:

- The time of the attack (*timestamp*)
- The resulting output of an attack, both in raw (*output_raw*) and parsed state (*output_parsed*), the format of the return (*output_format*), for example plain text or JSON and the result code, i.e. whether an attack was successfully executed or not (*output_code*).
- The information about the executed tool, i.e. the last executed command (*tool_command*), the name of the tool (*tool_name*) and the version (*tool_version*).
- The IP (*host_ip*) and the host name (*host_name*) of the host system.

3.2.8. Features

In order to provide the features defined in 3.1.4 in the CF-Framework, all necessary classes are implemented in the module **automation** and **campaign**. Figure 3.9 shows the **AttackGenerator** and **Template** classes on the left-hand side, which together implement the template feature.

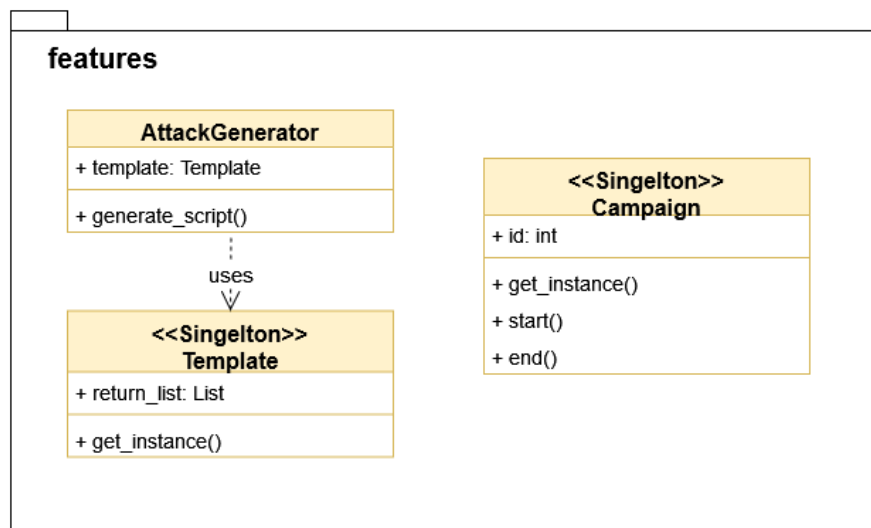


Figure 3.9.: Structure of the feature modules automation and campaign

The **Template** class contains a list of all templates used by the **AttackGenerator** class to ensure that a newly generated template does not already exist. If this is not the case, the attack generator creates the new template.

On the right-hand side of the figure is the **Campaign** class, which implements the campaign feature, by containing methods to creating and starting a campaign using the *start()* method and ending this campaign and writing it to the database using the *end()* method.

3.3. Database Architecture

The database architecture was carefully planned to ensure that data from the CF-Framework is stored in a structured and clear manner. Figure 3.10 illustrates the database architecture connected to the CF-Framework.

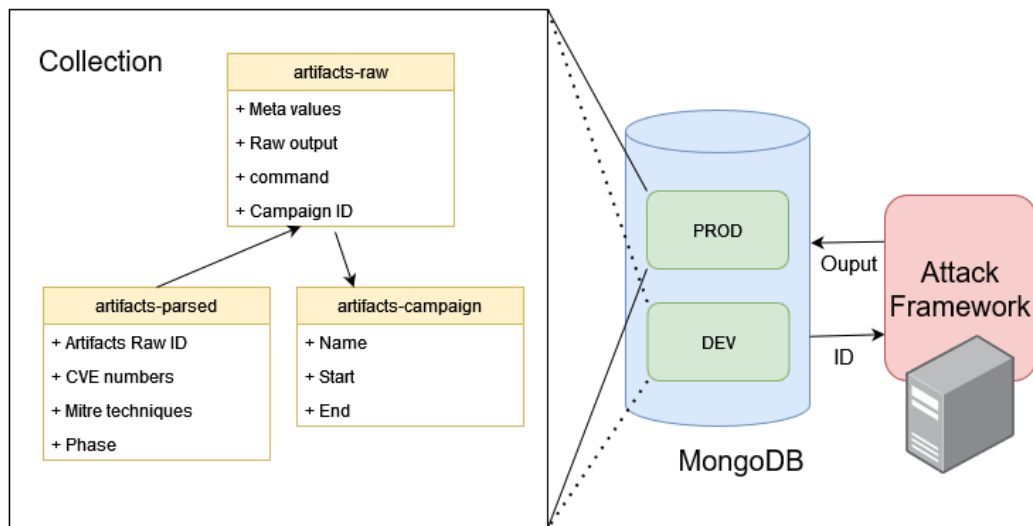


Figure 3.10.: Database Architecture

Within the MongoDB instance, two databases should be created to separate database entries that are created during testing of the CF-Framework from database entries that are created during trace generation in connection with the DFIR tools. The Production (PROD) database should be used for active track generation, while the Development (DEV) database is used when testing the CF-Framework. The schema of the PROD and DEV database is the same. Both databases consist of three collections.

The first collection `artifacts-raw`, which stores all the metadata as a document. In addition to the metadata, an entry in this collection can also contain a campaign ID if the entry was executed during a campaign. This campaign ID refers to the ID of the entry in the `artifacts-campaign` collection.

A created campaign is saved here as a document and contains the name and time of the start and end of the campaign, which is the second collection.

In the third collection, the processed data is written, which includes the phase of the attack, as well as a CVE number and reference to MITRE tactics and techniques. In addition, an artifacts raw ID within the document in the `artifacts-parsed` collection also refers to the corresponding document in the `artifacts-raw` collection from which the processed data originates.

3.4. Automation

When changes are made to the implementation of the CF-Framework or the database, it is important to update the versions running on the clusters. To automate this process and avoid manual updates on the clusters, a way of automating must be provided. For this the automation framework Ansible is used. The process of automation can be seen in the following figure 3.11.

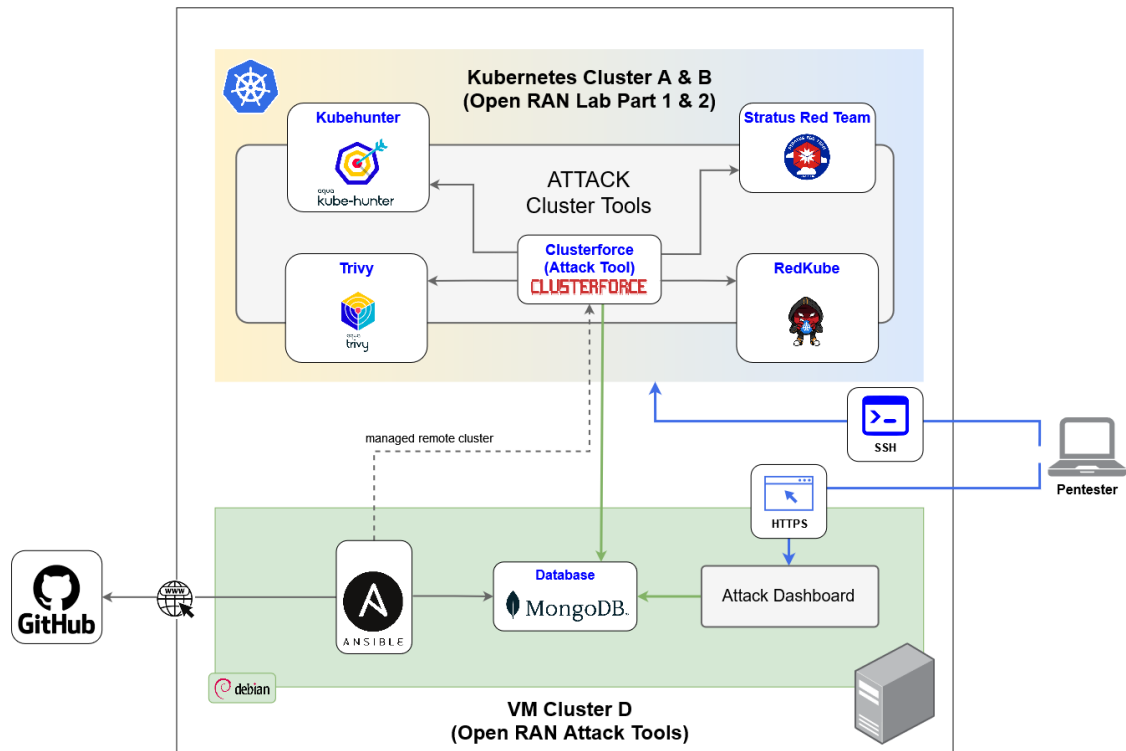


Figure 3.11.: Framework and Database Automation with Ansible

The figure is divided into two areas. The yellow-blue box represents clusters A and B, on which the O-RAN environment and the CF-Framework are deployed, while the green box shows attack cluster D, on which the database instance and the Ansible automation tool is running. Cluster D manages the automation with Ansible by updating both the database locally and the CF-Framework remotely on clusters A and B as required. To do that, Ansible retrieves the current version from Github and overwrites the old CF-Framework version with it.

3.5. Test strategy

Testing is divided into four areas. Firstly, the CF-Framework is tested for basis functionality. To do this, two use cases are to be tested: navigating through the menu and interacting with the metadata, including setting and retrieving it.

Secondly, follows a proof of concept testing, where Kubehunter is tested for functionality by looking at essential features such as setting parameters, executing an attack and evaluating the result in the database.

Thirdly, the features template, campaign and the execution of an attack in the context of a phase are tested. To do that all possible applications of the features, such as their commands for execution, are carried out and the result is analysed for completeness.

Fourthly, it should be possible to execute Tools, which are pre-mapped to a certain attack phase, so that the attack is also classified directly into one of the phases described in the section 2.1.5 in addition to the classification generated by the parser. This is tested by executing a tool over the menu entry attack phase, where the tools for the phases are listed, and checking the entry in the database.

4. Implementation

The implementation of the CF-Framework described below was done in collaboration with Jonas Arn Dieterich. While the architecture of the CF-Framework was mainly implemented in the context of this thesis, he implemented most of the tools provided by the CF-Framework. Some of the tools implemented by him can be seen in some of the figures, but are not part of this thesis and therefore will not get explained. Furthermore, the presented listings of the classes and methods are compromised on the fundamental functions.

4.1. Framework Architektur

4.1.1. Controller

To implement the **controller** module as described in 3.2.2, four classes have been implemented. Figure 4.1 shows the Python files belonging to the **controller** module. In addition, there is the file `__init__.py`, which is responsible for Python treating the directory as a module.

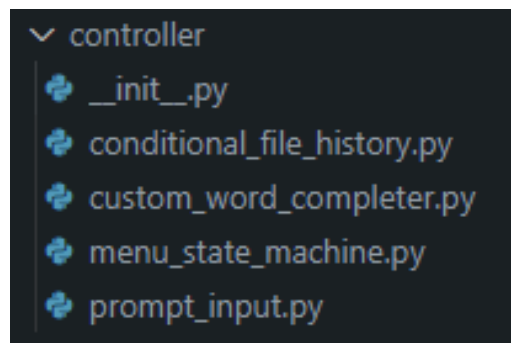


Figure 4.1.: Controller module - Files

The files contain the **ConditionalFileHistory**, **CustomWordCompleter**, **MenuStateMachine**, and **PromptInput** classes. Since the **CustomWordCompleter** and **ConditionalFileHistory** classes only provide part of the **PromptInput** class's library, they will be not mentioned further. The

PromptInput and **MenuStateMachine** classes are explained in more detail below.

PromptInput

The listing 4.1 displays a part from the **PromptInput** class, with the `get_input(prompt_text)` method. The `prompt_text` parameter defines the text preceding the input field, indicating the user's location in the menu. The default value for CF-Framework start is (START). For instance, if the user navigates from the Start menu state to Tools, the `prompt_text` changes to (START/TOOLS).

```

1 class PromptInput:
2     def __init__(self):
3         self.completer = None
4         self.history = None
5
6     def get_input(self, prompt_text):
7         if self.history is not None:
8             # Tools
9             session = PromptSession(
10                history=self.history,
11                completer=self.completer,
12            )
13        else:
14            # Menu
15            session = PromptSession(
16                completer=self.completer,
17                complete_style=CompleteStyle.READLINE_LIKE,
18            )
19
20        return session.prompt(prompt_text)

```

Listing 4.1: PromptInput class

The method initially verifies whether the class has a history object of the **ConditionalFileHistory** class. This is true when the CF-Framework is in the state of a tool. When a tool is executed, the entered commands for the tool are saved so that they can be executed again using the up and down arrow keys. Therefore, if an history object is available, a session object is created that supports a command history. However, if the CF-Framework is in the Menu state, a command history is not necessary, and the session object is created without a history. Despite this, Session includes a completer, which is an object of the **CustomWordCompleter** class, to provide the functionality to complete predefined keywords via the tab key.

Finally, the method returns the corresponding session, which is displayed by the `MenuStateMachine` class, which is explained below.

MenuStateMachine

Listing 4.2 below shows the implementation of the `MenuStateMachine` class with the class variables.

```

1 class MenuStateMachine:
2     def __init__(self):
3         self.current_state = Start()
4         self.global_var = GlobalVariables.get_instance()
5         self.prompt = PromptInput()
6         self.attack_generator = AttackGenerator()
7         self.campaign = Campaign.get_instance()

```

Listing 4.2: MenuStateMachine class - Variables class

As shown in the listing, the first variable is `current_state` (line 3), which stores the current state of the CF-Framework. Next the class contains the variable `global_var`, so the class can access the current metadata. In addition, the class contains the variable `prompt`, which provides the session associated with the state. Finally, it contains the variables `attack_generator` and `campaign`, to provide the template and campaign feature.

The class contains the method `run()`, which is responsible for running the CF-Framework and divides the incoming tasks, based on the user input to the right modules. The method is displayed on the listing 4.3 below.

```

1 def run(self):
2     try:
3         while True:
4             # Completer
5             self.prompt.set_completer(self.current_state.
6                 completer)
7             # History
8             self.prompt.set_history(self.current_state.history)
9             # User Input
10            user_input = self.prompt.get_input(helper.
11                create_cursor())
12            # Handle Input
13            if user_input.strip() == "exit":
14            elif user_input.strip() == "template":
15                self.attack_generator.generate_script()
16            elif re.search(r'\bcampaign\b', user_input):
17                try:

```



```

18         value = user_input.split(" ", 1)[-1]
19         if value.strip() == "info":
20             self.campaign.print_menu()
21         elif value.strip() == "start":
22             self.campaign.start()
23         elif value.strip() == "end":
24             self.campaign.end()
25     else:
26         new_state = self.current_state.
27         handle_user_input(user_input)
28         # Check if new state is a Tool
29         if isinstance(new_state, tool.Tool):
30             # Check tool is installed
31             if not new_state.check_requirements():
32                 print("Install Tool first")
33             else:
34                 self.current_state = new_state
35     else:
36         self.current_state = new_state

```

Listing 4.3: MenuStateMachine class - Run method

Inside the method, first the completer and the history from the current state are set. With that the prompt variable executes the *get_input()* method (Listing 4.1), to create the prompt where a user can enter the commands. When a command is entered, the user input is tested for some standard commands such as exit, to close the CF-Framework, template and campaign. When the user entered a command for navigating through the menu, or using a tool, the else condition in line 25 is reached. Now the user input must be evaluated by the corresponding state class. Each state class has implemented the *handle_user_input()* method for this purpose, which evaluates the input and returns the new state based on the evaluations (line 26-27). After receiving the new state, it is necessary to check whether its a tool or menu. When it is a tool, its necessary to check if the tool which the user wishes to access is installed on the system (line 29). At the end the *current_state* is updated with the new state (line 34 -36)

4.1.2. Tool

Figure 4.2 shows the **tool** module, which contains all implemented tools as separate subfolders.

The *base_classes* subfolder has three files: *local_env*, *tool_base*, and *wrapper_base*. Each file contains the base classes **LocalEnv**, **Tool**, and **Wrapper**, which are

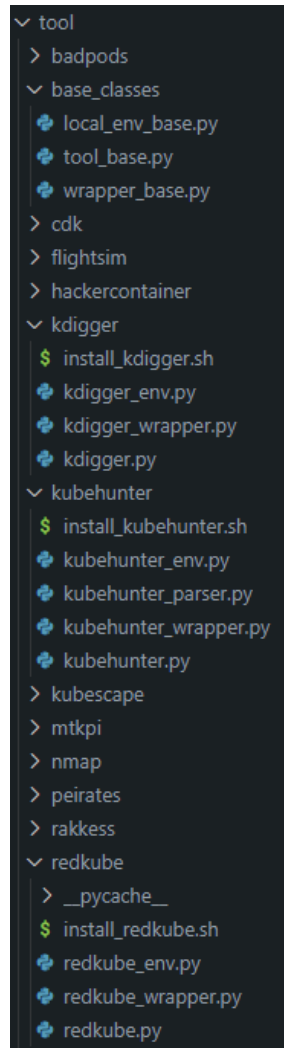


Figure 4.2.: Tool module - Files

described in 3.2.3. The subfolders for the tool implementations of Kubehunter, RedKube, and Kdigger each contain a bash script to install the tool itself. Additionally, they contain their subclass of the base class **LocalEnv**, **Tool**, and **Wrapper**. The Kubehunter subfolder is also supplemented by the *kubehunter_parser* file, which includes an initial parser implementation. Since the tool implementations only differ in terms of content and not implementation, Kubehunter is used to explain the tool implementation with all the necessary classes and files.

LocalEnv Base class

Listing 4.4 displays the implementation of the **LocalEnv** Base class. It includes two crucial methods: *set_variable_from_input()* and *to_json()*. The *set_variable_from_input()* method sets all local parameters when a user wants to change them, while the *to_json()* method transforms all values into a json object, in order to be

safed to the database.

```

1 class LocalEnv():
2     def set_variable_from_input(self, user_input):
3         command, variable, value = user_input.split(" ", 2)
4         new_var = self.translate_variable(variable)
5         # Check if Variable exists
6         if hasattr(self, new_var):
7             # Set Variable
8             setattr(self, new_var, value)
9             return True
10        else:
11            return False
12
13    def to_json(self):
14        data = {}
15        for attr_name, attr_value in self.__dict__.items():
16            key = attr_name.lstrip('_')
17            if attr_value is not None:
18                data[key] = attr_value
19
20    return data

```

Listing 4.4: LocalEnv Base class

To set the parameters the method *set_variable_from_input(user_input)* receives the user input. The method uses the user input and attempts to set the tool-specific parameter, Finally, the method determines if the variable to be set exists in the corresponding subclass by calling *hasattr* (line 6). When that is the case, the variable gets set below with *setattr* (line 8).

KubehunterEnv subclass

The following listing 4.5 displays the Kubehunter subclass **KubehunterEnv** for the base class **LocalEnv**.

```

1 class KubehunterEnv(LocalEnv):
2     def __init__(self):
3         super().__init__()
4         self._log_level = None
5         self._active_mode = "passive"
6         self._quick = False
7         self._report = "json"
8         self._mapping = False
9         self._statistics = False

```

```
10 self._subnet = "/24"
```

Listing 4.5: KubehunterEnv subclass - Parameter

Inside the subclass all Variables are defined, that can be set by an user via the inherit `set_variable_from_input(user_input)` method. For the subclass **KubehunterEnv**, an user can set these seven tool specific parameters. As default only the variable `active_mode`, `report` and `subnet` are initialised by the CF-Framework.

Wrapper Base class

The listing 4.6 below shows the implementation of the **Wrapper** base class.

```
1 class Wrapper():
2     def _execute_command(self, cmd, sudo=False, cwd_path=None):
3         # Execute Command
4         output = subprocess.Popen(
5             split_cmds,
6             cwd= cwd_path,
7             stdout=subprocess.PIPE,
8             stderr=subprocess.PIPE,
9             text=True,
10            bufsize=1,
11            universal_newlines=True
12        )
13        # Thread usage, to display output in realtime
14        output_lock = threading.Lock()
15        error_lock = threading.Lock()
16        error_thread = threading.Thread(...).start().join()
17        output_thread = threading.Thread(...).start().join()
18
19        output.wait()
20
21        # Safe Output
22        captured_output = ''.join(captured_output_lines)
23        captured_error = ''.join(captured_error_lines)
24
25        return Output(captured_output, captured_error,
26                    output.returncode)
```

Listing 4.6: Wrapper Base class class

It provides the subclasses with the method `_execute_command()`, which evaluates and executes the tool specific command. The standard library subprocess integrated in Python is used for this, which executes commands on the command line (line 4-12). To smoothly display the output of the tool, threads are used, which process this task

in the background, so the prompt is not freezing in the meantime (line 13-23). On this way the experience of using the tool in the CF-Framework corresponds to the experience of using the tool outside of the CF-Framework.

KubehunterWrapper subclass

The following listing 4.7 presents the implementation of the **KubehunterWrapper** subclass, which is based on the **Wrapper** base class.

```

1 class KubeHunterWrapper(Wrapper):
2     def __init__(self, env):
3         self.command = 'kube-hunter'
4         self.sudo = False
5         self.env = env
6
7     # Info
8     def help(self):
9         cmd = ['--help']
10        return self._execute_command(cmd, False)
11
12    # Scans
13    def remote_scan(self):
14        cmd = ['--remote']
15        return self._execute_command(
16            self._build_command(cmd, log_level=True,
17                                active_mode=True, << other FLAGS>> ))
18    def cidr_scan(self):
19        cmd = ['--cidr']
20        return self._execute_command(self._build_command(
21            cmd, <<FLAGS>>))
22    def interface_scan(self):
23        cmd = ['--interface']
24        return self._execute_command(self._build_command(
25            cmd, <<FLAGS>>))
26    def pod_scan(self):
27        cmd = ['--pod']
28        return self._execute_command(self._build_command(
29            cmd, <<FLAGS>>))

```

Listing 4.7: KubehunterWrapper subclass - Variables and pre-parameterised methods

As can be seen in the listing, the subclass has the following three variables:

- *final_command*, includes the shortcut to invoke the tool on the command line, in this case kube-hunter
- *sudo*, specifies whether a tool should be executed with highest privileges
- *env*, contains the LocalEnv subclass, to access the current parameters and flags

In addition to the variables, the subclasses always provides a *help* method. This implements the help or info command of the tools. This way, if a user wants to enter commands in addition to the pre-parameterised methods, he can display all options, which can be used for the tool. Furthermore, the subclass provides the pre-parameterised methods:

- *remote_scan()*, scans a cluster from outside for initial information
- *cidr_scan()*, scans a specific range of hosts, specified with the flag *subnet*
- *interface_scan()*, scans all network interfaces available
- *pod_scan()*, creates a privileged pod inside the cluster to scan.

To build the command, for the pre-parameterised methods, the *_build_command()* method is used inside of *_execute_command()* method. As parameter all, predefined flags are also transferred, which in the listing is marked as «*FLAGS*». The *_build_command()* method can be seen in the following listing 4.8.

```

1 def _build_command(self, cmd, <<FLAGS>>):
2     if log_level:
3         if self.env.get_log_level() in ["info", "warn", "debug"]:
4             cmd.extend(['--log', self.env.get_log_level])
5     if active_mode:
6         if self.env.get_active_mode() == "active":
7             cmd.append("--active")
8
9     # Further Flags
10    ...

```

Listing 4.8: KubehunterWrapper subclass - Build command

The function of the method is explained using the two flags *active_mode* and *log_level*. At the start, the method checks the pre-parameterised flags to determine how the command should be executed, by examining the flags passed in the if-conditions. When the flag is set, the next step is to check the value stored for the parameter in the associated **LocalEnv** class. If the user has not changed the value yet, it remains in the default configuration. However, if they want to execute the pre-parameterised method with a different value, they must adjust the parameter value via the console.

For instance, if a user specifies a log level that matches the requirements, i.e. has a value of info, warn or debug, the command will set the log level flag with the specified value. The same process is used for the *active_mode* flag. Since the default value of Kubehunter for this flag is passive, the flag gets added, when the flag is set by the user as active.

The building of an command is explained using the *remote_scan()* method with the flag *log_level*. On the listing 4.7 inside the method, the *-remote* flag set (line 14). The default value for the log level is none, as displayed in the listing 4.5, but if the user sets it to info, *-log info* will be added to the command and would be like *-remote -log info*. This way the command is constructed step by step with the parameters in **LocalEnv**. Finally, the method returns the complete command to the *__execute_command()* method, and the command gets executed.

Tool Base class

The **Tool** base class contains shared methods and variables for all implemented tools, as well as abstract methods that must be implemented in each tool's class. The listing will focus on the most important variables, the method for utilizing user input and inserting an attack into the database.

Variables

The most important variables are display in the listing 4.9.

```

1 class Tool(ABC):
2     def __init__(self):
3         # Global Variable Instance
4         self.global_var = GlobalVariables.get_instance()
5         self.global_env = self.global_var.get_env()
6         # Template
7         self.template = Template().get_instance()
8         # Campaign
9         self.campaign = Campaign.get_instance()
10        # Parser
11        self.parser = None
12        # DB
13        self.db = MongoDBWrapper(self.global_var.get_base_dir() +
14                                "database/config.ini")
15        # Commands
16        self.command_mapping = None

```

Listing 4.9: Tool Base class - Variables

The global variables *global_var* and *global_env* are particularly important, because they give access to the metadata or the status of the CF-Framework.

The next two variables, *template* and *campaign*, are used to implement the features of template generation and interaction with campaigns. Both variables are declared as singleton objects, which is a design pattern that ensures only one object of a class exists. In order to access the singleton object the *get_instance()* method is called, so all classes always access the same state and the same metadata values.

In order to guarantee access to the database, the class also includes the variable *db* which is an object of the database wrapper class.

The final variable, *command_mapping*, is a dict, which stores data in key:value pairs [33], that must be implemented by each tool subclass. The key is the user input and the value is the associated method, which needs to be executed.

Methods

In addition to the variables, the class provides the *handle_user_input()* method to handle the user input, as shown in listing 4.10.

```

1 def handle_user_input(self, user_input):
2     if user_input.strip() == "menu":
3         self.menu()
4     elif re.search(r'\bset\b', user_input):
5         self.local_env.set_variable_from_input(user_input)
6     elif user_input.strip() == "options":
7         self.settings()
8     else:
9         if user_input in self.command_mapping_db_ignore_db:
10            # handle help comands
11            self.out = self.command_mapping_db_ignore_db
12            [user_input]()
13        elif user_input in self.command_mapping:
14            # handle every case thats executes attack
15
16            # set timestamp start
17            self.global_env.set_timestamp_start(helper.
18            get_current_timestamp())
19            self.out = self.command_mapping[user_input]()
20
21            # Evaluate Output
22            if self.out:
23                # set timestamp end
24                self.global_env.set_timestamp_end(helper.
25                get_current_timestamp())

```



```

26         # set result code
27         if self.out.return_code is not None:
28             self.global_env.set_result_code(self.out.
29             return_code)
30
31         # handle output
32         if self.out.return_code == 0:
33             # add to template
34             self.template.add_to_result_list(
35             Result(<<METADATA>>))
36             # insert output into database
37             self.database_insert()
38     return self

```

Listing 4.10: Tool Base class - Handle user input

The method filters the input to determine whether the user wants to display the menu, options, or configure the tool specific parameters. It does this by checking for the keywords *menu*, *options*, and *set*. If none of these are entered, there are only two other possibilities. The user can call up a pre-parameterised method by entering a command.

Alternatively, the user can enter his own command to use the tool independently of the pre-parameterised method. The pre-parameterised methods are divided into two categories: commands that are helping commands like *help* (line 9-12) and those that execute an attack and therefore are written to the database (line 13-37). In both cases, an *command_mapping* dict gets provided with a keyword, so that the correct method can be found and executed.

To handle an executed command for an attack, first, the time at which the command was executed is saved. After that the command is executed and the output is saved to the variable *out* (line 19). Next, the variable is evaluated (line 22), the time is stopped (line 24-25), and the result code is saved (line 28-29). If the command is successful, the result code will be 0. In this case, all metadata resulting from the attack will be saved in the database (line 27-37).

To accomplish this, the method *database_insert()* is executed, as shown in listing 4.11.

```

1 def database_insert(self):
2     # connect
3     self.db.connect_db("collection")
4     # set Metadata
5     self.global_env.set_command(self.wrapper.final_command)
6     self.global_env.set_file_output_raw(self.out.captured_output)
7

```

```

8     # check if tool has parser
9     if self.parser is not None and self.global_env.
10    get_file_format() == "json":
11        self.parse_output()
12
13    metavalues = self.global_env.to_json(output=True)
14
15    # add campaign
16    if self.campaign.get_id():
17        metavalues["campaign_id"]=self.campaign.get_id()
18
19    # Insert
20    metavalues.update(self.local_env.to_json())
21    self.db.insert_document(metavalues)
22    self.global_env.set_variable_to_default(
23        self.global_env.translate_variable("file_output_raw"))
24
25    # Close Connection
26    self.db.close_connection()

```

Listing 4.11: Tool Base class - Insert into database

The method first establishes a connection to the database (line 3). After that the metadata get collected (line 5-6). When the tool, in which the attack got executed, provides a parser, the output gets parsed and added to the metadata (line 9-11). Is the attack part of a campaign, the campaign ID also gets added as a field to the metadata (line 16-17). At the end all accumulated information gets inserted into the database, by using the *insert_document()* method of the *db* variable (line 20-23), and the connection to the database is closed (line 26).

Additionally, the base class, defines three abstract methods that must be implemented by every subclass, which are shown in the listing 4.12.

```

1 @abstractmethod
2 def check_installed(self):
3     pass
4
5 @abstractmethod
6 def install_tool(self):
7     pass
8
9 @abstractmethod
10 def tool_version(self):
11     pass

```

Listing 4.12: Tool Base class - Abstract Methods

Method `check_installed` is used to check whether a tool is installed, method `install_tool` is used to install the tool, and `tool_version`, to display to current version of the tool.

Kubehunter subclass

The Kubehunter subclass is based on the **Tool** base class and evaluates user input for the Kubehunter tool. The subclass contains all variables and methods to handle user input for Kubehunter.

Variables

The variables are displayed in the following listing 4.13.

```

1 class Kubehunter(Tool):
2     def __init__(self):
3         super().__init__()
4         # Wrapper, LocalEnv, and Parser subclasses
5         self.local_env = KubehunterEnv()
6         self.wrapper = KubeHunterWrapper(self.local_env)
7         self.parser = KubehunterParser()
8
9         # Command Mapping
10        self.command_mapping = {
11            "remote": self.handle_remote,
12            "pod": self.handle_pod,
13            "cidr": self.handle_cidr,
14            "interface": self.handle_interface,
15        }
16        self.command_mapping_ignore_db = {
17            "--help": self.handle_help,
18            "help": self.handle_help,
19            "--list": self.handle_list,
20            "list": self.handle_list,
21        }
22        # Prompt
23        self.history = ConditionalFileHistory(self.global_var.
24        get_base_dir() + "history/.prompt_history_kubehunter")

```

Listing 4.13: Kubehunter subclass - Variables

In general, each subclass includes a variable that accesses tool-specific parameters via a **LocalEnv** subclass (in this case, **KubehunterEnv**, line 5), a variable assigned to the corresponding wrapper class (in this case, **KubehunterWrapper**, line 6) to interact with the tool and execute commands, and, if possible, a variable for the

corresponding parser class (in this case, **KubehunterParser**, line 7) to parse the results.

In addition, each tool contains the variables *command_mapping* (line 10-15) and *command_mapping_ignore_db* (line 16-20), which contains methods for the user input. For example when the user enters the command remote, the command gets handed down to the *command_mapping* dict as keyword. Does such a keyword exist, the dict calls the corresponding method. In this case, it would be the method *handle_remote*.

Methods

The *handle_remote* method is one of the four methods, that are implemented for handling all pre-parameterised commands and is shown in the following listing 4.14.

```

1 # Abstract Methods
2 def check_installed(self):
3     try:
4         subprocess.check_output(['kube-hunter', '--help'])
5         return True
6     except FileNotFoundError:
7         return False
8
9 def install_tool(self):
10    subprocess.call(["bash", self.global_var.get_base_dir() +
11                    "tools/kubehunter/install_kubehunter.sh",
12                    self.global_var.get_base_dir(),
13                    self.global_var.get_tool_dir()])
14
15 def tool_version(self):
16    return "0.6.8"
17
18 # InputHandle Methods
19 def handle_remote(self):
20    return self.wrapper.remote_scan()
21
22 def handle_pod(self):
23    return self.wrapper.pod_scan()
24
25 def handle_cidr(self):
26    return self.wrapper.cidr_scan()
27
28 def handle_interface(self):
29    return self.wrapper.interface_scan()

```

Listing 4.14: Kubehunter subclass - Methods

Additionally, the subclass contains also the method *handle_pod*, *handle_cidr* and *handle_interface* to handle the other four pre-parameterised commands. Inside of the methods, the corresponding method, which are shown in the listing 4.7, gets executed.

Furthermore, each subclass must implement three abstract methods as specified by the base class. These methods as well as the methods to handle the user input are shown in the listing 4.12. To implement the *check_installed* method, the Kubehunter help command gets executed. When the command is successfully, the tool is installed. The next abstract method is the *install_tool* method. The installation commands are defined in the script *install_kunehunter.sh*. The third abstract method *tool_version* returns the current tool version. In this case the current Kubehunter version is 0.6.8.

KubehunterParser class

The structure of the parser class with its variables and methods is shown in listing 4.15.

```

1 class KubehunterParser:
2     def __init__(self):
3         self.data = None
4         self.mapping = {
5             "generalsensitiveinformation": {"TA0001": ""},
6             "exposedensitiveinterfaces":
7             {"TA0001": ["T1133", "MS-TA9005"]},
8             ...
9         }
10
11     def parse_vulnerabilities(self):
12         vulnerabilities = self.data.get("vulnerabilities", [])
13         val = set()
14         for vulnerability in vulnerabilities:
15             subcategory = None
16             if vulnerability.get("category"):
17                 if len(self.split_and_clean(vulnerability.
18                     get("category"))==2:
19                     subcategory = self.split_and_clean(
20                         vulnerability.get("category"))[1].
21                         replace(" ", "").lower()
22
23                 val.add(self.get_info(subcategory))
24
25         mitre = []
26         cve = []

```

```
27     for item in val:
28         ta_number, ta_value1, ta_value2, cve_identifier = item
29         mitre.append({'ta_number': [ta_value1, ta_value2]})
30         cve.append(cve_identifier)
31     return mitre, cve
```

Listing 4.15: KubeHunterParser class

The class consists of two variables: *data*, which contains the data (line 3) for the result, and *mapping* (line 4-9), which provides a dict for mapping keywords to CVE and MITRE.

In addition, the class contains the method *parse_vulnerabilities* to evaluate the data using the dict in the *mapping* variable. The result is searched for certain keywords and the corresponding MITRE and CVE entries are extracted from the data and returned.

4.1.3. Menu

Figure 4.3 shows the **menu** module with the two subfolders *phase* and *startmenu*. In addition, the base class is contained in the *menu_base.py* file. Each class in the subfolders represents a state that the menu can have. The implementation of the base class and the implementation of the corresponding subclasses are described below. The structure of the subclasses is illustrated using the **Start** subclass.

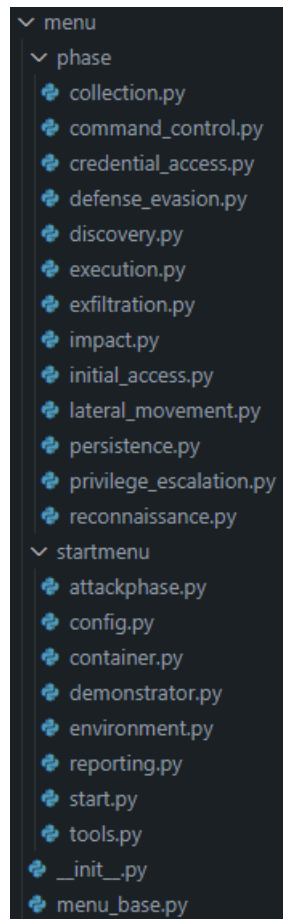


Figure 4.3.: Menu module - Files

Menu Base class

Listing 4.16 shows the base class of the **menu** module, which includes the variable *command_mapping*(line 3). Again this variable contains a corresponding method for every user input possible and every subclass needs to implement this variable.

```

1 class Menu(ABC):
2     def __init__(self):
3         self.command_mapping = None
4 
```

```

5     # Abstract Methods
6     def handle_user_input(self, user_input):
7         return self.command_mapping.get(user_input)()

```

Listing 4.16: Menu Base class

Similar to the tool base class, the menu base class also contains the *handle_user_input()* method (line 6-7), which is used in the **MenuStateMachine**. Since user input is limited to numbers that select a menu item, the method is shorter than the method in the tool base class.

Start subclass

Listing 4.17 displays the implementation of the **Start** subclass.

```

1 class Start(Menu):
2     def __init__(self):
3         super().__init__()
4         self.command_mapping = {
5             "1": self.attack_phase,
6             "2": self.tools,
7             "3": self.demonstrator,
8         }
9
10    # InputHandle Methods
11    def attack_phase(self):
12        return AttackPhase()
13    def tools(self):
14        return Tools()
15    def demonstrator(self):
16        return Demonstrator()

```

Listing 4.17: Start subclass

Similar to the **Tool** subclasses, the variable *command_mapping* is defined here as a dictionary. Each user input between one and seven corresponds to a specific method. For instance, the *attack_phase* method is referenced for user input one, which returns as need state the class **AttackPhase**. This allows the menu to be called by the **AttackPhase** subclass in the next run. This statement highlights a significant difference between the Tool subclasses and menu subclasses. Tool subclasses refer to themselves to remain in the tool, whereas menu classes always refer to another menu state.

4.1.4. Database

The class `MongoDBWrapper` provides interfaces to the database, establishing connection and writing data.

Establish connection

Listing 4.18 displays the connection setup.

```

1 class MongoDBWrapper:
2     def __init__(self, config_file):
3         self.config_file = config_file
4         self.client = None
5         self.database = None
6         self.collection = None
7
8     def connect_db(self, collect):
9         config = configparser.ConfigParser()
10        config.read(self.config_file)
11        host = config.get('MongoDB', 'host')
12        port = config.getint('MongoDB', 'port')
13        cert = config.get('MongoDB', 'client-cert')
14        ca_cert = config.get('MongoDB', 'ca-cert')
15        self.database=config.get('MongoDB','database')
16        self.collection = config.get('MongoDB',collect)
17
18        uri = f"mongodb://{host}:{port}/{database}?
19        authMechanism=MONGODB-X509&retryWrites=true&w=majority"
20
21        self.client = MongoClient(
22            uri,
23            tls=True,
24            tlsCertificateKeyFile=f"{cert}",
25            tlsCAFile = f"{ca_cert}",
26            tlsAllowInvalidHostnames=True
27        )

```

Listing 4.18: DatabaseWrapper class - Establishing connection

To connect to the database the method `connect_db method` is used. Inside the method the connection parameter `host`, `port`, `cert`, `ca_cert`, `database` and `collection` are configured (line 9-16). After that the connection is established (line 21-27). To connect to the database the database wrapper used certificates for authentication (line 24-26)

Write data

To write data into the database the *insert_document()* and *update_document()* methods are used. While the *insert_document()* method is used for writing new entries into the database, the *update_document()* method is used to update existing entries. The methods are shown in the listing 4.19.

```

1 def insert_document(self, document):
2     if self.client:
3         db = self.client[self.database]
4         coll = db[self.collection]
5         result = coll.insert_one(document)
6
7         return result.inserted_id
8
9 def update_document(self, document_id, key_to_update, new_value):
10    if self.client:
11        db = self.client[self.database]
12        coll = db[self.collection]
13
14        # Construct an update query to update the document
15        update_query = {"_id": document_id}
16
17        # Construct an update operation to set a new value
18        update_operation = {"$set": {key_to_update: new_value}}
19
20        # Create an UpdateOne object
21        update_request = UpdateOne(update_query, update_operation)
22
23    return coll.bulk_write([update_request])

```

Listing 4.19: DatabaseWrapper class - Write data

To write an entry into the database the *insert_document()* specifies the database and collection, where the data should be written to (line 3-4). Afterwards the data is inserted into the correct database and collection (line 5).

To update fields within a database entry, the *update_document()* method necessitates the *document_id*, *key_to_update*, and *new_value* as parameters (line 9). Subsequently, the database query is generated by creating the query and defining the new value (line 15-21). At the end the entry is updated (line 23).

4.1.5. State

The `MenuState` class records all possible states that a state can assume. It is implemented as an enum, which defines predefined constants for each state. Listing 4.20 illustrates that each state is initialized with an integer value.

```
1 class MenuState(Enum):
2     #Menu Base 0 - 10
3     START = 0
4     ATTACK_PHASE = 1
5     TOOLS = 2
6     DEMONSTRATOR = 3
7     ...
8
9     #Menu Attack Phase 20 - 32
10    RECONNAISSANCE = 20
11    INITIAL_ACCESS = 21
12    EXECUTION = 22
13    PERSISTENCE = 23
14    PRIVILEGE_ESCALATION = 24
15    DEFENSE_EVASION = 25
16    CREDENTIAL_ACCESS = 26
17    DISCOVERY = 27
18    LATERAL_MOVEMENT = 28
19    COLLECTION = 29
20    COMMAND_CONTROL = 30
21    EXFILTRATION = 31
22    IMPACT = 32
23
24    #Tools 40 - 60
25    KUBEHUNTER = 40
26    ...
27    REDKUBE = 44
28    ...
29    KDIGGER = 46
30    ...
```

Listing 4.20: MenuState class

This can be used to determine whether the status is a tool or a menu, by checking if the integer value falls between 0-32.

4.1.6. Environment

The module `environment` consists of two classes: `Environment` and `GlobalVariables`.

Environment

Listing 4.21 displays the `Environment` class, which contains all metadata which can be gathered in the background.

```
1 class Environment:
2     def __init__(self):
3         self._ip = helper.get_ip_address()
4         self._subnet = None
5         self._file_format = None
6         self._file_output_raw = None
7         self._file_output_parsed = None
8         self._command = None
9         self._tool_name = None
10        self._tool_version = None
11        self._timestamp_start = None
12        self._timestamp_end = None
13        self._attack_location = None
14        self._perspective = None
15        self._attack_phase = None
16        self._oran_component = None
17        self._result_code = None
18        self._hostname = helper.detect_hostname()
19        self._cve = None
20        self._mitre = None
```

Listing 4.21: Environment class - Metadata variables

When the CF-Framework is started, the `ip` (line 3) and `hostname` (line 18) variables are initialised with the IP address and host name of the host system. The remaining variables are initialised as empty, and can either be manually set via the console, or get assigned by the CF-Framework automatically when an attack is executed.

Global Variables

The **GlobalVariables** class consists of global variables and is also implemented as a singleton, as shown in figure 4.22.

```
1 class GlobalVariables:
2     _instance = None
3
4     @staticmethod
5     def get_instance():
6         if not GlobalVariables._instance:
7             GlobalVariables._instance = GlobalVariables()
8         return GlobalVariables._instance
9
10    def __init__(self):
11        self.menu_state = MenuState.START
12        self.env = Environment()
```

Listing 4.22: GlobalVariables class

It includes the current state, which is initialized with the start state from the **MenuState** class (line 11) and a variable of the **Environment** class (line 12). This means when a the controller or tool module wants to access the current metadata or CF-Framework state, it must be retrieve with the instance of the **GlobalVariables** class.

4.1.7. Features

Campaign

Besides the parser, the CF-Framework should implement the campaign and template feature. The campaign feature includes the **Campaign** class, as shown in the listing 4.23.

```
1 class Campaign:
2     _instance = None
3
4     def __init__(self):
5         self.name = None
6         self.id = None
7         self.start_date = None
8         self.end_date = None
9         self.description = None
10        self.targets = []
11
12
```

```
13     @classmethod
14     def get_instance(cls):
15         ...
16
17     def start(self):
18         # Dialogue to request user input
19         self.start_date = get_current_timestamp()
20         self.name = input
21         ("Enter the name of the campaign: ")
22         self.description = input
23         ("Enter a description for the campaign: ")
24         targets_input = input
25         ("Enter a list of targets (IPs or domains): ")
26
27         if self.name:
28             self.database_insert()
29
30     def end(self):
31         # connect
32         self.db.connect_db()
33         # update DB
34         self.end_date = get_current_timestamp()
35         self.db.update_document(self.id, "end_date", self.end_date)
36
37     def database_insert(self):
38         # connect
39         self.db.connect_db("collection-campaign")
40         # insert campaign into database
41         self.id = self.db.insert_document(self.to_json())
```

Listing 4.23: Campaign class

To start a campaign the *start* method is used. Inside the method, first the *start_date* (line 19), is recorded, followed by the option for the user to specify parameters such as the *name* (line 20-21), *description* (line 22-23), and *targets* (line 24-25), which could be the IPs or host names. The provision of campaign name is mandatory, while the description and the targets are optional. If the user provides a name for the campaign, all campaign-specific data will be written to the database (line 27-28) using the *database_insert()* method (line 37-41). When the user ends the campaign, the end time should be added to the campaign entry in the database. To accomplish this, the *update_document()* (line 35) method described in listing 4.19 is used.

Template

The implementation of the Template feature comprises two classes: **Template** (Figure 4.24) and **AttackGenerator** (Figure 4.25).

In the following listing, the **Template** class is displayed.

```

1 class Template:
2     _instance = None
3
4     def __init__(self):
5         self._result_list = []
6
7     @classmethod
8     def get_instance(cls):
9         ...
10
11    def add_to_result_list(self, item):
12        self._result_list.append(item)
13        return self._adjacent_duplicates()
14
15    def _adjacent_duplicates(self):
16        if len(self._result_list) < 2:
17            return False
18
19        for i in range(len(self._result_list) - 1):
20            if self._result_list[i].__eq__(
21                self._result_list[i + 1]):
22                del self._result_list[i + 1]
23                return True
24
25        return False

```

Listing 4.24: Template class

Similar to the Campaign class, the **Template** class is also implemented as a singleton. In addition, the class contains only the variable *result_list* (line 5).

If an attack is executed, the command is saved in the list. In addition to the list the class includes methods to add new entries to the list (list 11-13) and to optimise the list of commands by removing duplicates (list 15-23). For that the *_adjacent_duplicates()* method checks whether the same command has been entered twice. When this is the case, it is not included in the list. The list is then used by the second class **AttackGenerator**, which is shown in the listing 4.25, to generate the template.

```
1 class AttackGenerator:
2     def __init__(self):
3         self.script_name = "template_attack_script-" +
4             get_timestamp() + ".sh"
5         self.template = Template().get_instance()
6         self.results = self.template.get_result_list()
7         self.commands = []
8
9     def generate_script(self):
10        if self.template.get_result_list():
11            self.gen_script_name()
12            self.get_commands()
13            with open(GlobalVariables.get_instance().get_base_dir()
14                    + "automation/scripts/volatile/"
15                    + self.script_name, 'w') as script_file:
16                script_file.write("#!/bin/bash\n")
17                for cmd in self.commands:
18                    cmd = " ".join(cmd)
19                    script_file.write(cmd + "\n")
20                script_file.write("echo 'Attack automation
21                completed.'\n")
22                script_file.close()
```

Listing 4.25: AttackGenerator class

The **AttackGenerator** class contains the script name, which consists of the prefix *template_attack_script* and the current time (line 3-4). Additionally, it contains an instance of the **Template** class (line 5) to access the list of commands. With the use of the **generate_script()** method the template script gets generate by writing all commands from the list to a template script and saving it (line 16-21), to make it available for a user to execute again via the console menu.

4.2. Database Architecture

In the MongoDB architecture, two databases with three collections are specified. The PROD database is used during testing alongside the DFIR tools, while the DEV database is used for CF-Framework development and testing. Each database should contain the following three collections.

```
dev> show collections
artifacts-campaign
artifacts-parsed
artifacts-raw
```

Figure 4.4.: DEV Collections

Firstly, the Collection *artifacts-campaign* contains the campaign entries, secondly the collection *artifacts-parsed* contains the parsed metadata entries and finally, the collection *artifacts-raw* contains the attack entries with all the metadata.

4.3. Automation

The Automation covers two areas. Both the CF-Framework and the database deployment need to be automated. The snippets of Ansible playbooks shown below cover the most important main tasks. However, the playbooks consist of a large number of other tasks that are necessary to ensure that the main tasks can be performed without error. These are not discussed further below. Since the CF-Framework needs to authenticate itself to access the database, the authentication has to be automated as well. Additionally, the creation of users, for the CF-Framework or to manage the database, has to be automated as well.

4.3.1. Deployment CF-Framework

Automate the Deploying the CF-Framework to the clusters is managed with Ansible and involves two steps. The first step is to clone the CF-Framework repository from Github, which is shown in the listing 4.26.

```

1 - name: Clone GitHub repository to Ansible Master
2   tags: update
3   git:
4     repo: git@git.dn.fh-koeln.de:foran/foran-attack.git
5     dest: /tmp/attack
6     version: main
7     force: yes
8     clone: yes
9     update: yes
10    accept_hostkey: yes

```

Listing 4.26: Ansible - Cloning the CF-Framework

The repository gets cloned to the local `/tmp/attack` directory (line 4-5).

The next step is to, distribute the cloned repository to the O-RAN cluster A and B. This is shown in figure 4.27 below.

```

1 - name: Copy the cloned repository to worker nodes
2   become: yes
3   tags: update
4   synchronize:
5     src: "{{ SOURCE_DIR }}"
6     dest: "{{ BASE_DIR }}"
7     recursive: yes
8     delete: yes

```

Listing 4.27: AttackGenerator class

To accomplish this task, the source directory, which contains the cloned repository (line 5), as well as the destination directory on the O-RAN cluster where the repository will be copied (line 6), must be specified. The *SOURCE_DIR* and *BASE_DIR* values are variables for the source and destination directories. The source directory in this case is the */tmp/attack* directory, where the repository is located.

4.3.2. Database

In the following section the automation of the deployment of the database and the generation of all authentication certificates is shown.

MongoDB instance

To deploy the database, two main actions are necessary: installing the MongoDB packages, starting the MongoDB service, and creating the corresponding users. Listing 4.28 illustrates the installation of the MongoDB packages and the start of the services.

```
1 - name: Install MongoDB packages
2   apt:
3     name:
4       - mongodb-org
5       - mongodb-mongosh
6     state: present
7     update_cache: yes
8
9 - name: Enable and start the Mongod service
10  tags: update
11  systemd:
12    name: mongod.service
13    daemon_reload: yes
14    enabled: yes
15    state: restarted
```

Listing 4.28: Ansible - Install MongoDB packages

The first step in the listing is to install the MongoDB packages *mongodb-org* and *mongodb-mongosh* (line 4-5). The next step is to start the MongoDB service by defining the name and desired status of the service. In this case, the name of the service is *mongod.service* (line 12) and the selected state is 'restarted' (line 15) to ensure the service is restarted if it is already running. If the service is not yet running, it will simply be started.

Authentication

The implemented MongoDB instance utilises X.509 client certificate authentication. There are two actors who require access to the database: a database administrator for managing and a database user with limited rights to certain resources for the CF-Framework to use. The database user is used by the CF-Framework to read from and write to the database. Listing 4.29 illustrates the creation of the database user.

```

1 - name: Add foran to "$external" DB for X.509 Auth
2   tags:
3     - user_exist
4   ansible.builtin.shell:
5     cmd: |
6       mongosh foran --host localhost --eval '
7         db.getSiblingDB("$external").runCommand({
8           createUser: "CN=foran",
9           roles: [
10            { role: "readWrite", db: "dev" },
11            { role: "readWrite", db: "prod" },
12          ],
13          writeConcern: { w: "majority", wtimeout: 5000 }
14        })'
15   when: foran_check.stdout == "not_exists"

```

Listing 4.29: Ansible - Add foran user to MongoDB

The username assigned to the user is *CN=foran* (line 8), which gets full read and write access to both the PROD and DEV databases (line 10-11). The user name *CN=foran* is assigned because it is compared with the corresponding certificate and the common name specified in it. Authentication is only possible if the common name matches the common name used in the certificate.

The listing also shows that the user is added to the *\$external* database (line 7). This is used when users authenticate to MongoDB via an external authentication mechanism such as x.509 Client Certificate Authentication. To generate the certificate associated with the user, the Community Crypto Collection module provided by Ansible is used, with which keys and certificates can be created and signed. The creation of the certificate can be seen in the following listing 4.30.

```

1 # Create foran key
2 - name: Create private key for foran
3   community.crypto.openssl_privatekey:
4     path: "{{ SSL_DIR }}/foran.key"
5     type: Ed25519
6
7 # Create CSR for foran

```

```
8 - name: Create CSR for foran
9   community.crypto.openssl_csr_pipe:
10     privatekey_path: "{{ SSL_DIR }}/foran.key"
11     common_name: foran
12   register: csr
13
14 # Sign foran certificate
15 - name: Sign foran certificate with CA
16   community.crypto.x509_certificate_pipe:
17     csr_content: "{{ csr.csr }}"
18     provider: ownca
19     ownca_path: "{{ SSL_DIR }}/ca.crt"
20     ownca_privatekey_path: "{{ SSL_DIR }}/ca.key"
21     ownca_not_after: +365d # valid for one year
22     ownca_not_before: "-1d" # valid since yesterday
23   register: certificate
24
25 # Create foran.crt
26 - name: Create foran.crt
27   copy:
28     dest: "{{ SSL_DIR }}/foran.crt"
29     content: "{{ certificate.certificate }}"
30
31 # Create foran.pem
32 - name: Create foran.pem
33   assemble:
34     src: "{{ SSL_DIR }}"
35     dest: "{{ SSL_DIR }}/foran.pem"
36     regexp: '^(foran\.key|foran\.crt)$'
37     delimiter: '\n'
```

Listing 4.30: Ansible - Create Authentication Certificate for foran user

The Ed25519 encryption algorithm, specifically designed for digital signatures and key exchange protocols [34], is used to create a private key (line 5).

After that, a Certificate Signing Request (CSR) is generated for the private key (line 15-23). This cryptographic file contains the public key and the identity details of a company, and is used to request a digital certificate from a certificate authority (CA). With a CA certificate, the created certificates get signed, ensuring a secure authentication [35]. The generation of the CA certificate is also automated, as we serve as our own authority. This means the CA certificate signs both the admin and user certificate.

As common name *foran* is the specified (line 11). In the next step, the signed certificate for the foran user gets created (lin 25-29). Because the final certificate requires both the signed certificate and the key, the final step is to merge the created certificate and key into a single file (line 32-37). This resulting certificate *foran.pem* is used by the CF-Framework for authentication.

4.4. User Manual

In order to facilitate user familiarity with the CF-Framework, a user manual has been created. This manual lists all commands and their functions, as shown in figure 4.5.

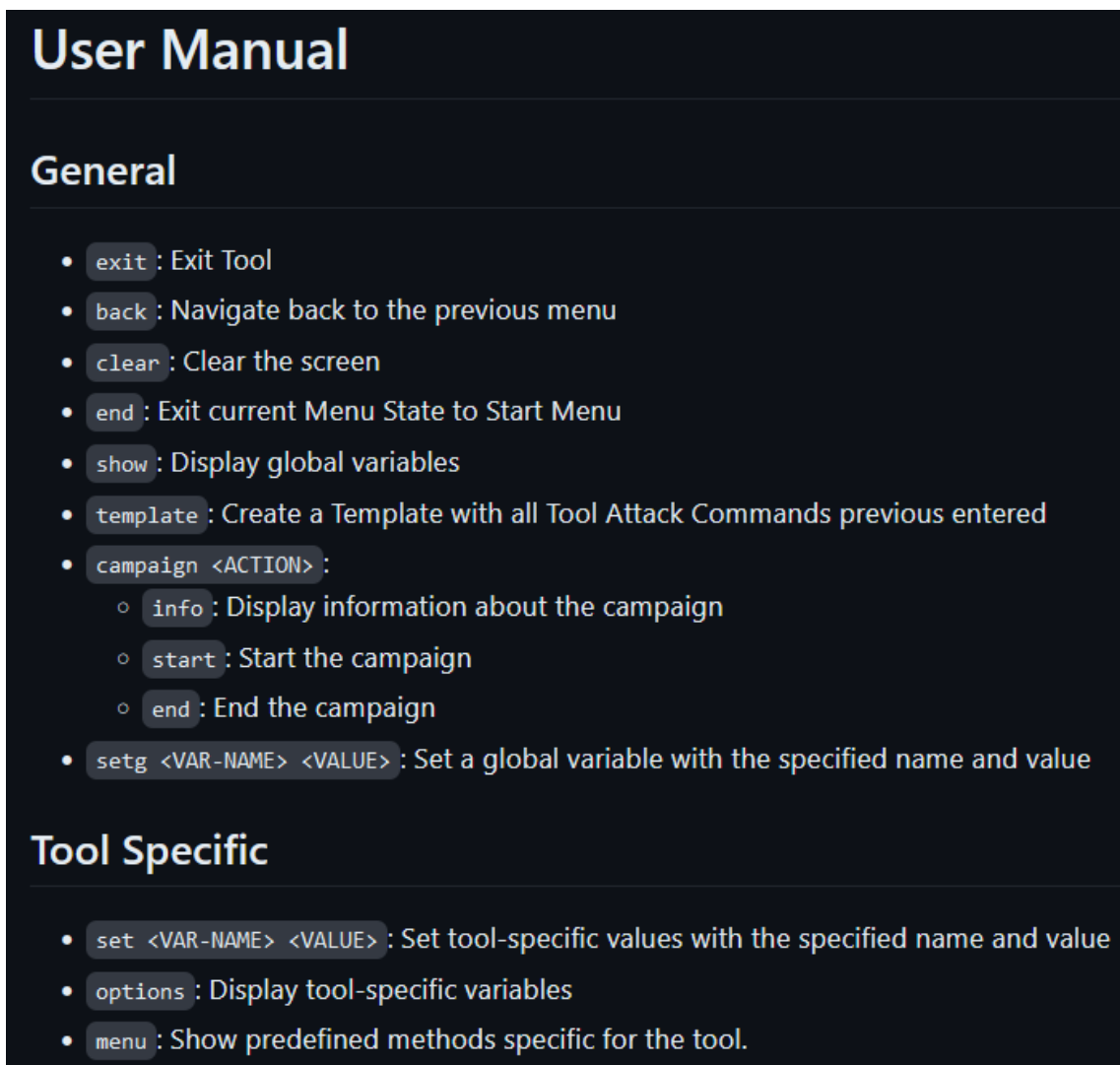


Figure 4.5.: CF-Framework User manual

The user manual divides the available commands into two categories: general com-

mands and tool-specific commands. General commands can be entered from any menu status and are not limited to a single tool. The following text describes the general commands:

- *exit*, to exit the CF-Framework
- *back*, to return to the previous menu state
- *clear*, to delete the history of the terminal
- *end*, to get to the start menu
- *show*, to display the metadata
- *template*, to create a Template with the Feature Template
- *campaign* <ACTION>, to create a campaign with the campaign feature. In doing so, <ACTION> can assume the three values info, start and end
- *setg* <VAR-NAME> <VALUE>, to set the metadata. The variable name and the desired value must be specified

In addition to the general commands, the CF-Framework also provides tool-specific commands, which are only available when the tool is being executed.

- *set* <VAR-NAME> <VALUE>, to set the tool-dependent parameters and flags
- *options*, to display the tool-dependent parameters and flags
- *menu*, to display the methods pre-parameterised for the tool

5. Testing

In Section 3.5 four areas for testing the CF-Framework are defined. These comprise the functional testing of the CF-Framework, the functional testing of tool, the testing of the implemented features and the testing of the execution of an attack via the attack phase menu to map the attack directly to a MITRE tactic. The test environment used is Cluster B with the Near Realtime RIC implemented on it.

5.1. Functional Testing CF-Framework

To test the CF framework for functionality, there are two use cases defined. The use cases include the following commands from the User Manual.

- *exit*
- *back*
- *end*
- *show*
- *setg*

Use Case: Menu Navigation

The figure 5.1 illustrates a use case in which a user navigates through different the menu states using the *back*, *end*, and *exit* commands.

```
(Start/Tools/Kubehunter)>> back

Menu options for TOOLS:
1: Kubehunter
  <<other Tools>>
5: Red-Kube
  <<other Tools>>
7: KDigger
  <<other Tools>>
(Start/Tools)>> end

Menu options for START:
1: Attack Phases
2: Tools
3: Demonstrator (One-Click Attack)
4: Environment
5: Reporting
6: Config
7: Container
(Start)>> exit
(venv) vagrant@near:~$
```

Figure 5.1.: Example Usage of Commands: back, end, exit

The Kubehunter tool is used by the user as shown in the figure. The user executes the *back* command to return to the menu, followed by the *end* command to go back to the start menu. Finally, the user exits the CF framework by executing the *exit* command.

Use Case: Metadata Interaction

The figure 5.2 illustrates a use case where a user can display and modify the metadata that has been collected up to that point.

```
(Start/Tools/Kubehunter)>> show

Parameter | Value
-----+-----
ip        | 10.0.0.22
tool_name | kubehunter
tool_version | 0.6.8
hostname  | near

(Start/Tools/Kubehunter)>> setg ip 10.0.0.22,10.0.0.50
(Start/Tools/Kubehunter)>> show

Parameter | Value
-----+-----
ip        | 10.0.0.22,10.0.0.50
tool_name | kubehunter
tool_version | 0.6.8
hostname  | near
```

Figure 5.2.: Pre-parameterised methods Kubehunter

To accomplish this task, the user first utilises the *show* command to display the metadata. The illustration shows that the host name, IP address, and the name and version of the tool currently have been set. The user now wishes to add the IP address 10.0.0.50. To do so, they use the command *setg ip 10.0.0.22,10.0.0.50*. Finally, the user displays the metadata again and it can be seen that the desired IP address has been successfully added. For instance, if an attack were to be carried out, both IPs would be utilised for the attack, if permitted.

5.2. Functional Testing Tool

The testing of an integrated tool is carried out below using Kubehunter. All essential functionalities of the tool, such as the setting of tool-specific parameters or the execution of an attack, are explained. The results are then analysed and interpreted. At the end, the entry in the database and the metadata it contains about the attack are analysed and the functionality of the parser is explained.

Pre-parameterised methods Kubehunter

One of the key features is the provision of pre-parameterised methods, as shown in figure 5.3 below.

```
(Start/Tools/Kubehunter)>> menu
```

Method	Description
--help or help	Show all available commands
--list or list	Displays all tests in kubehunter
remote	Hunts for weaknesses from outside the cluster.
pod	Hunts for weaknesses as a pod inside the cluster
cidr	Hunts for weaknesses inside the cluster. (default /24)
interface	Hunts for weaknesses inside the cluster, by scanning all interfaces

Figure 5.3.: Pre-parameterised methods Kubehunter

The text before the cursor displays the current position of the user inside the menu. In this case the user is located inside of Kubehunter. The status was reached via the Start and Tools menu items. To display the methods, first enter the *menu* command on the console. The four methods *remote*, *pod*, *cidr* and *interface* are offered for an attack. In addition, the *help* command can be used to display all available commands and *list* can be used to display all available tests.

Default Configuration of Kubehunter

Before executing an attack, we look at the default setting of the tool-specific parameters, which we can see in the following figure 5.4.

```
(Start/Tools/Kubehunter)>> options
```

Parameter	Value
IP	10.0.0.22
LOG_LEVEL	None
ACTIVE_MODE	None
QUICK	None
REPORT	json
MAPPING	False
STATISTICS	False
SUBNET	/24

Figure 5.4.: Default Configuration of Kubehunter specific Parameter

The parameters are displayed using the options command. The figure shows that an *IP*, a *subnet* and the *report* type of the result are set as parameters by default.

Parameter specification and attack

To execute an attack, a user can now set the parameters. For example, the *active_mode* is set for the attack. The setting of the parameter is shown in the following figure 5.5.

```
(Start/Tools/Kubehunter)>> set active_mode active
(Start/Tools/Kubehunter)>> options
```

Parameter	Value
IP	10.0.0.22
LOG_LEVEL	None
ACTIVE_MODE	active
QUICK	None
REPORT	json
MAPPING	False
STATISTICS	False
SUBNET	/24

```
(Start/Tools/Kubehunter)>> pod
['kube-hunter', '--pod', '--active', '--report', 'json']
```

Figure 5.5.: Set Kubehunter specific Attack Parameter and launch Attack pod

The mode is changed from *None* to *active* and the attack *pod* is executed. This

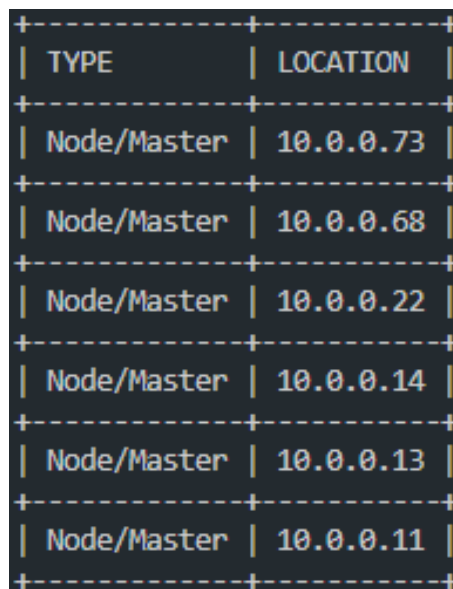
involves creating a pod with privileged privileges within the cluster, scanning the cluster for vulnerabilities and attempting to exploit those vulnerabilities using the active flag.

The command of the attack that was generated with the specified parameters can be seen at the end of the figure. It can be seen that both the `-active` flag for the active mode and `-report json` for the return type of the result are set. In order to be able to analyse the result, it is shown below in a human-readable format. This is derived from the resulting JSON object of the result.

The delivered result of the vulnerability analysis consists of three sub-results, Nodes Found, Services and Vulnerabilities.

Nodes

Figure 5.6 shows the nodes found.



```
+-----+
| TYPE          | LOCATION |
+-----+-----+
| Node/Master   | 10.0.0.73 |
+-----+-----+
| Node/Master   | 10.0.0.68 |
+-----+-----+
| Node/Master   | 10.0.0.22 |
+-----+-----+
| Node/Master   | 10.0.0.14 |
+-----+-----+
| Node/Master   | 10.0.0.13 |
+-----+-----+
| Node/Master   | 10.0.0.11 |
+-----+-----+
```

TYPE	LOCATION
Node/Master	10.0.0.73
Node/Master	10.0.0.68
Node/Master	10.0.0.22
Node/Master	10.0.0.14
Node/Master	10.0.0.13
Node/Master	10.0.0.11

Figure 5.6.: Vulnerability analysis - Nodes

As can be seen in the figure, a total of seven nodes has been found, which now can be investigated further.

Services

Next, the nodes are analysed for contained services, as shown in the following figure 5.7.

SERVICE	LOCATION	DESCRIPTION
Kubelet API (readonly)	10.0.0.73:10255	The read-only port on the kubelet serves health probing endpoints, and is relied upon by many kubernetes components
Kubelet API	10.0.0.73:10250	The Kubelet is the main component in every Node, all pod operations goes through the kubelet
Kubelet API	10.0.0.68:10250	The Kubelet is the main component in every Node, all pod operations goes through the kubelet
Kubelet API	10.0.0.22:10250	The Kubelet is the main component in every Node, all pod operations goes through the kubelet
Kubelet API	10.0.0.14:10250	The Kubelet is the main component in every Node, all pod operations goes through the kubelet
Kubelet API	10.0.0.13:10250	The Kubelet is the main component in every Node, all pod operations goes through the kubelet

Figure 5.7.: Vulnerability analysis - Services

The services themselves do not represent a vulnerability, but can be tested by the attacker for further vulnerabilities. For example, if a vulnerability is known for the

service, it may be possible to obtain sensitive data.

Vulnerabilities

Finally, the results provide the vulnerabilities found. These can be seen in the following figure 5.8.

ID	LOCATION	MITRE CATEGORY	VULNERABILITY	EVIDENCE
KHV044	10.0.0.73:10255	Privilege Escalation // Privileged container	Privileged Container	pod: calico-node-bbss5, container: calico-node, count: 1
KHV043	10.0.0.73:10255	Initial Access // General Sensitive Information	Cluster Health Disclosure	status: ok
KHV002	10.0.0.68:6443	Initial Access // Exposed sensitive interfaces	K8s Version Disclosure	v1.16.0
KHV002	10.0.0.22:6443	Initial Access // Exposed sensitive interfaces	K8s Version Disclosure	v1.16.0
KHV002	10.0.0.11:6443	Initial Access // Exposed sensitive interfaces	K8s Version Disclosure	v1.27.4
KHV052	10.0.0.73:10255	Discovery // Access Kubelet API	Exposed Pods	count: 55

Figure 5.8.: Vulnerability analysis - Vulnerabilities

The figure shows that six vulnerabilities were found. For each vulnerability, the location for the vulnerability, an associated MITRE tactic with the corresponding technique and the proof is listed. For example, a privileged container exists, which is available under 10.0.0.73:10255. An attacker could use this to perform operations

as a privileged user in order to obtain new information. It is important to note that because Kubehunter only displays the result and not the executed commands, the full extent of the traces generated can only be seen by analysing the logs filtered by the DFIR tools.

Database Entry

Together with all the collected metadata, this information is written to the database as an entry. The entry can be seen in the following figure 5.9.

```

_id: ObjectId("659833e53bd16fc2aa7ec8c3"),
ip: '10.0.0.22',
file_output_raw: '{"nodes": [{"type": "Node/Master", "location": "10.0.0.13"},
  "services": [{"service": "Kubelet API", "location": "10.0.0.13:10250"},
  "vulnerabilities": [{"location": "10.0.0.11:6443", "vid": "KIV002",
  "category": "Initial Access // Exposed sensitive interfaces"}
  ]}'

  <<Output>>
command: [ 'kube-hunter', '--pod', '--active', '--report', 'json' ],
tool_name: 'kubehunter',
tool_version: '0.6.8',
timestamp_start: ISODate("2024-01-05T16:52:25.705Z"),
timestamp_end: ISODate("2024-01-05T16:52:53.887Z"),
result_code: 0,
hostname: 'near',
mitre: [
  { TA0007: [ 'T1613', 'MS-TA9030' ] },
  { TA0001: [ ] },
  { TA0004: [ 'T1610', 'MS-TA9018' ] },
  { TA0001: [ 'T1133', 'MS-TA9005' ] }
],
active_mode: 'active',
report: 'json',
mapping: false,
statistics: false,
subnet: '/24'

```

Figure 5.9.: Database Entry for Vulnerability analysis

The result can be seen as a JSON object in the *file_output_raw* field, even though the output is limited for illustration purposes. The whole figure can be seen in the appendix B. The figure 5.9 also contains information such as the command used, the tool name and the time of the attack. You can also see that the result has already been parsed and a mapping to MITRE was successful. The result is written to the *mitre* field. The *mitre* field contains four figure tactics with corresponding techniques. The last value with the prefix MS corresponds to the tactic specified in the Microsoft Kubernetes ATT&CK matrix. As can be seen in 5.8, the attack found six vulnerabilities. However, the parser does not add duplicate entries. Although the *Initial Access* tactic with the *Exposed sensitive interfaces* technique occurs three

times, it is only included once in the matrix. This results in the four entries shown in the figure. It is noticeable that only the MITRE technique is included in one entry, while the Kubernetes ATT&CK matrix tactic and technique are excluded. This is due to Kubehunter assigning its own techniques to certain vulnerabilities, which are labelled as General. For instance, the second entry in figure 5.8 contains such a tactic, which is not recorded in the database during parsing.

5.3. Features Testing

The campaign, template, and tool execution are the features to be tested. The corresponding attack phase is mapped to the metadata. RedKube is used to test the Campaign feature, while both RedKube and Kubehunter are used to test the Template feature. Kdigger is used for the feature phase.

5.3.1. Campaign

The campaign feature has three key aspects that require testing. Firstly, it is important to test the starting and ending of a campaign. Additionally, an attack should be carried out using Redkube while a campaign is active. Finally, it is necessary to check both the *artifacts-raw* collection for attacks and the *artifacts-campaign* collection to ensure that new entries have been created. As the attack was carried out during an active campaign, it is necessary to include a field with the associated campaign ID in the entry to assign it to the correct campaign. Additionally, a campaign, with the corresponding ID must exist in the *artifacts-campaign* collection.

Campaign Usage

Figure 5.10 illustrates the beginning of a campaign and the execution of an attack in RedKube.

```
(Start/Tools/Redkube)>> campaign start
Enter the name of the campaign: RedKube Campaign
Enter a description for the campaign: RedKube Commands for Testing
Enter a list of targets (IPs or domains) separated by commas: 10.0.0.22
Creating Campaign...
Campaign ID: 659868e19e205a511c601311
Campaign: {'name': 'RedKube Campaign', 'start_date': datetime.datetime(2024, 1, 5, 20, 38,
24, 272825), 'end_date': None, 'description': 'RedKube Commands for Testing', 'targets':
['10.0.0.22']}
Campaign created successfully.
(Start/Tools/Redkube)>> discovery
Default mode is passive, for active mode use 'set mode active'
['sudo', 'python3', 'main.py', '--mode', 'passive', '--tactic', 'discovery']

<< Output Attack >>

(Start/Tools/Redkube)>> campaign end
Stopped Campaign...
```

Figure 5.10.: Feature - Campaign Usage

The figure shows the creation of a campaign named RedKube Campaign with the description RedKube Commands for Testing. Next, the *discovery* command from the Redkude tool is executed. After the attack has finished, the campaign is terminated, and the entries in the database collection can be analysed.

Database entries

Figure 5.11 displays the entry in the *artifacts-campaign* collection.

```
_id: ObjectId("659868e19e205a511c601311"),
name: 'RedKube Campaign',
start_date: ISODate("2024-01-05T20:38:24.272Z"),
end_date: ISODate("2024-01-05T20:39:47.515Z"),
description: 'RedKube Commands for Testing',
targets: [ '10.0.0.22' ]
```

Figure 5.11.: Database Entry in Collection artifacts-campaign

The entry includes the ID returned in figure 5.10, the specified *name* and *description*, as well as the *start_date* and *end_date* of the campaign. To test the feature for full functionality, view the entry for the attack in the *artifacts-raw* collection. See figure 5.12 for a section of the entry.

```
command: [
  'sudo',
  'python3',
  'main.py',
  '--mode',
  'passive',
  '--tactic',
  'discovery'
],
tool_name: 'red-kube',
tool_version: '1',
timestamp_start: ISODate("2024-01-05T20:39:01.189Z"),
timestamp_end: ISODate("2024-01-05T20:39:17.753Z"),
result_code: 0,
hostname: 'near',
campaign_id: ObjectId("659868e19e205a511c601311"),
mode: 'passive'
```

Figure 5.12.: Campaign ID Field in Collection artifacts-raw

The campaign ID, shown in 5.10 and 5.11, is contained in the *campaign_id* field at the end of the database entry. This confirms successful testing of the campaign feature's functionality.

5.3.2. Template

To test the template feature, it is necessary to execute attacks and generate a template from them. Figure 5.13 illustrates the execution of the attacks and the creation of the template.

```
(Start/Tools/Kubehunter)>> remote
['kube-hunter', '--remote', '10.0.0.22', '--report', 'json']
(Start/Tools/Redkube)>> discovery
Default mode is passive, for active mode use 'set mode active'
['sudo', 'python3', 'main.py', '--mode', 'passive', '--tactic', 'discovery']
(Start/Tools/Redkube)>> template

Script created: template_attack_script-20240105-215118.sh
```

Figure 5.13.: Feature - Template Usage

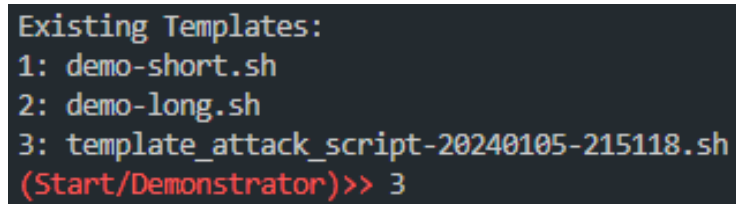
First, Kubehunter is used to perform a *remote_scan*. Next, the *discovery* attack is executed using Redkube. Finally, the template is generated with the *template* command. The name of the resulting template is *template_attack_script-2040105-215118.sh*, as shown in the figure.

The script's contents are displayed in the following listing 5.1.

```
1 #!/bin/bash
2 kube-hunter --remote 10.0.0.22 --report json
3 sudo python3 main.py --mode passive --tactic discovery
4 echo 'Attack automation completed.'
```

Listing 5.1: Template Bash Script

The script includes the two previously executed attacks(line 2-3) and can now be accessed and executed from the Demonstrator menu, as shown in figure 5.14.

A screenshot of a terminal window showing the output of a command. The text is as follows:

```
Existing Templates:
1: demo-short.sh
2: demo-long.sh
3: template_attack_script-20240105-215118.sh
(Start/Demonstrator)>> 3
```

Figure 5.14.: Demonstrator menu Entries

If the third menu item is selected, the attacks can be executed again to generate traces. This confirms the successful testing of the template feature.

5.4. Testing Phase Categorisation

Finally, it must be determined whether a tool is assigned to a phase when it is executed within that specific phase. In order to do this, Kdigger was executed during the Reconnaissance phase, as illustrated in figure 5.15.

```
(Start/Attack_phase/Reconnaissance/Kdigger)>> set output_format json
(Start/Attack_phase/Reconnaissance/Kdigger)>> set buckets version
(Start/Attack_phase/Reconnaissance/Kdigger)>> dig
['sudo', 'kdigger', 'dig', 'version', '--output', 'json']
{"bucket": "version", "result": {"buildDate": "2019-09-18T14:27:17Z",
"goVersion": "go1.12.9", "platform": "linux/amd64", "version": "v1.16.0"}}
```

Figure 5.15.: Set Kdigger specific Attack Parameter and launch Test Attack

As can be seen from the text in front of the cursor, the phase has been selected via the Attack Phase and Reconnaissance menu items. For testing, a scan is executed via the *dig* command to find out versions of components in the environment. The Database entry must now be checked for the field phase. The following figure 5.16 shows the database entry.

```
_id: ObjectId("6598800b4ca9e59457694c01"),
ip: '10.0.0.22',
file_output_raw: '{"bucket": "version", "result":
{"buildDate": "2019-09-18T14:27:17Z", "goVersion": "go1.12.9",
"platform": "linux/amd64", "version": "v1.16.0"}\n',
command: [ 'sudo', 'kdigger', 'dig', 'version', '--output', 'json' ],
tool_name: 'kdigger',
tool_version: '1',
timestamp_start: ISODate("2024-01-05T22:17:47.459Z"),
timestamp_end: ISODate("2024-01-05T22:17:47.499Z"),
attack_phase: 'Reconnaissance',
result_code: 0,
hostname: 'near',
```

Figure 5.16.: Database Entry for Kdigger Test Attack

Inside the entry you can see the *attack_phase* field with the value Reconnaissance. This means that a user can assign tools to a phase by selecting a tool from the Attack Phase menu.

6. Summary and Outlook

6.1. Summary

6.1.1. Objectives

In summary, the goal of implementing a framework for generating attack traces in O-RAN systems has been successfully achieved. The chosen structure of the framework architecture, with a central control unit and the use of different states, in conjunction with base classes for menu and tool classes, has provided a solid foundation on which to build in the future. With a view to the previously defined goal of making the framework dynamically extensible, tools and new menu states can be integrated into the framework without major code changes by implementing classes for the new states that inherit from the base classes.

In addition to the foundation, three tools have already been added to the framework. As shown, the integrated tools can be used in their full functionality via the framework, and also provide methods for pre-parameterised attacks. These provide the user with a way to generate traces via the framework without having to deal extensively with the tool itself, particularly with regard to the goal of user-friendliness.

In conjunction with the dynamic foundation, a database connection is also implemented in the framework. This is used to store the collected metadata for each attack. The main advantages of the database entries are that they can be compared with the logs generated by the DFIR tools to better identify and assign attacks, for example by comparing the timestamp of the logs with the timestamp of the database entry, and also found vulnerabilities are stored for further analysis and usage.

Furthermore the Campaign feature, which allows several attacks to be combined, the Template feature, which generates template so attacks can be repeat easily, and a parser for the Kubehunter tool have been integrated into the framework.

The parser enables the categorisation of attacks based on MITRE, Kubernetes ATT&CK Matrix, and CVE. Additionally, attacks can be manually mapped for every tool by executing them over the attack phase menu.

Because the framework is used on different clusters, it is important to automate the deployment of the framework on the clusters and also the deployment of the database

according to the implemented database connection. The automation was successfully implemented with Ansible, whereby both the database and the framework can be updated by executing the Ansible scripts created.

In conclusion, the framework plays a crucial role in the 5G-FORAN project, particularly in the active attack simulation sub-project. This is because the framework simplifies the generation of attack traces, which the DFIR tools uses as a basis to identify and evaluate these traces in the second sub-project.

6.1.2. Personal Motivation

Regarding my personal motivation, I gained valuable insights into O-RAN and Kubernetes technologies. During my bachelor's thesis, I focused on generating traces based on Kubernetes, which allowed me to gain practical experience with Kubernetes components such as pods and clusters, as well as testing them for vulnerabilities. The result has been in line with my expectations.

6.2. Outlook

6.2.1. Tools and Features

Due to the framework's dynamic structure, future efforts will focus on implementing additional tools and features.

Several tools have already been evaluated and will be integrated into the framework in the coming months. It can be said that evaluating and testing new tools that have the potential to generate traces on Kubernetes or ORAN will remain one of the most important tasks in the future.

Further parsers are also planned for implementation. Since the only functional parser is implemented for Kubehunter, further output of tools has to be analysed. If the analysis results show that implementing a parser for this tool is worthwhile, then a parser will be implemented

In addition, its planed to provided predefined templates to simplify the generation of tracks without the need to operate the tools directly. These templates will be expanded dynamically throughout the project, in consultation with Procyde. If specific attacks are easily identifiable and corresponding detection rules have been established, it is practical to offer a predefined template for each detection rule.

The final planned feature involves searching for exploits related to CVE numbers resulting from attacks. This allows users to continue attacks and generate additional traces that were not possible with previously used tools.

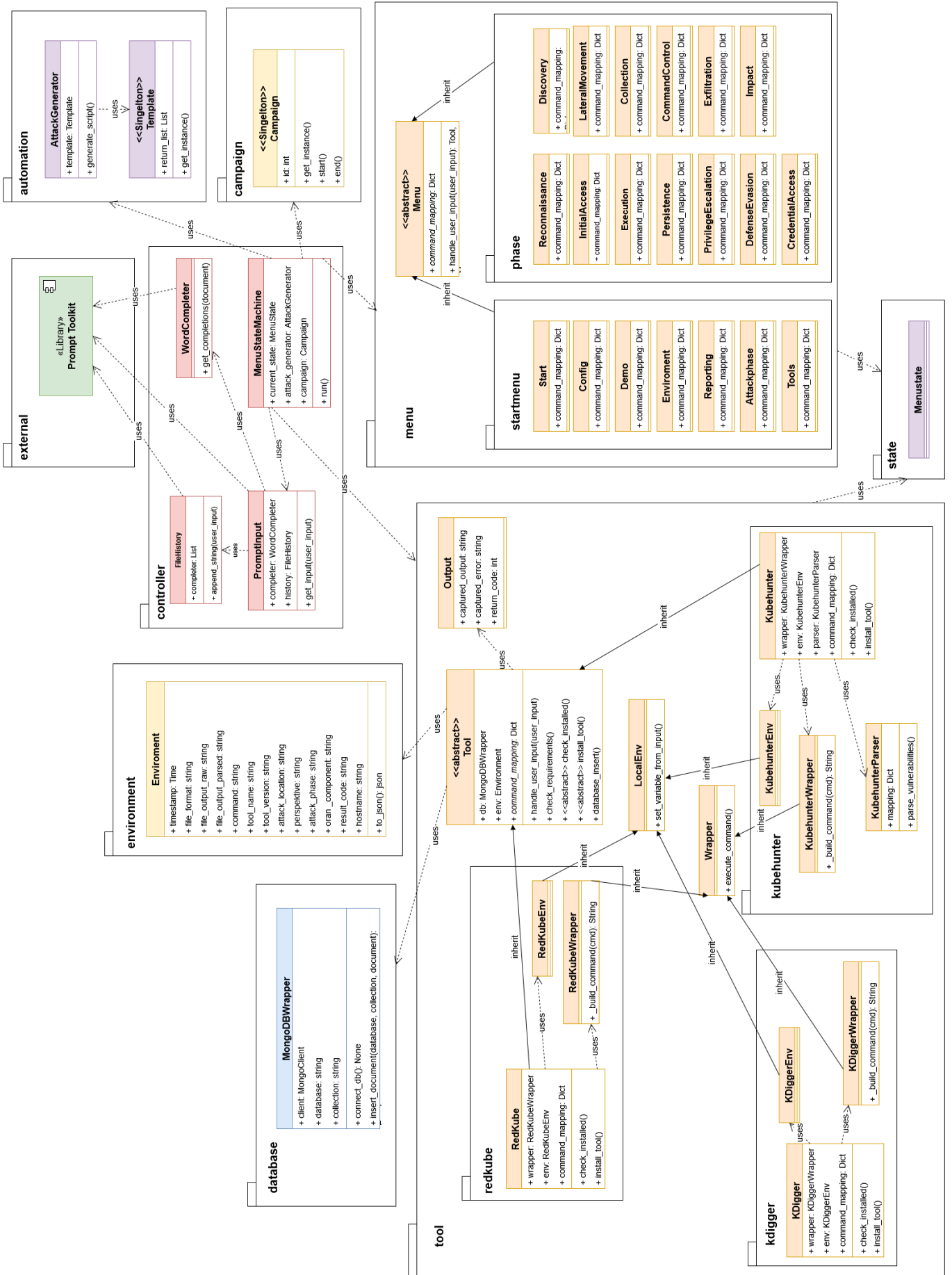
6.2.2. Attacker perspectives

The framework currently only maps one of the attacker perspectives defined in 2.1.8. To ensure the creation of traces based on other different attacker perspectives, it should also be possible to set environment variables and metadata, like different perspectives in a separate menu item. To enable the usage of different attack perspectives, it is necessary to automate the framework further for dynamic deployment to various locations.

6.2.3. Testing with DFIR tools

Besides modifying the framework, another important task is testing it in combination with DFIR tools. This is necessary to gain insight into the effectiveness of trace generation and identify areas where traces are not yet being generated. These findings can then be used to evaluate for new tools that specifically cover those areas. Since the framework is currently in a stage where intensive testing is possible, the next significant step is to test it in combination with DFIR tools. This will represent a central task in the further progress of the project.

A. Detailed CF-Framework Architecture



B. Detailed CF-Framework Architecture

Glossary

Word	Definition
Attack	An attack in context of this thesis, is the attempt to examine components for vulnerabilities and exploit them to generate traces recognisable to the DFIR tool.
Class	A Class is a blueprint for creating objects. It defines attributes and methods for objects instantiated from it. Classes can be organized hierarchically, with base classes serving as the foundation for more specialized subclasses [36].
Cluster	In Kubernetes, a cluster is a collection of interconnected nodes that work together to manage and execute containerized applications, ensuring efficient deployment, scaling, and coordination of containers in a distributed environment [13].
DFIR	DFIR is a cybersecurity field that involves investigating and responding to security incidents and digital crimes [37].
JSON	JSON is a lightweight format used to represent structured data in a way that is both human-readable and machine-readable [38].
Method	Methods are functions associated with an object. They define the behavior or actions that objects of a particular class can perform [36].
Module	A module, in the context of this thesis, is a logical entity that combines several Python files to perform a specific task within the framework.

Parameter Flag	/	Parameter and Flags in context of this thesis, are Variables than can be set by an user via the menu to interact with the CF-Framework.
Pod		In Kubernetes, a Pod is the smallest unit that can be deployed and can contain one or more containers. It is the fundamental component for running applications in a cluster [39].
Variable		Variables are names that represent values. They store and manage data within a program [36].
Vulnerability		A Vulnerability refers to a weakness or flaw in a system's design, implementation, or operation that could be exploited by attackers to compromise an system [40]

List of Acronyms

Acronym	Expansion
3GPP	Third Generation Partnership Project
5G	Fifth Generation mobile network
5GC	5G Core
5GS	5G Systems
AMF	Access and Mobility Management Function
BSI	Federal office for Information Security
CA	Certification Authority
CISA	Cybersecurity and Infrastructure Security Agency
CLI	Command Line Interface
CVE	Common Vulnerabilities and Exposures
DEV	Development
DFIR	Digital Forensics and Incident Response
DHS	Department of Homeland Security
FFRDC	Federally funded research and development centers
gNB	Next-Generation Node B
ID	Identification

NG-RAN	Next Generation Radio Access Network
NTS	Network Topology Simulator
O-CU	Open RAN Central Unit
O-DU	Open RAN Distributed Unit
O-RU	Open RAN Radio Unit
PROD	Production
RAN	Radio Access Network
RIC	RAN Intelligent Controller
SMO	Service Management and Orchestration
UE	User Equipment
UPF	User Plane Function
VM	Virtual Machine

Bibliography

- [1] BSI, *Cybersicherheit - KRITIS- Meldungen bis 2022*, August 2022. [Online]. Available: <https://de.statista.com/statistik/daten/studie/1230654/umfrage/anzahl-der-kritis-meldungen-an-das-bsi/> (visited on 13 January 2024).
- [2] S. Köpsell, A. Ruzhanskiy, A. Hecker, D. Stachorra, and N. Franchi, “Open-RAN Risikoanalyse 5GRANR”, pp. 6–41, February 2022. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/5G/5GRAN-Risikoanalyse.pdf?__blob=publicationFile&v=9 (visited on 13 January 2024).
- [3] A. Sultan, *5G System Overview*, August 2022. [Online]. Available: <https://www.3gpp.org/technologies/5g-system-overview> (visited on 13 January 2024).
- [4] O-RAN Alliance, *About us*. [Online]. Available: <https://www.o-ran.org/about> (visited on 13 January 2024).
- [5] Procyde and THKöln, “5G-FORAN Gesamtvorhabenbeschreibung”, p. 1, Sep. 2022.
- [6] ÜBER PROCYDE GmbH. [Online]. Available: <https://procyde.com/ueber-uns> (visited on 13 January 2024).
- [7] O-RAN Software Community, *O-RAN Software Community*. [Online]. Available: <https://o-ran-sc.org/> (visited on 13 January 2024).
- [8] Cisco, *What Is 5G? - How Does 5G Network Technology Work*. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/what-is-5g.html> (visited on 13 January 2024).
- [9] *5g-fig1.png (PNG-Grafik, 498 × 176 Pixel)*. [Online]. Available: <https://www.3gpp.org/images/2022/08/17/5g-fig1.png> (visited on 13 January 2024).
- [10] O.-R. ALLIANCE, “O-RAN.WG1.OAD-R003-v10.00.docx”, p. 15,
- [11] *Open RAN*. [Online]. Available: <https://firecell.io/learn/open-ran/> (visited on 13 January 2024).

-
- [12] *Kubernetes overview*. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/> (visited on 13 January 2024).
- [13] *What is Kubernetes Cluster? | VMware Glossary*. [Online]. Available: <https://www.vmware.com/topics/glossary/content/kubernetes-cluster.html> (visited on 13 January 2024).
- [14] Mitre, *MITRE ATT&CK®*. [Online]. Available: <https://attack.mitre.org/> (visited on 13 January 2024).
- [15] Mitre, *Tactics - Enterprise | MITRE ATT&CK®*. [Online]. Available: <https://attack.mitre.org/tactics/enterprise/> (visited on 13 January 2024).
- [16] *Tactics - Threat Matrix for Kubernetes*. [Online]. Available: <https://microsoft.github.io/Threat-Matrix-for-Kubernetes/> (visited on 13 January 2024).
- [17] Y. Weizman, *Secure containerized environments with updated threat matrix for Kubernetes*, March 2021. [Online]. Available: <https://www.microsoft.com/en-us/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/> (visited on 13 January 2024).
- [18] *What is a CVE?*, August 2020. [Online]. Available: <https://www.balbix.com/insights/what-is-a-cve/> (visited on 13 January 2024).
- [19] *What is MongoDB? Features and how it works – TechTarget Definition*. [Online]. Available: <https://www.techtarget.com/searchdatamanagement/definition/MongoDB> (visited on 13 January 2024).
- [20] *Was ist ein X.509-Zertifikat?*, Sep. 2019. [Online]. Available: <https://www.ssl.com/de/faq/Was-ist-ein-x-509-Zertifikat%3F/> (visited on 13 January 2024).
- [21] A. Hat Red, *How it works*. [Online]. Available: <https://www.ansible.com/overview/how-ansible-works> (visited on 13 January 2024).
- [22] *Python-prompt-toolkit*. [Online]. Available: <https://github.com/prompt-toolkit/python-prompt-toolkit> (visited on 13 January 2024).
- [23] *Kube-hunter*, January 2024. [Online]. Available: <https://github.com/aquasecurity/kube-hunter> (visited on 13 January 2024).
- [24] *Quarkslab/kdigger: Kubernetes focused container assessment and context discovery tool for penetration testing*. [Online]. Available: <https://github.com/quarkslab/kdigger> (visited on 13 January 2024).

-
- [25] *Lightspin-tech/red-kube: Red Team K8S Adversary Emulation Based on kubectl*. [Online]. Available: <https://github.com/lightspin-tech/red-kube> (visited on 13 January 2024).
- [26] B. Marijan, *CLI vs. GUI: What Are the Differences?* | phoenixNAP KB, February 2023. [Online]. Available: <https://phoenixnap.com/kb/cli-vs-gui> (visited on 13 January 2024).
- [27] *C vs C++ vs Python vs Java - Javatpoint*. [Online]. Available: <https://www.javatpoint.com/c-vs-cpp-vs-python-vs-java> (visited on 13 January 2024).
- [28] *Advantages and Disadvantages of C Language - Javatpoint*. [Online]. Available: <https://www.javatpoint.com/advantages-and-disadvantages-of-c-language> (visited on 13 January 2024).
- [29] *Advantages and Disadvantages of C++ Programming Language - Javatpoint*. [Online]. Available: <https://www.javatpoint.com/advantages-and-disadvantages-of-cpp-language> (visited on 13 January 2024).
- [30] *Advantages and disadvantages of Java - Javatpoint*. [Online]. Available: <https://www.javatpoint.com/advantages-and-disadvantages-of-java> (visited on 13 January 2024).
- [31] *Python Language advantages and applications*, October 2017. (visited on 13 January 2024).
- [32] IPSpecialist, *Ansible vs. Puppet vs. Chef*, August 2023. (visited on 14 January 2024).
- [33] *Python Dictionaries*, https://www.w3schools.com/python/python_dictionaries.asp. (visited on 14 January 2024).
- [34] *Ed25519 Key erstellen – Allerstorfer.at*. [Online]. Available: <https://www.allerstorfer.at/ed25519-key-erstellen/> (visited on 13 January 2024).
- [35] *What Is a Certificate Authority (CA)? - SSL.com*, <https://www.ssl.com/faqs/what-is-a-certificate-authority/>. (visited on 13 January 2024).
- [36] *9. Classes — Python 3.12.1 documentation*, <https://docs.python.org/3/tutorial/classes.html>. (visited on 15 January 2024).
- [37] *What is Digital Forensics and Incident Response (DFIR)? | IBM*. [Online]. Available: <https://www.ibm.com/topics/dfir> (visited on 13 January 2024).
- [38] *What is JSON*. [Online]. Available: https://www.w3schools.com/whatis/whatis%5C_json.asp (visited on 13 January 2024).

- [39] *What are Kubernetes Pods? / VMware Glossary*. [Online]. Available: <https://www.vmware.com/topics/glossary/content/kubernetes-pods.html> (visited on 13 January 2024).
- [40] *Understanding vulnerabilities*. [Online]. Available: <https://www.ncsc.gov.uk/information/understanding-vulnerabilities> (visited on 13 January 2024).

Selbstständigkeitserklärung

Ich erkläre an Eides statt, dass ich die vorgelegte Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

15.01.2024

Ort, Datum

Johannes Müller

Rechtsverbindliche Unterschrift