UiA University of Agder

Bachelor

Building a Framework for Automatic Testing of Intrusion Detection Software

Tor Edvard Røysland, Eirik Børø Gyring, Håvard Skordal Thorsen,
Odd-Egil Solheim Egelandsaa

SUPERVISOR

Sigurd Brinch

University of Agder, 2023

Faculty of Engineering and Science

Department of Engineering and Sciences

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Abstract

An important part of modern computer security consists of intrusion detection software. This software attempts to detect and mitigate intruders attempting to gain access to software systems. The University of Agder has plans to start a course on intrusion detection software in the foreseeable future. In this course the students will learn how intrusion detection systems work, and how to write their own.

In this thesis the group uses qualitative research methods to conduct an interview of the client and supervisor to obtain a general outline of the project. the Evolutionary development model of software by Hewlett-Packard is followed to develop this project from a simple prototype to a framework for intrusion detection testing, intended to be used in the future university course.

The group concludes that the software fulfills the demands set by the client, but contains many restrictions that may hinder or limit the use of the framework, and proposes some possible future development options and optimizations for the framework.

Contents

List of Figures

List of Listings

List of Tables

Abbreviations

The list below include different abbreviations and acronyms used throughout the project. These are available so that the reader can get a better understanding of these terms, avoiding any misunderstanding.

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command-Line Interface
DevOps	Development Operations
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EVO	Evolutionary Development
GUI	Graphical User Interface
UI/UX	User Interface / User Experience
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
JSON	JavaScript Object Notation
PCAP	Packet Capture
RE	Regular expression operations
Stdout	Standard Output
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
VSC	Version Control System

Table 1: List of abbreviations

Chapter 1

Introduction

1.1 Background

Modern secure systems should always take into account the potential of a skilled adversary attempting to gain access, steal information or perform sabotage. In recent decades such adversaries have become increasingly skilled and professional, representing countries, criminal organizations or other groups with a vested interest [1]. To prevent such groups from gaining access to critical systems, intrusion detection systems (IDS) are one of many security measures that exist to mitigate this threat.

An IDS is a software that aims to "defend a system by using a combination of an alarm that sounds whenever the site's security has been compromised, and an entity [...] that can respond to the alarm and take appropriate action" [2]. A modern IDS will typically scan network traffic for signatures or anomalies based on a set of rules, or using machine learning algorithms to detect possible intrusions into a system [3].

When developing any kind of software, it is important to verify whether it operates as intended and expected [4], and this is also true for intrusion detection systems. The University of Agder is planning a future course on creating IDS as mentioned in the assignment description in Appendix A. For this the university requires a test framework for the students to test the reliability of their intrusion detection systems.

1.2 Project objective

The objective of this project is to create a framework that allows students to push their IDS to a repository. This should start an automated testing process of the IDS, producing a human readable result for the user.

In addition to this, the group will attempt to fulfill the feature requests from the client outlined in the assignment description in Appendix A:

- Given a set of conditions, the student should be able to push their code to a git repository (Github or Gitlab).
- This should trigger an action that tests the IDS' capabilities with a predefined sequence of network packets.
- Provide a presentation of the results of the test.

1.3 Report outline

The **Theory** chapter contains background information and knowledge that will come in useful when reading this thesis.

The **Method** chapter explains the research and development methods used for this project.

The **Results** chapter is a thorough description on how the work in the project was performed, and the results that came with.

The **Discussion** chapter describes the groups discussions of the project results.

The **Conclusion** chapter summarizes concludes the groups project results.

Chapter 2

Theory

2.1 Intrusion Detection System

Intrusion detection systems are systems created to analyze network traffic and alert the user administrator when suspicious activity occurs. The software either scans the incoming network traffic for malicious packets or intrusions based on already known patterns, also called a signature-based method, the anomaly-method with the help of machine learning, or a hybrid of both these types. The three methods have these criteria [5] [3]:

- **Anomaly-based:** This method uses machine learning algorithms to detect new patterns as traffic is analyzed. This is used to detect new types of malicious attacks.
- **Signature-based:** This method is based on patterns already known and can be used to detect malicious traffic that the system recognizes.
- **Hybrid detection:** The hybrid detection method utilizes both the signature-based method with already known patterns and the anomaly-based machine learning algorithm for detecting new patterns.

The software is capable of running in different sections of the network. This can be on a device, in front of a firewall, on a router or a server. Intrusion detection systems are classified into 5 types [5]:

- **Network intrusion detection system:** Network intrusion detection systems are usually located within the network and is monitoring the traffic going in out of devices on a subnet level. If the IDS matches with any known attacks, the administrator will be alerted. This IDS can be useful to alert if someone is trying to attack the firewall.

- **Host intrusion detection system:** Host intrusion detection systems monitors incoming and outgoing packets on a specific device or host. This can be a critical host in a network that needs extra monitoring. The host intrusion detection systems can take snapshots of the file system and compare this to a previous snapshot to see if any files on the host has been deleted or edited. If so, the IDS will send an alert to the system administrator.

- **Protocol-based intrusion detection system:** Protocol-based intrusion detection systems is a system that usually sits in front of a server monitoring the traffic between devices and the server and analyzing specific set of protocols chosen to protect the server.

- **Application protocol-based intrusion detection system:** Application Protocol-based intrusion detection systems sits within the server on the application level and is monitoring traffic between the software and for example a database.

- **Hybrid intrusion detection system:** Hybrid intrusion detection systems are usually made by a combination of the other intrusion detection systems. This will result in a broader overview of the network all the way from the subnet to the application. This will be more secure and more effective than just one type of IDS operating alone.

2.2 Libraries

2.2.1 Scapy

Scapy is an interactive packet manipulation program based in Python which enables the user to send, sniff and forge network packets [6]. In other words, with the use of Scapy, it is possible to perform several different tasks such as network discovery, network testing, and intrusion detection. This is because Scapy can be used to construct custom made tools that can probe, scan or attack networks.

The main idea of Scapy is to provide users with a customizable tool capable of performing two simple tasks: sending packets and receiving answers. The user is able to define a number of packets that Scapy sends to a target, and then receives answers from said target to match these answers with the requests that was sent. Finally it returns a list of matched and unmatched packets. Scapy has great advantages compared to other networking tools, as other tools are often built for a specific purpose, and has limitations to almost only be used for that single purpose. According to the developers of Scapy, their program is on the

opposite side of this spectrum since it is just a baseline tool that can be built upon in many different directions [6].

2.2.2 Pytest

A common Python testing framework called Pytest is frequently used to test software programs and libraries [7]. It is built with powerful features that can scale to support complex functional testing while also making it simple to write short, readable tests. The fact that Pytest offers a straightforward and understandable syntax for defining test functions is one of the main advantages of using it. The naming convention for test functions in Pytest is "test_" and they are written as regular Python functions [8]. This allows the creation and organization of tests and enables the use of common Python building blocks like loops and conditional statements.

Another benefit of Pytest is that it supports a variety of testing scenarios, from straightforward unit tests to complex functional tests. For instance, Pytest can be used to test databases, REST APIs, web applications, and more. The use of fixtures, which are reusable pieces of code that can be used to create preconditions for tests as well as cleanup code that is executed after tests are finished, is also supported [9]. In addition to being adaptable and simple to use, Pytest offers strong features for test execution, test collection, and test discovery. In addition to providing thorough reporting and output for test results, it can automatically locate and run tests within a project [7].

2.3 Testing

Software testing is an essential phase in the software development life cycle, that involves determining whether a piece of software or application satisfies the requirements and operates as expected [4]. A dedicated team of software testers typically performs testing, using a variety of testing techniques and tools to find flaws, bugs, and other problems that might affect the functionality, performance, or security of the software product.

Software testing contains a wide spectrum of categories and types of testing. First and foremost, testing distinguishes between automated and manual testing. Where manual testing is performed by a person that is interactively using the application or software to test it, and automated testing is instead done by a machine executing test scripts written in advance [10]. Furthermore, there are several types of software testing that are normally used for dif-

ferent purposes. The types of software testing are usually put into three different categories [11]:

1. **Functional Testing**: Testing-types such as Unit Testing, Integration Testing, etc.
2. **Non-Functional Testing**: Testing-types such as Performance Testing, Stress Testing, etc.
3. **Maintenance and Regression testing**

Within these categories, there are several different types or strategies of software testing.

2.3.1 Black Box testing

Black Box Testing is a way of testing a software without prior knowledge about how it is setup or how it works. From the Appendix A - Glossary section (under "Black Box Testing") of the NIST special publication 800-192:

> "A method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied to virtually every level of software testing: unit, integration, system and acceptance". [12]

A Black Box Test is typically conducted in a way that the tester inputs data and monitors the output produced by the system being tested. This allows for the identification of the system's response time, usability issues, and reliability issues as well as how the system reacts to expected and unexpected user actions [13]. Because it tests a system from beginning to end, black box testing is an effective testing method. A tester can simulate user activity to check whether the system fulfills its promises, just as end users "don't care" how a system is coded or designed and expect to get a suitable response to their requests. A black box test assesses every relevant subsystem along the way, including the UI/UX, database, dependencies, and integrated systems, as well as the web server or application server [13]. Black Box Testing can be used in all the three categories mentioned in the 2.3 Testing section.

Functional testing

Specific features or functions of the software that is being tested can be tested using black box testing [13].

Non-Functional Testing

Beyond features and functionality, black box testing allows for the inspection of additional software components. A non-functional test examines "how" rather than "if" the software can carry out a particular task [13].

Regression and Maintenance Testing

To determine whether a new software version displays a regression, or a degradation in capabilities, from one version to the next, black box testing can be used. Regression testing can be used to test both functional and non-functional aspects of the software, such as when a particular feature no longer functions as expected in the new version or when a previously fast-performing operation becomes significantly slower in the new version [13].

2.3.2 White Box testing

White Box Testing or "Comprehensive Testing" is another type of conducting software testing. From the Appendix B - Glossary section (under "Comprehensive Testing") of the NIST special publication 800-137:

> "A test methodology that assumes explicit and substantial knowledge of the internal structure and implementation detail of the assessment object. Also known as white box testing".[14]

White Box Testing is an easier approach when it comes to automated tests. That, however, does not imply that it is impossible to automate testing with a black box approach [15]. The reason for this is simply due to knowing the system's internal structure and implementation details before running the tests. On the other hand, a White Box Testing approach is vulnerable to code changes, thus often requiring expensive maintenance to automate.

2.3.3 Grey Box Testing

Black box testing involves no knowledge of a system's inner workings, while white box testing requires complete knowledge. Grey box testing, on the other hand, is a compromise that involves testing a system while only knowing some of its internals. The three types of testing that it is most frequently used in are penetration, end-to-end, and integration [15]. Grey box testing combines feedback from developers and testers and can lead to testing approaches that are more successful. By concentrating testers on the user paths most likely to have an impact on users or cause a defect, it reduces the overhead necessary to perform functional testing of a large number of user paths [15].

2.4 Automation

2.4.1 Git

Git is a version control system (VCS) that lets software development teams work and write code on the same project independent of each other. The main project, also called main branch, is stored in a repository on a server. Version control systems lets you track every change made to the code base which lets the team see all the history and who made the change - and then roll back to an earlier stage of the project if something goes wrong in the development process. This feature works as a safety net and bypasses conflicts [16].

Types of version control systems [17]:

- **Local VCS:** This allows the developer to store the files and work on them locally.
- **Centralized VCS:** With this method all the changes are made on a single server.
- **Distributed VCS:** Distributed VCS: This involves cloning the code repository and committing changes directly.

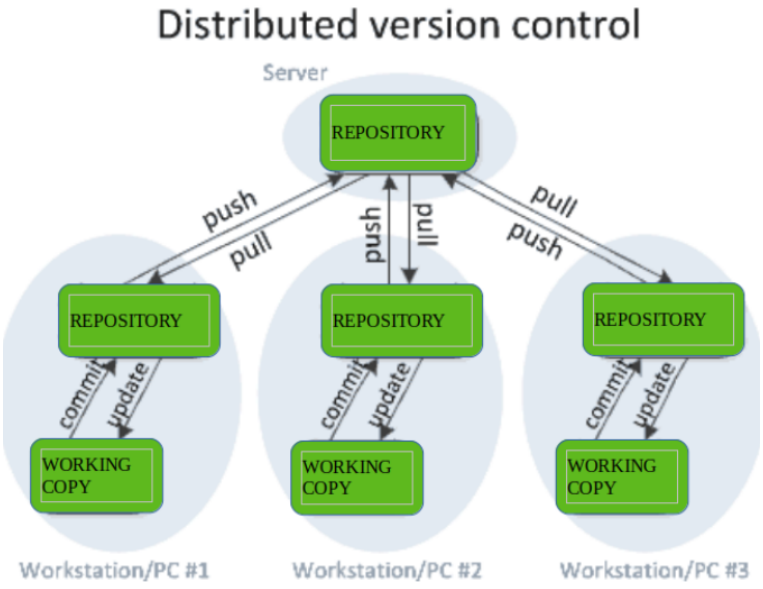

Figure 2.1: Distributed version control [17].

Distributed version control system has several different repositories as shown in Figure 2.1. Developers can clone the repository, which is the latest version of the code base, making this a unique working environment. After cloning, a pull command is needed every time to get the latest code. When the developers have added functions to the code, committed the changes and pushed these to the central repository, the changes will become visible for the others working on the same project [17].

2.4.2 Gitlab

GitLab is an open source code repository software development platform for DevOps projects. GitLab CI tools lets developers automate the testing and building phase of the software production and also comes with a fully functional dashboard [18].

DevOps is the term used to describe the workflow of the development cycle. This method of developing software is based on various tools to make the operation automated, effective, safe and reliable throughout the process. GitLab gives the development end-to-end potential with the DevOps method [19].

Continuous Integration

Continuous integration is a practice that lets developers merge their code into a central repository whenever they have been working on a new feature and wants to add this to the main build of the application. When someone merge their code to the repository, this code will automatically be built and tested in the same environment as all other merged code and therefore having the same test-base. This is to make sure that the code are not tested locally and some machine differs from another and can cause unexpected errors. If someone writes code with bugs, this will not affect the final build as the tests will fail before it reaches the main build. This results in improved software quality, early bug addressing and makes the process a lot more effective [20].

GitLab Runner

GitLab Runner is a software that is used together with GitLab to perform the CI/CD jobs in a pipeline. Runners can be installed on Linux, Windows, FreeBSD and macOS on a local machine or in a Docker container [21]. There are two main types of GitLab Runners:

- Shared runners: These runners are shared with the rest of the GitLab instance and can be used to perform jobs on several different projects with the same requirements [22].
- Specific runners: These runners are usually designed for a specific project with a specific set of requirements [22].

When the GitLab Runner is installed, the runner must be registered with the GitLab instance that the developers wants to run jobs on. There are two choices of GitLab instances, self-managed or using GitLab.com [21].

2.5 Docker

2.5.1 Containers

Docker is a software platform for running and developing applications. With Docker, developers can run and test the applications in an isolated environment separated from each other with the help of containers. Docker containers have namespaces where only that container is running. This namespace gets created when the container is started. With help from the Docker API or the CLI you can create, start, stop or delete containers [23].

Because the containers are isolated from each other, developers can run multiple containers at the same time on the same host. Containers do not require a lot of resources because they are lightweight and easy to run. Everything needed to run the application is located inside the container so there is no need for support from any software installed on the host. Containers are also easy to share between developers because the working environment stays identical on any host [23].

The Docker daemon called Dockerd is the background process that builds, runs and distributes the Docker containers. This daemon communicates with the Docker client using a REST API and handles the images, networks, volumes and containers. The way Docker orchestrates this is called a client-server architecture [23]. The docker architecture is outlined in figure 2.2.

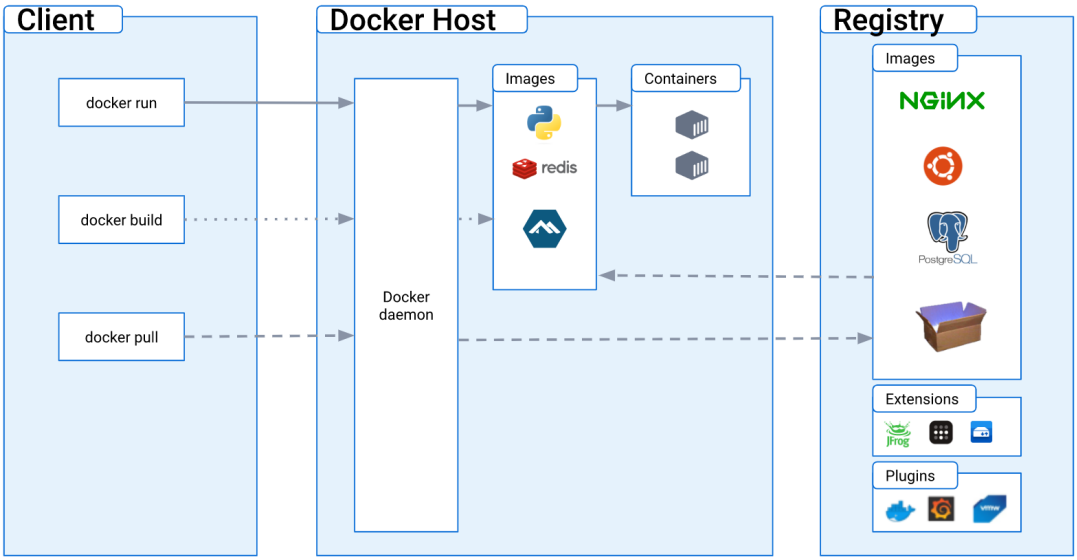

Figure 2.2: Docker architecture [23].

2.5.2 Images

When creating containers in Docker, this is based on a read-only template with specific instructions called an image. These images can be templates already created for a certain purpose, like a web-server, and then customized to the developers needs. The image is written inside a Dockerfile. This Dockerfile defines the container and involves all the configuration and options needed to run the application [23].

The instructions separately creates a layer in the image, and when the image gets rebuilt, only the layers that have been changed gets rebuilt. This is what makes images so lightweight and fast to work with. When the container gets shut down, every change within the running application gets deleted if it is not stored in persistent storage [23].

2.5.3 Docker-compose

Docker-compose is a tool for running multiple containers with different applications at the same time. Using Docker-compose involves creating a YAML file that contains all the information the applications needs to run, and by running this YAML file developers can start multiple services at once based on the configuration parameters set in the file. Docker-compose works well within all the stages of the application life-cycle [24].

- Development
- Staging
- Testing
- Production

When containers are running, developers can use commands for starting and stopping as well as viewing the status of the running services. Docker-compose has the benefits of creating more than one isolated environment on a single host and makes communication and support between the environments easy [24].

Chapter 3

Method

This chapter covers the research and development methods used in this project. Starting with the explanation of what kind method was used for research, how and what kind of data was collected. Further on the reader will get an overview of the development method that was used to develop the automatic IDS testing framework.

3.1 Research method

3.1.1 Qualitative Research

The research method chosen for the project was Qualitative research. Qualitative research is a method of inquiry that involves the collection and analysis of non-numerical data such as text, video, or audio [25]. Its aim is to comprehend concepts, opinions, or experiences and gain a deeper understanding of a problem or generate new research ideas.

3.1.2 Data collection

The architectural design of the automatic IDS testing framework was formed by settling the desired properties and results that the teacher and supervisor had foreseen they would get from this project. Prior to the start of this project, the group engaged in an informal meeting with the client and the supervisor of the project. The client referred to is the teacher of the University of Agder's next year subject (At the time of writing) "Inntrengerdeteksjon" Per-Arne Andersen.

Due to technical issues, the group was not able to recover the recording of this meeting. However, this meeting, and the notes taken from it is used as the main qualitative research. The meeting was conducted via Zoom.

The meeting form was set up to start with the client explaining the project assignment. Furthermore the client would present their opinions on how the test framework should work, what features that were preferable to look into, and some tips and ideas. After the client presented their view, the supervisor would present their view as well. The meeting ended with a few questions from the group members.

The data collection process also included secondary data collection. This was conducted through examination of government publications, academic search engines, and also documentations of the different services that were recommended to use for the design of our testing setup by the client and supervisor. Research on alternative services was also performed this way. The data that was needed for this project can be seen in listing 1.

1. Test data to be used as input to an IDS.
2. Data about different Intrusion Detection Systems to test the input data with.
3. Data about output from an IDS.
4. Data about Git repository and CI/CD.
5. Data about test scripts.

Listing 1: Data required for this project.

3.2 Evolutionary Development method

The project's development method is based on the Hewlett-Packard journal's article 4 Evolutionary Development Model (EVO) for Software [26]. Essentially, the EVO model breaks down the development process into smaller, sequential waterfall models. Each waterfall model ends with the users getting access to the product, and they provide feedback to help plan the next cycle. The development team takes this feedback into account and makes changes to the product, plans, or process accordingly. These cycles are usually short, lasting two to four weeks, and continue until the product is ready for shipment.

The EVO method can be illustrated as shown in Figure 3.1.

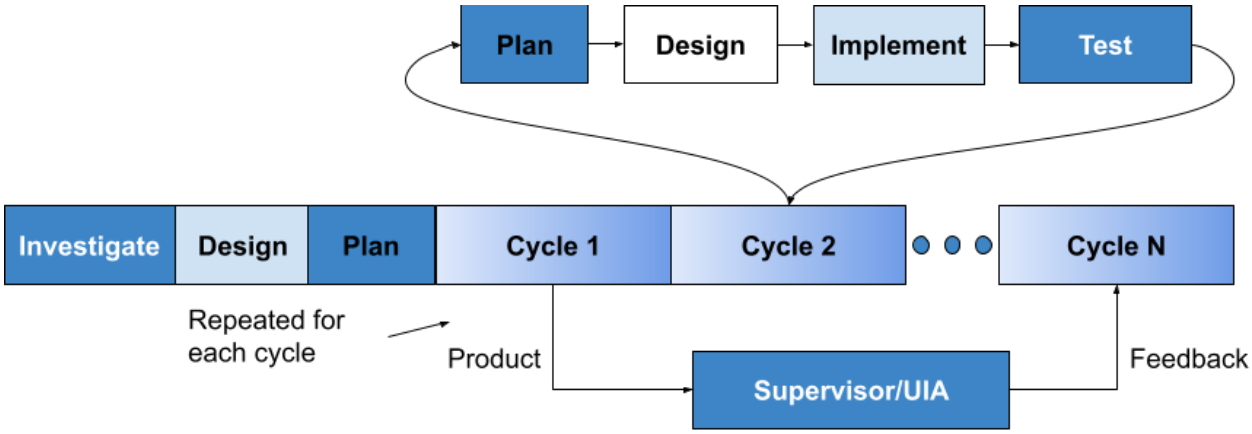

Figure 3.1: A self made illustration based on HP EVO method [26].

3.2.1 The Waterfall method

The Waterfall method is a structured and sequential approach to developing information systems, which involves the following stages [27]:

1. Requirement analysis and definition - During this stage, system services, constraints, and objectives are established by consulting with users. The resulting system specifications are then defined in detail.

2. System and software design - This stage involves dividing system requirements between hardware and software to create the overall system architecture. The software design phase includes identifying the fundamental software system abstraction and its interrelationships.

3. Implementation and unit testing - The software design is implemented as a series of programs or program units, which are then tested to ensure that each unit meets its specifications.

4. Integration and system testing - In this stage, the individual program units are combined and tested as a whole to ensure that they meet the software requirements. Once testing is completed, the software is ready to be delivered to the customer.

5. Operation and maintenance - This is typically the longest stage, during which the system is installed, used extensively, and maintained. Maintenance involves addressing errors that were not identified in previous stages, enhancing the implementation of system units, and improving system services in response to new needs.

The Waterfall method can be illustrated as shown in Figure 3.2.

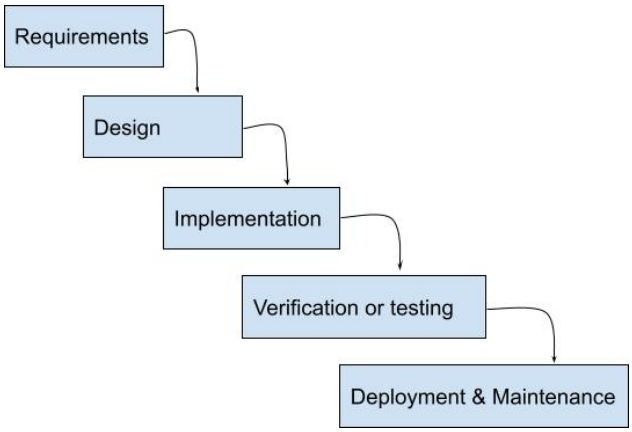

Figure 3.2: A self made illustration of the waterfall method

3.2.2 Implementing the EVO development model into our project

To make automatic IDS testing work, the group had to be able to create a setup that is working, test it to discover what could be done better or if the setup had any faults giving us the wrong result, or no result at all. The EVO method was to be implemented into the project in the early stages as soon as the setup was ready to go into development.

Investigate & Initial Design

The investigation stage was necessary to settle the outline and criteria for the setup in the cycles. This stage along with initial design is only used once in the EVO method, before the first cycle. The bare minimum criteria would be outlined in this phase of the development. The initial design phase covers how the model of the prototype would look like, this model would further on be put into a real design in cycle 1, which then got improved upon after the testing phases of the development cycles.

Cycles

The group set a 4 week time period for each cycle, including the planning, design, implementation and testing phases. The group estimated that a total of 4 cycles would be necessary to get a finalized setup.

Plan

The planning phases consisted of planning out how the design phase was to be carried out. The plan describes a scope of the testing setup, or in other words, what the group wanted to get from the cycle. This also included how advanced or simple the setup would be before the design was decided.

Design

The information and criteria that had been settled in the planning phase was used to design the setup itself. The input, testing and output was designed for each cycle.

Implement

The implement phase describes the technical implementation of the design. Since the research, planning and design phases were already finished, this would be considered the shortest phase in the cycle. This is where the group started to create the tests, and implemented the network data to be tested by the IDS, as well as formatting the output to be correctly ordered out of the test. If the group would stumble upon any required important changes in this phase, it would mean going back to the design phase to reevaluate the design.

Test

Before a cycle was ended, a testing phase would be conducted. This was to gather findings about how the framework performed, and what could be improved upon before the next cycle.

Chapter 4

Results

4.1 Data collection

The initial description, as seen in Appendix A, was: "In a new course covering Intrusion Detection Systems (IDS), the supervisor is looking for a way to automatically test IDS systems developed by the students". Other than that, the document briefly explained what kind of features were wanted. These are as follows:

- Given a set of conditions, the student should be able to push their code to a git repository (Github or Gitlab).
- This should trigger an action that tests the IDS' capabilities with a predefined sequence of network packets.
- Provide a presentation of the results of the test.

Other notes and ideas included in the document was:

- Make use of Gitlab CI/CD or Github actions.
- Run the code in either containers or virtual machines.

4.1.1 Qualitative data: Meeting with the client & supervisor

The group members were introduced to the project with the initial description, wanted features and ideas. The client shared their thoughts on the project. The group would, according to the client, preferably find some kind of 'dummy' malicious network traffic that could be in the form of a PCAP file, to use as input data to the IDS. Further on, the client explained that the malicious network traffic could be hosted via a cloud service. This cloud hosted "malware network", as they called it, could be connected to a virtual machine gateway that would send the malicious network traffic to anything that connected to it.

Furthermore, the client explained that when the students pushed their IDS to git, CI would facilitate a connection with a virtual machine acting as a gateway to the "malware network". The "malware network" would then send malicious traffic intended to test the IDS. CI automatic testing would then analyze the result coming out of the IDS and produce a JSON file that reported how the IDS performed along with organized results from the IDS itself. The client also suggested that this JSON file would get pushed to a database, e.g. Elastic Search.

The group's supervisor argued that another solution to the project would be to compress this down to only using CI/CD to push PCAP files with malicious network traffic into the IDS. The IDS would then output the results to the test script, which would then output a test result. The group agreed that this approach was more realistic, at least as a starting point.

Before ending the meeting, the group asked the client if there were any requirements to create their own IDS for the project, or if there were any IDS systems that could be used for the testing. Client responded that an IDS such as Snort could be used, however, they also recommended looking into other possible solutions.

4.2 Initial stage

The vision, expectations, and criteria for the project were clarified in the meeting. This lead to the first draft of the project design, seen in Figure 4.1.

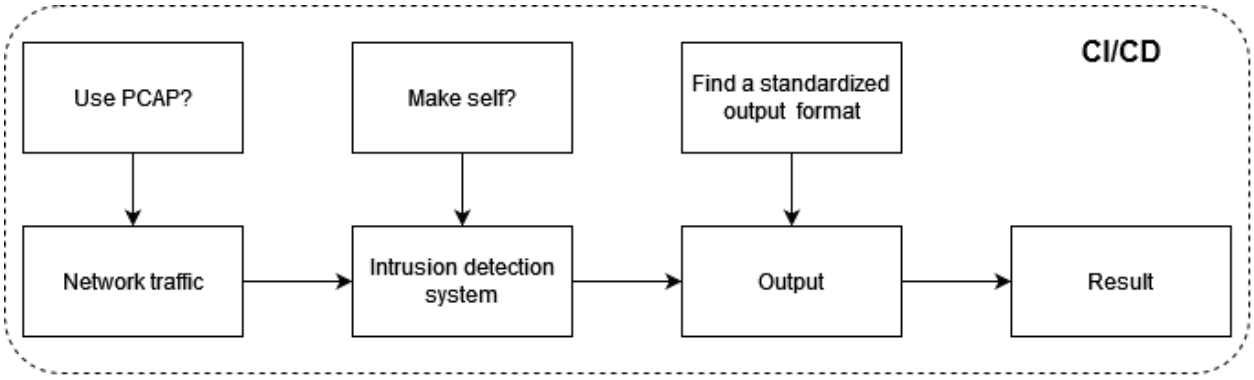

Figure 4.1: The original plan for the framework.

During the initial stage of the process, an investigation was performed in order to find a set of criteria for the IDS testing framework. There was also a need for a simple outline of the expected flow of the testing process. This outline served as a stepping stone for the development of the testing framework in future cycles.

4.2.1 Investigation

Before starting the first cycle, an investigation of what criteria the project should meet was performed. Looking at the task description, the test framework, in it's simplest form, should be able to run using captured network packets, and process the output of an intrusion detection software (IDS) after the network data had been processed by the IDS. Lastly the results of processing the IDS output should be output to the user, showing the results of the IDS test. Preferably this should all run as an automated CI/CD process on a git provided service, like Github or Gitlab.

In summary the IDS-test framework should be able to:

- Run as a CI/CD process on a git provider service (Like Github or Gitlab).
- Somehow provide packet capture data to the IDS.
- Capture the output of the IDS.
- Test if the output of the IDS is as expected.
- Present the results of the tests to the user in a readable way.

4.2.2 Design

Figure 4.2 shows a simple flowchart made to model the IDS test framework for the initial design phase. The flowchart illustrates the flow and interactions between the different components of the IDS test framework. In addition to this there was also a need to solve how the IDS output was going to look, as it would need to be parsed later in the process.

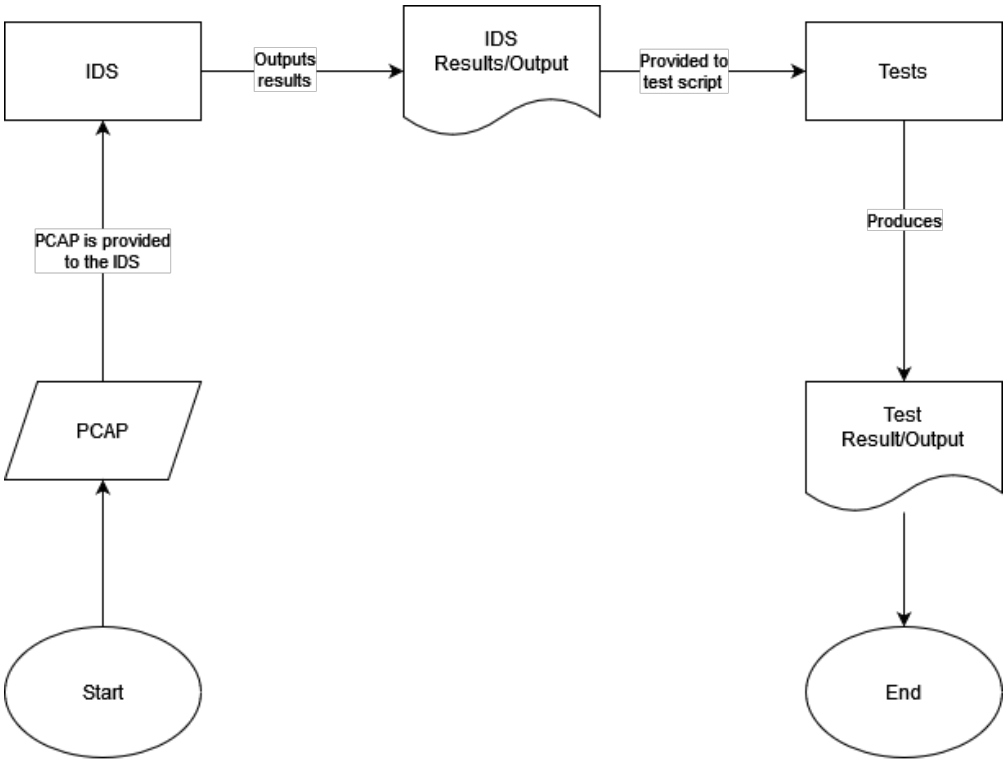

Figure 4.2: The initial design flowchart outlining the components of the test framework.

4.3 Cycle 1

4.3.1 Plan

During the first development cycle the plan was to start small with a simple prototype. The prototype would directly read packet capture data, and output a file using a well defined format for later parsing, as shown in Figure 4.3. This would enable the group to run tests offline, and facilitate a simple transfer from offline to online CI/CD testing later.

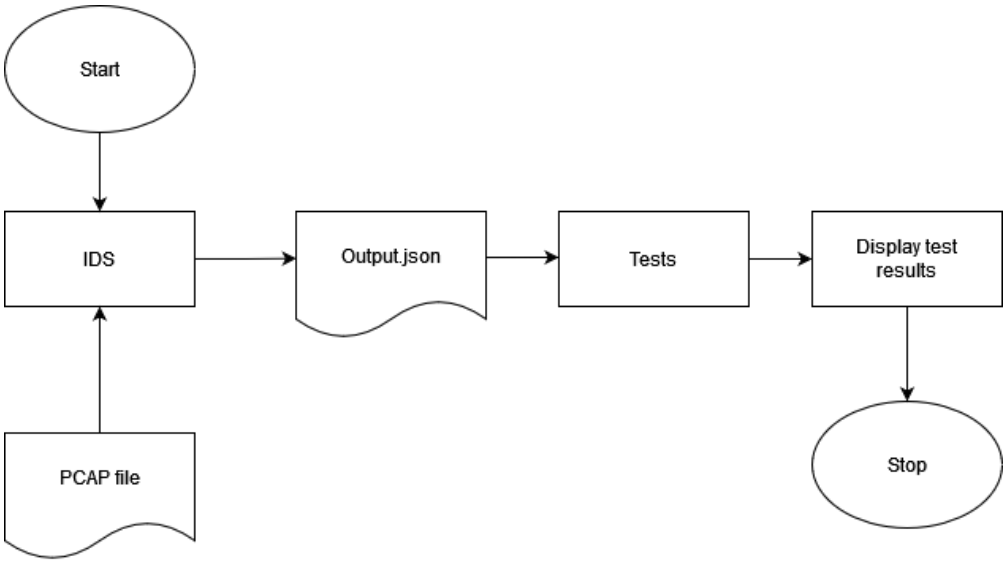

Figure 4.3: Plan for cycle 1.

4.3.2 Design

Despite it's small scope, the first cycle added a lot of restrictions on the design of the IDS. The group decided to make an IDS for the purpose of testing. This also allowed simple configuration changes to the IDS as needed. The IDS was made to be a signature based IDS, scanning for predefined patterns in packet payloads. After scanning the packets the IDS outputs a file containing information about which patterns were detected, the amount of patterns detected, and the payloads containing these patterns. Even though the IDS is not the focus of this project, it is an integral part of the first cycle. The IDS design has been reused for future cycles aswell.

The IDS must be able to:

- Process .pcap files (packet captures) directly.
- Use signature-based detection methods.
- Detect and store detection data.
- Output detection data as a file upon finishing.

Figure 4.4 shows a detailed flowchart showing how the IDS operates from start to finish while filtering packets.

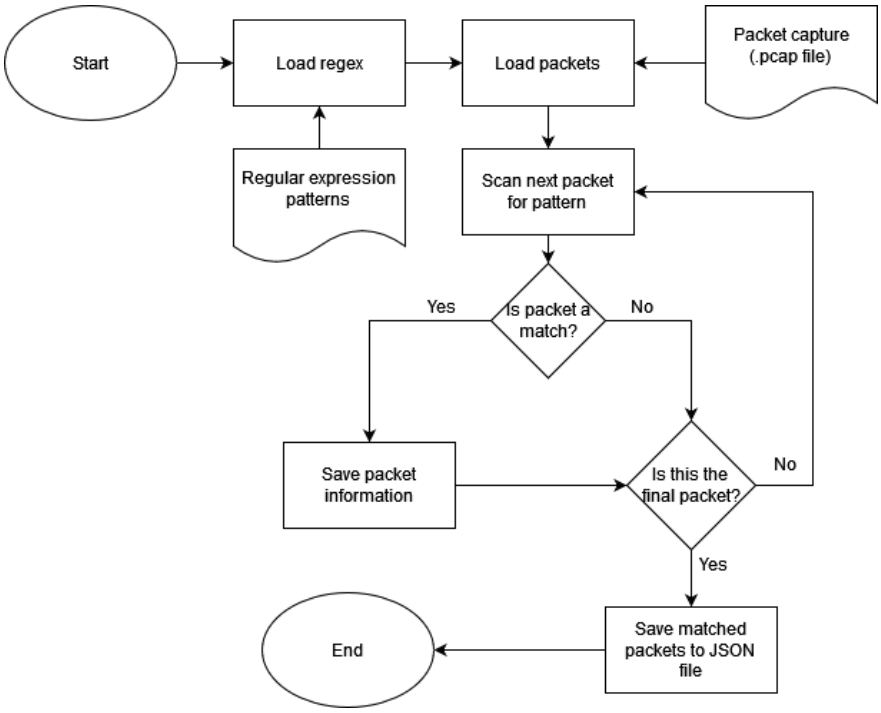

Figure 4.4: Intrusion detection system flowchart.

Testing was performed using Pytest, as it produced easily readable test output to the user. Using Pytest allowed the group to use Python's built in features for parsing JSON, eliminating reliance on more third party libraries.

The test script must be able to:

- Parse IDS output.
- Produce test results to the user.

Lastly the IDS output should be designed in such a way that it could be easily read by the test script. The testing script runs in Python, as mentioned previously, which provides native support for JSON files. JSON also allows the output to be structured like the IDS design mentions. For testing purposes the IDS output contains the following information, leading to the sample layout in Listing 2:

- Pcap filename.
- Packets scanned.
- Packets matched.
- Detection information, such as pattern, payloads and number of matches for this pattern.

```
{
    "filename": "<pcap filename>",
    "packets": "<total number of scanned packets>",
    "detections": "<number of packets matched or detected>",
    "detections_list": [
        {
            "pattern": "<detection pattern>",
            "detections": "<number of packets with this pattern>",
            "payloads": [
               "<list of payloads containing the pattern>"
                ]
        },
        {
            "pattern": "<detection pattern>",
            "detections": "<number of packets with this pattern>",
            "payloads": [
                "<list of payloads containing the pattern>"
                ]
        }
    ]
}
```

Listing 2: A sample .json file showing the output layout.

4.3.3 Implementation

IDS

The implementation of the IDS has stayed the same for future cycles. Looking at the design flowchart in Figure 4.4, the IDS first loads the patterns from an external file. The IDS for this project reads a text file containing regular expressions and loads them into a list. This list is later used when scanning the payloads for patterns.

After loading the regular expressions the IDS loads a packet capture file using the Scapy library [28]. Scapy allows easy access to read and edit packet payloads [6]. It also enables reading and editing packet information pertaining to different network layers, which is used in later cycles.

Once the patterns and packets have been loaded by the IDS, it iterates through the packets, and checks for pattern matches using the previously loaded regular expression with the Python re library [29]. If a match is made, relevant information is stored in a Python dict. The next packet is scanned after all regular expressions have been tested against the payloads of every packet. If no match is made, the program continues without storing information.

Once all payloads have been processed and scanned, the information previously stored in a dictionary is converted to json and saved to an output file (named "output.json" for ease of use). Once this file is saved the program exits.

Test script

The test script opens the json file as a fixture and passes it to the tests in Pytest. The Pytest script for the first cycle contains simple functions that assert that the data in the json file is correct, such as the packet capture file's filename, and the number of packets scanned. A snippet of code from the test script can be found in Listing 3.

```python
# Define a class and fixture for the test functions to use
class InputData:
    def __init__(self, filename):
        f = open(filename)
        self.data = json.load(f)
        f.close()

@pytest.fixture
def datafile():
    return InputData("output.json")

# Test filename
def test_filename(datafile):
    filename = datafile.data["filename"]
    assert filename == "input.pcap"

# Test number of packets scanned
def test_npackets(datafile):
    npackets = datafile.data["packets"]
    assert npackets == 101070
```

Listing 3: Example snippet from the first cycle Pytest script.

4.4 Cycle 2

4.4.1 Plan

Once the IDS and tests were finished, and were able to run offline, it was time to upload them to Git and automate the process. The plan for this cycle was simply to push the existing code to a repository, and have the tests run automatically on every future push, as outlined in Figure 4.5.

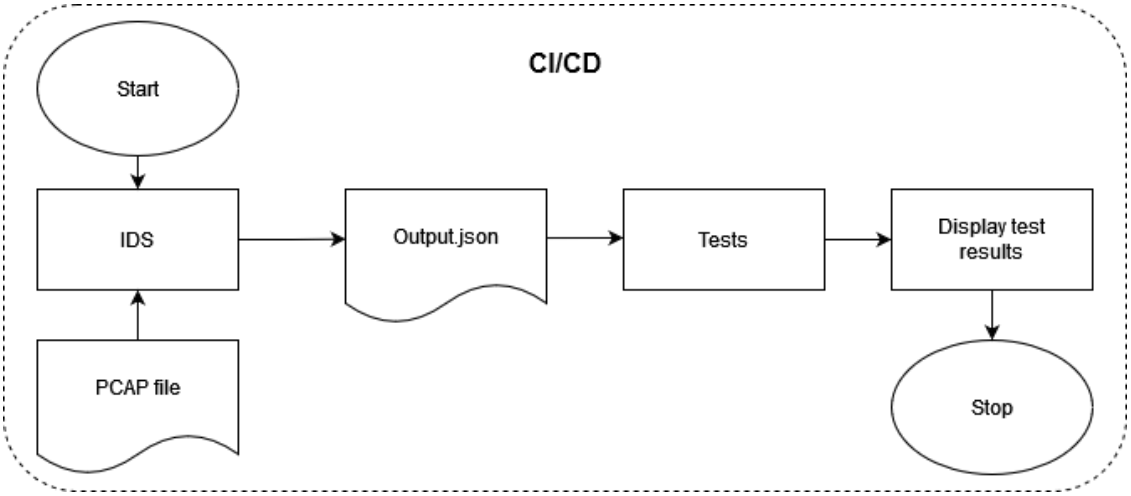

Figure 4.5: Plan for cycle 2.

4.4.2 Implementation

For this project Gitlab was chosen as the git platform. In addition to the code being pushed, a file called **.gitlab-ci.yml** is included in the root of the repository. The .gitlab-ci.yml file contains instructions on what to do and run every time code is pushed to the repository. The file for this implementation installs the latest version of pip, and some python libraries that the IDS and test script uses. It then gives the IDS' main.py script execution rights. Lastly the IDS is run (in our case with some parameters) and Pytest is invoked after the IDS has finished running. The file contents can be seen in Listing 4.

```
image: python:latest

variables:
    CI_DEBUG_SERVICES: "true"

test:
  before_script:
    - python --version  # For debugging
    - pip install --upgrade pip
    - pip install pytest argparse scapy-python3 scapy
    - chmod +x pcap/main.py
  script:
    - python3 pcap/main.py -f pcap/pcap/traffic.pcap
  after_script:
    - pytest -v --color=yes

```

Listing 4: The first gitlab-ci.yml script.

4.5 Cycle 3

4.5.1 Plan

Looking at the previous two cycles there are some major issues that must be addressed for the test framework to be feasible within a black box testing paradigm. Firstly the IDS in the previous cycles is a python program, and the .gitlab-ci.yml file has been tailored to this, including installing the IDS' dependencies. To allow IDS developers to run their program in an environment of their choice the IDS should be containerized.

Secondly the current IDS is constrained to read packet captures from a file. Allowing the IDS to listen to a network interface, and sending data into the IDS container allows for a more

realistic test scenarios. For this third cycle the IDS output is not of importance, but rather the network stream into the IDS container. In summary the plan for Cycle 3 is outlined in Figure 4.6, and the following list:

- Containerize the IDS.
- Run a program that processes packet capture files.
- Have the packet capture program send network traffic to the IDS container.

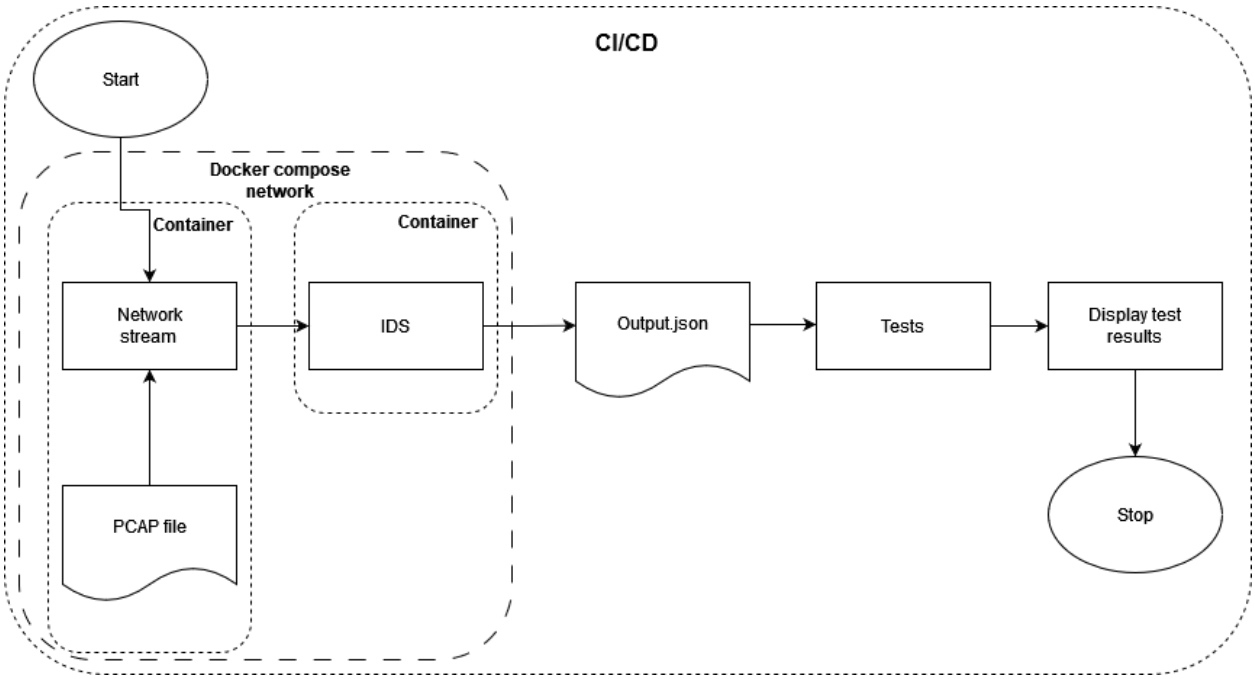

Figure 4.6: Plan for cycle 3.

4.5.2 Design

The design of the IDS, and the container it runs in, can now be left up to the IDS developers. The IDS only needs to listen to the default network interface, and preferably run in promiscuous mode to be able to read all network packets the container receives. The IDS output was of no concern this cycle, and is solved in Cycle 4.

The program reading and sending packets over the network should be able to edit packet information, so that it can reliably reach the intended recipient. Once all packets have been sent over the network, the program should shut down. This program should also run in a container, and be on the same network as the IDS to facilitate easier interaction between the two.

4.5.3 Implementation

To ensure that the black box testing worked for different IDS implementations, the python IDS was refactored and containerized in a Dockerfile. In addition to this a second IDS was developed in Rust using the same design as the Python IDS.

The script for the sender component reads packet capture files using scapy's rdpcap function, and opens a persistent layer 3 socket to use for sending packets over the network. The script then rewrites the destination MAC and destination IP address for the packet, and recalculates the checksum of the packet before sending it to a specified IP address. Packets that can't be sent are ignored, and continues to the next packet. The "sendTraffic" function in Listing 5 shows the implementation in code.

```
# Import scapy
from scapy.all import *
from scapy.utils import rdpcap
# Opens a pcap file and sends the contents to a specified receiver
def sendTraffic(self, pcapFile, receiverIP, receiverMAC):

    # Open a persistent socket to send packets
    socket = conf.L3socket()

    # Process packets, change receiverMac and reciver IP
    for packet in rdpcap(pcapFile):
        if IP not in packet:
            continue

        packet[Ether].dst = receiverMAC
        p = packet[IP]
        p.dst = receiverIP
        del(p.chksum)

        #Try to send packet using the persistent L3socket
        try:
            socket.send(p)
        except:
            pass
```

Listing 5: Implementation of the sender script.

To ensure that the sender script is able to reach the IDS container, docker compose is used to run both containers on the same network. This also allows setting custom IP and MAC addresses for the containers, and allows containers to be built from custom Dockerfiles. Given that the IDS has a predefined Dockerfile, using docker compose allows testing of that IDS simply by referring to the IDS' Dockerfile in the docker-compose.yaml file. In addition to the mentioned settings, docker compose also allows the IDS to be up and running before starting the sender script. The Docker-compose.yaml file is available in Listing 6.

```yaml
version: "3"

services:
  ids_container:
    build:
      # Build context may be either git repo or a folder
      context: ids/
      dockerfile: Dockerfile
    mac_address: 8a:ca:58:b9:e9:51
    networks:
      network:
        ipv4_address: 10.5.0.5

  sender:
    build:
      context: sender/
      dockerfile: Dockerfile
    networks:
      - network
    # Only start after the ids container has started
    depends_on:
      ids_container:
        condition: service_started

networks:
  network:
    ipam:
      config:
        - subnet: 10.5.0.0/16
          gateway: 10.5.0.1
```

Listing 6: Docker compose for container testing.

4.6 Cycle 4

4.6.1 Plan

During the third cycle the container output was of no concern, as no tests were run. Having the IDS output to a file within it's own container makes it harder to extract this data after the container is shut down. Also given that the IDS now listens to a network interface, it will be running indefinitely and will never shut down. To solve this the IDS should output to the standard output (stdout) for every detection, instead of a file. Once the IDS outputs to stdout, the logs from the container can be captured once it is shut down. In addition to this there should be some kind of mechanism that shuts down the IDS container after the network traffic is done transmitting. The plan for cycle 4 can be seen in Figure 4.7.

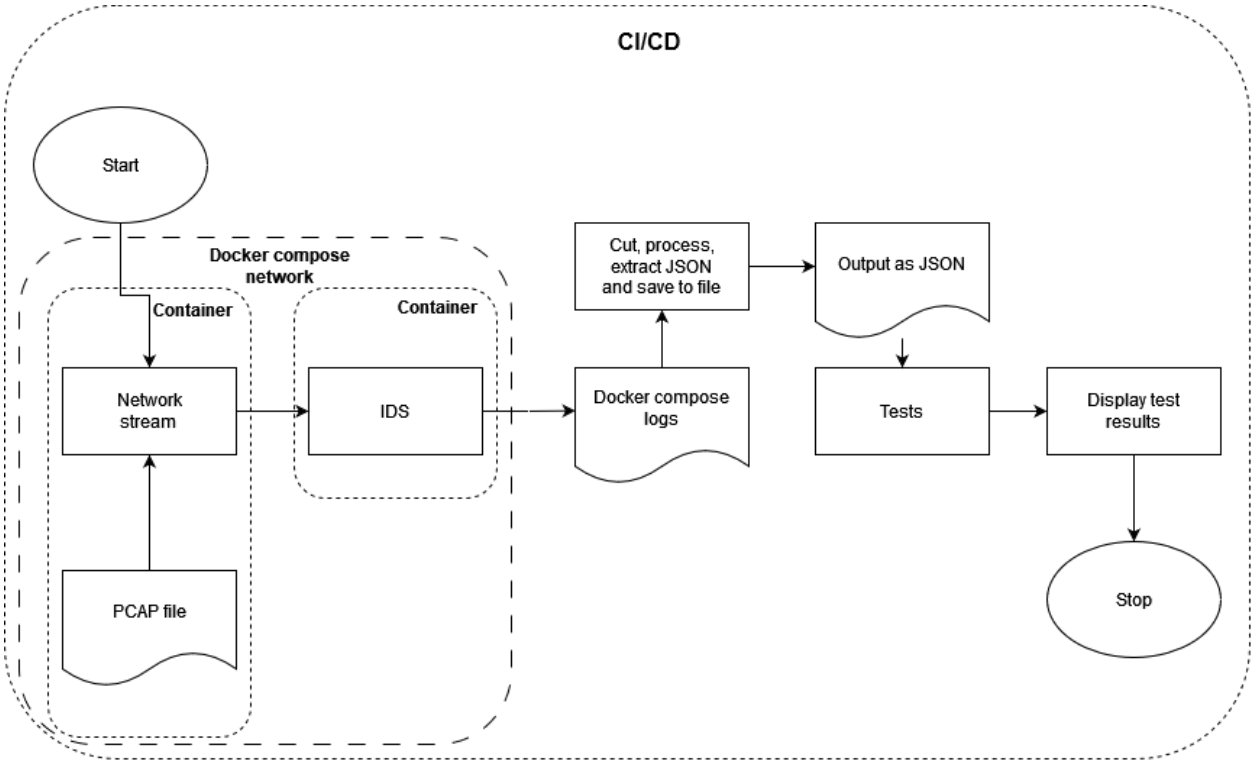

Figure 4.7: Plan for cycle 4.

4.6.2 Design and implementation

Moving away from the design in Listing 2, the IDS should now only output which patterns it is scanning for, and how many matches it has for each pattern. The IDS is still required to output this as JSON format, but it can now be output to stdout instead of a file. To collect the container output after the container is shut down, the docker compose logs are exported and processed into a final text file that the test script can open and run. The final gitlab-ci.yml in Listing 7 runs using a shell runner, and uses this script to run the network-and-IDS containers, and then runs the test. To shut down the IDS container the abort-on-container-exit flag is used, making the IDS container shut down as soon as the sender script shuts down.

```
image: python:latest

variables:
    CI_DEBUG_SERVICES: "true"

test:
  before_script:
    - pip install --upgrade pip
    - pip install pytest
    # Navigate into the appropriate folder containing docker-compose.yml
    - cd sender_and_ids
    # Remove old containers if they exist
    - docker compose down
    - docker compose build
    - docker compose up --abort-on-container-exit
  script:
    # Export the container logs, and process into a final file for testing
    - docker compose logs > logs.txt
    - cut -d "|" -f 2 logs.txt > logs_test.txt
    # Invoke pytest
    - python3 -m pytest -v --color=yes
```

Listing 7: The final gitlab-ci script, using a shell runner with preinstalled dependencies.

Chapter 5

Discussions

5.1 Research and development methods

The project objective was substantially assisted by the research and development techniques used in this project.

5.1.1 Research method

For research, qualitative data collection was performed during the meeting between the group, client and supervisor. This was discussed in the group internally beforehand to be used to get a deeper understanding of the specifications and opinions the client and supervisor had. As explained in the result section "4.1.1 Qualitative data: Meeting with the client & supervisor", there was an initial presentation on the client's opinions. These opinions were challenged by some of the supervisor's suggestions.

The client had foreseen that the framework would be built up by hosting a "malware network" in a cloud solution that included malicious network traffic data, connecting it to a gateway that sent this traffic to anything connecting to it. The group and the group's supervisor argued that this could get complicated to start the project with, and that the testing itself is the most important aspect of the project. It was agreed upon that the input data would be to use PCAP files directly into the IDS systems through CI.

Also, using a virtual machine for every IDS in CI could also cause problems because of the school's resources being insufficient. A mutual agreement of using containers was settled instead. This would save some of the school's resources, not needing to have a dedicated VM per test. The group also decided after recommendations from the supervisor to not focus on pushing the test results to a database, but instead working on getting the results

directly into the repository framework like Gitlab.

5.1.2 Development method

A decision was made that only using the waterfall method once would not be sufficient for the project. This was done under the justification that it was clear that some trial and error would be required to accomplish the project's goal. Smaller waterfall models in cycles lasting for about four weeks was the key to be able to plan, design, implement, test, and then evaluate the new design in the next cycle's planning phase.

The group did not perform any important revisions to the EVO method in chapter 3.2, other than skipping some phases that were unnecessary for the cycle. And, for the testing phases, the group decided to make some changes to the EVO model that would be a better fit to the project. Instead of conducting the test on other students or teachers for each cycle, the group decided that it would be better to just test the framework internally within the group. The reason for this is because the setup is supposed to be automatic, and should require little to no end-user input other then an IDS pushed to a repository. User-testing would be more appropriate for an application where there is a more direct program-interaction with the user's input.

5.2 Investigation, Design

The initial discussion with the teacher and supervisor resulted in a detailed project plan with many components, as seen in Figure 4.1. To keep the number of components manageable during the development stages, the group decided on a cyclical development process. This allowed the group to concentrate on a limited set of components at a time, and expand on those components as the project evolved.

5.3 Cycle 1

The goal of the first cycle was to create a simple prototype IDS that could read network packets, scan these for patterns, and output the result as a file on disk using a well structured format. There were several decision made for the first cycle that simplified or restricted the design of the project components, and was changed or expanded on in later cycles.

5.3.1 Implementing the IDS

Initially there was no intrusion detection software ready made for this project, but there was a need to have an IDS for testing purposes. A choice was made between using a known IDS like Snort [30], or make one. On one hand Snort offers the IDS capabilities we require, and more. It also offers logging capabilities which would allow the group to run tests on Snort's output after feeding it packets. Snort requires configuration to run, however, and it takes away control relating to the output formatting. In addition to this, the IDS is not the focus of the project. After some discussion a decision was made to make a "homemade" IDS. This would offer more control in relation to output formatting and general behaviour. It would also be quicker to develop, than learning Snort from scratch.

The group does not have the experience or technical knowledge to create an anomaly-based or hybrid IDS. The rapid development of a simple IDS was an important reason to create a signature-based IDS. The implementation was done with Python, as the group was familiar with this language, and the fact that Python offers a rich ecosystem of libraries to use [31].

5.3.2 Choosing IDS input

The choice was made to have the IDS read network packets directly from a packet capture file. This allowed for rapid debugging of the IDS during development. During testing this would allow for testing the logic the IDS uses to filter packets, without having to worry about components listening to network traffic within the IDS, or generating network traffic for the IDS to listen to. Despite allowing for simpler development and prototyping, having the IDS read packet captures directly is not a good long term solution. It imposes restrictions on the development of the IDS, and is an unrealistic mode of operation for an intrusion detection software. In the interest of keeping the scope of cycle 1 limited, the issue of changing how input was delivered to the IDS was pushed to a later cycle.

5.3.3 Choosing IDS output

When deciding on an output format for the IDS, it was important to use a well structured format. This way it could easily be parsed by a computer program and be used for testing. Because Python was already used for the majority of this project, and there is native JSON support built into the standard library [32], the decision was made to use this for the first cycle.

The output is saved to a file on disk. This was easier to implement than capturing the standard output after the fact. The downside of this output method was the test script needed to know the path and filename of this file in order to work, further restricting design. It is not the most realistic approach to IDS output, because of the restrictions imposed on the program. This will be remedied in later cycles.

5.3.4 Choosing test framework

The project needed a way of testing the IDS output. There are several test frameworks available, like Jasmine [33] that is compatible with, and written in JavaScript, or Avocado [34] that is a test framework that is written in Python. A common test framework used with Python applications is Pytest [35]. Pytest produces easily readable test output, can be used regardless of IDS implementation because of the JSON output, and the group has previous experience using the this framework. This resulted in Pytest being the test framework of choice for this project.

5.3.5 Testing

The initial tests were designed to be run simply for the purpose of testing the JSON output format, and prove Pytest worked for displaying results. It was unknown what tests the teacher or client had in mind for the finished product. These tests were therefore just designed to check if the IDS detected a certain number of packets containing a specific pattern, as a proof of concept.

The design of the JSON file contents was intended to allow an experienced test writer with knowledge of the content of the input packet capture files to write tests that could check, not only for positives, but false positives besides.

5.4 Cycle 2

Cycle 2 was centered around implementing automatic testing using CI, and getting all the components running in a git framework.

5.4.1 Choosing Gitlab

There was a need for a place to run the automated components of the project. There are several git provider platforms that could be used for this purpose, such as Gitlab, Github or Bamboo. The group had previous experience making CI scripts in Gitlab, and managing runners on this platform. Gitlab also runs internally on the university's servers. Alternatively the university could provide access to Bitbucket with Bamboo for testing, but there were some issues using this because of issues with elevated account clearance levels for students. Github does not run internally at the university, making Gitlab a more attractive choice. Because of the group's previous experience using Gitlab, and the fact it was provided by the university, Gitlab was chosen as the CI platform for this project.

5.4.2 White box

There are numerous issues with the current implementation of the test environment. Firstly the IDS requires dependencies to be installed directly on the runner. This makes an implementation-independent test environment very hard to implement, as people can write their IDS in whatever language they prefer. As of the second cycle the current solution is to use the CI script to install dependencies.

The second issues with the test environment is how the PCAP files are accessed directly by the IDS. This means that the IDS developer must refer to an already existing PCAP file in the repository. The PCAP file name and path must be known. This can be edited by the IDS developer, influencing the results of the test. It is unrealistic to have the test files lie in the IDS repository. Decoupling the IDS from the PCAP file is an issue for future cycles. The last issue with the test environment is how the output is stored by the IDS. The IDS must store the output in a location and with a filename that is known by the test script. This issue should somehow be handled by the CI-script and is also an issue for future cycles.

5.5 Cycle 3

From the limited scope of the first two cycles, a number of issues arose that needed to be solved in later cycles. Specifically the need to decouple the IDS from reading the PCAP files directly, and solving the possible dependency issues arising from potentially having multiple IDS implementations in different languages using different libraries.

5.5.1 Containerization

The first problem was to make any IDS implementation work, as long as it could produce an readable output. To do this the solution was to containerize the IDS, so the dependencies could be solved outside the test environment. This would enable students to make an IDS in their preferred programming language with any dependencies they required, without compromising the test environment. Containerizing the first step towards a black box testing paradigm. To test this the group rewrote the IDS in Rust and containerized it using a Dockerfile to ensure the black box capabilities of the framework.

Once the move to a containerized IDS was made, two new issues arose. The first issue was the need to decouple the IDS from the PCAP files. This meant the network packets should be sent to the container, instead of the IDS reading this directly. The solution to this was to create a separate container on the same network that reads the PCAP files and routes the content of these into the IDS container. To produce a network stream TCPreplay and Scapy was tested.

Initially the network stream was produced using Scapy. Scapy allowed rewriting of packet metadata. Using these capabilities Scapy allowed for quick rerouting of packets by editing this data before sending the packets. Because of a programming error, sending these packets was terribly slow. To remedy this the group attempted to switch to the TCPreplay suite. Using the TCPLiveReplay utility to rewrite packets on the fly provided to be a sensible option to Scapy. TCPLiveReplay is not an ideal choice, because the utility is not actively maintained. After performing some attempts at sending network traffic using TCPLiveReplay, it crashed because of corrupt packet data in the PCAP file. This resulted in another attempt at Scapy, and the programming error resulting in slow packet transfers was solved by establishing a persistent socket connection. This sped up the packet transfer rate by more than 400 times. The advantage of using Scapy over TCPLiveReplay was that Scapy allowed for a way to handle corrupt packets, allowing for error handling. The error handling, continuous support and python library implementation made Scapy a superior choice to TCPLiveReplay.

The script producing the network stream with Scapy was moved to a separate container. To make the IDS container and the network stream container communicate reliably Docker Compose was used. Docker Compose allowed to set static IP-addresses for the containers and define a closed network for the containers to communicate. In addition to this the docker compose file can be tailored to use docker images from different repositories, facilitating separation between the IDS repositories and the test repository. For the purposes of this project, docker compose offered advantages over virtual machines, which was another option. Docker compose is less resource intensive, requires less configuration and is quicker to start and stop containers.

5.6 Cycle 4

Cycle 4 would focus on the improvements made to the project, to make the IDS less dependent on the Framework, and lighten the restrictions that gave. The discussion will discuss these improvements.

5.6.1 Runners

The runners used until this point of the project where shared runners. The problem with shared runners is when there is need to change something on them. The shared runners would now have a problem running docker compose commands, because of a lack of privileges. This is a problem because there was a need for administrator rights to run docker in docker or docker compose. Because of this, there was a need to switch to a local runner.

The local runner was set up on a Ubuntu virtual machine that was provided by the supervisor. As the Ubuntu server was on the same network as the Gitlab there is no problem connecting these two together. The runner used was a shell runner because of the freedom this gave. There were also an attempt to use a docker runner, but this created some dependency issues with the docker in docker image. In the interest of time it was decided not to pursue this option. The difference between a shell runner and a docker runner, is that the shell runner [36] runs directly on the machine that hosts the runner. The docker runner [37] will be able to run in an docker environment, with all the benefits of a docker environment Multiple projects can run on the same machine, which is a plus considering that multiple students will probably test simultaneously, without interfering with each other. Docker environments will also run from a clean start and delete everything when done. The shell runner will be

the faster option, but will need dependencies to be preinstalled on the runners host machine. Artifacts need to be cleaned up after each run manually in the shell runner, or they may be left over from one run to the next. There is also the security aspect; the runner runs in the shell of the host machine. To our project the docker runner would give the most sense, but because of the issues with sorting out dependencies in the docker in docker container, and because of time constraints the runner was chosen to be a shell runner.

5.6.2 Stdout output

The output at this point was made into a file from the IDS. This file needed to be named right, and have a specific file path, in order to make the CI pipeline work properly. There was a need for a new way of obtaining this information, to make the restrictions on the development of the IDS as low as possible. There were a couple of choices to choose from: Make the students set up a server that would print the JSON format, as an API. This choice would lay more work to the students, even if there were an guide to make this API work. And would demand more workload on the CI pipeline. Another choice would make use of the stdout. This would mean that the students IDS only had to worry about having a readable JSON output, and dump that information straight to stdout. The only problem of this approach would be for the Pytest to obtain the output. this was solved by copying the docker logs, cut unnecessary information and extract them to a file. which later is read by the Pytest. The way that the logs were extracted was made without the need of the containers name, which meant no need for manually finding the name of the container. The downside to this approach is if the cut command would cut wrong, and delete valid output, as the cut command searches for the character "|". The character were chosen as every line starts with "docker container name" and the character "|". This may not be the optimal solution but worked fine with the IDS outputs that were used in testing, and may need to be edited in future cycles. A solution could be to have a script that reads the characters of the docker container plus 2 characters to delete the appropriate number of characters to make this a JSON only file.

5.6.3 Stop IDS from running after traffic is sent

The group has not made any mechanisms that stop the IDS from running after the sender script has finished sending network traffic. Ideally an interrupt signal of some kind should be sent from the sender script to the IDS. The solution ended up being using the "–abort-on-container-exit" option when running docker compose. When the sender script has finished producing network traffic and shuts down, the IDS container is forced to shut down as well.

The downside to this approach is that there may be packets that have not yet been processed by the IDS when it is forced to shut down. To remedy this the sender script waits for a few seconds before stopping, but this is not a good long term solution. Ideally implementing some kind of communication between the two containers, leading to a graceful IDS shutdown would be a better option.

5.6.4 The container output format

When looking at the previous cycles the IDS had output JSON formatted information to a file on disk. When the change was made to stdout, the output had to be simplified to be easier to work with. The IDS is expected to output one line for every detection it makes. Preferably it should print the pattern and the payload for later analysis and testing. Printing the payloads proved to produce output that was hard to parse, due to the payload packets being of different sizes and varying content types. The decision was made to have the IDS output a JSON object containing all patterns and matches for each pattern on a single line for each detection, as shown in Listing 8. This reduces the amount of information one can get from testing the IDS.

```
1 {"pattern1": number, "pattern2": number, "pattern3":  number}
```

Listing 8: Example stdout IDS output.

One alternative to JSON that was attempted was a comma separated values format (CSV). Using CSV it was possible to produce the same output as with JSON in a more compact format. The issue with using CSV over JSON was the fact that one requires a line containing the fields of the CSV, before the actual content of the CSV. In addition to this, attempts to output payloads (as with the JSON format) provided a challenge, because of the likelihood of commas appearing in printed payloads would create issues while parsing. In the end JSON was chosen as the best alternative between the two.

5.7 Future Cycles

Though the group has a working solution, there is always room for improvements to the project.

As mentioned previously the output format has been changed from the first, to the fourth cycle, and unfortunately provides a lot less information about the content the IDS has filtered in the newer cycles. Ideally the output should provide more information than just a list of patterns and number of detections. The group suggests that the output should contain some information about the packet itself, as well as a way to report the level of security threat a detected packet poses.

In addition to changing the output format of the IDS, the way the output is collected from the docker container is not a reliable process, and may result in errors parsing input from irregularities that may occur when testing other IDS. Gathering the logs and using a more specialized utility than cut may provide better, less error prone results.

A smaller issue is the fact that generating tests is left up to the user of the framework. Ideally there should exist a utility that enables one to automatically generate tests from a set of criteria, or from a set of PCAP files. In addition to this a utility that generates network packets with certain test related traffic would help facilitate easier testing. There are existing utilities that help generate network traffic, but having an "all in one" utility for the framework would be a good addition in future iterations.

Currently there is 1 working repository, with an all in one CI pipeline. Although this works, the optimal solution as stated previously should be a 2 repository solution. The students would have the Dockerfile put in a separate repository to make the students unable to access the test files and the testing environment. To reduce the risk of tampering with the results.

Chapter 6

Conclusion

In this project the objective was to create a framework that allows students to push their intrusion detection system to a repository. This should start an automated testing process of the intrusion detection system, producing a human readable result for the user.

In addition to this, the group attempted to fulfill the feature requests from the client outlined in the assignment description in Appendix A:

- Given a set of conditions, the student should be able to push their code to a git repository (Github or Gitlab).
- This should trigger an action that tests the IDS' capabilities with a predefined sequence of network packets.
- Provide a presentation of the results of the test.

To acquire these desired results and findings, the group conducted a qualitative research through a meeting with the client and supervisor of the project. This meeting gave the group a deeper understanding of the client and supervisor's thoughts and opinions, other than what was described in the assignment description. The Hewlett-Packard EVO method was used to divide the framework's development into several, smaller cycles instead of one large waterfall cycle.

During the course of this project the group created a small prototype intended to be a proof of concept. This prototype was then expanded on in subsequent development cycles, resulting in a test framework utilizing Docker Compose to create a closed network between two containers. One container producing network traffic to the other container containing the IDS. Pytest was utilized in all phases of the project to produce readable test results.

Given a set of conditions, the student should be able to push their code to a git repository. When pushing the IDS to a Git repository there are three conditions that needs to be fulfilled: The repository should contain a Dockerfile that solves dependencies and runs the IDS, The Docker Compose file needs to refer to the Dockerfile of the IDS, and the CI-script should be present in the repository. Given the set of conditions the Ci-pipeline is triggered, and the student's IDS should be processed through the pipeline, producing a readable presentation of the test results through the use of Pytest.

Looking at the feature requests from Appendix A, the IDS test framework is able to fulfill these requests. The group has produced a simple test framework to perform IDS testing. As mentioned in the discussion there are several aspects of this framework that could be changed or improved for a better user experience. Utilities to improve production of tests and network traffic, improvements to the container output format, and improvements to the processing of the container output is necessary to create a reliable test framework.

In conclusion the group has produced a simple test framework that is able to perform the basic functions requested by the client. There are however a number of improvements that should be done before this framework can be reliably used in practice.

Bibliography

[1] Fabio Roli Igino Corona Giorgio Giacinto. *Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues*. 2013.

[2] Stefan Axelsson. *Intrusion Detection Systems: A Survey and Taxonomy*. 2020.

[3] Checkpoint. *Intrusion Detection System (IDS)*. URL: https://www.checkpoint.com/cyber-hub/network-security/what-is-an-intrusion-detection-system-ids/. (accessed: 15.02.2023).

[4] IBM. *How does software testing work?* URL: https://www.ibm.com/topics/software-testing. (accessed: 28.04.2023).

[5] GeeksforGeeks. *Intrusion Detection System (IDS)*. URL: https://www.geeksforgeeks.org/intrusion-detection-system-ids/. (accessed: 09.02.2023).

[6] P. Biondi and the Scapy community. *Scapy - General Documentation - Introduction*. URL: https://scapy.readthedocs.io/en/latest/introduction.html#about-scapy. (accessed: 27.04.2023).

[7] H. Krekel and pytest-dev team. *pytest: helps you write better programs*. URL: https://docs.pytest.org/en/7.3.x/index.html. (accessed: 02.05.2023).

[8] H. Krekel and pytest-dev team. *Get Started*. URL: https://docs.pytest.org/en/7.3.x/getting-started.html. (accessed: 02.05.2023).

[9] H. Krekel and pytest-dev team. *API Reference*. URL: https://docs.pytest.org/en/7.3.x/reference/reference.html#fixtures. (accessed: 02.05.2023).

[10] Atlassian S. Pittet. *The different types of software testing*. URL: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing. (accessed: 28.04.2023).

[11] T. Hamilton. *What is Software Testing? Definition*. URL: https://www.guru99.com/software-testing-introduction-importance.html. (accessed: 28.04.2023).

[12] National Institute of Standards and Technology. *Verification and Test Methods for Access Control Policies/Models*. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-192.pdf. (accessed: 29.04.2023).

[13] Imperva. *Black Box Testing*. URL: https://www.imperva.com/learn/application-security/black-box-testing/. (accessed: 29.04.2023).

[14] National Institute of Standards and Technology. *Information Security*. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-137.pdf. (accessed: 28.04.2023).

[15] Imperva. *White Box Testing*. URL: https://www.imperva.com/learn/application-security/white-box-testing/. (accessed: 02.05.2023).

[16] Gitlab. *What is Git version control*. URL: https://about.gitlab.com/topics/version-control/what-is-git-version-control/. (accessed: 29.04.2023).

[17] GeeksforGeeks. *Version Control Systems*. URL: https://www.geeksforgeeks.org/version-control-systems. (accessed: 15.05.2023).

[18] TechTarget. *GitLab*. URL: https://www.techtarget.com/whatis/definition/GitLab. (accessed: 14.04.2023).

[19] AWS Amazon. *What is DevOps?* URL: https://aws.amazon.com/devops/what-is-devops/. (accessed: 14.04.2023).

[20] AWS Amazon. *What is Continuous Integration*. URL: https://aws.amazon.com/devops/continuous-integration/. (accessed: 14.04.2023).

[21] GitLab. *Install GitLab Runners*. URL: https://docs.gitlab.com/runner/install/. (accessed: 10.05.2023).

[22] EDUCBA. *GitLab Runner*. URL: https://www.educba.com/gitlab-runner/. (accessed: 29.04.2023).

[23] Docker Docs. *Docker Overview*. URL: https://docs.docker.com/get-started/overview/. (accessed: 29.04.2023).

[24] Docker Docs. *Docker Compose overview*. URL: https://docs.docker.com/compose/#docker-compose. (accessed: 07.05.2023).

[25] Pritha Bhandari. *What is Qualitative Research?* URL: https://www.scribbr.com/methodology/qualitative-research/. (accessed: 18.04.2023).

[26] Elaine L. May and Barbara A. Zimmer. *The Evolutionary Development Model for Software*. Aug. 1996. URL: https://www.hpl.hp.com/hpjournal/96aug/aug96a4.pdf. (accessed: 15.02.2023).

[27] Lucitasari D. Rachmawati, Khannan M. S. Abdul, et al. "Designing Mobile Alumni Tracer Study System Using Waterfall Method: an Android Based." In: *International Journal of Computer Networks and Communications Security* 7.9 (2019), pp. 196–202.

[28] Scapy community. *Scapy*. URL: https://scapy.net/. (accessed: 12.04.2023).

[29] Python. *re - Regular expression operations - Python 3.11.3 documentation*. URL: https://docs.python.org/3/library/re.html. (accessed: 12.04.2023).

[30] Snort. *Snort - Network Intrusion Detection & Prevention System*. URL: https://www.snort.org/. (accessed: 03.03.2023).

[31] Python. *The Python Standard Library*. URL: https://docs.python.org/3/library/. (accessed: 10.05.2023).

[32] Python. *json - JSON encoder and decoder*. URL: https://docs.python.org/3/library/json.html. (accessed: 10.05.2023).

[33] *Jasmine Documentation*. May 2023. URL: https://jasmine.github.io. (accessed 10.05.2023).

[34] *Avocado Framework*. Mar. 2023. URL: https://avocado-framework.github.io. (accessed 10.05.2023).

[35] *Pytest: helps you write better programs — Pytest documentation*. May 2023. URL: https://docs.pytest.org/en/7.3.x. (accessed 10.05.2023).

[36] *The Shell executor | GitLab*. May 2023. URL: https://docs.gitlab.com/runner/executors/shell.html. (accessed 11.05.2023).

[37] *Docker executor | GitLab*. May 2023. URL: https://docs.gitlab.com/runner/executors/docker.html. (accessed 11.05.2023).

Appendix A

Assignment description

AutomaticIDS.md 11/8/2022

Automatic IDS testing

Description

In a new course covering Intrusion Detection Systems (IDS), the supervisor is looking for a way to automatically test IDS systems developed by the the students.

Features wanted (not in order of priority)

- Given a set of conditions, the student should be able to push their code to a git repository (github or gitlab).
- This should trigger an action that tests the IDS' capabilities with a predefined sequence of network packets
- Provide a presentation of the results of the test

Notes & Ideas

- Make use of gitlab CI/CD or github actions
- Run the code in either containers or virtul machines
-