Automating a test strategy for a protocol decoder tool

by

Henrik Johansson

LIU-IDA/LITH-EX-A--08/038--SE

2008-09-03
Final Thesis

Automating a test strategy for a protocol decoder tool

by

Henrik Johansson

LITH-IDA/LITH-EX-A--08/038--SE

2008-09-03

Supervisor: Patrik Lindholm
Ericsson AB
Site Linköping

Examiner: Simin Nadjm-Tehrani
Linköpings universitet
Department of Computer and Information Science
Abstract

Within Ericsson AB, integration and verification activities is done on the network level in order to secure the functionality of the network. Protocol analysers are used to capture the traffic in the network. This results in many log files, which needs to be analysed. To do this, a protocol decoder tool called Scapy/LHC is used. Scapy/LHC is a framework that allows the users to write their own script to retrieve the data they need from the log files. The Scapy/LHC framework is incrementally developed as open source within Ericsson when there are needs for more functionality. This is often done by the users, outside normal working tasks. Because of this, there is almost no testing done to verify that old and new functionality works as expected, and there is no formal test strategy in use today.

The goal of this master’s thesis is to evaluate test strategies that are possible to use on the Scapy/LHC framework. To make the time needed for the testing process as short as possible, the test strategy needs to be automated. Therefore, possible test automation tools shall also be evaluated.

Two possible test strategies and two possible test automation tools are evaluated in this thesis. A test strategy, where the scripts that are written by the users are used, is then selected for implementation. The two test automation tools are also implemented. The evaluation of the implemented test strategy shows that it is possible to find defects in the Scapy/LHC framework in a time efficient way with help of the implemented test strategy and any of the implemented test automation tools.
Acknowledgment

I would like to express my thanks to my colleagues at Ericsson. Especially to my supervisor Patrik Lindholm, my secondary supervisor Jens Vevstad and my manager Roland Ævegran for making this thesis possible. I would also like to thank my examiner Simin Nadjm-Tehrani at IDA, Linköpings Universitet.
# Table of contents

1 Introduction ..............................................................................................................................1
  1.1 Background ......................................................................................................................1
  1.2 Purpose .............................................................................................................................1
  1.3 Method .............................................................................................................................2
  1.4 Requirements ....................................................................................................................2
  1.5 Limitations .......................................................................................................................2
  1.6 Target audience ................................................................................................................2
  1.7 Outline ..............................................................................................................................3

2 Theoretical background ............................................................................................................5
  2.1 Software testing ................................................................................................................5
    2.1.1 Testing levels ............................................................................................................5
    2.1.2 Regression testing ....................................................................................................6
    2.1.3 Test automation ........................................................................................................6
    2.1.4 Test automation architecture ....................................................................................7
  2.2 Protocol analyser ..............................................................................................................9
    2.2.1 Protocols ...................................................................................................................9
    2.2.2 Protocol stacks ..........................................................................................................9
    2.2.3 How a protocol analyser works ..............................................................................11
  2.3 The Scapy/LHC framework ...........................................................................................11
    2.3.1 Architecture ............................................................................................................12
    2.3.2 Development ..........................................................................................................13
    2.3.3 Testing ....................................................................................................................13
  2.4 Python .............................................................................................................................13
    2.4.1 GUI development ...................................................................................................14
    2.4.2 Unit testing framework ...........................................................................................14

3 Evaluation of test strategies ....................................................................................................15
  3.1 Requirements on the test strategy ...................................................................................15
  3.2 Automatic tests for the Scapy/LHC framework .............................................................16
  3.3 Test strategies ................................................................................................................16
    3.3.1 Using high-level test scripts ...................................................................................16
    3.3.2 Using unit tests on separate functions ....................................................................18
  3.4 Test automation tools .....................................................................................................19
    3.4.1 Using PyUnit together with TestOOB ...................................................................20
    3.4.2 Develop and implement a new test automation tool ..............................................23
  3.5 Selection of test strategy ................................................................................................24
  3.6 Selection of test automation tool ....................................................................................24

4 Implementation of test automation tools ................................................................................25
  4.1 Choice of programming language and development environment ................................25
  4.2 Implementing a tool using PyUnit together with TestOOB ..........................................25
    4.2.1 Comparator .............................................................................................................26
    4.2.2 Test case finder ......................................................................................................26
    4.2.3 Use cases ................................................................................................................27
  4.3 Implementation of a new test automation tool ...............................................................29
    4.3.1 System overview ....................................................................................................29
List of figures

Figure 1: Tools in a software-testing workbench.................................................................8
Figure 2: The package construction, from the application layer to the link layer.....................10
Figure 3: The protocol stack for a message sent from a web browser to a web server..............11
Figure 4: The Scapy/LHC framework architecture..............................................................12
Figure 5: Add a new test case to the test suit ......................................................................17
Figure 6: Run a test case .......................................................................................................18
Figure 7: Run a test case .......................................................................................................19
Figure 8: Test result when a test case is run with TestOOB ..................................................21
Figure 9: Test report in HTML format from TestOOB.........................................................22
Figure 10: Test automation tool architecture .......................................................................23
Figure 11: Directory structure for the test automation tool....................................................26
Figure 12: Statistics after all test cases have been run ..........................................................28
Figure 13: The difference log ...............................................................................................28
Figure 14: The test report in HTML format ..........................................................................29
Figure 15: Directory structure for the test automation tool....................................................30
Figure 16: The test report in HTML format generated by the report generator ....................31
Figure 17: The test automation tool with a test suite loaded ..................................................33
Figure 18: The test case configuration dialog .......................................................................34
Figure 19: The test automation tool after all test cases have been run ...................................35
Figure 20: The difference log ...............................................................................................36
Figure 21: Statistics after all test cases have been run ..........................................................37
Figure 22: A test case is added with the command line interface .............................................37
Figure 23: The test automation tool integrated in different versions of the Scapy/LHC framework ................................................................................................................38
Figure 24: Number of test cases that failed when changes were introduced in different protocols ..................................................................................................................44

List of tables

Table 1: Results from the measurements of elapsed time to add a test case..........................45
1 Introduction

This thesis has been prepared as a partial fulfilment of a Masters degree in Computer Science and Engineering at Linköping University. It is based on work carried out at Ericsson AB Linköping and performed under supervision of Simin Nadjm-Tehrani at department of computer and information science.

1.1 Background

Ericsson AB is a provider of telecommunication equipments and related services to mobile and fixed network operators globally. This thesis has been carried out within the System Integration & Verification organisation situated at Ericsson in Linköping, Sweden. System Integration & Verification is responsible for integration and verification on the network level.

When verification is performed on the network level, the tester must be able to see which and when messages are sent between the devices on the network. To do this, the tester uses protocol analysers. The protocol analysers generate many log files with network traffic that needs to be post processed and analysed. To do this, the log files are post processed by Python scripts, which are based on a framework called Scapy/LHC.

The Scapy/LHC framework is a protocol decoder tool that covers most network protocols that are used in mobile networks. The framework lets the user have easy access to all attributes in the protocols. It is therefore easy for users to write their own automatic analysing scripts when they need it. The Scapy/LHC framework is written in the programming language Python and is incrementally developed as open source within Ericsson when there are needs for more functionality. This is done outside normal working tasks and there is therefore often almost no time to test that old and new functionality actually works. Adding a new module might therefore negatively affect the earlier modules functionality.

1.2 Purpose

There is no formal test strategy for the Scapy/LHC framework in use today. This means that the testing of the framework is insufficient and is done manually, which is time consuming. Because of the time limitation for testing old and new functionality in the Scapy/LHC framework, there is a need for a test strategy for efficient and dynamic testing of the Scapy/LHC framework when new functionality has been added. This in order to verify that new and old functionality still works as expected. The goal of this master’s thesis is to elaborate and evaluate possible test strategies. The test strategy that is most appropriate to use on the Scapy/LHC framework should then be implemented. To make the test strategy more time efficient and to minimise the manual tasks, possible test automation tools will also be evaluated and implemented.

This thesis aims to answer the following questions.

- Which test strategies can be used on the Scapy/LHC framework in order to support verification of old and new functionality?
- Is it possible to automate tasks in the test strategies?
- Which test automation tools can be used together with the test strategies and the Scapy/LHC framework?
1.3 Method
To meet the goals in section 1.2, the following method was used. The project was divided into four phases.

1. Literature study
The Scapy/LHC framework is written in the programming language Python, therefore Python and Scapy/LHC were studied in order to understand how the framework works. To be able to evaluate possible test strategies and to get knowledge about how to design and implement a good test automation tool, theory about testing, test strategies and test automation were studied.

2. Evaluation of test strategies and test automation tools
Test strategies that are possible to use on the Scapy/LHC framework were evaluated and the one that was most appropriate was implemented. Possible test automation tools, which will support the test strategies and make them more effective, were also evaluated.

3. Implementation and testing
The selected test strategy could and two test automation tools were implemented. The implemented test automation tools were tested to make sure that everything works as expected.

4. Evaluation of implemented test strategy and test automation tools
The implemented test strategy was evaluated in order to see if the requirements are met. In order to decide which tool that is most appropriate to use with the implemented test strategy, the tools were also evaluated.

1.4 Requirements
After a discussion with an experienced user and developer of the Scapy/LHC framework, the following requirements were stated for this thesis:

- The Defect Detection Percentage (DDP) for the implemented test strategy shall be at least 50 % when the test strategy is run with 20 test cases.
- The average elapsed time to add a new test case shall be at most two minutes.

1.5 Limitations
This report focuses on test strategies and test automation. Theories about how to write good test cases is out of scope for this report and will not be discussed.

1.6 Target audience
The target audience for this report is mainly the people within Ericsson who develop and use the Scapy/LHC framework, but also others interested in test strategies and test automation. The report is written with the postulation that the reader has basic knowledge in computer science and software engineering, but need not to be an expert in the subject.
1.7 Outline

1. Introduction
In this chapter the background, the purpose and the requirements for this thesis are stated. The method that was used is also described.

2. Theoretical background
In this chapter the theoretical background about testing, test automation, protocol analysers and the Scapy/LHC framework are described. This will build a ground for the further discussion in this report.

3. Evaluation of test strategies
In this chapter is the test strategies that is possible to use on the Scapy/LHC framework evaluated. The test strategy that is most appropriate to use is then selected for implementation. To be able to automate the test strategy, two test automation tools is also evaluated.

4. Implementation of test automation tool
In this chapter the implementation of the test automation tools are described. These tools will support and automate the selected test strategy.

5. Evaluation of implemented test strategy
In this chapter the implemented test strategy and test automation tools are evaluated in order to determine whether they fulfil the requirements in chapter 1.4.

6. Conclusion and future work
In this chapter, the conclusions for this thesis are summarised. Possible future work is also discussed.
2 Theoretical background

This chapter includes some theoretical background about software testing, protocol analysers, the Scapy/LHC framework and Python. This will build a ground for the further discussion in this report.

2.1 Software testing

To be able to measure and guarantee some level of quality for a software system, some sort of testing has to be done. One problem with software testing is that it is possible to prove that bugs do exist but it is impossible to prove that bugs do not exist.

In software testing, a test case tests some functionality in the software is called a test case. It consists of input, output and order of execution. After a test case has been executed, the output can be analysed in order to decide whether result was correct. Take for example a function that takes an integer and returns true if it was even and false if it was odd. Then the input for a test case could be two for one test case and seven for another and the output could be true or false. The order of execution only matters if the output would be different if test case B is executed before test case A, instead of test case A is executed before test case B, or if test case B depends on test case A in some way. For example, if test case A writes something that test case B reads (Copeland, 2004).

If there are many test cases for a software system, they are often grouped together to one or more test suits. In this way, the test cases can be organised for different parts of the software system.

2.1.1 Testing levels

Software can be tested at many different testing levels. The testing levels are often performed within different steps in the development process. The way these levels are defined depends on the development process that is used. One common way to define the different levels is as unit testing, integration testing, system testing and acceptance testing (Copeland, 2004).

Unit testing

Unit testing is performed on individual units of the software. The goal is to find defects in code within these units. A unit means different things in different programming languages. In a procedural language, like the C language, a unit is a function, but in an object-oriented language, like Java, a unit is a class. The developer for a unit is often responsible for unit testing of that unit (Sommerville, 2004).

Integration testing

In integration testing, a subsystem is built from two or more units. The subsystem is then tested so that the units are functioning together. A unit can function in isolation but fail when interacting with other units. This can for example depend on interfaces are not being used correctly. The goal of integration testing is to find defects in the design (Sommerville, 2004).

System testing

In system testing, all subsystems that belong to the software and should be delivered with the software are integrated. The system is then tested so that all functionality works as expected. This will find defects in the functional specification (Sommerville, 2004).
Acceptance testing

Acceptance testing means that the software is tested against the requirements. When this test is completed successfully, the customer will accept the software (Copeland, 2004).

2.1.2 Regression testing

Regression testing is done in order to verify that no regression bugs have been introduced when the software is modified. This can happen when a bug is masked by another faulty function. When the other function is corrected, the masked bug will be exposed. Regression bugs can also be introduced when an old function is updated incorrectly. Regression testing should therefore be done every time the software is modified. This is done by running a test case against the software and then compares the output with the output from the same test case run against a previous version of the software. If the output is the same, the test has been completed successfully (MSDN, 2008).

An effective way to do regression testing is to build a library with test cases (MSDN, 2008). When the software is modified, all the test cases in the library should be run, and the output from the test cases and the output from the same test cases run against the previous version should then be compared against each other. This can also easily be automated to speed up the regression testing process.

2.1.3 Test automation

Test automation is a well known technique to speed up the testing process. The testing process that is most commonly automated is regression testing. Because in regression testing, the same test cases are run repeatedly. This very time consuming and repetitive work is well suited for automation.

When building a test automation tool, there are some things to think about in order to get a tool that is doing the right thing and that many people like to use. The first thing is to define the goals and the requirements for the automation project. This in order to know which type of test automation tool that is needed and which expectations there are on the tool. There are also some guidelines to follow in order to avoid some errors when the tool is designed and implemented (Pettichord, 2001).

- Reviewability

  The users must be able to understand what the test tool does. It is otherwise easily happened that the users begin to think that the tool does more then it actually does. Users then trusts the tool so much that they think it always provides the right answer. This in turn can lead to bugs not being found. In order to get reviewability, good documentation is needed but also well-structured code so a new developer easily can take over when the tool needs more or updated functionality (Pettichord, 2001).

- Maintainability

  It is common that a software system’s interfaces are changing during the development. It is therefore important that the test cases are easy to change so they can be run against the new version of the software. This can be done in many different ways. The challenge is to build a good abstraction layer so changes only need to done on some few places. One way to do this is to build a library that all test cases use. If the software changes the only thing that needs to
be changed to be able to run the test cases, is the library. It is also important that the tool is flexible, and not dependent on one particular application, so it can be used even in future projects (Pettichord, 2001).

- **Integrity**

  The user must be able to trust the output from the test tool. If the tool generates too many warnings about possible bugs when actually there are no bugs, i.e., false positives, the user loses trust in the tool and the tool becomes meaningless. On the other hand, if the tool does not report bugs when there actually are bugs, i.e., false negatives, the bugs will then be discovered later, maybe by the customer. In this case too the tool will not be trusted and most likely not used. To be able to get integrity, there must be a balance between false positives and false negatives. The tool must be sensitive enough to be able to find bugs, but not too sensitive. Another way to get better integrity is to improve the tool’s bug reports. This makes it easier for the user to decide if the warning was a false positive or a real bug (Pettichord, 2001).

- **Independence**

  When testing is done manually, the tester usually reuses components that were created for an earlier test case. This is okay because the tester can see if something seems to be strange. But the result will not be that good if the testing process is automated and the same strategy is used. If one test case fails, no test cases that depend on it will be able to run. It is therefore important that every test case is independent and can be run independently (Pettichord, 2001).

- **Repeatability**

  It is important that the test cases work the same way every time they run. If a test case is run and finds a bug, when the bug must be found next time the test case is run. Otherwise, it will be much harder for the developers to fix the bug (Pettichord, 2001).

In software testing, there is in general two main things that can be automated, the execution of test cases and the verification of test results.

### 2.1.4 Test automation architecture

When designing the architecture of a test automation tool, it is important to understand the goals for the tool. Which functionality needs to be included, and which tasks need to be automated. This is in order to get a tool that is as flexible, easy to use and easy to maintain as possible. A common type of automation architecture is to build a testing automation framework. A software test automation framework that consists of an integrated set of tools to supports the testing process is called a software testing workbench. A software testing workbench may include tools to support many different tasks. Some examples of tools that might be in a testing workbench are shown in Figure 1 (Sommerville, 2004).
Figure 1: Tools in a software-testing workbench

- **Test manager**
  The test manager runs the test cases with some inputs and collects the result from each test case. It also keeps track of the expected results from the test cases (Sommerville, 2004).

- **File comparator**
  In regression testing, a file comparator is used to compare the output from a test case with the output from the same test case run against a previous version of the software. The output from the previous version is in this case the test prediction, also called a “gold file”, in Figure 1. “Gold files” are described later in this chapter. The file comparator then reports if the files are equal or not (Sommerville, 2004).

- **Report generator**
  The report generator collects the results from the test cases and generates a test report there the user can, for example, see if a test case has passed or failed (Sommerville, 2004).

There are also some common architectures to support the goals and guidelines for the test automation tool.

- **Libraries**
  Libraries are often used to get an abstraction layer between the test cases and the software under test. All test cases can use the library, and when changes are made in the software interfaces, only the library has to be changed. Code pieces that are used very often in test cases can also be included in a library to make the test cases simpler and easier to write. There are also disadvantages with libraries, specially if the libraries are very big and complex. If something goes wrong with a test case, it is hard to know if the error lies in the test case or in the library. It can also be harder to understand the test cases because the reader needs to know the semantics of the libraries. It is therefore important to have relatively small, well designed and well documented libraries (Pettichord, 2001).
• **Data-driven tests**

Data-driven tests allow tests to be written in a table format. This makes it easier to write and read the test cases. This technique is useful when the test case writers do not have much experience in programming. Data-driven tests require that a parser is written to interpret the test cases. That is, a new small language needs to be developed, which can be time consuming. It can therefore be a good choice to use an existing language (Pettichord, 2001).

• **Heuristic verification**

One way to verify the test result is to run the test case and save the output to a file. The file is then analysed manually. If it seems to be correct, it is classed as a “gold file”. Next time the test case is run the output is compared with the “gold file”. If they are equal, the test case has passed. The disadvantage with this approach is that it will report every change in the output as an error, even if it is just a small change in some message that which is really an error. It can therefore be good to just verify the interesting part of the output (Pettichord, 2001).

The important thing when developing a test automation tool is to consider it as a normal software developing project. That is, goals and requirements must be specified (Pettichord, 2001).

2.2 **Protocol analyser**

When tests are done on devices that belong to a network, the tester must be able to see which packets have been sent, and to whom. This is in order to detect errors and to do measurements on, for example, the time it took to deliver a package. To do this, the tester uses a protocol analyser. To understand what a protocol analyser does, some basic knowledge about protocols and protocol stacks are needed.

2.2.1 **Protocols**

A network communication protocol is a set of rules that makes it possible for devices on a network to understand each other (WildPackets Academy, 2003). The sender constructs the message according to the rules and sends it to the receiver. The receiver knows the protocol rules and is therefore able to understand the message. This type of message is called a packet. For example, a web browser uses HTTP, HyperText Transfer Protocol, to communicate with a web server. When the user wants to load a web page, the web browser sends a packet, which contains “GET” followed by the address of the web page, to the web server. The web server responds by sending the web page to the web browser.

2.2.2 **Protocol stacks**

The protocols in a network device is organised in different layers, this is called a protocol stack. The protocol stack defines how protocols on different layers are connected to each other. When a device wants to send a message to another device on the network, the sender first constructs a packet on the highest level. The packet is then sent to the next layer in the stack at which more information is added. This continues until the lowest layer, the physical layer, is reached. The packet is then sent as a string of bits on the physical medium to the receiver. The receiver receives the packet on the lowest layer in the protocol stack, unpacks it, and sends it to the next layer. When the packet reaches the highest layer, the original message is extracted.
Take for example the web browser and the web server from section 2.2.1. First is the HTTP “GET” packet with the web page address is constructed. This layer is called the application layer. The packet is then sent to the transport layer where TCP is used. Here, information such as a port number, to know which application this packet belongs to, and a sequence number, to know in which order to handle this packet, is added. The TCP packet, with the HTTP packet inside, is then sent to the next layer. The next layer is the network layer where the IP protocol is used. Here, for example the IP-address for the sender and the receiver is added. The IP packet is then sent to the link layer where more information is added, for example, the MAC-addresses for the sender and the receiver. An Ethernet packet is constructed with the IP packet as payload, see Figure 2.

Figure 2: The package construction, from the application layer to the link layer

The packet has now reached the physical layer where it is translated to a binary bit stream on the cable. The web server receives the bit stream at the physical layer where it is translated back to an Ethernet packet and is forwarded to the link layer. Here is the IP package extracted from the Ethernet packet and is sent to the network layer. At each layer the information, added earlier from that layer, is removed, and the rest is sent to the next higher layer. When the packet reaches the application layer, the web server receives the original HTTP “GET” package with the web page address. This protocol stack example is illustrated in Figure 3. The figure shows the name for each layer and the protocols used in this example.
There exist many protocols for each layer and the protocols shown here are just some examples. The layer structure makes the software very dynamic. If, for example, a wireless network replaces the physical layer, the Ethernet cable, only the link layer needs to use another protocol. No modification needs to be done in the other layers.

### 2.2.3 How a protocol analyser works

A protocol analyser is a software program that is running on a regular computer. The program reads all packets that are sent on the network. Each packet is decoded and all information from each layer in the packet is presented in a human readable format. That is, the user of the protocol analyser is able to get the information stored in every field in every protocol used in the different layers. For example, the IP-address of the sender and the receiver from the network layer and sequence number from the transport layer. All information from the network traffic can be stored in log files for later analysis (WildPackets Academy, 2003).

Protocol analysers can also give statistical information about the network traffic. For example, number of packets per second and number of packets with errors. It is also often possible to filter the network traffic so only some packets are read. For example, read only HTTP packets from a specific sender (WildPackets Academy, 2003).

### 2.3 The Scapy/LHC framework

In this chapter the Scapy/LHC framework is described. All information is retrieved from the Scapy/LHC framework webpage on the Ericsson internal network.

The Scapy/LHC framework is a protocol-decoding tool implemented in the programming language Python. It is able to read many types of log files from different protocol analysers and provides functionality to retrieve packets from selected protocols. The user can easily write their own scripts to retrieve the information they need from the log files by using the Scapy/LHC framework. The main users of the Scapy/LHC framework are testers that do tests on the network, but the framework is available for all employees within Ericsson. The test activities often result in many log files that need to be analysed. To do this, the user uses the Scapy/LHC framework to write Python scripts that retrieve the information they need from the log files.
2.3.1 Architecture

The Scapy/LHC framework consists of different types of modules. The modules can be divided into four main types:

- **Framework system modules**
  These modules are the core of the framework. They contain file readers that can read different types of log files, Internet protocols like the IP protocol and protocol stacks. Some modules that contain functions for setting up the framework environment are also included.

- **Protocol stacks modules**
  These modules define how different messages in different protocols are connected. For example, how a message in a lower level protocol connects to a message in a higher-level protocol.

- **Protocol modules**
  A protocol consists of a collection of messages and rules that are used for communication on some layer in the protocol stack. A protocol module defines which messages that are used in that protocol.

- **Support modules**
  These modules contain several support tools like average and variance calculation.

On top of the four types of modules, there are the scripts, created by the users to retrieve data from the log files. The Scapy/LHC framework architecture is illustrated in Figure 4.

---

Figure 4: The Scapy/LHC framework architecture
The big advantage with the Scapy/LHC framework is that it is very flexible. If the framework needs support for a new type of log files, a new file reader can be written to decode the new file format. To get support for a new protocol, a new protocol module and a protocol stack module can be written. This makes it very easy to extend the framework and make it useful even in the future.

2.3.2 Development

The Scapy/LHC framework is developed as open source within Ericsson. There is no one in charge of the framework; instead, the users do the development when they need more functionality to be able to write their scripts. The most common type of change in the framework is support for a new protocol, added by defining new protocols and protocol stacks, or changes to existing protocols. Because of this, the developing process is very iterative. That is, the Scapy/LHC framework is extended with new functionality when there is need for it and is therefore always developed. All development of the framework is done outside normal working tasks so the time to spend on the framework is very limited.

2.3.3 Testing

Because of very limited resources, there are almost no test activities for the Scapy/LHC framework today, and there is no formal test strategy. The defects that are discovered are discovered by developers when implementing new functionality in the framework or by users when the framework is used.

2.4 Python

Python is an interpreted programming language implemented in the programming language C. The Python code runs on a virtual machine that is available for all major operating systems. A program that is written in Python can therefore run on many different platforms without being rewritten or recompiled. Python is distributed under open source, versions for new platforms is therefore always developed. Everything from small scripts to full-scale completely object-oriented applications can be written in Python. Python is a dynamic language, which means that data types are not declared explicitly. Instead, a variable with appropriate data type is created when the variable is assigned for the first time. The built-in data types are Numbers, Strings, Lists, Dictionaries, Tuples and Files. The memory management is completely handled by Python with the help of a garbage collector. This means that memory does not need to be created or deleted manually for variables. To test a piece of code in Python, it can just be entered in the Python interpreter application and every line of code is executed in real-time. (Python Software Foundation, 2008)

Many standard libraries come with the Python installation. These libraries cover a broad range of areas, for example a GUI toolkit, a unit testing framework, one XML parser and a lot more. There are also many third part libraries that do not come with the standard installation of Python but can be installed. Below are the libraries for GUI development and the unit-testing framework are described. (Python Software Foundation, 2008)
2.4.1 GUI development

For GUI development, Python comes with the Tkinter (Tk interface) module, which is the standard Python interface to the Tk GUI toolkit for Tcl. Tkinter is a GUI toolkit that offers native look and feel for all major platforms. The GUI toolkit makes it easy to create standard widgets on an application window, for example menus button and text areas. Event handlers can be connected to a widget, for example the button, so that some code is executed when the button is clicked. (PythonWare, 1999)

There are also many other GUI toolkits for Python that is not included in the standard installation. Some examples are wxPython based on wxWidgets, PyGtk based on Gtk, and PyQt based on the Qt application development framework. (Python Software Foundation, 2008)

2.4.2 Unit testing framework

The unit-testing framework that comes with Python is called PyUnit and is a Python version of JUnit, which is a unit-testing framework for the Java programming language. A class called `TestCase` in PyUnit represents a test case. Test cases can also be grouped together to a test suit. A test runner can run all test cases within a test suit automatically. PyUnit comes with two test runners, the `TextTestRunner` and the `GUI test runner`. The `TextTestRunner` reports the result from each test case in textual form to, for example, the Python interpreter session. With the GUI test runner, the test cases are run, and the result is reported, in a graphical user interface. (Purcell, 2005)
3 Evaluation of test strategies

In this chapter possible test strategies for the Scapy/LHC framework are evaluated. Two test automation tools that can be used to support and to automate these test strategies have been considered. Finally, one test strategy and the test automation tools are selected for implementation.

3.1 Requirements on the test strategy

Except for the requirements in chapter 1.4, the test strategy also needs to have some other properties to be appropriate to use with the Scapy/LHC framework. These requirements on the test strategy will be used to filter out test strategies that are possible to use on the Scapy/LHC framework.

The Scapy/LHC framework is changed and extended by many users all around the company when there is need for more functionality. It is therefore likely that regression bugs are introduced in the framework. To be able to test that old and new functionality work as expected, the test strategy needs to have support for regression testing, see chapter 2.1.2. The development of the Scapy/LHC framework is done outside normal working tasks. Hence, the time needed for the testing process and the number of human interactions needs to be as little as possible. The test strategy must therefore make it possible to automate the execution of test cases and the verification of test results.

The testing activities are intended to be carried out by the developers that write scripts, based on the Scapy/LHC framework, in Python. To make it easier for the developers, they should also be able to write the test cases in Python. It would also be advantageous if the test automation tool, and possible libraries, were written in Python. This would make it easier for the developers to understand and extend the tool, because the only language they need to know is Python. It will also make the tool operating system independent. The only thing needed on the computers to test the framework is Python, which is already needed by the Scapy/LHC framework. The last requirement is that the test automation tool must be free because there is no budget for the Scapy/LHC framework project.

The requirements on the test strategy and the test automation tool, except for the requirements in chapter 1.4, are summarised in the following list.

- Support for regression testing
- Automated execution and verification of test cases
- It shall be possible to write test cases in Python
- The test automation tool, and possible libraries, shall be written in Python
- The test automation tool must be free
3.2 Automatic tests for the Scapy/LHC framework

The first thing needed to be considered, to be able to fulfill the requirements in section 3.1, is the possibility to automate the testing process for the Scapy/LHC framework. It was shown, when the Scapy/LHC frameworks and Python was studied, that it is easy to run a function with some input, collect the return values, compare the values and to print them to a file or to the screen. It is also easy to run a Python script from another Python script and to open and compare files. This makes it possible to automatically execute test cases and then compare the output with an expected value with a test automation tool. The conclusion is that it is possible to automate tests for the Scapy/LHC framework.

3.3 Test strategies

After the literature study, the conclusion is that there are two main test strategies that can be used to test the Scapy/LHC framework. They require different amount of time and different knowledge about Scapy/LHC framework from the test case writers. They also have different degree of code coverage. In what follows the two test strategies are described and evaluated.

3.3.1 Using high-level test scripts

The reason to extend the Scapy/LHC framework is often that the user wants to write a script based on the Scapy/LHC framework but the framework does not have the wanted functionality. For example, the framework does not support decoding of some protocol. After the new functionality has been implemented, or simultaneously when implementing, the user also writes the script to check if the problem has been solved. These scripts also, even if they are not intended to, test many parts of the Scapy/LHC framework. These scripts print their output to the screen or to a file so the result can be analysed. The idea behind this strategy is to use these scripts.

After the new functionality and the script have been implemented, the user runs the script with some log file as input. The script has to print its output to a file. The output file is then manually analysed by the user. If everything seems to work as expected the file is classified as a “gold file”. If something seems to be wrong in the output file the script or the new implementation in the Scapy/LHC framework needs to be corrected. All these scripts and “gold files” are collected to a test suit there every script is a test case. How to create a new test case and add it to the test suit is illustrated in Figure 5.
Every time something has been changed in the Scapy/LHC framework all scripts in the test suit are run, i.e. regression testing. The output file is then compared against the “gold file” for that script. If the files are equal the Scapy/LHC framework has passed that test case, otherwise the difference between the output file and the “gold file” needs to be analysed in order to see if there are any errors in the Scapy/LHC framework. Figure 6 shows what is happening when a test case is run. This is done for every script in the test suit. It is important that a script is run with the same log file as input as when the “gold file” was created. Otherwise, the output from the script will be different when compared to the “gold file”. If the input log file or the output from the script is changed, the “gold file” also needs to be recreated in order to get a correct result from the comparison.
The advantage with this test strategy is that the person writing the test cases does not need to know so much about the Scapy/LHC framework. It is enough to know how to use it, for example know how to write a script based on the framework. The time needed to write the test cases is very small because the scripts are written anyway when the user want to use the new functionality. Only some extra time for adding the new test case to the test suit is needed. There are no requirements added to the scripts except that the output must be written to a file. The scripts are written in Python so the user does not need to learn a new programming language. It is also easy to automate this strategy with a test automation tool. These properties imply that the requirements in chapter 3.1 can be fulfilled.

This test strategy also has some disadvantages. The scripts will probably not give full code coverage. That is, there will probably still be code in the Scapy/LHC framework that is not tested. In this case, this is not a big issue because the code that is not tested by the scripts is probably not used very often, and is therefore less likely to be changed by the users than other parts of the framework. This will also make this code less error prone than the code that is changed often.

The approach to compare the output from the script with a “gold file” is very sensitive for changes in the output. For example, if there is just some small message that has been changed in the script, the comparator will give a warning, even if there is actually no bug. This can also be a positive thing because the comparator does not miss any bugs that change the output from the script.

The comparing process is also dependent on that same log file being used as input to the script every time it is run. Otherwise, the output from the script may be different compared to the “gold file”. It is therefore necessary to organise and save log files used as input, scripts and “gold files”. The output from the scripts also needs to be saved temporarily for the comparison and for analysis of possible differences.

**3.3.2 Using unit tests on separate functions**

This test strategy is based on traditional unit testing. A test case is written so that a function or a class is called with some parameters and the output is then compared against an expected value. If they are equal, the Scapy/LHC framework has passed the test case. If they are not equal, there can be a possible bug, so the output needs to be analysed. These test cases are then run every time
some change has been made in the Scapy/LHC framework. That is, regression testing. How a test case is run is illustrated in Figure 7.

![Figure 7: Run a test case](image)

An advantage with this test strategy is that full code coverage can be obtained, but this requires that many test cases are written to be able to test every single function, with every possible parameter, in the Scapy/LHC framework. This can be very time consuming. On the other hand, when this is done once, a full code converge regression test can easily be done every time the framework is changed, and the only thing that needs to be added is test cases for the new parts in the framework. Some other advantages with this strategy are that the only files that need to be organised are the test case files. No other input or output files are needed because all data can be saved in the test cases. The comparisons are then done inside the test cases. This test strategy allows test cases to be written in Python and it is also easy to automate execution and verification of test cases with a test automation tool. These properties fulfil the requirements in section 3.1.

The disadvantages with this test strategy are that the test case writers need to have good knowledge about how the Scapy/LHC framework works, for example which parameters a function can take and what values it can return. It also requires a lot of test cases to test the Scapy/LHC framework effectively.

If the test cases are made so that the output from the functions is written to files, there is nothing that stops that the strategy described in section 3.3.1 also uses this strategy on some test cases.

### 3.4 Test automation tools

A test automation tool is needed in order to execute of test cases and verify of test result effectively. The requirement in section 3.1 makes two alternatives available. The first alternative is to use the unit-testing framework PyUnit, which comes with Python, together with TestOOB, which is an extension to PyUnit. The second alternative is to develop and implement a new test automation tool. The decision ones the choice of automating tools is independent from the choice of test strategy. The two alternatives are described and evaluated in the following sections.
3.4.1 Using PyUnit together with TestOOB

As described in section 2.4.2, PyUnit is a unit testing framework that comes with Python. PyUnit is written in Python and it provides a test case class called `TestCase`, which can contain one or more tests. It also provides a test suit class called `TestSuit` that groups together several test cases. All test cases can automatically be run by running the test suit. TestOOB is an extension to PyUnit and provides some extra functionality. For example, it can print the test result with nicer formatting than PyUnit, and is able to print a test report in HTML format. The HTML report requires that an additional tool called 4Suit is installed. The idea is to use these tools to create a test automation tool.

A test case is developed in the following way. A new Python file is first created and a Python test case class is created within it. This class must inherit from the PyUnit class `TestCase`. An example test case is shown below.

```python
import unittest
class ExampleTestCase(unittest.TestCase):
    def setUp(self):
        # Initialise things needed during the test case.
    def tearDown(self):
        # Clean up after the test case has been executed.
    def testSomething(self):
        # This is the test, for example:
        result = function(input)
        assert result == "Expected result", "Some message"
if __name__ == "__main__":
    import testoob
    testoob.main()
```

A test case class contains three types of functions. The setup function is run before the tests are executed. Here, can data structures needed during the test case can be created and initialised. The teardown function is always run after the test case has been executed, even if something has gone wrong with the tests. This allows to clean up and remove things created during the test case. The last type of function is the test functions. Here one test function called `testSomething` is created. A test case class can contain as many test functions as needed. All test functions are executed when the test case is run. In the test function a function called `assert` is used to compare the test result with an expected result. If they are not equal, an exception is raised, which is taken care of by the PyUnit testing framework. The second argument to `assert` is a message that is printed if the test fails. After the tests have been executed, PyUnit reports for each test if it has failed or passed. The last three lines make it possible to run the test case individually from the command line by using TestOOB.

All these test case classes can then be grouped together into a test suit. This is done by using PyUnits test suit class, which is shown below.
testSuit = unittest.TestSuit()

```

testSuit.addTest(ExampleTestCase("testSomething"))
```

testSuit.addTest(ExampleTestCase("testSomethingElse"))

In this example a test suit and two tests are created, *testSomething* and *testSomethingElse*. The *ExampleTestCase* is then added to the suit. If all test functions start with a common pattern, for example *test*, another function called *makeSuit* can be used.

testSuit = unittest.makeSuit(ExampleTestCase, "test")

TestOOB can be used to automatically run all test cases in a test suit by calling its *main* method. To get detailed test results and use TestOOB to create a test report in HTML format, the method is called in the following way:

```

import testoob
testoob.main(testSuit, None, html="testReport.html", immediate=True, vassert=True)
```

Here are all test cases in the test suit called *testSuit* executed, the test report saved in the file *testReport.html*, and the test result from each test case reported immediately and in a verbose way. The output from TestOOB when a test case is run with these parameters is shown in Figure 8.

Three tests have been run, the first and the third test have passed but the second test has failed.

![Figure 8: Test result when a test case is run with TestOOB](image)

In this case TestOOB will also make a test report in HTML format. For this test case the test report looks like the HTML page in Figure 9.
It is easy to automatically execute all tests when they are in a single Python file, but to reduce the
time needed to add a new Python file with a test case, an automatic and dynamic test execution
script needs to be developed. The test execution script should search a test case directory for
available test cases and execute each test case. A new test case can then just be added to the test
case directory and is then executed when the test execution script is run.

If this test automation tool is used together with the high level script test strategy described in
chapter 3.3.1 the script code can be placed directly in the test function, or the script can be placed
in a separate Python file and the script can then be called from the test function with the Python
execfile function. The output from the script and the “gold file” is compared in the test function
after the script has been executed. To make the test functions simpler to write, a comparison
library can be developed to compare the two files and report the differences, if there are any, to a
log file. The library can then be used in every test function instead of implementing file
comparison over and over again.

The advantages with this test automation tool are that much of the automation functionality is
already implemented in PyUnit. PyUnit also has built-in functionality for comparing results with
expected results, and takes care if they are equal or not. By using TestOOB the result from each
test case is presented in a well formatted way. It is also easy to get a test report in HTML format
where test results can be reviewed later. The automated test execution script and the comparison
library can also easily be written in Python and then be used over and over again.

The disadvantages with this test automation tool are that the code needed to compare the test
result and the “gold file” needs to be added to each test case, if “gold files” are used, instead of
doing the comparison centrally in the test automation tool. But this code will just be a few lines if
a comparison library is used. The functionality is also to some degree limited to that provided by
PyUnit and TestOOB, because it could be a little bit tricky to extend these frameworks with new
functionality. Another disadvantage is that the test case writer needs to have knowledge about the
PyUnit test framework to be able to write a test case. The test case writer also always has to
implement a PyUnit test case class for each test case, even if the test strategy with high-level test
scripts, see section 3.3.1, is used.
The properties for this test automation tool described above satisfy the requirements in section 3.1 if used with any of the test strategies described in section 3.3.1 or section 3.3.2.

### 3.4.2 Develop and implement a new test automation tool

A new test automation tool can be implemented relatively easily in Python. By using the Tkinter GUI toolkit, described in chapter 2.4.1, a user-friendly graphical user interface to the test automation tool can be developed. The test automation tool will consist of five main parts, GUI, test runner, verifier, database handler and report generator. This architecture is illustrated in Figure 10.

![Test automation tool architecture](image)

**Figure 10: Test automation tool architecture**

The GUI provides all the functionality to the user. Here a new test suit can be created, saved and loaded. Test cases can be added and removed from a test suit and all, or just some, test cases in a test suit can be run. Dependent on which of the test strategies in section 3.3 that has been chosen, it may be possible to add an output file or a “gold file” for each test case. The GUI will also present the result of each test case, for example whether it failed or passed.

The test runner loads, with help from the database handler, the test cases in a test suit and executes each test case. It also keeps track of the results from the test cases and the expected results. The database handler stores and loads all information about a test suit. The information is stored in a database, for example an XML file. When a test case has been executed the verifier compares the output from the test case with the expected value and reports if there are any differences. If “gold files” are used the verifier can create a log file there the lines that are not equal in the output file and the “gold file” will be logged. After all test cases have been executed the report generator writes a test report in which the results from the test cases can be reviewed.

The advantages with this test automation tool is that the code needed to compare the output from a test case with the expected output can be placed centrally in the tool instead of in each test case. If the high-level test script strategy described in section 3.3.1 is chosen this means that the only code needed for a test case is the script. That is, the time needed to add an already written script as a test case will be very small and the test case writer does not need to now any special test library to write a test case. The output file and the “gold file” are configured via the GUI. If the
test strategy described in section 3.3.2 is chosen a similar solution can be made. It is also possible to add exactly the functionality that is needed, and the level of configuration possibilities that are wanted.

The disadvantage is that there is nothing that is already implemented. That is, all functionality that is needed must be designed and implemented, which requires some time.

With the properties described above this test automation tool will fulfill the requirements in section 3.1 with any of the test strategies in section 3.3.

3.5 Selection of test strategy
After the two test strategies have been analysed, advantages and disadvantages for the test strategies were compared against each other. Because simplicity and the time needed to write test case was prioritised, is was decided that the test strategy using high-level test scripts, described in chapter 3.3.1, was the most appropriate test strategy for the Scapy/LHC framework. The main reason was that the only thing needed for a test case is the script based on the Scapy/LHC framework. This minimises the time and the work needed for each test case and makes it simpler for the test case writers.

If instead the test strategy based on unit testing, see section 3.3.2, hade been chosen, full code coverage could be obtained. However, the requirement on time efficiency was more important in this case.

3.6 Selection of test automation tool
To make it easier to compare the test automation tools and to decide which test automation tool that is most appropriate to use when testing the Scapy/LHC framework, it was decided that both tools described in section 3.4 are implemented and compared.
4 Implementation of test automation tools

This chapter describes the implementation of the test automation tools presented in section 3.4. The tools are developed to support the test strategy using high-level test script, described in section 3.3.1, which was the test strategy selected in section 3.5.

4.1 Choice of programming language and development environment

Because of the requirements in section 3.1, I decided to implement the tools and the help libraries in the programming language Python. The users already write scripts, based on the Scapy/LHC framework, in Python, so this makes it easier for the users to change and extend the tool. Python is also a high-level object oriented language, which speeds up the implementation process and makes it easier in comparison to other low-level programming languages.

There are many development environments for Python. I have tested a few of them and decided to use Wing IDE from Wingware. This environment has good support for debugging and project handling. It also has an integrated Python shell in which code can be run directly. These features were not found in the other development environments tested.

4.2 Implementing a tool using PyUnit together with TestOOB

In this section is the implementation of a test automation tool that is based on PyUnit together with TestOOB described.

The main parts of this test automation tool are already implemented in PyUnit, for example automatic test case execution and some simple verification functions. As discussed in section 3.4.1, there are also modules that can be implemented to make it easier for the user to add and execute test cases. I have therefore implemented a file comparator and a test case finder. These are described in section 4.2.1 and 4.2.2.

The directory for the tool contains two directories, TestCases and TestReports. The TestCases directory contains one directory for each test case. Each of these directories contains the test case file, the script file for the test case, the input file for the script, the output file, the “gold file” and the difference log for a test case stored. The TestReports directory contains a test report in HTML format for each test session. This is illustrated in Figure 11.
4.2.1 Comparator

The comparator does two things. It compares the output file, generated by the script the last time it was run, with the “gold file” after the script in the test case has been executed and it writes a difference log file if the files where not equal. The comparison is done line by line and if the lines where not equal, the line number, the line from the output file and the line from the “gold file” is stored. After the whole output file has been compared to the “gold file”, the information about whether they where equal or not is returned. If the files where not equal, the information stored earlier when the differences where found is written to a log file. If the files where equal, the test case has succeeded and no log file needs to be written. The log file is a pure text file and is saved in the directory for each test case in the test case directory. Only the log file for the latest run of each test case is saved. The comparator can be used in the test cases by calling the compare method with the filenames for the output file and the “gold files” as input after the script file for the test case has been executed. This saves many lines of code compared to the alternative in which the files are compared and log files written in every test case.

4.2.2 Test case finder

The test case finder was implemented to make it easier for the user to run all test cases in the test case directory. When the user runs the test automation tool the test case finder seeks the test case directory for test cases and saves them in a list. The filename for a test case must end with _test.py to make it possible for the test case finder to find the test case. A PyUnit test suit is then created with all test cases found. The test case finder also generates a filename for the test report in HTML format. The test suit and the filename are given to TestOOB, which executes all the test cases and generates the HTML test report, which is saved in the test report directory as "year"-
After a test case has been executed, the test result for that test case is presented on the screen.

4.2.3 Use cases

Here follow two user cases to demonstrate how the test automation tool can be used.

Add a new test case:

To add the test case *GAN_Test*, the user creates a new directory called *GAN_Test* for the test case inside the test case directory. *GAN* stands for *Generic Access Network*. Inside that directory, the user creates a new Python module, called *GAN_Test*.py. The script file for the new test case, the input file to the script and the “gold file” are also placed in that directory. In the new module, the user imports the *os*, *unittest* and *comparator* modules. The user then creates a class, called *GAN_Test*, which inherits from the PyUnit test case class. Inside the class, the user creates three methods, *setUp*, *tearDown* and *testGAN_test*. In the *setUp* method the current directory is changed to the directory for this test case, the *GAN_Test* directory, and a *comparator* is created. In the *tearDown* method the current directory is changed back to the test case directory. The *testGAN_test* method is the method where the script is executed and the output file, generated by the script, is compared against the “gold file”. The user does this by calling the Python method *execfile* with the filename for the script, which executed the script. After the script has been executed, the user calls the *comparators compare* method with the filenames for the output file and the “gold file” as inputs. The *compare* method returns if the files were equal or not. The last thing the user does in the *testGAN_test* method is to call the *assert* method with the result from the comparator and a text to print if the files were not equal. Outside the class definition in the *GAN_Test* module, the user creates a *suit* method, which returns PyUnit test suit with all test functions in the *GAN_Test* class beginning with *test* as test cases. This method makes it possible for the test case finder to collect all test cases.

Here is the Python module *GAN_Test*.py that was written by the user.

```python
import os
import unittest
from comparator import *

class GAN_Test(unittest.TestCase):
    def setUp(self):
        os.chdir("GAN_Test")
        self.cmp = Comparator()
    def tearDown(self):
        os.chdir(os.pardir)
    def testGAN_test(self):
        execfile("GAN_Test.py")
        result = self.cmp.compare("GAN_Test_output.txt", "GAN_Test_gold.txt")
        assert result["equal"], "Not equal. See diff.log for more info."
```
def suit():
    return unittest.makeSuite(GAN_Test, 'test')

Run all test cases:

The user runs all the test cases in the test case directory by entering testlhc.py in the command line for the operating system. The tool starts to execute the test cases and after a test case has been executed, the test result for that test case is displayed. When all test cases have been executed, statistics about how many test cases were run, how long time it took and how many test cases failed are shown, see Figure 12.

![Figure 12: Statistics after all test cases have been run](image)

In this case has the test case GAN_Test_test failed. The user therefore goes to the GAN_Test directory to analyse difference log, GAN_Test/diff.log. There the user can see the line number of the line there the output file and the “gold file” were not equal, the content of the line in the output file and the line in the “gold file”, see Figure 13. Every line that is not equal in the output file and the “gold file” needs to be investigated by the user to see if there are any errors.

![Figure 13: The difference log](image)

The user has no more time to investigate right now and closes the tool. Later the user comes back and opens the HTML test report in the test report directory to get an overview of the latest test session. Here can the user see which test cases have failed and which test cases have executed successfully, see Figure 14. The user opens the file GAN_Test/diff.log in the test case directory and continues the investigation.
Figure 14: The test report in HTML format

4.3 Implementation of a new test automation tool

This chapter describes the implementation of the second test automation tool, as discussed in section 3.4.2.

4.3.1 System overview

The second test automation tool consist of five main parts, verifier, database handler, test runner and graphical user interface, as described in section 3.4.2. In addition, A command line interface was also developed for an alternative way to run the test cases. This can be used if the user prefers to work in command line environment instead of a graphical interface, or if the graphical user interface runs too slowly on a specific computer.

In the directory for the tool three directories are located, TestCases, TestSuits and TestReports. The TestCases directory consists of one directory for each test case. In each of these directories the script file, the input file for the script, the output file, the “gold file” and the difference log for a test case are stored. In the TestSuits directory the test suits are saved as XML-files, and in the TestReports directory a test report from each test session is stored in HTML-format. This is illustrated in Figure 15.
4.3.2 Report generator

The report generator writes a test report in HTML format. The test report contains the result for each test case run at that time. The HTML page contains a table with four columns, the name of the test case, the elapsed time to run the test case, if it has succeeded or failed and possible information about the test case. It was decided to use the HTML format because it makes it easier to get a good layout compared to plain text, and it can be viewed in any terminal with a web browser, see Figure 16. The test report is saved as “test_suit”_”year”_”month”_”day”_”hour”_”minutes”_”seconds”_report.html in the test report directory. This is in order to avoid name collision if many test sessions are run.
31

Figure 16: The test report in HTML format generated by the report generator

4.3.3 Verifier

The verifier takes the output file, generated by the script the last time it was run, and compares it against the gold file for that script. The comparison is done line by line and if any differences are detected, the line number, the line from the output file and the line from the gold file are stored. After the whole output file has been compared to the gold file and the files were not equal, the verifier writes a log file with all information stored earlier when the differences were found. If the files were equal, the test case has succeeded and no log file needs to be written. The log file is a pure text file and is saved in the directory for each test case in the test case directory. Only the log file for the latest run of each test case is saved.

4.3.4 Database handler

The database handler loads and stores data about a test suit, for example which test cases the test suit contains. This data is stored in XML format in the test suit directory. The XML-file is built-up in the following way.

```xml
<testsuit>
  <testcases>
    <testcase>
      <filename>TestCase1/testCase1.py</filename>
      <outputfile>TestCase1/outputFile.txt</outputfile>
      <goldfile>TestCase1/goldFile.txt</goldfile>
    </testcase>
    <testcase>
      <filename>TestCase2/testCase2.py</filename>
      <outputfile>TestCase2/outputFile.txt</outputfile>
      <goldfile>TestCase2/goldFile.txt</goldfile>
    </testcase>
  </testcases>
</testsuit>
```

The XML-file is generated with the `mindom` module, which is an XML parser that comes with Python. This module is also used when a test suit is loaded from a XML-file. The advantages of
using XML instead of other database solutions is that no database system needs to be installed on the computer, and the XML-files can also directly be read and edited in clear text. This makes the tool less dependent on the environment it is running on. Another alternative was to develop a new file format specific for this tool, but the advantages with the XML-format is that the test suits in the XML-files are less dependent on this test automation tool, and it is possible to use them in other applications.

4.3.5 Test runner

The test runner is the kernel in this test automation tool. It handles all information about the test cases in the test suit currently loaded. If an XML-file is given, the test runner uses the database handler, described in section 4.3.4, to load all test cases in the test suit. The information stored for each test case is the filename of script to execute, the filename of output file, the filename of the “gold file”, the filename of the difference log, information about whether the test case has succeeded or failed and the time it took to execute the test case. The test runner is also responsible for the execution of selected test cases. After a test case has been executed, the test runner gives the filenames of the output file and the “gold file” for that test case to the verifier, described in chapter 4.3.3, to decide if the test case has succeeded or failed. When the test runner has executed all selected test cases it provides the test results to the report generator, described in section 4.3.2, which returns the file name of the test report.

4.3.6 Graphical user interface

To make it easier for the user to use the test automation tool, a GUI was developed. Tkinter was chosen as GUI toolkit, because it is the only GUI toolkit that comes with Python. This makes the test automation tool not dependent on any external package except for Python, which is the only software that needs to be installed to be able to use the tool. The goal was to make the GUI as simple and easy to use as possible with just necessary functionality.

The main features that were implemented are the following.

- Create a new test suit
- Open/save a test suit
- Add/remove test cases to/from a test suit
- Configure output file and “gold file” for each test case
- Run all or selected test cases in the test suit
- The test result is presented after each test case has been executed
- After all selected test cases have been executed statistics from the test results are presented. This includes whether all tests have succeeded or if some have failed.
- View the difference log for each failed test case within the GUI
- Access the test report in HTML format from the GUI

Two user cases to demonstrate how the tool can be used are presented below.
Open a test suit and add a new test case:

To start the test automation tool, the user runs the Python file *LHCTester.py*. The tool is opened with a new empty test suit. The user opens a test suit that is saved on the disk, by open the *File* menu and then chooses *Open*. A file dialog is shown on the screen, standing in the test suit directory, to allow the user to choose a test suit. The user chooses the file *testSuit.xml* and clicks the *Open* button. The test suit is now loaded and the test cases are shown in the test case list, see Figure 17.

![LHCTester – testSuit.xml](image)

Figure 17: The test automation tool with a test suit loaded

The user now adds another test case to this test suit by clicking on the *Add test case...* button. A file dialog is shown; standing in the test case directory, where the user can choose which script file to use for the new test case. The user goes to the *GAN_Test* directory, selects the *GAN_Test.py* script and clicks the *Open* button. The test case is now added to the test case list and is called *GAN_Test/GAN_Test.py*. The name is shown in gray, which means that no output file or “gold file” is selected and the test case can therefore not be run. The user selects the new test case in the test case list and right-clicks to open the popup menu and chooses *Edit test case*. The test case configuration dialog is shown on the screen, see Figure 18.
The user clicks on the Browse... button for the output file, selects `GAN_Test/GAN_Test_output.txt` from the file dialog, and clicks the Open button. The user then clicks on the Browse... button for the “gold file”, selects `GAN_Test/GAN_Test_gold.txt` from the file dialog, and clicks the Open button. To save the configuration and close the test case configuration dialog, the user clicks the OK button. The name of the new test case is now shown in black and is it now able to run.

Run all test cases in a test suit:

The user has opened the test suit `testSuit.xml` as described in the previous example, and all test cases in the test suit are listed in the test case list. The user now clicks on the Run all button and the test cases starts to execute. After a test case has been executed, the test result for that test case is shown in the test result area. When all test cases in the test suit have been executed, statistics from the test cases is shown in the bottom of the test result area. The user is told if all test cases have succeeded or if there are any failures, how many test cases run, the elapsed time, how many test cases succeeded, how many test cases failed and a link to the test report in HTML format. This time, the user is told that some test cases have failed and the user needs to investigate the possible error, see Figure 19.
The first test case, *GAN_Test/GAN_Test.py*, failed, therefore the user right-clicks on it in the test result area and chooses *Show difference log*. A new window is created and the difference log for that test case is shown. There the user can see the line number of the line where the output file and the "gold file" were not equal, the line in the output file and the line in the "gold file", see Figure 20. Every line that is not equal in the output file and the "gold file" needs to be investigated by the user to see if there are any errors.
The user has no more time to investigate right now and closes the tool. Later the user comes back and opens the HTML test report in the test report directory to get an overview of the latest test session. Here the user can see which test case that has failed and where the difference log is saved. The user opens the file $GAN_Test/diff.log$ in the test case directory and continues the investigation.

These were just some examples of how to use the main features in the test automation tool.

### 4.3.7 Command line interface

A Command Line Interface, or CLI, to the test automation tool was developed because in some situations it could be easier and faster to use. This could be, for example, if the GUI feels too slow on the computer or if the user prefers to work with command line interfaces. The main features in the GUI is also implemented in the CLI but if the user, for example, wants to change the name of the output file for an existing test case, the GUI is needed or the XML-file for the test suit could be edited manually. The features that are implemented in the CLI are described below. All commands are entered in the command line for the operating system.

- `LHCTesterL.py <test suit>`

  Runs all tests cases in the test suit given by the user. The test suit must be a XML file in the format specified in section 4.3.4. After a test case has been executed, the test result for that test case is presented. When all test cases in the test suit have been executed, statistics from the test cases is presented on the screen. The user is told if all test cases have succeeded or if there are any failures, how many test cases run, the elapsed time, how many test cases succeeded, how many test cases failed and the filename of the test report in HTML format, see Figure 21.
- LHCTesterL.py <test suit> -a | --add <script file>

This command adds a test case to the test suit specified. The option can be given in any of the formats, -a or --add, followed by the filename of the script to use in the new test case. The user then presses the enter-key and is asked to enter the filenames for the output file and the “gold file”, see Figure 22. The test case is then saved in the test suit. If no filename for the output file or the “gold file” was given, the test case will not be able to run until both filenames are set.

- LHCTesterL.py <test suit> -r | --remove <index>

This command removes a test case from a test suit. The option can be given in any of the formats, -r or --remove, followed by the index of the test case to remove. The index can be found by using the --show option described below.

- LHCTesterL.py <test suit> -s | --show

This command shows a list with the test cases in the test suit specified. The test cases are listed with an index that can be used if the user wants to remove the test case from the test suit. If no filename for the output file or the “gold file” is set, the text “Not runnable” is printed after the name of the test case. The option can be given in any of the formats –s or --show.

The above is just an overview of the features in the CLI.
4.4 Distribution of files

It is important to distribute the test automation tool and test cases in an efficient way. This reduces the time needed to test the Scapy/LHC framework. There is one main version of the Scapy/LHC framework but there are also some specialised versions that also need to be tested. Test cases must therefore be handled for different versions of the framework.

Today the version control system Subversion is used to handle updates to the Scapy/LHC framework. When a user wants to install the Scapy/LHC framework the user checks out the latest version from the Subversion server. If the user changes something in the framework, for example adds support for a new protocol, and wants it to be available as part of the framework, the user submits the changes to the Subversion server.

The easiest way to distribute the test automation tool and the test cases is to add them on the Subversion server. The root directory of the test automation tool is placed in the directory for the Scapy/LHC framework version that should be tested. That is, one copy of the test automation tool is placed in each version of the Scapy/LHC framework, see Figure 23. The test automation tool then automatically uses the right version of the framework when the test cases are run. Each copy of the test automation tool contains a TestCases directory, so each version of the Scapy/LHC framework can have different test cases. If a user wants to add a test case, the user adds the test case to the TestCases directory and submits the changes to the Subversion server.

---

**Figure 23: The test automation tool integrated in different versions of the Scapy/LHC framework**

No changes to the Scapy/LHC framework or new test cases should be submitted to the Subversion server until all test cases, old and new, for that version of the framework have been successfully executed without any failures. Test reports generated by the test automation tool should not be submitted to the Subversion server unless necessary, i.e. for sharing with colleagues, because they will fill up the TestReports directory and make it harder to find the right test report. They will also allocate unnecessary disk space.
5 Evaluation of implemented test strategy and tools

In this chapter, the method and the results from the evaluation of the implemented test strategy and test automation tools are presented.

5.1 Method

To be able to evaluate whether the test strategy and the test automation tools fulfils the requirements in section 1.4, measurements must be done to see if the strategy is working and how time efficient the automation tools are. But it is not trivial to measure how good a test strategy is. Fewster and Graham (1999) give some ideas about what can be measured in their book Software Test Automation. They have divided the metrics into seven categories:

- Maintainability
- Efficiency
- Reliability
- Flexibility
- Usability
- Robustness
- Portability

It is possible to place some of the metrics in more than one category. For example, average elapsed time to add a test case can be placed in both usability and in efficiency.

I have chosen to use five metrics from three of the categories. These metrics were chosen because they will give the most useful information about whether or not the test strategy and the test automation tools fulfil the requirements and whether the goals are met. The definition of each metric and how the metrics should be measured are described below.

5.2 Measurements on implemented test strategy

The following metrics were used to evaluate whether the implemented test strategy fulfilled the requirements in section 1.4.

5.2.1 Efficiency

Metrics in this category give information about how efficient the test strategy is, for example, in finding bugs and in time needed to do test activities (Fewster and Graham, 1999). I have chosen three of the five metrics from this category because efficiency was the main goal for the test strategy.

Defect Detection Percentage (DDP)

This metric gives information about how effective the test strategy is in finding bugs. The definition of DDP is defects found by testing, divided by the total known defects (Fewster and Graham, 1999).

$$DDP = \frac{\text{defects found by testing}}{\text{total known defects}}$$
In Fewster and Graham (1999) total known defects is defined as defects found by testing plus the total number of defects found afterwards. Because of limitations in this thesis, it is not possible to study how many defects that were not detected by the test strategy were discovered later. Instead, a number of defects will be introduced in the current version of the Scapy/LHC framework. Because of properties of the test strategy, the current version of the Scapy/LHC framework is presumed to be correct. That is, without any defects. Therefore, the introduced defects are the total known defects.

This metric will be measured by introducing a defect in the current version of the Scapy/LHC framework. The test strategy is then run with 20 test cases against this version of the framework, and the result, whether or not the defect is detected, is recorded. This is repeated with 20 different defects. After this has been done, the DDP for the test strategy is calculated. As described in section 1.4, the DDP for the test strategy must be at least 50%. To be able to introduce realistic defects, an experienced user of Scapy/LHC was interviewed about which defects that are most commonly introduced by users then they add a new protocol to the framework.

**Average elapsed time to execute a set of test cases**

This metric gives information about how long it takes to execute a set of test cases (Fewster and Graham, 1999). The smaller this value is the more test cases can be executed in a time period. This saves resources and, maybe, more defects can be found.

This metric will be measured by executing 20 test cases and record the time elapsed. This value is dependent on how busy the computer is, so the test cases are run three times to get an average value.

**Average elapsed time to find a defect in the code**

This metric gives information about how long time it takes from a defect is detected by the test strategy until the user has localised the defect in the code, if the defect is localised in the code. The smaller this time is the more efficient is the test strategy. The time needed is dependent on how much information that is given, and how good it is, then a defect is found by the test strategy.

To measure this metric, T three defects were introduced to be found by the test strategy running with 20 test cases. Another person will then run the test strategy with 20 test cases and then try to localise the three defects in the code.

### 5.2.2 Maintainability

Maintainability means that it is easy to keep the tests updated if the software is changed. It is common that software is changed frequently and it is therefore important that the updating of the tests does not require too much resources (Fewster and Graham, 1999).

**Number of test cases that fails when a change in a protocol is introduced**

This metric gives information about how many test cases have to be changed if a change is introduced in a protocol. It is important that the type changes that occur frequently do not invalidate a large number of test cases (Fewster and Graham, 1999).

This metric will be measured by introducing a change in a protocol within the Scapy/LHC framework. The test strategy will then be run with 20 test cases, and the number of test
cases that failed will be recorded. This will be repeated with three different changes to get statistics about how changes in different protocols influences the test cases.

5.3 Measurements on implemented test automation tools

The following metric was used to evaluate whether the implemented test automation tools fulfilled the requirements in section 1.4, and to decide which test automation tool is most appropriate to use together with the implemented test strategy and the Scapy/LHC framework.

5.3.1 Usability

To make the automation tool and the test strategy effective, it must also be easy to use. This can be measured in many ways, but the most important thing for the test automation tool is that the time needed to add a new test case is small.

**Average elapsed time to add a test case**

This metric gives information about how much time needed when a new test case should be added. This is important because every time a protocol is added to the Scapy/LHC framework or something else is changed in the framework, new test cases needs to be added. The time required to do this should be as small as possible. This metric will also be used to decide which of the two test automation tools described in chapter 4 is most appropriate to use to test the Scapy/LHC framework. The biggest difference between the test automation tools is the way test cases are handled. It was also an important goal for this thesis that test cases should be handled in a time efficient way. As described in section 1.4, the average elapsed time to add a test case must be as most two minutes.

This metric will be measured for both test automation tools. A user first adds three new test cases with help of the first tool and the elapsed time is recorded. This will be repeated with the same three test cases with the second tool.
6 Results
This chapter presents the results from the evaluation of the test strategy and the test automation tools. The results from the measurements are presented for each of the metrics described in section 5.2 and 5.3. Finally in this chapter, I try to answer the question: How good is the test strategy?

6.1 Measurements on implemented test strategy
In this section, the results from the measurements on the implemented test strategy are presented.

6.1.1 Defect Detection Percentage (DDP)
The evaluation was performed by introducing 20 defects, one at a time. The test strategy was then run with the 20 existing test cases to see if the defect was detected.

The result was that ten defects were found and ten defects were not found by the test strategy. This gives that the DDP for the test strategy is 50%. That was exactly the lower limit of the DDP in the requirements, (see section 1.4 for the requirements). This means that the DDP requirement is fulfilled, even if it is on the limit.

This measurement of the DDP does not tell the whole truth because the measurement was not done in a completely realistic situation. The idea with the test strategy is that when a new protocol is added or something else is changed in the Scapy/LHC framework, a new test case shall be added. If the test automation tool tells that there are no defects, and a defect is discovered later anyway, a new test case shall be added to make sure that the defect never will be introduced again. The number of test cases will also be more then 20 in the future. This will make the DDP for this test strategy to increase over time, but because of the time restriction for this thesis, it is not possible to measure the DDP over a longer period. The measured DDP value gives nevertheless some indication on how efficient the test strategy is and can be compared against future measured DDP values. According to Fewster and Graham (1999) who subscribe to Gilb’s Law:

“Gilb’s Law: anything can be made measurable in some way, which is superior to not measuring it at all (Gilb, 1988).”

This means that even if it is not possible to measure something in a perfect way, is it nevertheless better than not measuring it at all.

6.1.2 Average elapsed time to execute a set of test cases
The 20 test cases were executed three times and the average elapsed time to execute 20 test cases was calculated. The average elapsed time was calculated to 373.52 seconds, with a minimum of 373,41 seconds and a maximum of 373,77 seconds. This gives an average elapsed time to execute one test case of 18.68 seconds. The measured values imply that the time needed to execute 100 test cases completely automated is a little bit over half an hour. This is an acceptable time, compared to the expected time needed to do manual tests on every protocol. The average size of the log files that where used by the test cases was 2.38 MB, so 100 test cases could test the Scapy/LHC framework with 238 MB of data.
The measurements were done on a computer with a CPU with the speed 2.0 GHz and 4 GB of RAM. The values will of course vary depending on the CPU speed, available RAM, operating system, the size of the log files used by the test cases and many more factors.

6.1.3 Average elapsed time to find a defect in the code

Three bugs were introduced in the Scapy/LHC framework and the test strategy was run with the test automation tool described in section 4.3 and 20 test cases. It does not matter which of the test automation tools that is used in this case, because the output from the 20 test cases will be the same. A chosen person was then trying to find the three defects in the framework with help of the test results.

The result was that all three defects were found in 4 minutes and 41 seconds. This shows that it is possible to find defects in the code with the help of the output from the test cases. The average elapsed time to find a defect in the code is 1 minute and 34 seconds. This value does not tell so much about what the average elapsed time to find a defect in the code will be in the future, because it is very dependent on the person who does the tests, which development environment that is used and the type of defect. This measurement was only done with one person and one development environment. However, the result gives an indication that it is possible to find defects in the code with help of the test results in a reasonable time.

6.1.4 Number of test cases that fail upon protocol change

To measure this metric, a change was introduced in a protocol and the test strategy was run with 20 test cases. The number of test cases that failed was then counted. This was repeated with three different changes in three different protocols. The results are shown in Figure 24.

![Figure 24: Number of test cases that failed when changes were introduced in different protocols](image-url)
The measurements show that a greater number of test cases will be affected if a change is introduced in lower layer protocol, like the IP protocol, compared to if the change is introduced in a higher level protocol like the HTTP protocol. This is due to the fact that a great number of higher level protocols use IP in the network layer. If something is changed in the IP protocol, all test cases that test a protocol that uses IP in the network layer will fail. If something instead is changed in the HTTP protocol on the application layer, only the test cases that explicitly test the HTTP protocol will be affected. This is a good thing because in this case is it more likely that changes are introduced in the protocols on the application layer than in the protocols in the network layer. For each higher level protocol there will just be a smaller number of test cases. The conclusion is that the most frequent types changes will not invalidate a great number of test cases, but if something is changed in a protocol that many other protocols and test cases depend on, it is possible that a greater number of test cases will be affected.

### 6.2 Measurements on implemented test automation tools

In this section, the results from the measurements on the implemented test automation tools are presented.

#### 6.2.1 Average elapsed time to add a test case

Three new test cases were first added by a chosen person with help of the test automation tool based on PyUnit described in section 4.2, and the time was recorded. The same thing, with the same test cases and the same person, was then repeated with the second implementation of a test automation tool described in section 4.3. The average elapsed time to add a test case was then calculated. The results are shown in Table 1.

<table>
<thead>
<tr>
<th>Test automation tool:</th>
<th>Elapsed time to add three test cases (seconds):</th>
<th>Elapsed time to add one test case (seconds):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on PyUnit</td>
<td>196</td>
<td>65</td>
</tr>
<tr>
<td>New implementation</td>
<td>84</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 1: Results from the measurements of elapsed time to add a test case

Both test automation tools fulfill the requirements in section 1.4. That is, the average elapsed time to add a new test case is less than two minutes. However, the time needed to add a new test case in the tool based on PyUnit is more than double the time needed to add a new test case in the second implementation. The goal was that the test strategy should be time efficient. Therefore, the new implementation of a test automation tool seems to be the most appropriate tool to use in order to automate the test strategy. When the tools were tested, some other advantages were discovered that make the new implementation of a test automation tool more time efficient than the tool based on PyUnit. For example, access to the difference log via the tool. This is also a positive usability aspect.
6.3 Summary
The results show that the test strategy and the test automation tools fulfil all requirements on DDP and average elapsed time to add a test case, stated in section 1.4. They also show that:

- Defects can be localised in the code with the help of the output from the test cases
- The Scapy/LHC framework can be tested with a reasonable amount of data per time unit.
- The most frequent types of changes in the Scapy/LHC framework do not invalidate a large number of test cases.
- The new implementation of test automation tools, described in section 3.4.2, was the most appropriate automation tool to use.

Beside this, the test strategy and the test automation tools have properties that fulfil all the requirements stated in section 3.1.

So, how good is the test strategy? These measurements have not been done in a completely realistic situation, but they still show that the test strategy is able to find defects in the Scapy/LHC framework in a time efficient way with the help of any of the implemented test automation tools. How good it really is could only be shown by using the test strategy in a realistic situation over a longer period.
7 Conclusions and future work

In this chapter the conclusions for this thesis are summarised. Possible future work is also discussed.

7.1 Conclusions

The implementation and automation of the test strategy have made it possible to test old and new functionality within the Scapy/LHC framework in a time efficient way every time changes have been introduced. This makes it possible to increase the quality of the framework. The most important measure to make this test strategy effective is to add new test cases every time the Scapy/LHC framework has been changed.

The measurements done on the test strategy were not done in completely realistic situations and the result may therefore not tell the whole truth. To get a more complete picture of how good the test strategy is the measurements should be done over a longer period when the test strategy is used practically. This was not possible to do in this project because of limitation in time.

7.2 Future work

Tools and frameworks, similar to the Scapy/LHC framework, are now developed for the next generation of mobile communication. It should be possible to implement this test strategy and a test automation tool even for these tools and frameworks. The test strategy will be more effective in these cases because it could be introduced in the very beginning of the development process.
Glossary

CLI – Command Line Interface
DDP – Defect Detection Percentage
GUI – Graphical User Interface
HTML – HyperText Markup Language
HTTP – HyperText Transfer Protocol
IP – Internet Protocol
MAC – Media Access Control
OSI – Open System Interconnection
TCP – Transmission Control Protocol
XML – eXtensible Markup Language
Bibliography


Python Software Foundation (2008). *Python Programming Language* [www]

PythonWare (1999). *An Introduction to Tkinter* [www]

Purcell, Steve (2005). *PyUnit – the standard unit testing framework for Python* [www]

Pettichord, Bret (2001). *Seven Steps to Test Automation Success* [www]


Ericsson AB (2007). *LHC framework* [www]
   <Ericsson Internal Network> Retrieved 20/1 2008.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Henrik Johansson