Tuomas Parttimaa

TEST SUITE OPTIMISATION BASED ON RESPONSE STATUS CODES AND MEASURED CODE COVERAGE

Master’s Thesis
Degree Programme in Computer Science and Engineering
April 2013
ABSTRACT

A software test suite often comprises of thousands of distinct test cases. Therefore, the execution of an unoptimised test suite might waste valuable time and resources. In order to avoid the unnecessary execution of redundant test cases, the test suite should be optimised to contain fewer test cases. This thesis focuses on the optimisation efforts of a commercially available Hypertext Transfer Protocol (HTTP) fuzzing test suite.

A test setup was created for the optimisation purposes. The test setup consisted of the given fuzzing test suite, five different HTTP server implementations used as test subjects, and a code coverage measurement tool. Test runs were executed against the five test subjects with the test suite, and at the same time code coverage was measured from the test subjects.

In this thesis, three different types of test suite optimisation algorithms were implemented. The original test suite was optimised by applying the optimisation algorithms to the results of the test runs. Another set of test runs were performed with the optimised subset suites, while again measuring code coverage from the same test subjects. All of the coverage measurement results are presented and analysed. Based on the code coverage analysis, the test suite optimisation algorithms were assessed and the following research results were obtained.

Code coverage analysis demonstrated with a strong degree of certainty that a variation in the response messages indicates which test cases actually exercise the test subject. The analysis showed also with a quite strong degree of certainty that an optimised test suite can achieve the same level of code coverage which was attained with the original test suite.

Keywords: fuzz testing, coverage measurement, coverage testing, software testing, hypertext transfer protocol.

TIIVISTELMÄ

Ohjelmistojen testisarja koostuu usein tuhansista erilaisista testitapauksista. Testisarjan suorittamisessa voi mennä hukkaan arvokasta aikaa ja resурсseja, jos testisarjaa ei optimoida. Testisarja tulee optimoida siten, että se sisältää vähän testitapauksia, jotta epäolennaisten testitapausten tarpeeton suoritus vältetään. Tämä diplomityö keskittyi kaupallisesti saatavilla olevan Hypertext Transfer Protocol (HTTP) -palvelimien fuzz-testisarjan optimointirykkyyteen.


Koodikattavuusanalyysin avulla saatiin varmuus siitä, että muutos vastausviesteissä ilmenee, mitkä testitapaukset oikeasti käyttävät testikohdetta. Analyysilla saatiin myös kohtalainen varmuus siitä, että optimoidulla osajoukkosarjalla voidaan saavuttaa sama koodikattavuus, joka saavutettiin alkuperäisellä testisarjalla.

Avainsanat: fuzz-testaus, kattavuuden mittaus, kattavuustestaus, ohjelmistotestaus, hypertext transfer protocol.
# TABLE OF CONTENTS

ABSTRACT ................................. 9
TIIVISTELMÄ ................................ 9
TABLE OF CONTENTS ................. 9
FOREWORD ................................. 9
LIST OF ABBREVIATIONS AND SYMBOLS ...

## 1. INTRODUCTION ................................................................. 9

## 2. SOFTWARE TESTING ..................................................... 11

### 2.1. Fuzz Testing ................................................................. 11

#### 2.1.1. Fuzzer Categorization ........................................... 12

#### 2.1.2. Mini-Simulation Method ........................................ 12

### 2.2. Code Coverage ............................................................. 13

#### 2.2.1. Line Coverage ....................................................... 14

#### 2.2.2. Branch Coverage ................................................... 14

#### 2.2.3. Function Coverage ................................................ 15

### 2.3. Code Coverage Measurement ........................................ 15

#### 2.3.1. Source Code Level Instrumentation ....................... 15

#### 2.3.2. Binary-Level Instrumentation .................................. 16

## 3. HYPERTEXT TRANSFER PROTOCOL (HTTP) ....................... 18

### 3.1. Client/Server Message Exchange ................................... 18

#### 3.1.1. Request Messages .................................................. 19

#### 3.1.2. Response Messages ................................................. 19

### 3.2. Request Methods .......................................................... 20

### 3.3. Response Status Codes ................................................ 21

#### 3.3.1. Response Status Code Classes ................................. 21

## 4. TEST ENVIRONMENT ....................................................... 22

### 4.1. Research Questions ....................................................... 22

### 4.2. Terms of the Work ........................................................ 23

### 4.3. Solution Choices of the Test Setup ............................... 23

### 4.4. Test Suite Analysis ........................................................ 24

#### 4.4.1. Structure of the Test Suite ....................................... 25

#### 4.4.2. Instrumentation Method ........................................... 26

#### 4.4.3. Test Run Result Files .............................................. 26

### 4.5. Test Subject Selection ................................................... 27

#### 4.5.1. Nginx ................................................................. 28

#### 4.5.2. Lighttpd ............................................................... 29

#### 4.5.3. Tntnet ................................................................. 29

#### 4.5.4. Hiawatha .............................................................. 29

#### 4.5.5. Apache httpd ....................................................... 30

#### 4.5.6. Other Candidates for Test Subjects ........................... 30

### 4.6. Code Coverage Measurement Tool Selection .................. 31

#### 4.6.1. GNU Compiler Collection (GCC) .............................. 31

#### 4.6.2. Gcov ................................................................. 31

#### 4.6.3. Lcov ................................................................. 32

#### 4.6.4. Other Candidates for the Code Coverage Measurement Tool .... 33
5. TEST SUITE OPTIMISATION ............................................................ 35
  5.1. Description of the Optimisation Methods .......................................... 35
    5.1.1. Test Suite Optimisation Algorithm No. 1 ............................... 36
    5.1.2. Test Suite Optimisation Algorithm No. 2 ............................... 38
    5.1.3. Test Suite Optimisation Algorithm No. 3 ............................... 40
  5.2. Test Setup Control Script .............................................................. 41
  5.3. Practical Trial ............................................................................ 42
    5.3.1. Result Validation .............................................................. 42
6. TEST RESULTS ............................................................................... 44
  6.1. Code Coverage Measurement Results Using the Original Test Suite ...... 45
  6.2. Optimisation Results as Subset Test Suites ........................................ 46
    6.2.1. Reduction in the Size of the Test Suite .................................... 47
  6.3. Code Coverage Measurement Results Using the Subset Test Suites ...... 48
    6.3.1. Results Using the Subset Test Suites Optimised with Algorithm No. 1 ................................................................. 49
    6.3.2. Results Using the Subset Test Suites Optimised with Algorithm No. 2 ................................................................. 50
    6.3.3. Results Using the Subset Test Suites Optimised with Algorithm No. 3 ................................................................. 51
7. DISCUSSION ................................................................................... 53
  7.1. Answers to the Research Questions ................................................ 53
  7.2. Assessment of the Algorithms ....................................................... 54
  7.3. Known Limitations ................................................................. 55
  7.4. Future Development ................................................................. 56
8. CONCLUSION ............................................................................... 58
9. REFERENCES ................................................................................. 60
10. APPENDICES .................................................................................. 68
FOREWORD

This research was conducted while working for Codenomicon Oy. The topic for this Master’s Thesis evolved into its final form in March 2012. The development of the test setup was finished by June 2012, and the code coverage measurements were performed in July 2012. The implementation of the optimisation algorithms and the processing of the test results were carried out in July 2012. The writing process was started in October 2012 and completed in the spring of 2013. It is easy to say that this thesis has been the greatest endeavour and achievement in my life thus far.

First and foremost, I would like to cordially thank my parents and siblings. I am most indebted to you for all your love and support given during my studies. Secondly, I would like to express my sincere compliments to my thesis adviser, Lauri Piikivi from Codenomicon Oy for his valuable help and guidance. I especially want to thank the supervisor and the second examiner of this thesis, Professors Juha Röning and Janne Heikkilä from the University of Oulu for their important feedback. I want to give my gratitude to Heikki Kortti for proofreading my thesis.

I would like to profoundly acknowledge Codenomicon Oy and my fellow co-workers for the challenging opportunity to write this thesis. Furthermore, I would like to express my appreciation to Rauli Kaksonen from Codenomicon Oy for originating an apt topic. A special thanks goes to my highly revered friends from the ABS. I am most grateful for all of the terrific years spent amongst you. I would like to express my gratitude to Pekka Mäkinen for availing me with the peculiarities of \LaTeX.

Finally, I would like to take this opportunity to thank my beloved Aino-Kaisa for all her kind support and encouragement throughout this journey.

Oulu, April 24th, 2013

Tuomas Parttimaa
**LIST OF ABBREVIATIONS AND SYMBOLS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABNF</td>
<td>Augmented Backus-Naur Form</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>bash</td>
<td>Bourne Again Shell</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CentOS</td>
<td>Community ENTerprise Operating System</td>
</tr>
<tr>
<td>CSV</td>
<td>comma-separated values</td>
</tr>
<tr>
<td>C/DC</td>
<td>Condition/Decision Coverage</td>
</tr>
<tr>
<td>DCC</td>
<td>Dynamic Code Coverage</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix</td>
</tr>
<tr>
<td>GPLv2</td>
<td>GNU General Public License version 2</td>
</tr>
<tr>
<td>GPLv3</td>
<td>GNU General Public License version 3</td>
</tr>
<tr>
<td>GWS</td>
<td>Google Web Server</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText [sic] Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext [sic] Transfer Protocol</td>
</tr>
<tr>
<td>HTTP/0.9</td>
<td>Hypertext [sic] Transfer Protocol version 0.9</td>
</tr>
<tr>
<td>HTTP/1.0</td>
<td>Hypertext [sic] Transfer Protocol version 1.0</td>
</tr>
<tr>
<td>HTTP/1.1</td>
<td>Hypertext [sic] Transfer Protocol version 1.1</td>
</tr>
<tr>
<td>HTTP/2.0</td>
<td>Hypertext [sic] Transfer Protocol version 2.0</td>
</tr>
<tr>
<td>httpd</td>
<td>Hypertext [sic] Transfer Protocol Daemon</td>
</tr>
<tr>
<td>IANA</td>
<td>Internet Assigned Numbers Authority</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IIS</td>
<td>Internet Information Services</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol version 6</td>
</tr>
<tr>
<td>ISO/IEC</td>
<td>International Organization for Standardization/International Electrotechnical Commission</td>
</tr>
<tr>
<td>LDRA</td>
<td>Liverpool Data Research Associates</td>
</tr>
<tr>
<td>LF</td>
<td>line feed</td>
</tr>
<tr>
<td>LGPL</td>
<td>GNU Lesser General Public License</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>light (footprint) and httpd</td>
</tr>
<tr>
<td>LOC</td>
<td>lines of code</td>
</tr>
<tr>
<td>LTP</td>
<td>Linux Test Project</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Modified Condition/Decision Coverage</td>
</tr>
<tr>
<td>MCC</td>
<td>Multicondition Coverage</td>
</tr>
<tr>
<td>MS</td>
<td>Microsoft</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
<tr>
<td>NCSA</td>
<td>National Center for Supercomputing Applications</td>
</tr>
<tr>
<td>OHS</td>
<td>Oracle HTTP Server</td>
</tr>
<tr>
<td>OUSPG</td>
<td>Oulu University Secure Programming Group</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>PROTOS</td>
<td>Security Testing of Protocol Implementations project</td>
</tr>
</tbody>
</table>
RFC Request for Comments
SIP Session Initiation Protocol
SNMP Simple Network Management Protocol
SQA Software Quality Assurance
SUT Software Under Test
SUT System Under Test
TCP Transmission Control Protocol
TLS Transport Layer Security
URI Uniform Resource Identifier
VM Virtual Machine
W3 World Wide Web
W3C World Wide Web Consortium
WWW World Wide Web

$\Delta U$ The difference between the total numbers of the particular coverage unit in two separate test runs

$C$ Code coverage

$C_b$ Branch coverage

$C_f$ Function coverage

$C_l$ Line coverage

$N$ The number of elements accessed

$N_b$ The number of branch coverage conditions met

$N_{bt}$ The total number of branches

$N_f$ The number of functions executed

$N_{ft}$ The total number of functions

$N_l$ The number of source code lines accessed

$N_{lt}$ The number of source code lines without comments

$N_t$ The total number of elements

$S$ The size reduction as a percentage

$T$ The size of the original test suite as measured in the number of test cases

$T_{opt}$ The size of the optimised test suite as measured in the number of test cases

$U_{opt}$ The total number of the particular coverage units in a test run with an optimised test suite

$U_{orig}$ The total number of the particular coverage units in a test run with the original test suite
1. INTRODUCTION

Software testing is an important part of the software development process. The purpose of software testing is both to find bugs from a System Under Test (SUT) and to validate that the SUT is working correctly. Testing methods are traditionally divided to white box and black box testing. White box testing is also known as structural testing, and it focuses on the implementation details. Black box testing is also termed functional testing. In it, the SUT is treated as a black box. Black box testing is done by providing inputs to the software and verifying that the behaviour of the SUT is in conformance to a specification. It is not necessary for the user to know any details of the implementation. [1, pp. 5–11]

Fuzz testing or fuzzing is a black box testing method that focuses on the robustness of the software. In fuzz testing, invalid and unexpected inputs are sent to the SUT, while monitoring its behaviour for faults. It is an automated or semi-automated testing process that involves manipulating and repeatedly supplying input data (negative tests) to the SUT through its external interfaces for processing [2, p. 22]. The purpose of fuzzing is to find security-related defects and other undesired behaviour. Fuzz testing can help uncover both reliability and robustness flaws in the SUT, while also facilitating the analysis of any found defects by recording the details of the failure conditions and the events that occurred before them. [3, pp. 24–25, 30–31]

Code coverage is an often used metric in software testing. It provides a quantitative measurement of the exercised source code after test execution, usually expressed either as a percentage or a ratio of the number of covered source code lines and the total number of lines. Code coverage analysis is a white box testing technique which requires access to the source code of the SUT. Analysis helps in finding out which parts of the source code have not yet been tested, and vice versa. The main idea behind coverage testing is that the unexercised parts of the source code are presumed to contain errors. In addition, coverage analysis can help identify redundant test cases which do not increase coverage. [4, pp. 105–108]

An unoptimised test suite might involve many redundant test cases which waste valuable testing time, resources, and generate useless test data to be processed. The losses add up significantly if the same test suite is executed many times. To reduce these factors, the test suite should be optimised to contain only a minimum amount of test cases. Ideally, an optimised test suite would achieve the same level of code coverage as an unoptimised test suite in a much shorter testing time as fewer test cases need to be executed.

The scope of this thesis is on the optimisation of a commercially available Hyper-text Transfer Protocol (HTTP) server test suite. The optimisation methods developed in this thesis are based completely on the analysis of test run results. An adequate sampling of different HTTP server implementations are chosen as test subjects, and code coverage is measured from all of them throughout the testing. Optimisation methods are applied to the attained test run results, which include HTTP response status codes from the SUTs, and code coverage measurement results. The results of the test suite optimisation are validated using rudimentary code coverage analysis.

Based on the code coverage analysis, this thesis answers the following research questions:

1. Does the variation in protocol response messages indicate which test groups
should be tested more thoroughly, i.e. does the variation in responses from the SUT highlight those test groups which actually exercise the SUT?

2. Can the same level of code coverage be achieved with a smaller amount of test cases, i.e. with an optimised test suite which does not contain test groups in which responses from the SUT remain the same?
2. SOFTWARE TESTING

Software testing has two major goals. The first goal is to find bugs from the SUT, and the second is to validate that the SUT is working correctly. Testing activities also support software quality assurance (SQA) by feeding gathered information back to programmers who can use the information to prevent bugs and improve future software. The first goal of software testing is reached by performing negative tests whose main purpose is to break the SUT, or to demonstrate that it contains bugs. The second goal is achieved by performing positive tests in order to ensure that the SUT is working according to specifications. [5, pp. 7–8]

Software testing methods are traditionally divided into white box testing and black box testing. In addition, a combination of the previous two methods is called grey box testing. In white box testing (also known as structural testing), the implementation details of the SUT are known and taken into account. White box testing requires complete access to the source code of the SUT. Ideally, white box testing would test all of the logical paths of the source code, but such exhaustive testing is not realistic even for simple programs. [6, pp. 9, 37]

In black box testing (also termed functional testing), the SUT is considered as a black box, which means that there is no knowledge of the internal behaviour and the implementation of the SUT. Ideally in black box testing, a tester would feed both all of the possible valid and invalid inputs to the SUT. However, doing this would require an infinite number of test cases. Therefore, it is not possible to attain the ideal. The output of the SUT is monitored during tests, and the aim is to find situations where the SUT does not act in conformance to the given specification. [6, pp. 8–9]

2.1. Fuzz Testing

Fuzz testing or fuzzing is one form of black box testing that focuses on the robustness of the software, and the history of fuzzing is relatively short in comparison to other software testing techniques. The earliest known reference to fuzz testing is from 1988, when Professor Barton P. Miller led a project to develop a simple tool to test the robustness of UNIX utility programs [7, p. 1]. Inspired by the syntax testing described by Boris Beizer in [1] and the earlier work of Barton P. Miller, Oulu University Secure Programming Group (OUSPG) took a more systematic approach to fuzzing when Security Testing of Protocol Implementations (PROTOS) project was started in 1999. The aim of the project was to produce fuzzing test suites which would be first available to software vendors, and eventually also to the general public. [3, pp. 23–24], [8]

Fuzzing is a negative black box testing technique which aims to find security flaws and robustness defects in the SUT. The fuzz testing process starts with feeding unexpected and malformed inputs to the SUT through external interfaces in a repeated manner. Response messages sent by the SUT can be logged and analysed. The SUT is also monitored for any faulty behaviour throughout the testing process. To ease the analysis of any found defects, failure monitoring should include a recording of the events leading to the failure and the failure itself. Ideally, this process is highly automated, with the aim being to perform as much testing as possible in a short time span without producing a large amount of redundant testing data. [3, pp. 24–25, 30–31]
2.1.1. Fuzzer Categorization

A fuzz testing tool is generally called a fuzzer. Takanen et al. describe a four-way categorization of fuzzers which is founded on test case complexity. The following categorization is sorted from the least intelligent fuzzers to the most intelligent ones. The first category is static and random template fuzzers, which are typically used to test simple request/response protocols, or file formats. They generate arbitrary input data, and have very little or no structural or semantic knowledge at all about the protocol used by the SUT. The second category of fuzzers is block-based fuzzers, which contain an implementation of the basic structure of a request/response protocol. When a fuzzer possesses structural details of the protocol, it can focus on testing specific parts of the protocol messages at a time, while possibly retaining other parts of the message valid and intact. [3, pp. 26–27]

The third category is dynamic generation or evolution-based fuzzers, which are able to learn the protocol used by the SUT based on the interpretation of the messages received from the SUT or recorded from similar protocol exchanges between the SUT and other protocol implementations. Therefore, it is not a necessity for the fuzzer to have preliminary knowledge of the protocol under fuzz testing. The last category of fuzzers is model-based or simulation-based fuzzers, which incorporate either an implementation of the protocol model used by the SUT, or a simulation of the interactions within the protocol. A fully model-based fuzzer is able to achieve full interoperability with the SUT, which means that the fuzzer exercises the input handling routines of the SUT thoroughly. The model-based approach also enables fuzzing the latter parts of a complex message sequence. [3, pp. 26–29]

2.1.2. Mini-Simulation Method

A mini-simulation method can be used for the functional modelling of a protocol message exchange which is usually described in a protocol specification. This method was originally developed for functional robustness testing by the PROTOS project as a joint effort between OUSPG and VTT Technical Research Centre of Finland. [9, pp. 4, 54] In addition to the test suites created in the PROTOS project, the mini-simulation method has served as the basis of commercially available test suites made by Codenomicon Oy [10].

The original design goal for the method was the ability to produce a large amount of intelligently fuzzed protocol messages. This meant that only one or a few elements in the messages are fuzzed at a time, any checksums or other dynamic runtime protocol values are calculated correctly, and the fuzzed messages are transported to the SUT successfully. The mini-simulation method is meant to be a debugging and testing aid instead of being a design method. [9, pp. 54–56] Figure 1 depicts the mini-simulation method in use for testing purposes [11].
When the mini-simulation method is applied to a protocol specification, the result is a minimal model of the protocol entity in a higher-order attribute grammar. The mini-simulation model consists of a master specification (simulation grammar) and a configuration. The mini-simulation model is derived from the configuration which reads the master specification and modifies the grammar in the specification into an interaction model and test cases. [9, pp. 55–58]

The master specification uses context-free grammar, also called Backus-Naur Form (BNF) [12], as the basis for the mini-simulation language notation to describe typical error-free protocol exchanges in a simple, human-readable format. The simplicity and flexibility of the master specification is retained by keeping the configuration separate from it. The master specification is not intended to be a complete, fully functional protocol implementation, although the derived model can be used as a prototype implementation. [9, pp. 55–58]

2.2. Code Coverage

The concept of code coverage was introduced in 1963 [13], and it has since become an often used metric in software testing. Code coverage provides a quantitative measure of the exercised source code after test execution. Therefore, code coverage is a direct measure of test quality and an indirect measure of the quality of the SUT. [4, pp. 105–107]

Code coverage measurement results are usually represented as a percentage or a ratio of the number of elements exercised and the total number of elements. Code coverage is calculated by dividing the number of the executed elements by the total number of elements. The basic form of the code coverage calculation is defined as the following equation:

$$ C = \frac{N}{N_t}, $$(1)

where $N$ is the number of elements accessed, and $N_t$ is the total number of elements. [14, p. 20]

Code coverage analysis identifies the parts of the code that have not been exercised during testing, and vice versa. The idea behind coverage testing is that the unexercised parts of the code are presumed to include errors. Therefore, when more testing is done to increase code coverage, confidence that the SUT works correctly is gained. Code coverage gives a measure of test completeness. [15, pp. 428–429]

The process of code coverage analysis is a white box testing technique, as it requires access to the source code of the SUT. Often the source code needs to be recompiled in order to support code coverage analysis. In addition, code coverage analysis
helps testers generate a more thorough set of tests, and it can also help in the identification of redundant test cases which do not increase coverage. [4, pp. 107–108]

Code coverage is a measurement of a coverage criterion which specifies a set of paths that should be tested and covered. A path is a sequence of operations from the start of a program to an exit point. The simplest and smallest path is a single line of code. Instead of the impossible goal of testing all the logical paths, a coverage criterion defines an achievable amount of testing. [16, p. 43]

However, complete coverage of any criterion does not guarantee that the code is error-free. There are more than one hundred different types of coverage criteria known. [17] Three common coverage criteria used in this thesis are further described in Sections 2.2.1, 2.2.2, and 2.2.3.

2.2.1. Line Coverage

Line coverage metric measures the amount of source code lines exercised during testing. Line coverage is the weakest coverage criterion, because it does not take into account that a single code line may comprise complex decision making based on the values of variables. Instead, the criterion requires that a source code line is simply executed. [16, p. 43]

A full one hundred percent line coverage is attained when all source code lines have been executed. Line coverage is calculated by dividing the number of covered code lines by the total number of source code lines. The line coverage calculation is defined as the following equation:

\[
C_l = \frac{N_l}{N_{lt}},
\]

where \(N_l\) is the number of source code lines accessed, and \(N_{lt}\) is the number of source code without comments. [14, p. 21]

2.2.2. Branch Coverage

Branch coverage metric measures the amount of branches exercised during testing. Contrary to line coverage, branch coverage takes into account different branching possibilities by checking whether all Boolean expressions are evaluated to both true and false. [18]

A full one hundred percent branch coverage is obtained when both all of the source code lines and all the branches have been executed [16, p. 43]. Branch coverage is calculated by dividing the number of covered branches by the total number of branches. The branch coverage calculation is defined as the following equation:

\[
C_b = \frac{N_b}{N_{bt}},
\]

where \(N_b\) is the number of branch coverage conditions met, and \(N_{bt}\) is the total number of branches. [14, p. 22]
2.2.3. Function Coverage

Function coverage metric measures the amount of functions called during testing. The measurement results provide useful information in the preliminary testing phases, when it is important to have quick knowledge that the testing has covered at least partially all areas of the SUT. [18]

A full one hundred percent function coverage is reached when all functions have been called and covered. Function coverage is calculated by dividing the number of covered functions by the total number of functions. The function coverage calculation is defined as the following equation:

\[ C_f = \frac{N_f}{N_{ft}}, \]  

(4)

where \( N_f \) is the number of functions executed, and \( N_{ft} \) is the total number of functions. [14, p. 23]

2.3. Code Coverage Measurement

In order to be able to measure code coverage information, the SUT must be instrumented. Instrumentation is a process in which instrumentation code is added to the SUT. The added instrumentation code enables the collection of run-time information, such as code coverage, without affecting the original functionality of the SUT. Other purposes for instrumentation are e.g. analysis, optimisation, and testing of the SUT. In general, there are two different approaches to code instrumentation: source code level instrumentation, and binary-level instrumentation. There are numerous commercial and open source [19] tools for code coverage measurement for both of these approaches. [20, pp. 47–51]

2.3.1. Source Code Level Instrumentation

Figure 2 depicts how source code level instrumentation is applied to the software compilation process [21, p. 2]. Source code level instrumentation is applied to the source code of the SUT before it is compiled into object code. Source-level instrumentation is always specific to each programming language, and it also requires disk space to store trace information. Any available implementation details about the SUT benefit source-level coverage tools, since they reduce the amount of the instrumentation code, which eventually reduces both the execution time and the resulting trace size. [20, pp. 53–55]
2.3.2. Binary-Level Instrumentation

Binary-level instrumentation can be done to the machine code in two different points of the software compilation process: to the object code prior to linking, or to the executable code after linking. The former option for binary-level instrumentation is object code level instrumentation, which is also called pre-linking instrumentation. Figure 3 depicts how object code level instrumentation is applied to the software compilation process. Object code level instrumentation is done after the source code is compiled to object code but before it is linked into executable code. [20, pp. 51–53]

The latter option for binary-level instrumentation is executable-level instrumentation, which is also called post-linking instrumentation. Figure 4 depicts how executable-
level instrumentation is applied to the software compilation process. Executable-level instrumentation is performed after the object code is linked into executable code. Neither of the binary-level instrumentation approaches requires access to the source code of the SUT. Therefore, binary-level instrumentation is programming language-independent. [20, pp. 51–53]

Figure 4. Executable-level instrumentation in the software compilation process.
3. HYPERTEXT TRANSFER PROTOCOL (HTTP)

The Hypertext Transfer Protocol (HTTP) is an application-level request/response protocol for distributed, collaborative, hypermedia information systems. The stateless client/server communications in HTTP take place over Transmission Control Protocol (TCP) [22] connections. The most common use of HTTP is in the World Wide Web (WWW or W3) [23] as a generic communication protocol between web servers, proxies, and Internet browsers. [24, pp. 1–13]

The HTTP specification is developed and maintained by the Internet Engineering Task Force (IETF). The first version of HTTP [25], later referred to as HTTP/0.9, was originally proposed by Sir Tim Berners-Lee in 1991. Standardization led to the second version, HTTP/1.0 [26], which was defined in Request for Comments (RFC) 1945 technical specification in 1996. The World Wide Web Consortium (W3C) supported the standardization efforts of the latest version of HTTP, HTTP/1.1 [24], which was defined in RFC 2616 in 1999. HTTP/1.1 is the current Internet standard for the protocol. The HTTP/1.1 specification is currently under revision alongside with the development of the HTTP/2.0 specification [27]. [24, pp. 1–7]

3.1. Client/Server Message Exchange

The HTTP specification uses Augmented Backus-Naur Form (ABNF) notation [28] to define the syntax of two message types: request and response messages. The specification also defines that HTTP messages use American Standard Code for Information Interchange (ASCII) [29] coded character set. A typical successful and error-free client/server HTTP message exchange is shown in Figure 5 as a Message Sequence Chart (MSC) [30]. An HTTP client (e.g. a browser) is a program that creates connections to a server in order to transmit HTTP requests for resources. An HTTP server is an application program that accepts incoming connections and serves the requests with HTTP responses that contain the resources requested by the client. [24, pp. 8–15], [31, p. 7]

![Figure 5. MSC of a successful HTTP message exchange.](image-url)
3.1.1. Request Messages

All HTTP request and response messages consist of the following sections: a start line, zero or more header lines, an empty line, and an optional message body. An example of an HTTP request message from a client to a server is shown in Listing 1. The first line of the request message is a request line which consists of a method token, a Uniform Resource Identifier (URI), and the HTTP protocol version used by the client. Request method tokens are further discussed in Section 3.2. The URI identifies the target resource in which the method is applied to. Lines 2–9 are header lines which are used to pass additional information about the request and the client itself. Line 10 is an empty line which indicates the end of the header section. The presented example of a request message does not contain an optional message body section. [24, pp. 31–39]

```
GET / HTTP/1.1
Host: 10.10.3.218:8075
User-Agent: HTTP Test Suite (X11; U; Linux x86_32; en-US; rv:1.8.0.10) (Test case number:0)
Accept: text/xml, application/xml, application/xhtml+xml, text/html;q=0.9, text/plain;q=0.8, image/png,*/*;q=0.5
Accept-Language: en-us, en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8;q=0.7, *;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Listing 1. An HTTP request message.

3.1.2. Response Messages

An example of an HTTP response message from a server to a client is shown in Listing 2. The first line of the response message is a status line which comprises of the HTTP protocol version used by the server, a numeric status code, and a textual description of the status code. The numeric status codes and their descriptions are further explained in Section 3.3. Lines 2–8 are header lines which are used to pass additional information about the response, the server, and access to the target resource. Line 9 is an empty line which indicates the end of the header section. The optional message body is on lines 10–18, and the message body section uses HyperText Markup Language (HTML) [32]. [24, pp. 31–43]
### 3.2. Request Methods

The request method token is part of the request line in an HTTP request message. The method token indicates the method which has been requested by the client, and it also indicates what the client expects as a successful outcome of the method. The HTTP specification defines eight different request methods which are described below. In addition, an extension method is defined to mean that also additional methods may be used to support future requirements. By convention, methods are defined in all-uppercase and as case-sensitive. All HTTP server implementations are required to support at least GET and HEAD methods. [24, p. 36], [33, p. 20]

The GET method requests that the HTTP server sends the target entity in its current representation. The server transmits the target resource to the client in a response message. The HEAD method works in the same way as the GET method, except that the server does not include a message body in the response message. This method is used by the client to obtain the server headers, and testing hyperlink validity and accessibility. [24, pp. 53–54], [33, p. 24]

The POST method requests that the HTTP server accepts the entity included in the payload of the request message as data to be processed by the target resource. The actual data processing is completely dependent on the server configuration. The PUT method requests that the server either creates or replaces the target resource with the entity enclosed in the request message payload. The DELETE method requests that the server removes the target resource. [24, pp. 54–56]

The TRACE method is used to request a testing loopback (on the application level) for the HTTP request message. This method allows the client to observe what is actually received at the other end, and to use that data for testing or diagnostic information. The OPTIONS method requests information about the communication methods that the HTTP server supports on the target resource. Alternatively, this method can also query the server for options which are generally supported. The HTTP specification has reserved the CONNECT method for the use by HTTP proxies to create a tunnel to
3.3. Response Status Codes

The status line in an HTTP response message consists of the protocol version information, a numeric status code, and a textual description of the status code. The status code is a three-digit integer result code of the attempt to understand and serve the request. The numeric status code is intended for machine use, whereas the short textual description (also called a reason phrase) is intended to be a human-readable representation of the status code. The HTTP specification defines forty individual status codes. In addition, an extension code is defined to mean that also additional status codes may be used to support future requirements. [24, pp. 39–41]

HTTP client programs are not required to understand the meaning of all individual status codes. Instead, client programs are required merely to understand the class of the response which is indicated by the first digit of the status code. The HTTP specification classifies response status codes into five classes: informational 1xx, successful 2xx, redirection 3xx, client error 4xx, and server error 5xx. [24, pp. 39–41] The class notation, e.g. 1xx is used to denote status codes ranging from 100 to 199, and so forth.

3.3.1. Response Status Code Classes

The informational class indicates a temporary response, meaning that the request was successfully received, and the processing is ongoing at the server side. In this class, the response message consists only of the status line and optional header fields. The client must always be prepared to accept one or more 1xx response messages prior to a regular response. The successful class indicates that the request from the client was successfully received, understood, and accepted by the server. The redirection class indicates that further action needs to be taken by the client in order to complete the request. If the required action is another HTTP request, the client may send the request without any user interaction. [24, pp. 40, 57–65]

The client error class indicates that the client has made an error e.g. syntax error in the request message or the request cannot be fulfilled. The server error class indicates that the server has failed. Either the server has made an error or the server is incapable of carrying out the valid request. In both the client error class and the server error class, the response message includes an explanation of the error situation and an indication whether it is a permanent or a temporary condition. [24, pp. 40, 65–71]
4. TEST ENVIRONMENT

This thesis work is done in an isolated test environment which allows simultaneous setup of several Hypertext Transfer Protocol (HTTP) servers, HTTP clients, and a code coverage measurement tool. An overview of the test setup is illustrated in Figure 6. The test setup comprises of three parts: a test suite, a sampling of different HTTP server software implementations used as test subjects, and a code coverage measurement tool used to generate measurement results for validation purposes. The complete test setup is built in an isolated test environment to minimise possible external influences.

![Figure 6. Overview of the test setup.](image)

The focus of this thesis is on the optimisation of the HTTP server test suite. All the test runs and code coverage measurements should be reproducible using the designed test setup in the depicted test environment. The only predefined part of the test setup is the test suite, Defensics HTTP Server Test Suite from Codenomicon Oy. It is a commercially available test suite designed for robustness and security testing of HTTP server implementations [34]. The test suite acts as an HTTP client in the test setup.

The setup is divided on two separate machines. The predefined test suite resides on a modern (Intel Core i5-540M processor) personal computer (PC) running a 64-bit version of Microsoft (MS) Windows 7 [35] operating system (OS). The HTTP servers and the code coverage measurement tool reside on a virtual machine (VM) running a 64-bit version of Fedora GNU (GNU’s Not Unix)/Linux OS version 16 (“Verne”) [36]. The VM runs in Oracle VM VirtualBox, a virtualisation software package [37]. The software version used in this thesis is Oracle VM VirtualBox 4.1.8.

The different parts of the test setup are further described as follows. A detailed explanation of the test suite is provided in Section 4.4. The selection criteria for test subjects (from here on referred to as SUTs) and details of all of the five HTTP server software implementations are described in Section 4.5. The selection criteria for the code coverage measurement tool and details of the chosen tool are discussed in Section 4.6.

4.1. Research Questions

The basic assumption which leads into the research questions is that software test suites consisting of thousands of unique test cases might involve redundant test cases which actually do not increase code coverage at all. Therefore, an unoptimised test suite might waste valuable testing time and consume resources unnecessarily. The losses add up, especially if the unoptimised test suite is used frequently e.g. in regression or nightly testing. In order to reduce wasted testing time and resources, the test suite should be optimised to contain the minimum number of test cases which are still able to retain the same level of code coverage as the unoptimised test suite.
As a detailed recapitulation paraphrased from Chapter 1, the goal of this thesis is to answer the following research questions.

1. Does the variation in the HTTP response messages (specifically, the response status code from the SUT) indicate which test groups should be tested more thoroughly, i.e. does the variation in the responses from the SUT highlight those test groups which actually exercise (cover the code in) the SUT?

2. Can the same level of code coverage (measured from the SUT) be achieved with a smaller amount of test cases, i.e. with an optimised test suite which does not contain those test groups (individual test cases or test groups) in which the responses from the SUT remain the same?

Answers to these research questions can only be obtained by firstly designing and creating a complete test setup in which both the test runs and the test run results are reproducible. Secondly, the optimisation problem in the context of the predefined test suite requires that a processing method is designed to enable the optimisation based on the test run results. The processing method is the actual implementation of the designed optimisation algorithm, which produces a proposal for an optimised test suite consisting of either a subset of test groups or test cases.

4.2. Terms of the Work

The test setup must incorporate a sampling of different HTTP server implementations to be used as SUTs. The sampling of the SUTs must also be compatible with the predefined test suite. The test setup must also be simultaneously apposite for code coverage measurement. The measurement of code coverage from the different SUTs should work independently, i.e. the measurement of code coverage should not affect the functionality of the selected SUTs.

The processing method must be applicable for use with the test run result files generated by the predefined test suite. The processing method should create an output (a proposal) which should be readily usable in the predefined test suite. The processing of the test run results with the designed method should be a completely independent process which should not affect the functionality of the test setup. The processing method is entirely founded on the optimisation of the test run results, which means that the fault detection abilities of the test suite are not to be taken into account.

4.3. Solution Choices of the Test Setup

Quite a few choices have to be made in order to construct a complete test setup. First of all, a sampling of suitable test subjects (SUTs) must be put together. To make the test run results in this research generally more valid, the sampling of the SUTs should comprise of a sufficient amount (e.g. five) of HTTP server implementations. The only predefined part of the test setup, the test suite, restricts the test subjects to servers which have HTTP capabilities. A downside in using merely one test suite is that the results are not as generally valid as if similar test run results were also available from other test suites.
These days there are numerous open source and commercial HTTP servers available. Open source implementations were selected instead of closed-source options as SUTs, because the open source implementations are available free of charge and their usage does not require going through any kind of evaluation or registration process. The choice to use only open source SUTs also has a clear disadvantage, because it rules out some notable SUT candidates such as MS Internet Information Services (IIS) [38].

The choice to use open source SUTs affects the selection of the OS to be used in the test setup as an installation platform. The same choice also affects the selection of the code coverage measurement tool. Open source SUTs will most likely require to be compiled with the compilation tools from the GNU Compiler Collection (GCC) [39], or they are at least compatible with the GCC. The GCC is used as a standard compiler in GNU/Linux OS, which was selected to be used in the test setup mainly for compatibility reasons.

The choice to use the GCC is supported by the fact that the GCC includes a code coverage measurement tool called gcov, which is also applicable for use in this thesis. The downside in using gcov is that it supports merely the measurement of line, branch, and function coverage instead of supporting the measurement of more complex code coverage metrics. [40, pp. 615–617]

The selection to use gcov as the code coverage measurement tool dictates the approach how the code is also instrumented in the SUTs during the test runs. Gcov instruments code at the binary level (explained in Section 2.3.2), i.e. at the object code level to be precise, instead of instrumenting at the source code level. This means that although code instrumentation is done at the binary level, access to the source code is required. [40, pp. 615, 617–618]

The source code of the SUT must be compiled using the compilers incorporated in the GCC in order to be able to use the gcov tool [40, p. 615]. The requirement for source code access is a drawback in the use of gcov. An alternative would be to use a different tool, e.g. a code coverage measurement tool that works entirely at the binary level without the need for the source code access.

4.4. Test Suite Analysis

Defensics is a commercially available test platform from Codenomicon Oy that provides model-based fuzz testing suites for over 200 different communication protocols. The test platform provides fully automated and unattended testing of security and robustness of any supported protocol implementation. [34] Defensics is written in the Java [41] programming language, and is compatible with MS Windows 7 and Community ENTerprise Operating System (CentOS) Linux [42] OSs [43]. The software version used in this thesis is Codenomicon Defensics 10.1.15.

HTTP Server Test Suite is a commercially available test suite for Defensics. HTTP Server Test Suite is model-based test suite for automated black box negative testing purposes. The test suite can be used for robustness and security testing of Hypertext Transfer Protocol (HTTP) server implementations in an isolated laboratory environment. HTTP Server Test Suite complies with the following IETF recommendations: the HTTP/1.0 specification RFC 1945 [26] and the HTTP/1.1 specification RFC 2616.
The software version used in this thesis is HTTP Server Test Suite 3.1.0.

The robustness test cases (negative tests) in HTTP Server Test Suite are derived from the protocol specifications with the method described in Section 2.1.2. The test cases systematically break all the elements and structures defined in the HTTP protocol specifications. The fact that any protocol specification allows an infinite number of negative test cases (further explained in Chapter 2) is resolved by compiling only an intelligently chosen subset of all the possible negative test cases. [45, p. 2]

### 4.4.1. Structure of the Test Suite

HTTP Server Test Suite is designed to act as a malicious HTTP client, which sends malformed or invalid HTTP request messages to the SUT. Each of the automatically produced malformed (anomalous) HTTP requests is considered as a single test case. The test suite involves ten different kinds of HTTP message sequences, from which the most commonly used sequence is selected to be used in this thesis. [46]

The chosen message sequence is named *HTTP request and receive response*, and it is portrayed in Figure 5. In this message sequence, the SUT may or may not respond to the anomalous HTTP request message received from the test suite. [46] The selection to use merely one message sequence is supported by the fact that all of the SUTs do not support other message sequences than *HTTP request and receive response*. Therefore, they may not interoperate with the test suite if other sequences are used.

The structure of the test suite is depicted in Figure 7. All in all, the test suite consists of 673 331 unique test cases (i.e. anomalous HTTP request messages) when the chosen message sequence is used with the default settings related to the test case selection and to the automatic test case generation process. The test cases are grouped into 809 distinct test groups, where a single test group contains test cases with different kinds of anomalies for each specific part of the HTTP request message. In addition to all of the anomaly test cases, the test suite includes also valid (i.e. error-free) test cases for testing and verification of the interoperability with the SUT.

![Figure 7. Structure of the test suite.](image)

An example of a valid test case is represented in Listing 1, and an example of an anomalous HTTP request message (i.e. a test case) from the test suite (which acts as a client) to a SUT (which acts as a server) is presented in Listing 3. The example test case incorporates an overflow anomaly which is located on the request line, and the actual anomaly is a repetition of eight octets involving "HTTP".
By default, all test cases (i.e. all test groups) from the test suite are selected to be executed, but the selection of test cases and test groups is user-definable in the following two conventions. The first way is to simply define one or more test case indices and separate every index with a comma. In this convention, the test suite will execute the user-specified test cases only. The latter way is to define one or more test group names and separate each test group name with a comma. The test suite will execute all of the test cases from the user-specified test groups. [44]

4.4.2. Instrumentation Method

Defensics supports several different instrumentation methods, which can be used to monitor the status of a SUT, i.e. whether it failed or not during a test run. These methods include TCP connection-based instrumentation, valid case instrumentation, Simple Network Management Protocol (SNMP) instrumentation, and external instrumentation using system commands and user-defined scripts.

As stated in Chapter 3, HTTP is a TCP connection-oriented protocol. Therefore, TCP connection-based instrumentation was chosen as the instrumentation method for all of the test runs in this thesis. This means that the verdict for the test case is founded on the functionality of the TCP connection, i.e. a working TCP connection results as pass verdict for the test case, and vice versa.

4.4.3. Test Run Result Files

In addition to summary information and detailed message data, Defensics saves test run information into a statistics.csv file in comma-separated values (CSV) format during a test run. An example of a statistics.csv file from a test run where a total of four test cases have been executed is represented in Listing 4. The first line contains the column names for the data saved into the CSV file. Each of the remaining lines represents a single test case. All of the values are separated by commas, and the lines are separated by line feed (LF) characters.
The relevant values for this thesis are found under the following titles: `test-group`, `index`, and `status`. Test-group incorporates the name of a test group for a test case in question, e.g. the name of the test group is `http-suite.valid` for the first test case which shown in line 2. Index displays the test case index value, e.g. 0 for the first test case. Status contains the response status code received from the SUT during the execution of the particular test case, e.g. the value of the response status code is 200 in the first test case. The status value is empty if the SUT does not respond at all.

### 4.5. Test Subject Selection

In order to make the results of this thesis generally as valid as possible, the test subjects were primarily selected based on their popularity and availability. The four most popular HTTP server software implementations according to the Netcraft December 2012 web server survey are listed in Table 1 [47]. Both the most popular, Apache httpd [48], and the third most popular, Nginx [49, p. 1], HTTP server software implementations are open source and thus freely available.

The most prominent commercial HTTP server is Microsoft IIS, which runs on an estimated 17.6 percent of all Internet servers. Google Web Server (GWS) was not even considered as a test subject as it is Google’s proprietary server software. This means that it is not available to the general public at all, although it runs on an estimated 3.4 percent of all Internet servers.

The Netcraft December 2012 web server survey is lacking a detailed account of the rest of the 11.1 percent of the web servers, although in an earlier Netcraft survey the fifth most popular HTTP server software was Lighttpd [50]. The market share of Lighttpd is quite marginal in comparison to the four more popular ones.
<table>
<thead>
<tr>
<th>Name of the HTTP Server</th>
<th>Number of websites</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd</td>
<td>352,951,511</td>
<td>55.7%</td>
</tr>
<tr>
<td>MS IIS</td>
<td>111,570,010</td>
<td>17.6%</td>
</tr>
<tr>
<td>Nginx</td>
<td>76,460,756</td>
<td>12.1%</td>
</tr>
<tr>
<td>GWS</td>
<td>21,870,614</td>
<td>3.5%</td>
</tr>
<tr>
<td>Others</td>
<td>70,853,673</td>
<td>11.1%</td>
</tr>
</tbody>
</table>

A total of five different HTTP server software implementations were selected as SUTs. The following information is listed for all the chosen SUTs: name of the HTTP server implementation, implementation language, support for and compatibility with different OSs, usage statistics if they are available, software licensing model, software version, and the TCP port number assigned to the SUT.

All of the five SUTs fulfil the selection criteria described below. The last three criteria are highly influenced by the chosen code coverage measurement tool, which is further discussed in Section 4.6, and by the selected OS of the test setup, which is explained in the beginning of Chapter 4.

- The HTTP server software (from here on referred to as the SUT) is being developed actively.
- The SUT’s latest release version from the stable branch is not older than one year.
- The SUT is open source software, i.e. the source code of the SUT is available free of charge.
- The SUT is preferably implemented in either the C or the C++ programming language.
- The SUT supports GNU/Linux OS.

TCP ports ranging from 8075 to 8079 were assigned to the SUTs in the order in which they are presented in the following Sections. According to the Internet Assigned Numbers Authority (IANA) registry, the selected port numbers are not assigned to any specific service [51]. Therefore, the chosen ports can be used in this thesis.

### 4.5.1. Nginx

**Nginx** (pronounced “engine-x”) is a high performance HTTP server initially written by Igor Sysoev in the C programming language. Nginx is available for GNU/Linux, FreeBSD, MS Windows, and other modern OSs. [49, pp. 1, 48, 54] Some heavily loaded websites such as WordPress and Foursquare use Nginx as web server software [52]. According to the Netcraft December 2012 web server survey, Nginx runs on an estimated 12.1 percent of all Internet servers [47].

Nginx is licensed under the 2-clause BSD-like (Berkeley Software Distribution) license [53]. The software version used in this thesis is Nginx 1.0.15.
4.5.2. Lighttpd

*Lighttpd* (pronounced “lighty”) is a high performance HTTP server originally written by Jan Kneschke [54] in the C programming language. The name was initially spelled as LightTPD, an abbreviation of the words “light” (footprint) and “httpd” (Hypertext Transfer Protocol Daemon). Lighttpd is available for GNU/Linux, FreeBSD, and other modern OSs. [55, pp. 1, 7–10]

Some heavily loaded websites such as YouTube and Wikimedia use Lighttpd as web server software [55, p. 7]. According to the Netcraft March 2011 web server survey, Lighttpd is the fifth most popular HTTP server software, although its market share is marginal in comparison to the more popular ones [50].

Lighttpd complies with the following IETF recommendations: the HTTP/1.0 specification RFC 1945 [26] and the HTTP/1.1 specification RFC 2616 [24]. Lighttpd is not compliant with the chunked POST request messages defined in the RFC 2616. [56] Lighttpd is licensed under the revised BSD license [57]. The software version used in this thesis is Lighttpd 1.4.30.

4.5.3. Tntnet

*Tntnet* is a modular HTTP server originally written by Tommi Mäkitalo in the C++ programming language [58]. It is used for hosting web applications written in a specific template language called ecpp, which enables embedding C++ code inside web pages [59]. Tntnet supports Portable Operating System Interface (POSIX) compatible OSs [58]. The market share of Tntnet is arguably minimal, since it has not been noted in the web server popularity surveys.

Tntnet complies only with the GET and POST request methods from the following IETF recommendations: the HTTP/1.0 specification RFC 1945 [26] and the HTTP/1.1 specification RFC 2616 [24]. [59] Tntnet is licensed under the GNU Lesser General Public License (LGPL) [58]. The software version used in this thesis is Tntnet 2.1.

4.5.4. Hiawatha

*Hiawatha* is a lightweight and highly secure HTTP server written by Hugo Leisink in the C programming language. Hiawatha is available for GNU/Linux, FreeBSD, and MS Windows OSs. [60], [61] The market share of Hiawatha is arguably minimal, since it has not been noted in the web server popularity surveys.

According to personal communication with Hugo Leisink, Hiawatha is for the most part, but not 100 percent, compliant with the following IETF recommendations: the HTTP/1.0 specification RFC 1945 [26] and the HTTP/1.1 specification RFC 2616 [24]. In addition, Hiawatha complies with the URI generic syntax specification RFC 2396 [62]. Hiawatha is licensed under the GNU General Public License version 2 (GPLv2) [63]. The software version used in this thesis is Hiawatha 8.1.
4.5.5. Apache httpd

The Apache HTTP Server, from here on referred to as Apache httpd, is a powerful and flexible HTTP server written in the C programming language [64], [65]. Apache httpd was initially a series of “patches” written by the Apache Group members using Rob McCool’s NCSA (National Center for Supercomputing Applications) httpd software version 1.3 as a codebase. The server architecture has since been completely redesigned and re-implemented. [66]

Apache httpd has been the most popular HTTP server on the Internet since 1996. Apache httpd runs on most versions of UNIX, MS Windows, and other modern OSs. [48] According to the Netcraft December 2012 web server survey, Apache httpd runs on an estimated 55.7 percent of all Internet servers [47].

Apache httpd complies with the following IETF recommendations: the HTTP/1.0 specification RFC 1945 [26] and the HTTP/1.1 specification RFC 2616 [24]. In addition, Apache httpd complies with the URI generic syntax specification RFC 2396 [62]. [67] Apache httpd is licensed under the Apache License version 2.0 [64]. The software version used in this thesis is Apache httpd 2.4.1.

4.5.6. Other Candidates for Test Subjects

All other HTTP server software implementations that were considered as possible candidates for test subjects were ruled out based on the selection criteria described in Section 4.5. All closed-source HTTP server software implementations were rejected, most notable among them being MS Internet Information Services (IIS), and Oracle HTTP Server (OHS), whose popularity has drastically diminished in recent years according to Netcraft December 2012 web server survey [47]. In addition to the five chosen SUTs, all of the open source HTTP server software implementations that were considered for test subjects are shown in Table 2.

Table 2. Open source HTTP server software and their implementation languages

<table>
<thead>
<tr>
<th>Name of the HTTP Server</th>
<th>Implementation Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLone Embedded Web Server</td>
<td>C</td>
</tr>
<tr>
<td>Caudium [68]</td>
<td>C and Pike [69]</td>
</tr>
<tr>
<td>Cherokee Web Server [70]</td>
<td>C, Python and JavaScript</td>
</tr>
<tr>
<td>node.js [71]</td>
<td>C++ and JavaScript</td>
</tr>
<tr>
<td>Tornado [72]</td>
<td>Python</td>
</tr>
<tr>
<td>Yaws [73]</td>
<td>Erlang [74]</td>
</tr>
<tr>
<td>Apache Tomcat [75]</td>
<td>Java</td>
</tr>
<tr>
<td>Jetty [76]</td>
<td>Java</td>
</tr>
</tbody>
</table>

KLone Embedded Web Server was considered for a test subject but it was eventually rejected because it does not support the chosen code coverage measurement tool [77]. Similarly, the following HTTP server software implementations were rejected because they are implemented either entirely or partially in programming languages which are not supported by the selected code coverage measurement tool: Caudium, Cherokee Web Server, node.js, Tornado, and Yaws.
The supported languages of the chosen code coverage measurement tool are further explained in Section 4.6. Apache Tomcat and Jetty server software were also considered but were not selected as test subjects due to the preferred programming languages discussed in Section 4.5.

### 4.6. Code Coverage Measurement Tool Selection

The chosen code coverage measurement tool fulfils the selection criteria described below. The last two criteria are highly influenced by the selected OS of the test setup, which is further described in the beginning of Chapter 4.

- The code coverage measurement tool (from here on referred to as the tool) is being developed actively.
- The tool’s latest release version from the stable branch is not older than one year.
- The tool is preferably open source software.
- The tool supports GNU/Linux OS.

#### 4.6.1. GNU Compiler Collection (GCC)

A code coverage measurement tool named gcov was chosen from the GNU Compiler Collection (GCC). The GCC consists of a full tool chain for compiling software implemented in the following programming languages: C, C++, Java, etc. The GCC is licensed under the GNU General Public License version 3 (GPLv3). [40, pp. 3, 649] The software version used in this thesis is the GCC 4.6.3, which means that the software version of the gcc compiler, the g++ compiler, and the gcov tool is 4.6.3 for all of them.

The relevant compilers in this thesis are the C compiler, gcc, and the C++ compiler, g++, as they fulfil the programming language criterion explained in Section 4.5. The gcc compiler supports different C dialects, including the original American National Standards Institute (ANSI) C standard (X3.159-1989) [78], and different corrected editions of the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) C standard [79]. The g++ compiler supports ISO C++ standard. [40, pp. 3, 5–6]

#### 4.6.2. Gcov

In addition to the compilers, the GCC incorporates gcov, a tool for measuring code coverage. Gcov requires that the source code of the SUT is compiled with the GCC in order to perform code coverage measurements which mean that the source code of the SUT must also be available. Gcov supports measurement of line, branch, and function coverage. [40, pp. 615–617]

The gcov tool is used in four different phases: [80, p. 278]
1. Instrumentation of the SUT’s object code during the software compilation process.

2. Code coverage data collection during the execution of the SUT.

3. Code coverage data extraction when the execution of the SUT finishes.

4. Code coverage analysis and presentation at the post mortem phase.

In order to enable code coverage measurements in the first phase, the source code of the SUT must be compiled with the following two GCC options for CFLAGS:

```
-fprofile-arcs -ftest-coverage
```

These two options tell the compiler to generate and involve additional information into the object files during the software compilation process, and to create the following files needed by gcov: data (.gcda files) and note (.gcno files) files for each of the source code files. [40, pp. 76–77, 617–621]

The code instrumentation is done at the object code level during the software compilation process (as discussed in Section 2.3.2), which means that gcov does not modify the source code in any way. The instrumented object code must be linked with the following GCC option for LDFLAGS: [40, pp. 76–77, 617–621]

```
-lgcov
```

In the second phase, the code coverage data is collected during the execution of the SUT. The execution of the SUT can be done in concert with a test suite. In the third phase, gcov requires that the execution of the SUT stops and the SUT exits successfully before the code coverage data files (.gcda files) can be produced. In the final phase, the code coverage data is integrated into a textual representation (.gcov files) which can be used in further analysis. [40, pp. 615–621]

### 4.6.3. Lcov

The lcov tool is an extension of the gcov tool that generates a graphical representation from the textual output created by gcov. Lcov was originally developed by the Linux Test Project (LTP) for Linux kernel code coverage measurements, but it can also be used for code coverage measurements on standard user space software. The extension is essentially a set of scripts written in the Perl [81] programming language. The scripts produce HTML output with different views for source code, file, and directory levels. [82] The software version used in this thesis is lcov 1.9.

The use of lcov for code coverage measurements is illustrated in Figure 8. Lcov first calls the gcov tool to generate the code coverage data files (.gcda files) for all of the source code files. Then lcov calls a Perl script (geninfo.pl) to create a single code coverage data file (.info file) which represents the code coverage data. And finally, lcov calls another Perl script (genhtml.pl) to produce a multi-page HTML output, which represents the collected code coverage data. [80, pp. 284–285]
4.6.4. Other Candidates for the Code Coverage Measurement Tool

All other code coverage measurement tools that were considered to be used in this thesis were ruled out based on the criteria described in Section 4.6. An open source tool entitled xCover [83] was considered but eventually rejected because the compilation of the different versions all failed, and the tool is also outdated.

In addition to the selected tool, the coverage measurement features of all of the commercial coverage tools that were considered are shown in Table 3. The surveyed coverage tools are Dynamic Code Coverage (DCC) [84], BullseyeCoverage [85], Liverpool Data Research Associates (LDRA) Testbed [86, pp. 26–27], TCAT C/C++ [87], Testwell CTC++ [88], and Squish Coco [89]. Dynamic Code Coverage was considered but rejected because it supports only Solaris [90] OS, and the tool is also outdated.

Table 3. Code coverage measurement tools and their coverage support

<table>
<thead>
<tr>
<th>Tool</th>
<th>line</th>
<th>branch</th>
<th>function</th>
<th>decision</th>
<th>condition</th>
<th>C/DC</th>
<th>MC/DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCC</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bullseye</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LDRA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TCAT</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTC++</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Squish</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The summary shows that LDRA Testbed and the DCC tools support all the same coverage metrics that the selected gcov tool does, and even a few more. The most extensive support for coverage measurement is found from LDRA Testbed, Testwell CTC++, and Squish Coco tools. In addition to the features mentioned, Testwell CTC++ tool supports Multicondition Coverage (MCC). Conversely, TCAT C/C++, and Bullseye-Coverage tools support only less than three types of different coverage metrics.

Feature-wise, a tool that supports complex coverage metrics such as Condition/Decision Coverage (C/DC) or Modified Condition/Decision Coverage (MC/DC) would
be an ideal choice, as these complex metrics give a better insight to the quality of
the testing than simpler ones. E.g. the MC/DC coverage criterion is required for
the testing of the most critical (Level A in the DO-178B document) software used in
aviation systems [91, pp. 1, 7].

The commercially available code coverage measurement tools were ruled out in
favour of an open source tool (i.e. gcov). The advantage in selecting the gcov tool
instead of a closed-source tool is that the tool is available free of charge and the usage
does not require going through any kind of evaluation or registration process. The gcov
tool is considered to be sufficient for use in this thesis, although it lacks the support
for the more complex metrics. The gcov tool supports the measurement of line, branch
and function coverage only.
The test run results made with the complete setup are processed using the designed optimisation algorithm in order to be able to answer to the research questions discussed in Section 4.1. The designed optimisation method is based on the processing of the test run results containing the response status codes received from the SUT in every test case.

The fact that the optimisation method is founded merely on the processing of the response status codes is a disadvantage in comparison to more advanced optimisation methods, which also take into account that the fault detection abilities of the test suite should not weaken due to the test suite optimisation. A benefit in the designed optimisation method is the fact that it can also be applied to test run results from other applicable protocols, i.e. request/response protocols in which the response messages incorporate status codes. One example of such an applicable protocol could be e.g. the Session Initiation Protocol (SIP) [92, pp. 28–29].

The designed optimisation method is almost completely independent from the test setup (explained in Chapter 4), because the optimisation method is not specific to any SUT or code coverage measurement tool. The advantage is that the optimisation method can also be applied at an abstract level to test run results generated with an entirely different kind of a test suite.

Another alternative solution would be a more advanced optimisation method, in which the optimisation would be based simultaneously on the processing of the test run results and on the code coverage measurement results. This alternative solution requires a test setup where the code coverage is measured separately for each test case, and the precise measurement results of each test case are submitted to the advanced optimisation method.

5.1. Description of the Optimisation Methods

In order to answer the research questions (described in Section 4.1), three alternative optimisation algorithms were designed and created to solve the test suite optimisation problem. The basis for the development of the optimisation algorithms was that the original test suite is presumed to always contain redundant test cases in which the code coverage does not increase.

All three optimisation algorithms are completely founded on the processing of the test run results produced by the predefined test suite. This is due to the basic assumption that a change in the response status code received from the SUT actually implicates that the SUT was exercised and presumably a different part of the SUT’s code was covered.

Conversely, the same assumption affects the design of the developed optimisation algorithms. All of the algorithms assume that the SUT does not exercise and cover new parts of the code in a scenario where two or more consecutive HTTP response messages contain the same response status code. Therefore, the optimisation algorithms leave aside the test cases in which the response status code remains unaltered from the second test case onwards.

Another key factor in the design phase of the optimisation algorithm was the struc-
ture of the test suite (i.e. the test suite consists of test groups, which consist of test cases), which served as the basis for how the subset (i.e. the actual result of the optimisation) is formed by the algorithm. All of the three designed algorithms differ from each other in the way how the subset of either test groups or test cases is selected by the algorithm.

The first algorithm generates a proposal for an optimised test suite as the result of the optimisation, and the proposal consists of a subset of test groups only. The latter two algorithms create proposals for optimised test suites comprising of subsets of test cases. The optimisation result in all three algorithms is represented in a format which is directly usable in the predefined test suite (as described in Section 4.4.1).

The contents of the test run result files are processed in sequential order, i.e. a result of one test case at a time, and a test group after another. The particular test run result file to be processed is the statistics.csv file, which is further explained in Section 4.4.3. Because the predefined test suite consists of test cases which are grouped into test groups, the optimisation is done by reducing either the number of selected test cases or the number of chosen test groups in the given test suite.

The three optimisation algorithms and their differences are further discussed in Sections 5.1.1, 5.1.2, and 5.1.3. The actual implementation of all three developed optimisation algorithms was done in the Python [93] programming language. The software version used in this thesis is Python 2.6.5. The implementation of the optimisation algorithms is presented in Appendix 1.

5.1.1. Test Suite Optimisation Algorithm No. 1

The idea behind the first algorithm is that a majority out of the 40 or more possible and distinct response status codes (further explained in 3.3) are presumably received during the execution of different test groups. This assumption was taken based on the fact that the test suite consists of test groups which are all meant for fuzzing different parts of the HTTP request message (as discussed in 4.4.1).

The first optimisation algorithm focuses on a single test group at a time. During the optimisation, the algorithm checks if both the same response status code has already been received in an earlier test group, and whether the name of the test group is already included in the result list. This means that the algorithm presumes that a single response status code is covered simply by executing a single test group, and all the other test cases in which the same response status code is received from the SUT are completely ignored.

A pseudo-code presentation of test suite optimisation algorithm No. 1 is shown in Algorithm 1, and the actual implementation of the algorithm is provided in Appendix 1. As a result of the optimisation, the algorithm produces an optimisation proposal consisting of test group names only.

The optimisation proposal is a result list in which the members (the names of the test groups) have been selected based on whether the response status code and the test group are both new for the test case in question. A single test group is considered to be interesting optimisation-wise (i.e. chosen by the algorithm to be included into the resulting subset suite) if and only if both the response status code is new and the name of the test group is not already included in the result list. If the response status code is
known or the name of the test group is already included in the result list, the algorithm
presumes that the same effect on the SUT is covered with a test group that has already
been chosen.

Algorithm 1: Pseudo-code of test suite optimisation algorithm No. 1

Data: A statistics.csv file from a test run.
Result: A list of test groups.

1 begin
2     the result list = ∅;
3     the temporary list = ∅;
4     while not at the end of the statistics.csv file do
5         read next line from the statistics.csv file;
6         if (value of status /∈ the temporary list) and (name of test − group /∈
7             the result list) then
8             the value of status ∪ the temporary list;
9             the name of test − group ∪ the result list;
10         end if
11     end while
12     return the result list
13 end

The input data for the algorithm is a statistics.csv file from the test run result files
generated in the first phase, as described in Section 5.3. In the beginning of the algo-

rithm, two empty lists are declared, one for the result list and the other for a temporary
list. The content of statistics.csv file is read one line at a time until all the lines have
been read and processed by the algorithm. Every line that is read contains values for
status and test-group of the test case in question, as explained in Section 4.4.3. An
empty value for status indicates that the SUT did not respond at all during the test case
in question, and no response is considered to be one type of response status code by
the algorithm.

After reading one line, the algorithm checks whether the value of status is already
included in the temporary list, and whether the value of test-group is already involved
in the result list. If neither of the if-statements is true, the value of status is appended
to the temporary list, and the value of test-group is appended to the result list. After all
the lines from the statistics.csv file have been read and processed, the algorithm returns
the definitive result list.

An example of the algorithm’s output, i.e. an example result of the optimisation is
presented in Listing 5. The first line indicates the number of different instances of test-
group from which the optimisation result comprises of. The second line incorporates
the names of the test groups represented in the test group selection convention (as
described in Section 4.4.1). This means that the optimisation result can be directly fed
to the predefined test suite in order to execute the optimised subset suite.
Optimised test groups contain 2 items which are:

```
```

Listing 5. An example of the output of algorithm No. 1.

### 5.1.2. Test Suite Optimisation Algorithm No. 2

The idea behind the second optimisation algorithm is to focus on the test cases in a single test group at a time, because the predefined test suite consists of test groups, each of which is meant for fuzzing a different part of the HTTP request message (as explained in Section 4.4.1). The changed focus in comparison to the first algorithm means that knowledge about the response status codes in other test groups is completely ignored. During the optimisation, the algorithm checks if the response status code has already been received with an earlier test case in the test group in question. This means that the optimisation result might consist of several test cases (although from different test groups) in which the response status code received from the SUT is in fact the same.

A pseudo-code presentation of test suite optimisation algorithm No. 2 is shown in Algorithm 2, and the actual implementation of the algorithm is presented in Appendix 1. As a result of the optimisation, the algorithm produces an optimisation proposal comprising of test case indices only. Although the next algorithm and algorithm No. 2 create similar kinds of optimisation results, it is noteworthy that the design for this algorithm is a bit more complex.

The optimisation proposal is a result list in which the members (the indices of test cases) have been selected based on whether the response status code for the test case is new in the test group. A single test case is considered to be interesting optimisation-wise (i.e., chosen by the algorithm to be included into the resulting subset suite) if and only if the response status code has not been received in the previous test cases of the test group in question. If the response status code is already known, the algorithm presumes that the same effect on the SUT is covered with a test case which was earlier selected from the same test group.
Algorithm 2: Pseudo-code of test suite optimisation algorithm No. 2

**Data:** A *statistics.csv* file from a test run.

**Result:** A list of test cases.

```
begin
  the result list = ∅;
  while not at the end of the *statistics.csv* file do
    read next line from the *statistics.csv* file;
    if value of test-group ≠ value of current-group then
      set the value of test-group as the value of current-group;
      the temporary list = ∅;
    end if
    if value of status ∈ the temporary list then
      the value of index ∪ the result list;
    end if
    the value of status ∪ the temporary list;
  end while
return the result list
end
```

The input data for the algorithm is a *statistics.csv* file from the test run result files produced in the first phase, as discussed in Section 5.3. In the beginning of the algorithm, an empty result list is declared. The content of statistics.csv file is read one line at a time until all the lines have been read and processed by the algorithm. Each read line contains values for status, test-group, and index of the test case in question, as described in Section 4.4.3. An empty value for status indicates that the SUT did not respond at all during the test case in question, and no response is considered to be one type of response status code by the algorithm.

After reading one line, the algorithm examines whether the value of test-group is the same as the value of current-group. If it is not equal, the value of test-group is set as the value of current-group, and an empty temporary list is declared. Then, the algorithm checks whether the value of status is already included in the temporary list. If the outcome of the if-statement is false, the value of index is appended to the result list. Regardless, the value of status is appended to the temporary list. After all the lines from the statistics.csv file have been read and processed, the algorithm returns the definitive result list.

An abridged example of the algorithm’s output, i.e. an example result of the optimisation is represented in Listing 6. The first line indicates the number of different instances of index from which the optimisation result comprises of. The second line incorporates the indices of the test cases presented in the test case selection convention (as described in Section 4.4.1). This means that the optimisation result can be directly fed to the predefined test suite in order to execute the optimised subset suite. A part of the second line has been omitted on purpose to shorten the example listing.
Optimised test cases contain 839 items which are:
0, 1, 19, 30, 428, [SNIP], 672379, 672509, 672564, 673325, 673326

Listing 6. An example of the output of algorithm No. 2.

5.1.3. Test Suite Optimisation Algorithm No. 3

The idea behind the third algorithm is to completely ignore knowledge about the current test group for the test case in question, and to focus on single test cases only. This assumption was made based on the fact that the test suite consists of only 809 distinct test groups, whereas there is a far greater number of test cases (673 331) which means that optimisation-wise it might be better to make the optimisation decisions based on the results attained with the larger scope (all the test cases instead of one group at a time). During the optimisation, the algorithm pays attention to the variation of the response status codes in between two consecutive test cases only. This means that the optimisation result might consist of several test cases in which the response status code is in fact the same.

A pseudo-code presentation of test suite optimisation algorithm No. 3 is shown in Algorithm 3, and the actual implementation of the algorithm is presented in Appendix 1. As a result of the optimisation, the algorithm generates an optimisation proposal comprising of test case indices only. Although the previous algorithm and algorithm No. 3 create similar kinds of optimisation results, it is noteworthy that the designs differ and the design for this algorithm is a bit simpler.

The optimisation proposal is a result list in which the members (the indices of test cases) have been selected based whether the response status code received from the SUT in the test case has changed in comparison to the response status code received in the preceding test case. A single test case is considered to be interesting optimisation-wise (i.e. chosen by the algorithm to be included into the resulting subset suite) if and only if the response status code altered between the test cases in question. If the response status code remains the same, the algorithm presumes that the test case had no effect on the SUT, and hence the test case did not exercise it.

**Algorithm 3**: Pseudo-code of test suite optimisation algorithm No. 3

**Data**: A statistics.csv file from a test run.  
**Result**: A list of test cases.

1 begin
2 the result list = ∅;
3 while not at the end of the statistics.csv file do
4 read next line from the statistics.csv file;
5 if value of status ≠ value of the status of the previous test case then
6 the value of index ∪ the result list;
7 end if
8 set value of status as the value of the status of the previous test case;
9 end while
10 return the result list
11 end
The input data for the algorithm is a *statistics.csv* file from the test run result files produced in the first phase, as explained in Section 5.3. In the beginning of the algorithm, an empty *result* list is declared. The content of *statistics.csv* file is read one line at a time until all of the lines have been read and processed by the algorithm. Every read line contains values for *status*, *test-group*, and *index* of the test case in question as discussed in Section 4.4.3. An empty value for *status* indicates that the SUT did not respond at all during the test case in question, and *no response* is considered to be one type of response status code by the algorithm.

After reading one line, the algorithm examines whether the value of *status* has changed in comparison to the value of the status of the previous test case. If the outcome of the if-statement is true, the value of *index* is appended to the *result* list. Regardless, the value of *status* is set as the value of previous test case’s status. After all the lines from the *statistics.csv* file have been read and processed, the algorithm returns the definitive *result* list.

An abridged example of the algorithm’s output, i.e. an example result of the optimisation is presented in Listing 7. The first line indicates the number of different instances of *index* from which the optimisation result consists of. The second line includes the indices of the test cases presented in the test case selection convention (as described in Section 4.4.1). This means that the optimisation result can be directly fed to the predefined test suite in order to execute the optimised subset suite. A part of the second line has been omitted on purpose to shorten the example listing.

```
1 Optimised test cases contain 158 items which are:
2 0, 19, 22, 30, 32, [SNIP], 658877, 658878, 668779, 668781, 673326
```

Listing 7. An example of the output of algorithm No. 3.

### 5.2. Test Setup Control Script

In addition to the three designed and implemented optimisation algorithms, a Bourne Again Shell (bash) [94] script was generated to remove the manual work of the following repetitive tasks in controlling the test setup:

- Start all of the SUTs concurrently.
- Shutdown all of the SUTs concurrently.
- Create code coverage measurement reports to a specified directory from coverage data files of all of the SUTs.
- Reset all of the counters from the coverage data files of all of the SUTs.
- Delete the produced code coverage measurement reports from a specified directory.

The bash script is presented in Appendix 2. The software version used in this thesis is GNU bash 4.2.24.
5.3. Practical Trial

In order to answer the research questions (described in Section 4.1) and to validate these answers, a practical trial was done by using all three of the designed optimisation algorithms. The practical trial was done in the following four phases.

In the first phase, normal test runs using the original test suite are performed against the five selected SUTs in the defined isolated test environment. The chosen message sequence for the regular test runs, and the test case count of the test suite are further explained in Section 4.4.1. The first phase produces altogether five test run results, i.e. one test run result for each of the SUTs.

During the normal test runs, all of the received HTTP response status codes are stored by the predefined test suite to the test result files, as discussed in Section 4.4.3. During this black box testing process, code coverage is measured simultaneously from all of the SUTs. The code coverage is measured in the four steps described in Section 4.6.2.

In the second phase of the practical trial, each of the test run results generated in the first phase are processed. The processing is done by applying the three different optimisation algorithms (further explained in Sections 5.1.1, 5.1.2, and 5.1.3) to the test run result files. Each one of the algorithms creates a separate proposal for an optimised test suite as the result of the optimisation phase. Altogether, the second phase produces 15 separate optimised subset suites out of the five test run results which were acquired in the first phase. An optimised test suite consists of a subset of test groups or test cases of the predefined test suite.

In the third phase, regular test runs are performed against the same five SUTs in the same test environment using the 15 distinct subsets of the original test suite that were produced in the second phase. All of the received HTTP response status codes are stored, and the code coverage is measured in a similar fashion to the first phase. The third phase produces altogether 15 code coverage measurement results, i.e. one coverage result from each of the test runs which were executed using the optimised subset suites.

The code coverage measurement results are comparable against the code coverage measurement results generated in the test runs performed in the first phase. The code coverage measurement results from both the test runs using the original test suite (in the first phase) and from the test runs using the subset test suites performed in the third phase are presented in Chapter 6.

5.3.1. Result Validation

In the fourth and final phase, the answers to the research questions are validated. In order to validate whether the designs of the developed optimisation algorithms are considered successful or unsuccessful, an acceptance criterion is set. The design of the algorithm is considered successful if the code coverage measurement result attained with an optimised subset test suite meets the acceptance level.

The acceptance criterion is set to 5 percent of coverage, which is measured as the difference of the coverage level attained with the optimised subset suite and the coverage level acquired with the original test suite. The acceptance criterion allows a
moderate reduction in the level of code coverage due to the nature of the developed algorithms, i.e. the optimisation is done completely based on the processing of the test run results (as described in the beginning of Chapter 5) instead of using a more sophisticated code coverage measurement method as an aid in the optimisation phase.

In an acceptable optimisation result, the level of attained code coverage is allowed to diminish no more than 5 percent in comparison to the coverage acquired with the original suite. If the code coverage achieved with an optimised subset suite fulfils the criterion, then the design of the optimisation algorithm is considered successful. On the other hand, if the code coverage level diminishes more than 5 percent, the optimisation result does not fulfil the acceptance criterion. In that case, the design of the optimisation algorithm is considered unsuccessful.

The answers to the research questions are validated with the aid of the acceptance criterion, which means that the answers to both questions are validated using code coverage analysis. In the fourth phase of the practical trial the results of the code coverage measurement stored in both the normal test runs performed with the original test suite (in the first phase of the practical trial) and the regular test runs performed with the optimised subset suites (in the third phase) are compared, and conclusions are drawn from the measures using elementary code coverage analysis.

The answer to the first research question is validated as follows. The variation in the response status codes in HTTP response messages indicates that the test case or test group actually exercises the SUT, if an optimised subset test suite is able to meet the acceptance level.

If the acceptance criterion is fulfilled by a code coverage measurement result attained using an optimised test suite, a conclusion is drawn that the alteration in the status codes indicates which test cases exercised the SUT more thoroughly and were able to cover more code in it. On the other hand, if the optimised subset test suite is unable to meet the acceptance level, an evident conclusion cannot be drawn about whether the change in the status codes indicates which test cases or test groups actually exercise the SUT.

The answer to the second research question is validated as follows. The same level of code coverage is achievable with an optimised test suite (i.e. a subset suite which does not contain those individual test cases or test groups in which the response status codes received from the SUT remain the same), if an optimised subset suite is able to meet the acceptance level.

If the code coverage measurement result achieved using an optimised subset test suite fulfils the acceptance criterion, a conclusion is drawn that it is possible to attain the same level of code coverage with a smaller amount of test cases (i.e. subset suite). On the other hand, if the code coverage result acquired with the subset suite does not meet the acceptance level, the conclusion is contrary. The code coverage analysis and the validation phase are further described in Chapter 7.
6. TEST RESULTS

In the first phase of the practical trial, normal test runs were performed against all five SUTs using the original test suite involving 673 331 test cases, as discussed in Section 4.4.1. In the first phase, five separate test run results (i.e. one for each of the chosen SUTs) were created, and they served as the basis for the next phase in which the test run results were processed.

The response status codes from the HTTP response messages received from all the SUTs during test runs were stored to the test run results. The amounts of distinct status codes received in the test runs are represented in Table 4. A message sequence where no response at all was received from the SUT was considered to be one type of response status code for each of the SUTs.

Table 4. Number of different response codes received with the original test suite

<table>
<thead>
<tr>
<th>Name of the HTTP Server Software</th>
<th>Number of Different Response Status Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>8</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>7</td>
</tr>
<tr>
<td>Tntnet</td>
<td>3</td>
</tr>
<tr>
<td>Hiawatha</td>
<td>7</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>10</td>
</tr>
</tbody>
</table>

The maximum amount of different response status codes received in a single test run with the original test suite was only ten, although the HTTP/1.1 specification defines forty individual response status codes and in addition an extension code (as described in Section 3.3). The relatively small amount of different response status codes can be explained by the fact that the predefined test suite comprises of test cases which are malformed HTTP request messages (discussed in Section 4.4.1).

A more extensive use of different response status codes could be expected from a protocol-specific conformance or regression test suite which does not emphasise the negative testing aspect. On the other hand, it is also possible that some of the SUTs have not implemented different response status codes at all, or the SUTs might make a simple use of generic response status codes in various situations. A generic use could be the following: response status code 200 to denote OK in all positive scenarios, and status code 400 to denote Bad syntax in all negative scenarios.

A conclusion can be drawn from the numbers that Tntnet is possibly the simplest HTTP server software implementation in the group of five, as only three different status codes were received, and one of them was no response. In comparison, Apache httpd seems to have the most extensive implementation of different response status codes.

All three of the developed optimisation algorithms base their optimisation efforts on knowledge about the alteration of the response status code in a test run. Therefore, theoretically this means that the optimisation of the test suite is hardest when the optimisation is founded on the test run result of Tntnet, and vice versa when the optimisation is based on the test run result of Apache httpd.

In addition to no response, two of the most common response status codes which were received from the SUTs in all test runs were status codes 200 and 400. Response
status code 200 denotes OK (from the successful class), whereas status code 400 stands for Bad syntax (from the client error class) [24, pp. 38, 42].

6.1. Code Coverage Measurement Results Using the Original Test Suite

The summary of the code coverage measurement results from the test runs using the original suite is shown in Table 5. The results comprise of the measured line, branch and function coverage from all five SUTs. The coverage quantities are presented both as a percentage and as a ratio of the covered code and the total amount of code.

Table 5. Code coverage results from test runs with the original test suite

<table>
<thead>
<tr>
<th>Metric</th>
<th>SUT</th>
<th>Nginx</th>
<th>Lighttpd</th>
<th>Tntnet</th>
<th>Hiawatha</th>
<th>Apache httpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Lines</td>
<td>Percentages</td>
<td>27.3 %</td>
<td>55.7 %</td>
<td>41.4 %</td>
<td>31.2 %</td>
<td>32.6 %</td>
</tr>
<tr>
<td>Branches</td>
<td>Percentages</td>
<td>21.5 %</td>
<td>44.9 %</td>
<td>20.4 %</td>
<td>22.0 %</td>
<td>20.4 %</td>
</tr>
<tr>
<td>Functions</td>
<td>Percentages</td>
<td>36.8 %</td>
<td>62.1 %</td>
<td>54.5 %</td>
<td>48.3 %</td>
<td>41.0 %</td>
</tr>
</tbody>
</table>

The figures show the total number of lines of code (LOC), the amount of branches, and the number of functions for each of the SUTs. The numbers suggest that Lighttpd and Hiawatha are the smallest implementations according to code size, whereas Nginx and Apache httpd are many times more complex HTTP server software implementations.

All in all, there are considerable differences in the amount of code covered with the original test suite. The results show that the most code was covered from Lighttpd (all of the coverage metrics are over 44%), and the least was covered from Nginx (all of the coverage metrics are under 37%). The lowest individual coverage result was 20.4% for the branch coverage of Tntnet and Apache httpd. On the other hand, the highest individual coverage result was 62.1% for the function coverage of Lighttpd.

The relatively low coverage reached with the original suite is partially explained by the fact that a modern HTTP server is a complex piece of software which often supports many other protocols in addition to HTTP. Therefore, some parts of the code in the SUT are not testable with plain HTTP messages, from which the test suite solely consists of. Another reason might be that the debugging features were disabled in the SUTs, and fixed configurations (e.g. Internet Protocol version 6 (IPv6) capabilities were disabled) were used in the SUTs throughout the testing [14, pp. 36, 40].

The configuration was also fixed in the test suite, as only one message sequence out of the ten available ones was used in all of the test runs, and all of the settings related to test case selection and to the automatic test case generation process were set to default values instead of the available maximum, which would have produced four times as much test cases as the default settings do (further discussed in Section 4.4.1).
6.2. Optimisation Results as Subset Test Suites

In the second phase, the original test suite incorporating 673,331 test cases grouped in 809 distinct test groups was optimised using the three developed optimisation algorithms (described in Sections 5.1.1, 5.1.2, and 5.1.3).

The optimisation was founded on the processing of the five test run results (containing the response status codes received from the SUTs) created in the first phase. Every optimisation result was considered as a separate proposal for an optimised test suite. The second phase generated altogether 15 distinct optimised subset test suites, as the three separate optimisation algorithms were each applied to the five test run results. The optimisation algorithms produced different sized subset suites. The sizes are listed in Tables 6 and 7.

Table 6. Optimisation results in the form of sizes (as measured in the number of test groups) of the subset suites

<table>
<thead>
<tr>
<th>Based on the Test Run Results of</th>
<th>Algorithm No. 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>8</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>7</td>
</tr>
<tr>
<td>Tntnet</td>
<td>3</td>
</tr>
<tr>
<td>Hiawatha</td>
<td>7</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>10</td>
</tr>
</tbody>
</table>

The optimisation result of algorithm No. 1 is represented in the form of the number of test groups (shown in Table 6) in the subset suite, because it is the only algorithm which creates an optimisation result comprising of whole test groups instead of individual test cases. For the latter two algorithms, the optimisation result is presented merely in the form of the number of test cases in the subset suite (listed in Table 7). The sizes of the subset suites produced by algorithm No. 1 are also represented in the form of the number of test cases in order to make the comparison of the subset suite sizes easier.

Table 7. Optimisation results in the form of sizes (as measured in the number of test cases) of the subset suites

<table>
<thead>
<tr>
<th>Based on the Test Run Results of</th>
<th>Algorithm No. 1</th>
<th>Algorithm No. 2</th>
<th>Algorithm No. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>11832</td>
<td>1687</td>
<td>38270</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>4020</td>
<td>1890</td>
<td>26931</td>
</tr>
<tr>
<td>Tntnet</td>
<td>840</td>
<td>839</td>
<td>158</td>
</tr>
<tr>
<td>Hiawatha</td>
<td>3334</td>
<td>1079</td>
<td>978</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>15511</td>
<td>1810</td>
<td>44355</td>
</tr>
</tbody>
</table>

It is clear that the optimisation based on the test run result of Tntnet generated extremely small subset suites due to the smallest number of different response codes used by the SUT (described in the beginning of Chapter 6). E.g. the optimisation result
of algorithm No. 3 based on the test run result of Tntnet is significantly smaller than the subset suite based on the optimisation of the test run result of Apache httpd. Therefore, theoretically this means that the subset suites created from the test run result of Tntnet are the least likely to attain the same code coverage level as the original test suite.

All in all, the use of optimisation algorithm No. 2 resulted in the smallest subset suite sizes out of the three algorithms. Optimisation algorithm No. 3 produced the biggest subset suite sizes. Theoretically this means that the optimisation results from algorithm No. 3 have the best chance to achieve the same code coverage level which was obtained in the first phase with the original test suite (shown in Table 5).

It is also worth noting that optimisation algorithm No. 1 optimised heavily the number of test groups. E.g. the optimisation result based on the test run result of Lighttpd involves only seven test groups among the original 809 test groups.

### 6.2.1. Reduction in the Size of the Test Suite

The reduction in the sizes of the subset test suites in comparison to the original test suite were calculated using the test case count in each of the subset suites (listed in Table 7) and the amount of test cases in the original test suite as variables in the calculation. The test suite size reduction calculation is defined as the following equation:

\[
S = \frac{T - T_{opt}}{T} \times 100, 
\]

where \(S\) is the size reduction as a percentage, \(T\) is the size of the original test suite as measured in the number of test cases, and \(T_{opt}\) is the size of the optimised test suite as measured in the number of test cases. [95, p. 47], [96, p. 20] The reductions in the sizes of the test suites are represented in Table 8.

<table>
<thead>
<tr>
<th>Based on the Test Run Results of</th>
<th>Algorithm No. 1</th>
<th>Algorithm No. 2</th>
<th>Algorithm No. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>98.2 %</td>
<td>99.7 %</td>
<td>94.3 %</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>99.4 %</td>
<td>99.7 %</td>
<td>96.0 %</td>
</tr>
<tr>
<td>Tntnet</td>
<td>99.9 %</td>
<td>99.9 %</td>
<td>99.9 %</td>
</tr>
<tr>
<td>Hiawatha</td>
<td>99.5 %</td>
<td>99.8 %</td>
<td>99.8 %</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>97.7 %</td>
<td>99.7 %</td>
<td>93.4 %</td>
</tr>
</tbody>
</table>

The numbers clearly confirm that the test suite size reduction is dramatic when each of the developed algorithms is used in the optimisation. The reduction in the size of the suite means that a considerable amount of valuable test time can be saved in every test run in comparison to a test run using the complete test suite. The vast reductions in the suite sizes hint that the optimised subset suites with such a small number of test cases have to work exceptionally well in trying to reach the same level of code coverage as the original test suite did.

The level of code coverage might show the actual cost of the optimisation inflicted by the fact that a vast amount of test cases are not executed if the testing process is based on the execution of an optimised test suite. The level of code coverage might
decline, simply because a far smaller amount of test cases are executed. On the other hand, a benefit of the optimisation is savings in the testing time and resources which are no longer allocated due to the execution of an optimised subset test suite instead of the complete test suite.

The execution times of the test runs were measured in the first and third phase of the practical trial. All of the attained execution times are not comparable and statistically valid as such, because the tests were performed with approximately five simultaneous test runs. The parallel test runs generated non-stable load to the VM hosting the SUTs, and thus they had an effect on the validity of the test execution times.

The percentages for the test suite size reduction (shown in Table 8) serve as a rough estimate for saved test execution time, which means that theoretically altogether 93.4 percent (an approximation) or more can be saved and reduced from the test execution time by using any of the optimised subset suites generated within this thesis.

6.3. Code Coverage Measurement Results Using the Subset Test Suites

In the third phase of the practical trial, regular test runs were performed against all five SUTs using the subset test suites originated in the second phase as the result of the optimisation efforts. Altogether, 15 normal test runs were performed in the third phase. One test run was executed for each optimised subset suite.

The test runs were carried out with merely five subset suites at a time, and each had a separate HTTP server as a SUT. Firstly, the five subset suites created as the result of optimisation done with algorithm No. 1 were used. Then, the five subsets generated as the result of optimisation done with algorithm No. 2 were used, and so forth.

The complete results of the measured code coverage from all of the 15 different test runs in the third phase are presented in the following three Sections. In addition to the code coverage results, the changes in the coverage results are shown. The alterations are shown in comparison to the code coverage results achieved using the original test suite.

The differences were calculated by reducing the coverage figures obtained in the first phase (shown in Table 5) from the coverage results attained in the third phase. The basic form of the difference calculation is defined as the following equation:

\[ \Delta U = U_{\text{opt}} - U_{\text{orig}}, \]

where \( U_{\text{opt}} \) is the total number of the particular coverage units in a test run with an optimised test suite, and \( U_{\text{orig}} \) is the total number of the same particular coverage units in a test run with the original test suite.

In addition, the response status codes from the HTTP response messages received from all the SUTs during the test runs with the optimised subset suites were stored to the test run results. The amounts of distinct response status codes received in the test runs are represented in Table 9. A message sequence where no response at all was received from the SUT was considered to be one type of response status code for each of the SUTs.
Table 9. Number of different response codes received with the subset suites

<table>
<thead>
<tr>
<th>Based on the Test Run Results of</th>
<th>Algorithm No. 1</th>
<th>Algorithm No. 2</th>
<th>Algorithm No. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>10</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Tntnet</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Hiawatha</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Apache httpd</td>
<td>11</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

The amount of different response status codes remained the same in six test runs out of fifteen when using the subset suites. The amount of distinct status codes decreased only in two test runs, and increased in the remaining seven test runs. The two losses in the received amount of different status codes were with the subset suites produced by algorithm No. 2. It should be noted that the number of response status codes remained the same or increased only with the subset suites created by algorithms No. 1 and No. 3.

The changes in the amounts of different response status codes (received during the test runs) indicate that the earlier state in the SUT’s code execution might have an effect on the forming of the response status code for the next response message. This means that in order to receive exactly the same response status codes in two successive test runs, the same test cases ought to be executed in the exact same order.

6.3.1. Results Using the Subset Test Suites Optimised with Algorithm No. 1

Optimisation algorithm No. 1 (detailed account in Section 5.1.1) focused on a single test group at a time. The actual selection of the optimisation-wise interesting test groups was done based on the fact whether the same response status code has already been received in an earlier test group, and whether the name of the test group is new in the test case in question. A test group was selected by the algorithm if the response status code and the name of the test group were both new.

A summary of the code coverage measurement results from the test runs using the first series of five subset suites (which were the result of optimisation using algorithm No. 1) is shown in Table 10. As in the first phase, the results consist of the coverage quantities of line, branch and function coverage from all of the five SUTs as ratios and percentages. In addition, the numbers show the differences between the coverage result with the optimised suites and the coverage with the original suite.
Table 10. Code coverage results from test runs with the subset suites (algorithm No. 1)

<table>
<thead>
<tr>
<th>Metric</th>
<th>SUT</th>
<th>Nginx</th>
<th>Lighttpd</th>
<th>Tntnet</th>
<th>Hiawatha</th>
<th>Apache httpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Lines</td>
<td></td>
<td>6628 /26211</td>
<td>4067 /7850</td>
<td>1905 /5562</td>
<td>1807 /6232</td>
<td>6452 /20409</td>
</tr>
<tr>
<td>Percentage</td>
<td></td>
<td>25.3 %</td>
<td>51.8 %</td>
<td>34.3 %</td>
<td>29.0 %</td>
<td>31.6 %</td>
</tr>
<tr>
<td>Δ Code Lines</td>
<td>−538</td>
<td>−303</td>
<td>−396</td>
<td>−137</td>
<td>−197</td>
<td></td>
</tr>
<tr>
<td>Δ Percentage</td>
<td>−2.1 %</td>
<td>−3.9 %</td>
<td>−7.1 %</td>
<td>−2.2 %</td>
<td>−1.0 %</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td>3543 /18186</td>
<td>1949 /4882</td>
<td>2227 /14072</td>
<td>1013 /5344</td>
<td>3138 /16592</td>
<td></td>
</tr>
<tr>
<td>Percentage</td>
<td>19.5 %</td>
<td>39.9 %</td>
<td>15.8 %</td>
<td>18.9 %</td>
<td>18.8 %</td>
<td></td>
</tr>
<tr>
<td>Δ Branches</td>
<td>−372</td>
<td>−245</td>
<td>−645</td>
<td>−162</td>
<td>−252</td>
<td></td>
</tr>
<tr>
<td>Δ Percentage</td>
<td>−2.0 %</td>
<td>−5.0 %</td>
<td>−4.6 %</td>
<td>−3.0 %</td>
<td>−1.5 %</td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td>396 /1110</td>
<td>241 /398</td>
<td>547 /1068</td>
<td>122 /263</td>
<td>566 /1396</td>
<td></td>
</tr>
<tr>
<td>Percentage</td>
<td>35.7 %</td>
<td>60.6 %</td>
<td>51.2 %</td>
<td>46.4 %</td>
<td>40.5 %</td>
<td></td>
</tr>
<tr>
<td>Δ Functions</td>
<td>−13</td>
<td>−6</td>
<td>−35</td>
<td>−5</td>
<td>−6</td>
<td></td>
</tr>
<tr>
<td>Δ Percentage</td>
<td>−1.2 %</td>
<td>−1.5 %</td>
<td>−3.3 %</td>
<td>−1.9 %</td>
<td>−0.4 %</td>
<td></td>
</tr>
</tbody>
</table>

Out of all of the fifteen different optimisation results attained in the practical trial, the best optimisation result was achieved with algorithm No. 1 when the optimisation was based on the test run result of Apache httpd. Almost the same amount of code was covered with the subset suite as with the original test suite.

The amount of covered code reduced considerably with the subset suites based on the test run result of Lighttpd and Tntnet in comparison to the coverage results obtained with the original test suite.

The coverage result for Nginx is the worst amongst all three optimisation results, but on the other hand the change in the amount of the covered code was not actually significant (the maximum change being of only −2.1 %). Similar deduction can be made from the coverage result of Hiawatha where the coverage level was retained reasonably close to the level reached with the original test suite.

Although none of the optimised subset suites attained the same level of coverage as the original test suite did, optimisation algorithm No. 1 performed reasonably well in the optimisation based on three test run results (Nginx, Hiawatha, and Apache httpd) and poorly in the optimisation based on the two remaining ones (Lighttpd, Tntnet).

### 6.3.2. Results Using the Subset Test Suites Optimised with Algorithm No. 2

Optimisation algorithm No. 2 (described in more detail in Section 5.1.2) focused on the test cases in a single test group at a time. The actual selection of the optimisation-wise interesting test cases was done based on the fact whether the same response status code had already been received with an earlier test case in the test group in question. A test case was selected by the algorithm if the response status code had not been received in the previous test cases in the test group in question.

The summary of the code coverage measurement results from the test runs using the second series of five subset suites (which were the result of optimisation using
The best optimisation results were obtained with optimisation algorithm No. 2 when it was applied to three out of the five different test run results. These results were from the test runs of the following SUTs: Lighttpd, Tntnet, and Hiawatha. The optimisation result based on Hiawatha is an exemplary result, because the amount of the covered code was reduced no more than –2.0 %.

None of the achieved code coverage results is the worst for any of the chosen SUTs in comparison to the other algorithms. The highest decline in the results was –3.7 % in the branch coverage of Lighttpd. It should be noted that it was the best optimisation result founded on the test run result of the SUT in question.

To summarize, optimisation algorithm No. 2 was the most successful optimisation algorithm amongst the three developed ones, although all of the subset suites failed to reach the same code coverage level as the original test suite did.

### 6.3.3. Results Using the Subset Test Suites Optimised with Algorithm No. 3

Optimisation algorithm No. 3 (explained in more detail in Section 5.1.3) focused merely on single test cases. The actual selection of the optimisation-wise interesting test groups was done based on the fact whether the response status code alters in between two consecutive test cases. A test case was selected by the algorithm if the response status code differed from the preceding test case.

The summary of the code coverage measurement results from the test runs using the third and the last series of five subset suites (which were the result of optimisation using algorithm No. 3) is shown in Table 12. As in the first phase, the results consist
of the coverage quantities of line, branch and function coverage from all the five SUTs as ratios and percentages. In addition, the numbers show the differences between the coverage result with the optimised suites and the coverage with the original suite.

Table 12. Code coverage results from test runs with the subset suites (algorithm No. 3)

<table>
<thead>
<tr>
<th>Metric</th>
<th>SUT</th>
<th>Nginx</th>
<th>Lighttpd</th>
<th>Tntnet</th>
<th>Hiawatha</th>
<th>Apache</th>
<th>httpd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Lines</td>
<td>Percentage</td>
<td>6776 /26211</td>
<td>4120 /7850</td>
<td>1901 /5562</td>
<td>1785 /6232</td>
<td>6254 /20409</td>
<td></td>
</tr>
<tr>
<td>∆Code Lines</td>
<td>Percentage</td>
<td>−390</td>
<td>−250</td>
<td>−400</td>
<td>−159</td>
<td>−395</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td>Percentage</td>
<td>3646 /18186</td>
<td>1998 /4882</td>
<td>2220 /14072</td>
<td>995 /5344</td>
<td>2967 /16592</td>
<td></td>
</tr>
<tr>
<td>∆Branches</td>
<td>Percentage</td>
<td>−269</td>
<td>−196</td>
<td>−652</td>
<td>−180</td>
<td>−423</td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td>Percentage</td>
<td>400 /1110</td>
<td>245 /398</td>
<td>548 /1068</td>
<td>123 /263</td>
<td>561 /1396</td>
<td></td>
</tr>
<tr>
<td>∆Functions</td>
<td>Percentage</td>
<td>−9</td>
<td>−2</td>
<td>−34</td>
<td>−4</td>
<td>−11</td>
<td></td>
</tr>
</tbody>
</table>

The worst result for the optimisation based on the test run result of Hiawatha was obtained using algorithm No. 3. The code coverage result for Hiawatha is not so bad, as the biggest fall was in the branch coverage, which fell −3.4 %.

Algorithm No. 3 proved to be the best when optimisation was founded on the test run result of Nginx. The amount of covered code is near the quantities attained with the original test suite. The maximum reduction in the coverage results was only −1.5 %, which means that the subset suite in question achieved the second best code coverage result of all the 15 different optimised suites in comparison to the coverage results with the original test suite.

None of the optimised subset suites reached the same level of coverage as the original test suite did. But optimisation algorithm No. 3 performed well when the optimisation was based on the test run results of all of the SUTs except Tntnet, which proved to have the toughest test run results for all of the optimisation algorithms to process. All in all, the obtained code coverage levels were not generally as good as with algorithm No. 2, but at the same time they were slightly better than with algorithm No. 1.
7. DISCUSSION

In the fourth and the final phase of the practical trial, the answers to the research questions (discussed in Section 4.1) were validated with the aid of the acceptance criterion. The code coverage measurement results from the third phase (the test runs with the optimised subset suites) were analysed and compared to the code coverage measurement results from the first phase (the test runs with the original test suite), and conclusions were drawn from the measures.

7.1. Answers to the Research Questions

The answer to the first question, whether the variation in the response status codes in HTTP response messages indicates that the test case actually exercised the SUT, is quite evident. The answer was validated by using rudimentary code coverage analysis, with the aid of the acceptance criterion (described in Section 5.3.1). 12 out of the 15 different comparison results for the code coverage measurement results attained with the optimised suites in comparison to the coverage results acquired with the original test suite (detailed account in Tables 10, 11, and 12) fulfilled the acceptance criterion. Therefore, a favourable conclusion was drawn as the answer for the first question.

The result, 12 out of 15, implies with a quite strong degree of certainty that the variation in the response status codes indicates those test cases or test groups which actually exercise the SUT. The certainty would be even stronger, if the coverage results attained from two SUTs (Lighttpd and Tntnet) that used the smallest amounts of distinct response status codes (as shown in Table 4) would not have been taken into account in the validation phase. The poorer performance of the optimised subset suites based on the test run results generated by Lighttpd and Tntnet is mostly explained by the fact that the optimisation efforts are completely founded on knowledge about the alteration of the response status codes which were received from the SUTs.

The answer to the second research question, whether the same level of code coverage is achievable with an optimised subset suite, is as evident as the answer to the first question. The second answer was similarly validated using elementary code coverage analysis and with the aid of the same acceptance criterion. Corresponding to the first question, a favourable conclusion was drawn as the answer for the second question, because 12 out of the 15 different comparison results for the code coverage results attained with the optimised suites in comparison to the coverage measurement results acquired with the original test suite (details in Tables 10, 11, and 12) fulfilled the acceptance criterion.

The amount of covered code did not decrease drastically when the coverage measurement was done using the optimised subset suites. The result, 12 out of 15, implies with a quite strong degree of certainty that the same level of code coverage is achievable with an optimised suite. The certainty would be even stronger, if the SUT (Tntnet) which used the smallest amount of distinct response status codes (as shown in Table 4) would not have been taken into account in the validation phase.

Alas, none of the three optimisation algorithms succeeded in producing an optimised subset suite which is able to achieve exactly the same amount of code coverage as the original test suite did. This is partially explained by the fact that no response
at all was received from the SUT for a majority of test cases due to the nature of the test cases (i.e. negative tests). The test cases in the predefined test suite are essentially invalid and malformed HTTP request messages, for which the test suite does not even require or expect a response message (as explained in Section 4.4.1).

On the other hand, this confirms that some parts of the SUT’s code are covered during the parsing of the received anomalous HTTP request messages, even if the SUT does not send any response at all to the test suite in a majority of the test cases.

7.2. Assessment of the Algorithms

The assessment of the optimisation algorithms was done based on the attained code coverage measurement results from the practical trial. Code coverage analysis and the acceptance criterion (described in Section 5.3.1) were used as assistance in the assessment.

The most effective algorithm was optimisation algorithm No. 2, as it reached the best optimisation result for three SUTs out of the group of five. All five of the code coverage results from the test runs with the optimised subset suites met the acceptance criterion. Therefore, optimisation algorithm No. 2 is regarded as highly successful in the optimisation efforts. The only downside in the optimisation results was that algorithm No. 2 performed worse than the others when the optimisation was founded on the test run result of *Apache httpd*.

Optimisation algorithm No. 2 focused on single test cases in a single test group at a time. A test case was selected by the algorithm, if the response status code had not been received in the preceding test cases in the test group in question (detailed account in Section 5.1.2). The algorithm acquired the best results amongst the three algorithms, because it possessed optimisation-wise valuable knowledge about the status codes received earlier in the test group in question.

The second best performer was algorithm No. 3, because four out of the five code coverage results from the test runs with the optimised subset suites met the acceptance criterion. Therefore, optimisation algorithm No. 3 is considered mainly successful in the optimisation efforts. Algorithm No. 3 attained relatively the second best optimisation result out of the all 15 different optimisation results when the optimisation was based on the test run result of *Nginx*. But on the other hand, the algorithm achieved the weakest optimisation result for one of the SUTs out of the five.

Optimisation algorithm No. 3 focused merely on single test cases. A test case was selected by the algorithm, if the response status code had changed from the previous test case (discussed in Section 5.1.3). The moderate lack in the performance of the algorithm is due to the fact that no response at all was received from the SUT for a majority of the test cases which is due to the nature of the test cases (negative tests) in the suite. The algorithm might have performed better if a response status code had been received in all of the test cases.

The third and the last in the ranking was algorithm No. 1. The algorithm is regarded as moderately successful in the optimisation efforts, because only three out of the five code coverage results from the test runs with the optimised subset suites met the acceptance criterion. The poorest optimisation results were obtained with algorithm No. 1, as two optimisation results amongst the five were poorer in comparison to
the results attained with the other algorithms. It is noteworthy that algorithm No. 1 managed to achieve the best result when the optimisation was founded on the test run result of Apache httpd, although it was generally considered as the poorest performer amidst the developed algorithms.

Optimisation algorithm No. 1 focused on a single test group at a time. A test group was chosen by the algorithm if both the response status code received from the SUT and the name of the test group were both new for the test case in question (as explained in Section 5.1.1). The moderate performance of the algorithm is partially due to the selection method which relied heavily on the fact that the HTTP/1.1 specification defines forty individual response status codes and in addition an extension code (as described in Section 3.3) which can be used in HTTP response messages. The actual maximum amount of different response status codes received in a single test run was as low as ten. The lack in the use of different response status codes is partially explained by the fact that the predefined test suite comprises mostly of test cases which are essentially malformed HTTP request messages.

The reduction in the test execution times was not used as an argument in the assessment (as discussed in Section 6.2.1). The attained execution times were not comparable as such due to the simultaneous and fluctuating load generated by the parallel test runs. A rough estimate was given that approximately 93.4 percent or more can be reduced from the whole test execution time, because the optimised subset suites comprised of a far smaller amount of test cases than the original test suite did.

The sampling of five test subjects consisted of three HTTP server software implementations which can be considered highly relevant due to a significant user base, and of two more trivial HTTP server implementations (discussed in Section 4.5). The significance of the most popular HTTP servers is reflected in the amount of lines of source code in the SUT. Nginx and Apache httpd are both approximately three times larger by the amount of source code than the rest of the SUTs. The trivial nature of Tntnet and Hiawatha was not used as an argument in the assessment of the optimisation algorithms, or in how they fulfilled the acceptance criterion.

### 7.3. Known Limitations

There is a prerequisite in the use of the developed optimisation algorithms. A test run must be performed before any of the algorithms can be used, because the algorithms base their optimisation efforts on the processing of the test run result containing response status codes. This means that optimisation cannot be performed unless a valid test run result exists.

The optimisation algorithms cannot be applied to all kinds of test run results, as the optimisation efforts rely on knowledge about the response status code received from the SUTs. Therefore, the developed algorithms are applicable only to test run results attained using a protocol in which the response message contains a status code. Such an applicable protocol would be e.g. the Session Initiation Protocol (SIP).

There is a known limitation in the code coverage comparison performed in the final phase of the practical trial. The comparison was done by comparing only the quantity levels of the covered code, which means that the actual content of the coverage data was not compared. The content of the coverage data was not compared line by line, as
the basic assumption was that the optimised subset suites cannot cover such code parts which were not covered with the original test suite. The assumption was made because all of the developed optimisation algorithms are founded merely on the reduction of the amount of test cases to be executed.

In this thesis, three different optimisation algorithms were developed, and all of them were applied to the five test run results from the different SUTs. Hence, the optimisation result was the fifteen distinct subset suites which were used and experimented only on the same specific SUT whose test run result was the basis for the optimisation. To make the research results generally more valid, more research could have been done, i.e. each of the fifteen subset suites could have been used on all of the five SUTs, whereupon the outcome would have been quite ample due to 75 distinct code coverage measurement results instead of the fifteen which were dealt with in this thesis.

The gcov tool provided only the following three kinds of code coverage measurement results: line, branch, and function coverage. In order to provide other kinds of coverage measurement data, a different coverage measurement tool should be selected (discussed in Section 4.6.4). E.g. the use of Testwell CTC++, and LDRA Testbed would provide Modified Condition/Decision Coverage (MC/DC) results from the SUT, which would give a better insight to the quality of the testing.

All of the code coverage measurement results in this thesis include the covered code from the start-up and shutdown phases of the SUT’s execution in addition to the code covered during the test case execution (as discussed in Section 4.6.2). To make the coverage results more exact, a code coverage measurement tool which supports disabling of the coverage measurement during certain periods of the SUT’s execution could be used. An alternative option is to edit out the code which was covered in the start-up and shutdown phases from the coverage data.

### 7.4. Future Development

All of the optimisation efforts were completely based on the processing of the response status codes in HTTP response messages. The optimisation could be based on slightly more advanced decision making, if also header lines from the HTTP response messages (described in 3.1.2) were observed and stored during the test runs. E.g. the selection of optimisation-wise interesting test cases could be founded on knowledge about the change in the amount of different header lines (or on the value changes in the header lines) which could be interpreted by the algorithm as a change in the SUT’s behaviour.

The predefined test suite is intended merely for fuzz testing of HTTP server implementations. On the other hand, a modern HTTP server is often a very complex piece of software which tends to support various other protocols than HTTP, e.g. the Transport Layer Security (TLS) [97] protocol. In order to improve the code coverage levels reached in this thesis, the other protocols supported by each SUT should also be tested.

There are quite a few ways to make the results of this research purportedly more valid and general. Firstly, the optimisation algorithms could be tried on the other suitable test subjects which were mentioned in Section 4.5.6. The research could be extended to cover also commercial HTTP server software, such as Microsoft IIS, which is the second most popular HTTP server software according to surveys (explained
in Section 4.5). In addition, HTTP proxy server implementations could be used as additional test subjects.

Secondly, the topic of this research could be extended to cover other protocols which make similar use of the status codes in the response messages as the HTTP protocol. Such an applicable protocol would be e.g. the Session Initiation Protocol (SIP) [92, pp. 28–29].

The maximum amount of different response status codes received in a single test run was only ten in this thesis, although the HTTP/1.1 specification defines forty individual response status codes and in addition an extension code (as described in Section 3.3). The lack in the use of different response status codes is partially explained by the fact that the predefined test suite comprises for the most parts of test cases which are essentially malformed HTTP request messages. Extensive use of different response status codes could be expected from a protocol-specific conformance or regression test suite which does not emphasise the negative testing aspect.

Also other HTTP server test suites could be tried, e.g. conformance or regression test suites could provide different code coverage results. Bearing in mind that if a regression suite is optimised with the algorithms developed in this thesis, the same amount of code might no longer be covered as was with the original test suite. Therefore, test runs with the full regression test suite should not be neglected, but instead should be done on a regular basis.

On the other hand, the predefined test suite offers several ways to extend the research. One is to use the other message sequences than the HTTP request and receive response which was used in this thesis. Another alternative is to change all the settings related to the test case selection and generation in the test suite to the maximum value, which results in an approximately four times bigger test suite than with the default settings which were used in this thesis. As a curiosity, the test suite even offers setting for the theoretical creation of an unlimited number of test cases, but the test execution time also increases in the same proportion. The level of the code coverage could increase due to the use of the other message sequences or due to the larger number of executed test cases.

Code coverage could be measured on a test case by test case basis with a different coverage measurement tool. This improvement would amend the precision of the code coverage results, and the way how the measurement results can be presented. It would also enable the use of existent optimisation methods in which the redundancy of a test case could be deduced based entirely on the code coverage data.
8. CONCLUSION

The scope of this thesis was on the optimisation of Defensics Hypertext Transfer Protocol (HTTP) server test suite, which is a commercially available test suite designed for robustness and security testing of HTTP server implementations. In this thesis work, an answer was given to the two research questions with the aid of code coverage analysis and an acceptance criterion.

In order to answer to the research questions, first a complete test setup was designed and created. This test setup comprised of the given test suite, a sampling of five different HTTP server software implementations, and the selected code coverage measurement tool.

Secondly, three different kinds of test suite optimisation algorithms were developed. The optimisation of the predefined test suite was based completely on the processing of the test run results which were acquired using the created test setup. The decisions in the optimisation were made based on knowledge about the distinct response status codes received from the SUT during the test runs. All three different optimisation algorithms approached the test suite optimisation with different solutions. The developed test suite optimisation algorithms are also applicable to test run results generated with other request/response protocols in which the response messages include status codes.

The first one of the algorithms focused on a single test group at a time, and the optimisation results comprised of whole test groups. The optimisation algorithm selected a test group, if both the response status code was new and the name of the test group was not already included in the result list. This approach was taken based on the fact that the given test suite consists of test groups, each of which is meant for fuzzing a different part of the HTTP request message. It was also presumed that a majority of the possible distinct response status codes are received during the execution of different test groups.

Algorithm No. 2 focused on the test cases in a single test group at a time, and the optimisation results consisted of single test cases. The optimisation algorithm chose a test case, if the response status code received from the SUT had not been received in the preceding test cases of the test group in question. The former approach to the test suite optimisation was changed, and the key factor in the change of the focus was the structure of the predefined test suite, i.e. the test suite consists of test groups, each of which is meant for fuzzing a different part of the HTTP request message.

The third algorithm focused on single test cases only, and the optimisation results comprised of test cases. The optimisation algorithm selected a test case, if the response status code altered in comparison to the response status code of the previous test case. The last of the three different approaches was taken based on the fact that the test suite consists of only few hundred distinct test groups, whereas the test suite comprises of a multiple times greater number of test cases.

The code coverage measurement results from the test runs using the original test suite and all of the optimised subset suites were provided by the performed practical trial. The coverage measurement results were compared, and conclusions were drawn from the measures using code coverage analysis.

The most efficient of the developed algorithms was algorithm No. 2, because all five of the code coverage measurement results met the acceptance criterion, and the
optimisation result was the best for three SUTs out of five in comparison to the results achieved with the other algorithms.

A majority of the code coverage measurement results attained with the optimised subset suites fulfilled the acceptance criterion. Therefore, a favourable conclusion was drawn to the following research questions.

The result of the practical trial implied with a strong degree of certainty that the variation in the response status codes of the HTTP response messages indicates which test cases or test groups actually exercise the SUT.

The result of the practical trial also implied with a quite strong degree of certainty that the same level of code coverage from the SUT can be achieved with an optimised subset test suite.
9. REFERENCES


10. APPENDICES

Appendix 1  Python script for test suite optimisation using developed algorithms

Appendix 2  Bash script for controlling SUTs and code coverage measurement
Appendix 1. Python script for test suite optimisation using developed algorithms

```
#!/usr/bin/python
# -*- coding: latin-1 -*-

""
@author: Tuomas Parttimaa
@copyright: 2012 Tuomas Parttimaa
@license: 3−Clause BSD License

Copyright (c) 2012 Tuomas Parttimaa
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

""

import sys
import csv
import os

def statparser(filename, algorithm):
    ""
    Function for the three different types of optimisation algorithms which were developed
    ""
    prevstatus = [] # Status of the previous test case
    statuses = [] # All statuses from a test run
    tempstatuses = [] # A temporary status list
    tempcases = [] # A temporary test case list
    tempgroups = [] # A temporary test group list
```
```python

testcases = []  # The optimisation result as a subset of test cases
testgroups = []  # The optimisation result as a subset of test groups

# Optimisation algorithm No. 1
if algorithm == 1:
    resultfile = 'optimised1-result.txt'
    try:
        statfile = open(filename, 'r')
        reader = csv.reader(statfile)
        rownum = 0
        # Go through rows
        for row in reader:
            # Header row
            if rownum == 0:
                header = row
            # Other rows
            else:
                # Replace temp lists with the actual ones
                tempstatuses = statuses[:]
                tempcases = testcases[:]
                tempgroups = testgroups[:]
                newstatus = False
                newtestcase = False
                newgroup = False
                # Is it new status?
                newstatus = datachecker(tempstatuses, row[3])
                if newstatus is True:
                    # Is it new testcase?
                    newtestcase = datachecker(tempcases, row[2])
                    if newtestcase is True:
                        # Is it new testgroup?
                        newgroup = datachecker(tempgroups, row[1])
                        if newgroup is True:
                            datachecker(statuses, row[3])
                            datachecker(testcases, row[2])
                            datachecker(testgroups, row[1])
                rownum += 1
    except:
        print("Optimisation failed. Stopping the script.")
        exit(0)
    finally:
        statfile.close()

# Optimisation algorithm No. 2
elif algorithm == 2:
    resultfile = 'optimised2-result.txt'
    try:
        statfile = open(filename, 'r')
        reader = csv.reader(statfile)
        rownum = 0
        # Go through rows
        for row in reader:
            # Header row
            if rownum == 0:
```
header = row
# Other rows
else:
    newstatus = False
    newtestgroup = False
    # Is it new testgroup?
    newtestgroup = datachecker(testgroups, row[1])
    if newtestgroup is True:
        # Clear temporary status list
        tempstatus = []
        # Is it new status?
        datachecker(statuses, row[3])
        # Is it new status in temp list?
        newstatus = datachecker(tempstatus, row[3])
        if newstatus is True:
            # Is it new testcase?
            datachecker(testcases, row[2])
            rownum += 1
except:
    print("Optimisation failed. Stopping the script.")
    exit(0)
finally:
    statfile.close()

# Optimisation algorithm No. 3
elif algorithm == 3:
    resultfile = 'optimised3_result.txt'
    try:
        statfile = open(filename, 'r')
        reader = csv.reader(statfile)
        rownum = 0
        # Go through rows
        for row in reader:
            # Header row
            if rownum == 0:
                header = row
            # Other rows
            else:
                newstatus = False
                newtestcase = False
                # Did status change from previous case
                newstatus = datachecker(prevstatus, row[3])
                if newstatus is True:
                    # Clear previous and add current
                    prevstatus = []
                    datachecker(prevstatus, row[3])
                    # Is it new testcase?
                    datachecker(testcases, row[2])
                    rownum += 1
    except:
        print("Optimisation failed. Stopping the script.")
        exit(0)
    finally:
        statfile.close()

# Print the result of optimisation to stdout
print("Optimised test groups with algorithm No. %d: %s" % (algorithm, resultfile))
print "Optimised test cases with algorithm No. %d: %s \\
% (algorithm, testcases)
print "Statuses with algorithm No. %d: %s \\
% (algorithm, statuses)
# Write the same result to a result file
print "Starting to write these optimisation results to the following result file: \\
"to the following result file": resultfile
def datachecker(givelist, givendata):
    """
    Function to check if a given data is already in a given list
    """
    for item in givelist:
        if item == givendata:
            return False
    # Data not known, store it
    givelist.append(givendata)
    return True

def filechecker(filename):
    """
    Function to check if the given optimisation result file already exists
    """
    # Prompt user if result file already exists
    if os.path.isfile(filename) == True:
        print "File %s already exists!" % (filename)
        while 1:
            choice = raw_input("Overwrite (y)es or (n)o: ")
            # Delete existing file
            if choice == "yes" or choice == "y":
                os.remove(filename)
                # Check if delete successful
                if os.path.isfile(filename) == False:
                    print "Deleting of %s was successful." % (filename)
                    break
                else:
                    print "Deleting of %s was unsuccessful." % (filename)
                    exit(0)
            elif choice == "no" or choice == "n":
                print "Results were not stored. Stopping the script."
                exit(0)
def writeparsed(filename, givendata, dataname):
Function to save the given optimisation results to a file

if os.path.isfile(filename) == False:
    # Write to a new result file
    resultfile = open(filename, 'w')
else:
    # Append to an existing result file
    resultfile = open(filename, 'a')
try:
    length = len(givendata)
    if dataname != "statuses":
        resultfile.write("Optimised%s contain%d items which are:
             
% (dataname, length))
    else:
        resultfile.write("The%s contains%d items which are:
             
% (dataname, length))
    i = 1
    # Separate items with comma
    for item in givendata:
        resultfile.write("%s" % item)
        if i != length:
            resultfile.write(",")
        i += 1
        resultfile.write("\n\n")
finally:
    resultfile.close()

def main():
    """
    Main function
    """
    print "Optimisation No.1 starting."
    statparser('statistics.csv', 1)
    print "Optimisation No.1 finished.\n\nOptimisation No.2 starting."
    statparser('statistics.csv', 2)
    print "Optimisation No.2 finished.\n\nOptimisation No.3 starting."
    statparser('statistics.csv', 3)
    print "Optimisation finished."

if __name__ == "__main__":
    """
    The main entry point
    """
    main()
Appendix 2. Bash script for controlling SUTs and code coverage measurement

srv.sh

```bash
#!/bin/bash
# @author: Tuomas Parttimaa
# @copyright: 2012 Tuomas Parttimaa
# @license: 3−Clause BSD License
#
# Copyright (c) 2012 Tuomas Parttimaa
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# 1. Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
# 2. Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
#    with the distribution.
# 3. The name of the author may not be used to endorse or promote
#    products derived from this software without specific prior written
#    permission.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS
# OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
# DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
# DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
# GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
# NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
# Check if non−root user
if [ $EUID != 0 ]; then
    echo "Use sudo"
    read −p "Press any key ..."
else
    if [ "$1" == "" ]; then
        echo "Use either one of these arguments: start | stop | report | cleanup | delrep"
        echo "start: start http servers"
        echo "stop: stop http servers"
        echo "report: make code coverage reports after test runs"
        echo "cleanup: remove code coverage related files"
        echo "delrep: remove code coverage reports"
        read −p "Press any key ..."
else
        ARG="start"
        # Check supplied argument
        if [ "$1" == "$ARG" ]; then
            # Check if process already running
```
SERVICE='nginx'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "$SERVICE is already running!"
else
echo "Starting $SERVICE!"
/usr/local/nginx/sbin/nginx
fi
SERVICE='lighttpd'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "$SERVICE is already running!"
else
echo "Starting $SERVICE!"
/usr/local/sbin/lighttpd -f /etc/lighttpd/lighttpd.conf
fi
SERVICE='ttnet'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "Ttnet is already running!"
else
echo "Starting Ttnet!"
(cd /opt/ttnet-2.1/myfirstproject/ && valgrind /usr/local/bin/ttnet -c ttnet.conf > /dev/null 2>&1 & )
fi
SERVICE='hiawatha'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "Hiawatha is already running!"
else
echo "Starting Hiawatha!"
/usr/local/sbin/hiawatha -d > /dev/null 2>&1 &
fi
SERVICE='httpd'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "Apache httpd is already running!"
else
echo "Starting Apache httpd!"
/usr/local/apache2/bin/httpd -k start -X > /dev/null 2>&1 &
fi
fi
ARG="stop"
# Check supplied argument
if [ "$1" == "$ARG" ]; then
# Check if process already running
SERVICE='nginx'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "Stopping $SERVICE!"
/usr/local/nginx/sbin/nginx -s quit
else
echo "$SERVICE is not running!"
fi
SERVICE='lighttpd'
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
echo "Stopping $SERVICE!"
PID=`ps -eo pid, args | grep $SERVICE | grep -v grep | cut -c1-6`
kill -SIGINT $PID
else
echo "$SERVICE is not running!"
fi
SERVICE=tntnet
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
  echo "Stopping $SERVICE!"
PID=`ps -eo pid, args | grep $SERVICE | grep -v grep | cut -c1-6`
  kill -SIGINT $PID
else
  echo "$SERVICE is not running!"
fi
SERVICE=hiawatha
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
  echo "Stopping $SERVICE!"
PID=`ps -eo pid, args | grep $SERVICE | grep -v grep | cut -c1-6`
  kill -SIGINT $PID
else
  echo "$SERVICE is not running!"
fi
SERVICE=httpd
if ps ax | grep -v grep | grep $SERVICE > /dev/null then
  echo "Stopping $SERVICE!"
PID=`ps -eo pid, args | grep $SERVICE | grep -v grep | cut -c1-6`
  kill -SIGINT $PID
else
  echo "Apache $SERVICE is not running!"
fi
ARG="clean up"
# Check supplied argument
if [ "$1" == "$ARG" ]; then
  lcov --directory /opt/nginx-1.0.15 --zerocounters
  lcov --directory /opt/lighttpd-1.4.30 --zerocounters
  lcov --directory /opt/tntnet-2.1 --zerocounters
  lcov --directory /opt/hiawatha-8.1 --zerocounters
  lcov --directory /opt/httpd-2.4.1 --zerocounters
  rm /opt/nginx-1.0.15/nginx-1.0.15.info
  rm /opt/lighttpd-1.4.30/lighttpd-1.4.30.info
  rm /opt/tntnet-2.1/tntnet-2.1.info
  rm /opt/hiawatha-8.1/hiawatha-8.1.info
  rm /opt/httpd-2.4.1/httpd-2.4.1.info
fi
ARG=" report"
# Check supplied argument
if [ "$1" == "$ARG" ]; then
  # Nginx
  lcov --directory /opt/nginx-1.0.15 --capture --output-file /
opt/nginx−1.0.15/nginx−1.0.15.info — base directory /opt/nginx−1.0.15
lcov — remove /opt/nginx−1.0.15/nginx−1.0.15.info " /usr*" — output /opt/nginx−1.0.15/nginx−1.0.15.info
genhtml −o /opt/report/1 /opt/nginx−1.0.15/nginx−1.0.15.info
cp /opt/nginx−1.0.15/nginx−1.0.15.info /opt/report/1/nginx−1.0.15.info

# Lighttpd
lcov — directory /opt/lighttpd−1.4.30/src — capture — output — file /opt/lighttpd−1.4.30/lighttpd−1.4.30.info
genhtml −o /opt/report/2 /opt/lighttpd−1.4.30/lighttpd−1.4.30.info
cp /opt/lighttpd−1.4.30/lighttpd−1.4.30.info /opt/report/2/lighttpd−1.4.30.info

# Ttnet
lcov — directory /opt/ttnet−2.1 — capture — output — file /opt/ttnet−2.1/ttnet−2.1.info
lcov — remove /opt/ttnet−2.1/ttnet−2.1.info " /usr*" — output /opt/ttnet−2.1/ttnet−2.1.info
genhtml −o /opt/report/3 /opt/ttnet−2.1/ttnet−2.1.info
cp /opt/ttnet−2.1/ttnet−2.1.info /opt/report/3/ttnet−2.1.info

# Hiawatha
lcov — directory /opt/hiawatha−8.1 — capture — output — file /opt/hiawatha−8.1/hiawatha−8.1.info
lcov — remove /opt/hiawatha−8.1/hiawatha−8.1.info " /usr*" " /opt/hiawatha−8.1/polarssl/*" — output /opt/hiawatha−8.1/hiawatha−8.1.info
genhtml −o /opt/report/4 /opt/hiawatha−8.1/hiawatha−8.1.info

# Apache httpd
lcov — directory /opt/httpd−2.4.1 — capture — output — file /opt/httpd−2.4.1/httpd−2.4.1.info
genhtml −o /opt/report/5 /opt/httpd−2.4.1/httpd−2.4.1.info
cp /opt/httpd−2.4.1/httpd−2.4.1.info /opt/report/5/httpd−2.4.1.info

fi
ARG="delrep"

# Check supplied argument
if [ "$1" == "$ARG" ]; then
DIRECTORY="/opt/report/1"
if [ −d "$DIRECTORY" ]; then
    echo "Delete $DIRECTORY?"
    rm −Ir $DIRECTORY
fi
DIRECTORY="/opt/report/2"
if [ −d "$DIRECTORY" ]; then
    echo "Delete $DIRECTORY?"
    rm −Ir $DIRECTORY
fi
DIRECTORY="/opt/report/3"
if [ −d "$DIRECTORY" ]; then
    echo "Delete $DIRECTORY?"
    rm −Ir $DIRECTORY
fi
DIRECTORY="/opt/report/4"
if [ −d "$DIRECTORY" ]; then
    echo "Delete $DIRECTORY?"
    rm −Ir $DIRECTORY
fi
DIRECTORY="/opt/report/5"
if [ -d "$DIRECTORY" ]; then
  echo "Delete "$DIRECTORY"?"
  rm -Ir "$DIRECTORY"
fi

DIRECTORY="/opt/report/5"
if [ -d "$DIRECTORY" ]; then
  echo "Delete "$DIRECTORY"?"
  rm -Ir "$DIRECTORY"
fi
fi
fi
fi