TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

Tuomas Haanpää

FUZZ TESTING COVERAGE MEASUREMENT BASED ON ERROR LOG ANALYSIS

Diplomityö
Tietotekniikan tutkinto-ohjelma
Huhtikuu 2016

OULUN YLIOPISTO
UNIVERSITY of OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Tuomas Haanpää

FUZZ TESTING COVERAGE MEASUREMENT BASED ON ERROR LOG ANALYSIS

Master's Thesis
Degree Programme in Computer Science and Engineering
April 2016

Haanpää T. (2016) Fuzz Testing Coverage Measurement Based On Error Log Analysis. Degree Programme in Computer Science and Engineering, University of Oulu, Oulu, Finland. Master's thesis, 70 p.

ABSTRACT

Fuzz testing is a black box testing method in which a SUT is subjected to anomalous inputs in order to uncover faults such as crashes or other incorrect behaviour. A defining attribute of any black box testing is the assumption that the inner workings and the source code of the SUT are unknown. This lack of information adds an element of difficulty to the task of estimating test coverage.

During testing a SUT typically produces a log of error conditions and other events which were triggered by the testing process. This log data is available even when the source code is not. The purpose of this study was to research whether a meaningful metric of test coverage could be extracted from the log data. The goal was to discover a metric similar to code coverage, but applicable to black box testing.

A hypothesis was presented that a large variety of observed events translated to great code coverage as well. To extract this metric, a rudimentary pattern recognition algorithm was devised in order to automatically classify the events encountered during a test run.

Measurements were performed on three open source SUTs representing three widely used communication protocols. Log analysis results were compared to code coverage measurements in order to study any possible correlation between them.

The results were positive, as the study showed clear correlation between the code coverage metric and the log analysis results for two of the three case studies. Further study is required to establish whether the studied log analysis method is generally applicable.

Keywords: software testing, black box testing, code coverage, edit distance, string classification

Haanpää T. (2016) Fuzz-testauksen kattavuusmittaus perustuen virhelokianalyysiin. Oulun yliopisto, tietotekniikan koulutusohjelma. Diplomityö, 70 s.

TIIVISTELMÄ

Fuzz-testaus on mustalaatikkotestausmenetelmä, jossa testikohteesta pyritään löytämään vikoja altistamalla se virheelliselle syötteelle. Mahdolliset ohjelmistoviat ilmenevät kaatumisina tai muuna virheellisenä toimintana. Mustalaatikkotestaukselle ominaista on se, että kohteen sisäistä toimintaa ja lähdekoodia ei tunneta, mikä tekee testauskattavuuden arvioinnista ongelmallista.

Testauksen aikana kohde tavallisesti tuottaa lokitiedoston, joka sisältää kohteessa havaitut virhetilat. Tämä lokiaineisto on käytettävissä myös silloin, kun lähdekoodia ei tunneta. Tämän tutkielman tarkoituksena on selvittää, onko mahdollista kehittää mittaustekniikka testikattavuuden arviointiin lokiaineiston perusteella. Tämä mittaustekniikka muistuttaisi koodikattavuusmittausta, mutta sitä voisi soveltaa myös mustalaatikkotestauksen yhteydessä.

Tutkielmassa esitetty hypoteesi oli se, että mikäli lokissa havaitaan suuri määrä erilaisia virhetiloja, myös koodikattavuus olisi korkea. Mittausten suorittamiseksi kehitettiin alkeellinen hahmontunnistusalgoritmi, joka luokitteli testauksen aikana kerätyn lokiaineiston.

Mittaukset toistettiin kolmella testikohteella, joiden lähdekoodi oli avointa, ja jotka edustivat yleisesti käytettyjä tietoliikenneprotokollia. Lokianalyysituloksia verrattiin koodikattavuusmittaustuloksiin, jotta mahdollinen korrelaatio tulosten välillä havaittaisiin.

Tutkimuksen tulokset olivat positiiviset, sillä kahdessa esimerkkitapauksessa kolmesta havaittiin selkeää korrelaatiota koodikattavuusmittausten ja lokianalyysitulosten välillä. Menetelmän yleinen sovellettavuus vaatii kuitenkin lisätutkimusta.

Avainsanat: ohjelmistotestaus, mustalaatikkotestaus, koodikattavuus, editointietäisyys, merkkijonojen luokittelu

CONTENTS

ABSTRACT

TIIVISTELMÄ

FOREWORD

ABBREVIATIONS

1. INTRODUCTION 8

2. SOFTWARE TESTING 9
2.1. Testing methodologies 9
2.2. Fuzzing 10
2.3. Code coverage as a testing metric 11
2.3.1. Statement coverage 12
2.3.2. Branch coverage 12
2.3.3. Function coverage 12

3. LOG ANALYSIS 13
3.1. Event classification 13
3.2. String distance metrics 15

4. CASE STUDIES 17
4.1. LDAP 17
4.1.1. OpenLDAP 18
4.1.2. OpenLDAP logging 19
4.2. SSL/TLS 20
4.2.1. OpenSSL 21
4.2.2. OpenSSL logging 21
4.3. HTTP 22
4.3.1. Apache HTTP Server 23
4.3.2. Apache HTTP Server logging 23

5. IMPLEMENTATION 24
5.1. Log collection 25
5.1.1. Fuzzing solution 25
5.1.2. Log collector 26
5.1.3. Code coverage measurements 28
5.2. Log analysis 29
5.2.1. Preprocessing 29
5.2.2. Classification 30
5.2.3. Postprocessing 32

6. RESULTS **33**
6.1. OpenLDAP results 33
6.2. OpenSSL results 37
6.3. Apache HTTP Server 43

7. DISCUSSION **46**
7.1. General observations 46
7.2. Research question 46
7.3. Future development 47

8. CONCLUSION **49**

9. REFERENCES **51**

10. APPENDICES **54**

FOREWORD

A large part of the work for this thesis was completed in the winter of 2015-2016, although some of the research was done in the previous year. The implementation of the measurement software as well as the measurements themselves took place from August to October, while the writing of the thesis was finished in April.

I would like to thank all those who provided assistance and guidance during this project, most notably my thesis supervisors Juha Röning and Thomas Schaberreiter, as well as my technical supervisor Rauli Kaksonen who initially suggested the topic of the thesis to me. Special thanks to Heikki Kortti for proofreading the thesis. I would also like to thank Lauri Piikivi and my team at Synopsys.

Finally I would like to thank my parents, Pertti and Marjukka, and my sister Tiina for their support, as well as all of my friends from the ABS.

Oulu, Finland April 22, 2016

Tuomas Haanpää

ABBREVIATIONS

C	Code coverage
C_b	Branch coverage
C_f	Function coverage
C_l	Line coverage
C_s	Statement coverage
N	Number of elements accessed
N_b	Number of branches executed
N_f	Number of functions executed
N_l	Number of lines executed
N_s	Number of statements executed
N_{bt}	Total number of branches
N_{ft}	Total number of functions
N_{lt}	Total number of lines
N_{st}	Total number of statements
N_t	Total number of elements
CI	Continuous Integration
CSV	Comma separated values
CVSS	Common Vulnerability Scoring System
DoS	Denial of Service
GCC	GNU Compiler Collection
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
LDAP	Lightweight Directory Access Protocol
LDAPv3	Lightweight Directory Access Protocol version 3
LTP	Linux Test Project
NCSA	National Center for Supercomputing Applications
OUSPG	Oulu University Secure Programming Group
RFC	Request for Comments
SNMP	Simple Network Management Protocol
SSL	Secure Sockets Layer
SUT	System Under Test
TCP	Transmission Control Protocol
TLS	Transport Layer Security
WWW	World Wide Web

1. INTRODUCTION

Software testing is an activity aimed at discovering faults in software products [1 p. 25]. Software testing can be divided to white box testing, in which the party performing the testing has access to the software source code, and black box testing, in which they do not.

Fuzzing or robustness testing is a black box testing method which consists of intentionally feeding broken or unusual inputs to the external interfaces of the software product [2 p. 24]. The aim of fuzzing is to reveal anomalous behaviour, such as crashes, memory leaks or denial-of-service conditions.

Code coverage is a metric used for measuring which parts of program code were executed during the execution of the program. Code coverage can be used to estimate the efficacy of testing, and while covering a part of the code does not guarantee that it is fault-free, it is certain that a fault residing in untested code will not be revealed [3].

Measuring code coverage requires access to source code, which makes it unsuitable for pure black box testing. This limitation also limits its use for measuring fuzzing effectiveness, as fuzzing falls into the black box domain of testing.

Modern software products produce a large amount of event log data to be used for debugging, forensics, performance analysis, etc. [4]. This log data is accessible even when software source code is not. The goal of this thesis is to investigate whether an event log data of a System Under Test (SUT) can be used to estimate a testing coverage metric similar to code coverage. It seems reasonable to assume that triggering a variety of log events and error conditions during testing translates to large code coverage as well.

The research question this thesis posits and aims to answer is whether a metric for testing coverage can be extracted from the SUT event log which satisfies the following conditions:

1. The metric has to consistently correlate with other, well known coverage metrics, such as code coverage

2. Calculating the metric must not require access to, or understanding of, the inner workings of the SUT

3. The metric has to be applicable to a variety of test targets

2. SOFTWARE TESTING

Software quality control is a collection of activities undertaken to ensure that a software product is of acceptable quality [5 p. 412]. These activities include a variety of software testing techniques. The purpose of testing is to discover problems in the software product [1 p. 25].

Software testing has its roots in the activity of debugging, often performed concurrently with software development, by the developers themselves. As the discipline of software engineering matured and the developed programs grew in size and complexity, the need for a more formal testing methodology was recognized as well. In his seminal work on the subject of software testing published in 1979, Myers recognizes that software developers should not test their own programs [6 p.12-13]. Today most software organizations employ QA teams dedicated entirely to software testing.

A more recent development in software development practices is agile software development, which focuses on iterative methods and a quick release cycle. In the classic waterfall method of development, testing takes place entirely after coding but before deployment and maintenance [7]. This rigid view is fundamentally incompatible with agile development.

Extreme Coding and Test-Driven Development have popularized the concepts of self-testing code and Continuous Integration (CI) [8]. Automated tests for functionality are written while, or even before, the functionality itself is implemented. CI ensures that every time new program code is added, the software under development is rebuilt and all existing tests are executed in order to prevent regression.

2.1. Testing methodologies

There are two general views which software testing can adopt: internal and external [5 p. 484]. Internal testing, also known as white box testing, focuses on the internal workings of the product, such as the logical paths and the collaborations of components. External testing, or black box testing, focuses on the external interfaces, the inputs and the outputs and their relationships, assuming no knowledge of the internal structure of the software.

Complete access to source code is a prerequisite for white box testing, but it is the only testing methodology which in principle makes 100% testing coverage possible [2 p. 80]. By viewing the source code, it is possible to identify every path of execution and ensure that they are all traversed in the course of the testing [9 p. 8]. However, it must be noted that this does not provide a guarantee that there are no bugs in the software. In addition, white box testing may result in a number of false positives, since it also tests code paths that may never be traversed in actual use [10 p. 9].

Black box testing is also called behavioral testing, and it is based on the requirements and the specification of the software [9 p. 8]. The purpose of behavioral testing is to ensure the software behaves as specified when exposed to different inputs. This method of testing is also called functional testing, but this term is misleading because black box testing can be used to test aspects other than functionality [2 p. 84].

Both white box and black box test methods have their merits [11]. Theoretically behavioural testing is sufficient to find all flaws, but it may take an infinite amount of

time to do so. Conversely, complete white box testing takes a finite amount of time, but it will not discover bugs which are only visible during execution.

Testing methods can be further divided into positive and negative testing. Positive testing aims to verify that the software performs its specified functionality when given valid input. Negative testing aims to verify that invalid or unexpected inputs do not cause the software to fail.

2.2. Fuzzing

Fuzzing is a negative black box testing technique where the SUT is fed unexpected or invalid data through its external interfaces. The purpose of fuzzing is to discover flaws which compromise its security, cause denial or degradation of service, or any other unwanted behaviour. [2 p. 24]

At its heart, fuzzing is a fairly simple brute force approach to vulnerability discovery. In their book, Sutton et al. describe it as "throwing everything but the kitchen sink at the target and monitoring the results" [10 p. 12]. However, they also state that it is possible to incorporate information on the SUT's inner workings to make fuzzing more surgical, which would make it more of a gray box than a black box technique.

Fuzzing as a term dates back to 1989, when Barton P. Miller developed Fuzz, a simple fuzzer which generated random character data in order to test the robustness of core UNIX utilities [12]. Although Fuzz was primitive, it was successfully used to demonstrate that 24-33% of the tested utilities were prone to crashing or hanging when presented with unexpected input data. Due to his pioneering work, Barton Miller is considered by many to be the father of fuzzing [10 p. 22].

A more systematic approach to fuzzing was taken by the OUSPG research group of University of Oulu in 1999 [11]. The mini-simulation method designed by OUSPG was implemented in a series of protocol test suites named PROTOS. Instead of simply generating random data, the PROTOS test suites each simulated a protocol implementation and were capable of generating valid protocol messages complete with e.g. correctly calculated checksums in place. The test suites were then capable of inserting anomalies into specific message fields, systematically covering the protocol specification. PROTOS test suites were developed for a number of protocols, including LDAPv3, HTTP and SNMP.

The design of the anomalies included within the PROTOS test suites drew inspiration from syntax testing devised by Boris Beizer [13 p. 284]. Beizer suggested anomalies such as elements with illegal contents, correct elements in incorrect contexts, illegal delimiters and empty input. The best test cases are those which supply the test target with data that is difficult to classify either as correct or incorrect.

In 2001, following the conclusion of the PROTOS project, Codenomicon Oy was founded by some of the members of the original research group to develop a commercial fuzzing solution based on the work done at OUSPG [14]. As part of their Defensics product line Codenomicon developed test suites for over 200 protocols. In 2015 Codenomicon was acquired by Synopsys, which continues to develop Defensics products.

2.3. Code coverage as a testing metric

It seems like an attractive proposition to prove that a software product does not contain bugs by testing it completely. However, complete testing is likely not possible in a finite amount of time, and only the simplest of programs can be shown to be bug-free. The question of allocating time for testing is not how much time it takes to test the product completely, but how much time it takes to test it adequately.

Various testing metrics have been utilized to measure the completeness of testing efforts. Metrics such as input space coverage, specification coverage and code coverage all have been used to measure fuzzing effectiveness to varying levels of accuracy [2]. This work focuses on code coverage, as it is believed to most closely reflect the results obtained from log analysis.

Code coverage metrics are used in software testing to indicate what percentage of the program code is exercised by the tests. Measuring a code coverage metric of any kind requires access to source code, which means they are only truly applicable to white box testing.

There are over a hundred recognized code coverage metrics available [15]. Most of these are outside the scope of this study, and only the few most relevant ones are discussed in this study. The most basic definition for code coverage is

$$C = \frac{N}{N_t} \tag{1}$$

where N is the total number of elements accessed and N_t is the total number of elements. The element type is dependent on the type of code coverage metric used. [16 p. 20].

Testing which strives to reach high or complete coverage is called coverage testing [16 p. 20]. Marick defines a high level of testing to be 90% [17]. The specific level of testing considered adequate depends on the organization doing the testing, but as stated before, in white box testing high levels of coverage are both attainable and desirable.

It must be noted that high level coverage does not guarantee that all bugs are discovered. Even a piece of code covered by testing may still contain a bug which manifests only under specific conditions. Still it is clear that a bug or a vulnerability residing in untested and unexecuted code most certainly will not be discovered [3]. While high coverage does not guarantee that every bug will be found, low coverage guarantees that most bugs will not be found.

In the context of fuzzing, code coverage metrics are somewhat more problematic. High levels of coverage are essentially unattainable. In a study performed by Rontti, code coverages reached by fuzzing OpenSSL and Zebra were all below 30% [16 p. 44]. This can be explained by the fact that fuzzing test cases are typically designed in black-box manner to cover protocol or interface specifications, not program code. Therefore only the code reachable through the tested interfaces can even theoretically be covered. For example, a program may have code to parse a configuration file, and this internal code is not reachable through an external interface [2 p. 60].

2.3.1. Statement coverage

Statement coverage is the simplest form of code coverage. It simply measures how many statements of the source code were executed out of the total number of statements available. The definition for statement coverage is

$$C_s = \frac{N_s}{N_{st}} \tag{2}$$

where N_s is the number of statements executed, and N_st is the total number of statements in the source code. [16 p. 20]

A metric very similar to statement coverage is line coverage, which evaluates non-comment lines of the source code instead of statements. Line coverage is more affected by programming style, because many languages allow multiple statements per line. Line coverage is defined as

$$C_l = \frac{N_l}{N_{lt}} \tag{3}$$

where N_l is the number of lines executed, and N_lt is the total number of lines in the source code. [16 p. 20]

2.3.2. Branch coverage

Branch coverage is a code coverage metric which calculates the number of branch conditions explored, such as if statements and loops. Full coverage requires execution of every statement, and that every branching condition is evaluated both true and false in the course of the testing. Branch coverage is defined as

$$C_b = \frac{N_b}{N_{bt}} \tag{4}$$

where N_b is the number of branch coverage conditions met, and N_bt is the total number of possible branch coverage conditions. [16 p. 20]

2.3.3. Function coverage

Function coverage gives a very rough estimate for code coverage by measuring how many functions are called during the execution. It is mostly used to check if there are modules or functions that are not called at all. Function coverage is defined as

$$C_f = \frac{N_f}{N_{ft}} \tag{5}$$

where N_f is the number of functions called, and N_ft is the total number of possible functions in the program. [16 p. 20]

3. LOG ANALYSIS

Modern computer systems produce a considerable amount of event log data that are used for purposes such as fault diagnostics, security incident reporting, performance analysis, etc. A typical event log is a text file to which the system continuously appends lines containing information on the events as they occur at a level of detail which is typically configurable by the user [18]. Log analysis can even be used to predict hardware failures before they occur to a large degree of accuracy [19].

Due to aforementioned benefits and the sheer amount of log data produced by an even moderately sized system, which makes manual parsing of said data impractical, there has been a growing interest in automated log analysis as a tool for system monitoring [4]. However, this task is complicated by the fact that there is no uniform, commonly accepted syntax for event logging.

Although there is an expired draft standard for a universal format for logger messages, no final specification exists [20]. The current state of affairs is that different software projects are left to define their own logging formats, which are often inconsistent and provide a varying level of detail on the workings of the software. Without clear guidelines on how to implement event logging, developers may misjudge the severity of the events as they are logged, and the content of the log files may evolve with new versions of the software [19].

In the context of fuzzing, log analysis offers an attractive method of estimating test coverage. As test cases are executed, different events and error conditions are triggered in the SUT. It seems intuitive that different events are produced and reported by different areas in the program code, in which case an event log showcasing a great variance of reported events would indicate better test coverage. This would mean that analyzing the SUT event log could provide a metric similar to code coverage, but applicable even when there is no access to the source code. Lack of access to source code must be assumed in the context of any black box testing [10 p. 19].

3.1. Event classification

Before a meaningful metric can be extracted from an event log, separate events must first be distinguished from each other. Due to the lack of uniform logging standards discussed in the previous chapter, this can prove to be a considerable technical challenge depending on the type of information that the analysis is aiming to extract from the event log.

A typical event log is a text file containing one or more lines of text for each event. Designing a system which automatically distinguishes different log events from each other is essentially a classification problem in the field of pattern recognition.

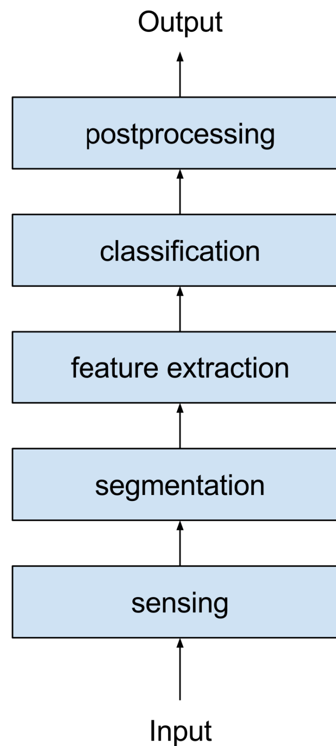

Figure 1. Components of a pattern recognition system

Figure 1 represents a simplified version of the components of a general pattern recognition system [21 p. 10]. The remainder of this chapter describes these components as they relate to the topic of log event classification.

The initial task of sensing consists of collecting the input data. This task is typically handled by the tested software itself which outputs the event log data either to a file or a standard output or standard error streams, or to a standard logging facility of the operating system. However, the system software may require configuration, for example for the amount of debug data included in the log.

Segmentation refers to distinguishing individual log events from one another. Segmentation is considered one of the most fundamental problems of pattern recognition in general [21 p. 10]. An event may be represented by any number of lines of log data without any immediately obvious structure. If the event is triggered by a protocol message received by the SUT, it may be stored in the log, further complicating the analysis.

The purpose of the feature extractor is to recognize in the event the data pertinent for classification, and to discard the rest. The goal is to identify the distinguishing features which are common for events in the same class, but different to events of other classes [21 p. 11]. These features are then utilized by the classifier to place the event in its appropriate class.

It must be noted that the distinction between a feature extractor and a classifier is not clearly defined. If the feature extractor is powerful and capable of extracting a set of clearly defined features with unambiguous decision criteria, the implementation of

the classifier becomes trivial. Conversely a highly sophisticated classifier can function even without clearly defined features. The distinction is practical rather than theoretical [21 p. 11].

The final step of the system is postprocessing. The classifier generally does not make a decision by itself, but instead recommends a course of action. Postprocessor may have information not available to the classifier, such as context-specific knowledge or outputs of other classifiers based on the other feature vectors. Postprocessor makes the final classification decision based on all available information [21 p. 13].

A further complication to the event classification problem is that the system may not have information on all of the possible classes of events, especially when the source code is not available. While it may be possible to extract different event messages from the software binary, identifying the event messages from all of the textual data may not be a simple task. Even if all event messages are correctly identified, not all of the events may be possible to trigger by fuzzing, or by any means at all.

3.2. String distance metrics

As described in the previous chapter, the complexity of the classification step is highly dependent on the quality of the features that can be extracted. In some cases, it may be as straightforward as a numerical event identifier, which makes classification trivial. In other cases, classification may become a much less clearly defined operation of clustering similar events based on some metric or metrics. Since the data is textual, edit distance metric seems like an intuitive choice.

Minimum edit distance is a well established metric for comparing similarity of two strings. Given symbol sequences X and Y, edit distance represents the minimum cost of transforming X into Y through a series of elementary weighted edit operations, such as insertion, deletion or substitution of a symbol [22]. Minimum edit distance was named by Robert Wagner and Michael Fisher in 1974, but it was independently discovered by multiple researchers [23 p. 154].

A well known edit distance metrics is the Levenshtein distance proposed by Vladimir Levenshtein in 1966 [23 p. 154-155]. The original Levenshtein distance is calculated as the minimum number of insertion, deletion and substitution operations, each with an equal cost of 1. Levenshtein also proposed an alternative version of the metric, in which the substitution operation is not allowed, doubling its cost to 2 since it has to be implemented as a combination of a deletion and an insertion.

Algorithm 1 Minimum edit distance

function MIN-EDIT-DISTANCE($target$, $source$) **return** $mindistance$
  $n \leftarrow LENGTH(target)$
  $m \leftarrow LENGTH(source)$
  Create distance matrix $distance[n+1, m+1]$
  $distance[0, 0] \leftarrow 0$
  for each column i from 0 to n **do**
    for each row j from 0 to m **do**
      $distance[i, j] \leftarrow MIN[distance[i-1, j] + inscost(target_i),$
        $distance[i-1, j-1] + substcost(source_j, target_i),$
        $distance[i, j-1] + delcost(source_i)]$
    end for
  end for
end function

Algorithm 1 illustrates a general edit distance algorithm [23 p. 156]. Minimum distance between two symbol sequences of arbitrary lengths is calculated by first creating a distance matrix with one column for every symbol in the target sequence and one row for every symbol in the source sequence. Every cell *distance[i,j]* represents the edit distance between the *i* first characters of the target and the *j* characters of the source.

Levenshtein distance was selected for this work, as it is well known, simple to calculate and applicable to strings of differing lengths. However, as Levenshtein distance is based only on the number of edits required, it does not account for the string lengths as a factor. It is clear that the distance of two edits is more significant for strings of three characters, than the distance of three edits is for strings of nine characters. [22].

A number of algorithms have been suggested for the task of string distance normalization [24]. The normalization algorithm chosen for this work is shown in equation 6.

$$NED(X, Y) = min\{\frac{W(P_{x,y})}{|X| + |Y|}\} \tag{6}$$

The normalized edit distance from symbol sequence X to sequence Y is calculated by dividing the edit distance with the sum of the sequence lengths. The normalization method violates triangle inequality and thus can not be considered a true mathematical metric [25]. It is quick to calculate however, and the accuracy is considered adequate for the purposes of this study.

4. CASE STUDIES

To test the metric presented in this study, three case studies were performed using different software products as test targets. While selecting the test targets, an effort was made to ensure that the test targets represent a variety of widely used software products and protocols. A vital criteria were also that the test targets were open source and implemented in C/C++ programming languages, so that the same method of code coverage measurement could be used for each SUT.

In this chapter each protocol and the associated SUT is described in detail. The results of the measurements performed for each case study are described in chapter 6.

4.1. LDAP

LDAP (Lightweight Protocol Access Protocol) is an application level network protocol for accessing directory services over an IP network [26]. LDAP is based on an earlier and more extensive directory service protocol known as X.500. While there is no direct mapping between LDAP operations and X.500 operations, LDAP essentially implements a subset of X.500 functionality.

A directory is a database which holds information on a set of entities. The fundamental characteristic which distinguishes a directory from other types of databases is that the information held by a directory is assumed to be relatively static [27 p. 8]. A directory can be described as "high-read, low-write", meaning that in typical use entries are read more often than edited.

While LDAP was originally intended only as a protocol to provide access to existing X.500 directory servers, it has to come to be synonymous with a specific type of directory architecture [27 p. 19]. The current and most widely used version of LDAP protocol is called LDAPv3, which was most recently revised in 2006. Its technical specifications are described in IETF RFCs 4510-4519[28].

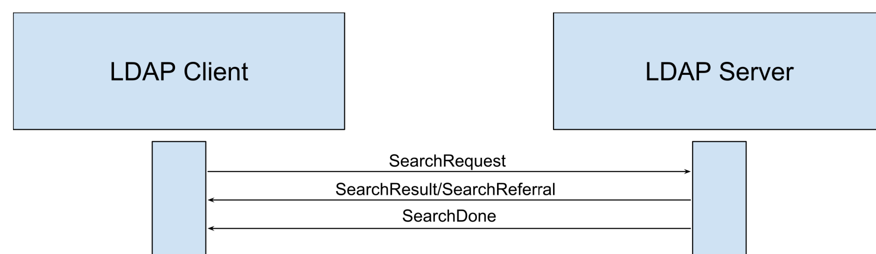

Figure 2. LDAP Search messaging sequence

Figure 2 illustrates a normal LDAP search operation. The client sends a search request which contains a search filter and other parameters related to the search and the expected results. The server responds with a number of Result and Referral messages, the latter of which may indicate another LDAP server to query. Finally the server returns a Done message, which indicates that all results have been returned.

4.1.1. OpenLDAP

OpenLDAP is a suite of LDAP servers, clients, utilities and libraries implementing a comprehensive implementation closely adhering to the LDAPv3 specification. Started in 1998, after the original LDAP project run by the University of Michigan became defunct, the OpenLDAP project salvaged the codebase of the existing reference implementation and continued its development. OpenLDAP is overseen by the OpenLDAP foundation, and it is included in almost every major Linux distribution [27 p. 19-20].

The LDAP server utility of OpenLDAP is called slapd (Stand-alone LDAP daemon). For the purposes of this study, slapd was configured with a default directory configuration, and a directory entry was added in order for the LDAP search operation to return a valid result. LDAP search was the only operation tested. This choice was made because the search operation is read-only, which means that the directory does not need to be reset between test cases for consistent testing, and the search operation contains a number of configurable options, resulting in a large variety of test cases.

By default, slapd uses the syslog utility for event logging, but OpenLDAP administrator's guide recommends that for debugging purposes syslog should be bypassed and the log data should be read from slapd's stderr output [29]. This is due to the slow and lossy nature of syslog, which discards messages if it does not have computing resources to process them properly.

Slapd defines a number categories of log messages. Any number of these messages can be enabled or disabled, depending on the type of information required. The complete listing of log message categories can be seen in Table 1 [29].

During the testing of OpenLDAP slapd utility, Codenomicon Defensics LDAP Server Test Suite version 3.0.0 was used for testing. Testing covered only the LDAP Search operation.

4.1.2. OpenLDAP logging

OpenLDAP offers a number of different debug levels, offering a varying amount of data concerning the events encountered. The different debug levels are described in table 1.

Table 1. slapd debug levels

Level	Keyword	Description
-1	any	enable all debugging
0		no debugging
1	(0x1 trace)	trace function calls
2	(0x2 packets)	debug packet handling
4	(0x4 args)	heavy trace debugging
8	(0x8 conns)	connection management
16	(0x10 BER)	print out packets sent and received
32	(0x20 filter)	search filter processing
64	(0x40 config)	configuration processing
128	(0x80 ACL)	access control list processing
256	(0x100 stats)	stats log connections/operations/results
512	(0x200 stats2)	stats log entries sent
1024	(0x400 shell)	print communication with shell backends
2048	(0x800 parse)	print entry parsing debugging
16384	(0x4000 sync)	syncrepl consumer processing
32768	(0x8000 none)	only messages that get logged whatever log level is set

The debugging level 0x8000 was considered an adequate compromise between log verbosity and information content. It seems reasonable to assume that levels which trace function calls (0x1 and 0x4) would provide useful information as well, but during the processing of the logs it became apparent that even the most rudimentary log verbosity produced logs which took a lot of processor time to process due to the Levenshtein distance calculations. A more efficient classification algorithm needs to be devised if this line of study is to be investigated in further depth.

```
1 55feaf43 connection_operation: conn 2910 unknown LDAP request 0x0
2 55feaf43 ber_get_int returns 0xffffffffffffffff
3 55feaf43 connection_operation: conn 2912 unknown LDAP request 0x0
```

Listing 1. OpenLDAP log

Listing 1 contains a sample from the OpenLDAP event log. A log event is preceded by an 8 digit hexadecimal timestamp, followed by the event description. As the format lacks simple event identifiers, it was technically the most challenging format for automatic parsing. The OpenLDAP logging is used to illustrate the implementation of the log event classification system in Chapter 5, which contains a more in-depth exploration of the processing of this log format.

4.2. SSL/TLS

TLS (Transport Security Layer) is a successor protocol to SSL (Secure Sockets Layer), which was originally developed to allow secure communication over a TCP channel. The core services of SSL and TLS include message confidentiality, message integrity and endpoint authentication [30 p. 737].

SSL was originally developed by Netscape Communications. The initial version SSL 1.0 was completed in 1994, but it suffered from major technical shortcomings and was only used internally by Netscape [31]. At the end of the same year Netscape released Netscape Navigator, a browser which implemented an improved SSL 2.0. This version still had some shortcomings, and in 1996 the specification for SSL 3.0 was published.

In 1996 IETF Transport Security Layer Working Group (IETF TLS WG) was formed in order to continue the development of the SSL protocol. The initial version of TLS protocol was published in 1999 as TLS 1.0. The differences between TLS 1.0 and SSL 3.0 are minor, but significant enough that the two protocols do not interoperate, though TLS 1.0 implements a mechanism to downgrade connection security to SSL 3.0 [32]. The most recent official version of TLS is TLS 1.2, though a working draft exists for TLS 1.3.

From a technical standpoint SSL/TLS implements an additional layer between the transport layer and the application layer, which offers similar services as TCP but with added security. Originally intended for HTTP, SSL/TLS is applicable to any application level protocol which utilizes TCP for transport [30 p.737-738]. It is a common practice to designate secure versions of application protocols by appending the letter S to their names: HTTP over TLS is called HTTPS, LDAP over TLS is called LDAPS, etc.

Figure 3 depicts the messaging sequence for a full TLS handshake [32]. During the sequence the client and the server establish each others' identities by exchanging certificates, negotiate session keys and other security parameters, and finally initiate secure data exchange.

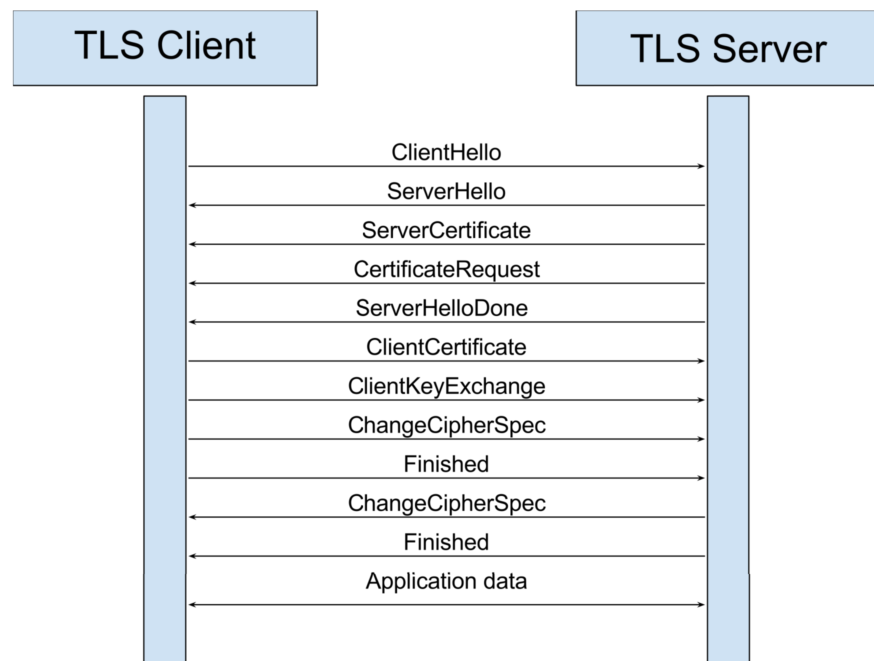

Figure 3. TLS messaging sequence

4.2.1. OpenSSL

The OpenSSL Project was founded in 1998 as a successor to SSLeay library when its development ceased [33 p. 21]. OpenSSL continued to work on the codebase of SSLeay to develop a fully open source SSL/TLS toolkit and cryptography library.

OpenSSL is licensed for use under Apache License 1.0, and it is or has been bundled by default in a number of operating systems both in the desktop and the server domains. This includes major Linux distributions, such as Debian, Ubuntu, Fedora and CentOS.

During the testing the OpenSSL s_server sample server utility was fuzzed using Codenomicon Defensics TLS Server Test Suite version 7.3.1. The fuzzing covered every message in basic handshake, re-handshake and abbreviated handshake test sequences. SSL3, TLS 1.0 and TLS 1.1 were covered. RSA was the only key exchange algorithm covered.

4.2.2. OpenSSL logging

Listing 2 presents a sample of the OpenSSL logging format. A total of five distinct events are present in the listing.

```
1  140672205448856:error:1408A0A0:SSL routines:ssl3_get_client_hello:
2  length too short:s3_srvr.c:947:
3  140672205448856:error:1408A0A0:SSL routines:ssl3_get_client_hello:
4  length too short:s3_srvr.c:947:
5  140672205448856:error:1408F081:SSL routines:SSL3_GET_RECORD:block
6  cipher pad is wrong:s3_pkt.c:452:
7  140672205448856:error:1408F081:SSL routines:SSL3_GET_RECORD:block
8  cipher pad is wrong:s3_pkt.c:452:
9  140672205448856:error:1408A09F:SSL routines:ssl3_get_client_hello:
10 length mismatch:s3_srvr.c:1147:
```

Listing 2. OpenSSL log

Brief investigation revealed that the eight digit hexadecimal value present in the first part of the log line is an event identifier. Based on the assumption that this identifier uniquely identifies a log event, supported by a comparison of identifiers and log events, the preprocessing of OpenSSL saves only this identifier per event for further processing.

```
1 1408A0A0
2 1408A0A0
3 1408F081
4 1408F081
5 1408A09F
```

Listing 3. Preprocessed OpenSSL log

Listing 3 shows the same log sample after preprocessing. Preprocessing is described in further detail in Chapter 5.2.1.

4.3. HTTP

The Hyper-Text Transfer Protocol (HTTP) is an essential technology of the World Wide Web (WWW) [30]. HTTP is an application level protocol which uses TCP for transport, and it is used by a web client to request objects such as HTML documents, images, applets, etc. from web servers. The technical specification for HTTP can be found in RFCs 1945 and 2616.

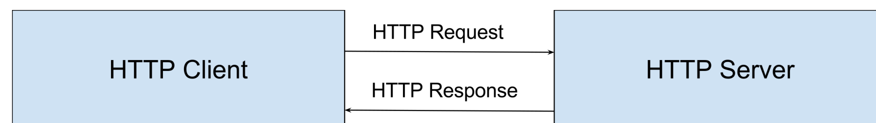

Figure 4. HTTP messaging sequence

Figure 4 depicts a normal HTTP message exchange. The client sends a request for a service or a resource, and the server responds with either the response which indicates the success or failure. The response message may also contain the requested resource.

4.3.1. Apache HTTP Server

Apache HTTP Server is based on the work done by Rob McCool who developed the HTTP daemon (httpd) while working for the National Center for Supercomputing Applications (NCSA) at University of Illinois [34]. When McCool left NCSA, the Apache Group was founded to carry on the development of httpd in 1995. The first public release of the Apache httpd was version 0.6.2 in April 1995.

Codenomicon Defensics HTTP Server test suite version 4.4.0 was used to fuzz Apache httpd. The test sequence was an HTTP Request-Response exhange, and the test suite covered every HTTP method.

4.3.2. Apache HTTP Server logging

In a manner similar to OpenSSL, httpd utilizes event identifiers in its logging format. The log format and the level of information logged are configurable in httpd [35]. The default format using log level "error" was used, which is illustrated by listing 4.

```
1  [Fri Dec 18 16:10:37.030794 2015] [core:error] [pid 9996:tid
2  140472634779392] [client 127.0.0.1:49856] AH00135: Invalid method
3  in request \xff\xfb\x01\xff\xfb\x03\xff\xfd\x18\xff\xfd\x1f /
4  HTTP/1.1
5  [Fri Dec 18 16:10:37.052229 2015] [core:error] [pid 9996:tid
6  140472626386688] [client 127.0.0.1:49860] AH00126: Invalid URI in
7   request \x1b]2;All your terminal are belong to us! Case index
8   1303\x07 / HTTP/1.1
9  [Fri Dec 18 16:10:37.074025 2015] [core:error] [pid 9996:tid
10 140472617993984] [client 127.0.0.1:49864] AH00135: Invalid method
11 in request GET\x7fGET / HTTP/1.1
```

Listing 4. Apache HTTP Server log

The events are identified by an event identifier beginning with prefix "AH". Similar to OpenLDAP, httpd stores many of the anomalies in the event log which makes the raw log large and difficult to parse. Fortunately the event identifier is simple to extract, making the preprocessed log much more compact. Listing 5 displays the log of listing 4 after it has been preprocessed. The preprocessing is described in chapter 5.2.1.

```
1  AH00135
2  AH00126
3  AH00135
```

Listing 5. Preprocessed Apache HTTP Server log

5. IMPLEMENTATION

This chapter describes in detail how the concepts presented in chapters 2 and 3 were implemented in order to find an answer to the research question presented in this study. Most of this chapter uses log excerpts from the OpenLDAP error log to illustrate the phases of the event classification system as described in chapter 3.1.

The primary research question of this study is whether a meaningful test coverage metric can be extracted from the event log produced by the SUT during fuzzing. In order to be useful, the metric needs to satisfy the following conditions:

1. The metric has to consistently correlate with other, well known coverage metrics, such as code coverage
2. Calculating the metric must not require access to, or understanding of, the inner workings of the SUT. In other words, it must be applicable to black box testing
3. The metric has to be applicable to a variety of test targets

Due to criteria 2 and 3, the analysis of the log data should not focus on the semantic data of the logged events, which are heavily target-specific. Listing 6 displays a log event from httpd log which is used as an example to illustrate the concept.

```
1 [Fri Dec 18 16:10:37.030794 2015] [core:error] [pid 9996:tid
2 140472634779392] [client 127.0.0.1:49856] AH00135: Invalid method
3 in request \xff\xfb\x01\xff\xfb\x03\xff\xfd\x18\xff\xfd\x1f /
4 HTTP/1.1
```

Listing 6. httpd log event

The event log entry contains information which would be useful in a diagnostics context. The log entry includes a process ID, the IP address and TCP port of the test suite, as well as a description of the error, which in this case is a malformed HTTP method. An automatic system capable of analyzing all of the semantic information contained in every log entry would need to be complex and tailored for each individual SUT, which does not serve the requirements of this study. Instead the focus should be on the analysis of the metadata on the logged events, such as how many distinct log events were observed and whether encountering a previously unobserved log event indicates that the testing has reached a previously uncovered section of the SUT.

In order to find an answer to the research question, a technical solution was developed with two phases:

1. Collect log data and a reference metric
2. Analyze the collected log data in order to extract a log-based coverage metric that correlates with the reference metric

The technical solution is described in detail in this chapter.

5.1. Log collection

This section describes the solution for generating and storing log data for later analysis. In order to analyse the results, it is not enough to only store the raw event log produced by the SUT. It is also necessary to retain the information on which test case triggered which event, in order to be able to rerun specific event categories for further analysis.

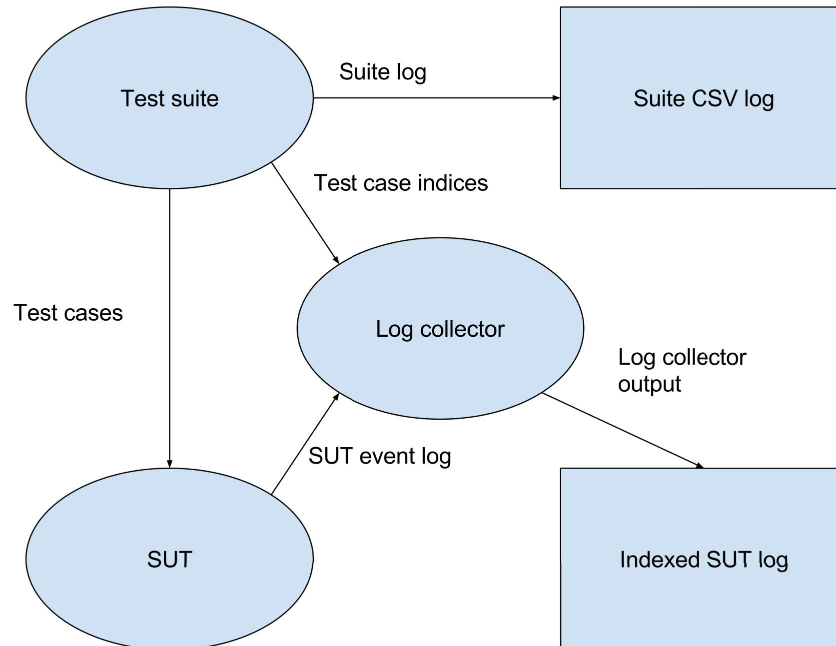

Figure 5. Illustration of log collection system

Figure 5 illustrates the general structure of the log collection system. The fuzzing solution, or test suite, produces test cases to fuzz the SUT. The log collector is a utility implemented in Python, which reads the test case indices and the SUT's event log in real time as the test cases progress. The final output of the system are the test suite's log in CSV format, as well as the SUT's event log complete with test case indices separating the logged events. The following subsections provide further detail on the individual components of the system.

5.1.1. Fuzzing solution

Codenomicon Defensics test suites were used to fuzz the test target. Defensics test suites are based on the mini-simulation method developed as part of the PROTOS research project in Oulu University. Defensics allows for detailed configuration of the test material, allowing the testing to focus on a particular section of the protocol specification. Defensics can also be configured to execute external scripts during the testing process, which can be leveraged to send information on the test case over TCP.

Defensics test suites are model-based fuzzers which implement a model of the protocol specification. The test suite's compatibility with the SUT is verified by executing

a *valid case*, which is a regular protocol interaction. During the actual testing, this same interaction is repeated for each test case, but with anomalies injected into the data fields of the protocol messages. After every test case the SUT is instrumented by repeating the valid case in order to check whether a Denial of Service (DoS) condition has been triggered.

5.1.2. Log collector

The log collector utility is implemented as a Python 2.7 script. It is a multithreading TCP server, which combines textual messages from multiple sources into a single, coherent output file. The collector receives its input from two sources: The test suite sends test case indices as the test run progresses, and at the same time, a simple shell utility tracks the SUT event log file and sends new lines to the collector as they appear.

To illustrate the function of the log collector, Listing 7 represents a sample of raw event log produced by OpenLDAP server utility slapd during fuzzing.

```
1 55feaf43 connection_operation: conn 2910 unknown LDAP request 0x0
2 55feaf43 ber_get_int returns 0xffffffffffffffff
3 55feaf43 connection_operation: conn 2912 unknown LDAP request 0x0
```

Listing 7. OpenLDAP log

This log represents three log events triggered by three sequential test cases, but this information is lost if only the raw log data is viewed. To reproduce specific log events after the fact it is necessary to know which test cases caused the events.

The log collector combines the raw log with test case indices sent by the test suite. These test case indices can be used later to identify and re-execute any test case as desired. The output of the log collector is illustrated by the Listing 8.

```
1 !TESTCASE: 1939
2 55feaf43 connection_operation: conn 2910 unknown LDAP request 0x0
3 !TESTCASE: 1940
4 55feaf43 ber_get_int returns 0xffffffffffffffff
5 !TESTCASE: 1941
6 55feaf43 connection_operation: conn 2912 unknown LDAP request 0x0
```

Listing 8. OpenLDAP log with added test case indices

In addition to the output of the collector, Defensics produces a suite log in csv format illustrated by Listing 9. This log contains information on the test cases themselves, which can be used for further analysis. The listing illustrates the data retained by the test suite and the format it is stored in.

```
1 2015-09-20 16:06:11.377,LDAPv3.TCP.ldap-search.ldap-search-req.
2 search-req.V.Message.messageID.L,1939, , , ,1,631,pass,0.023,
3 9.3, , ,471405ee1a1ff95c
4 2015-09-20 16:06:11.400,LDAPv3.TCP.ldap-search.ldap-search-req.
5 search-req.V.Message.messageID.L,1940, , , ,1,846,pass,0.022,
6 9.3, , ,622d63ce953afb53
7 2015-09-20 16:06:11.422,LDAPv3.TCP.ldap-search.ldap-search-req.
8 search-req.V.Message.messageID.L,1941,, , ,1,1062,pass,0.023,
9 9.3, , ,eabfa0313dbf7d90
```

Listing 9. Defensics CSV statistics

Each line in the file describes a test case as a list of fields separated by comma characters. To further illustrate the contents stored in this presentation, the first line of the excerpt is dissected. This line is displayed in Listing 10.

```
1 2015-09-20 16:06:11.377,LDAPv3.TCP.ldap-search.ldap-search-req.
2 search-req.V.Message.messageID.L,1939, , , ,1,631,pass,0.023,
3 9.3, , ,471405ee1a1ff95c
```

Listing 10. Single test case as Defensics CSV representation

Table 2 shows the contents and the description of every field as they pertain to this line of the CSV file.

Table 2. Description of the CSV file format of Defensics test suite results

2015-09-20 16:06:11.377	Timestamp
LDAPv3.TCP.ldap-search...	Test group
1939	Test case index
None	Status code
None	Number of input events
None	Number of input octets
1	Number of output events
631	Number of output octets
pass	Diagnosis
0.023	Execution time
9.3	CVSS score
None	Instrumentation rounds
None	Unused
471405ee1a1ff95c	Hash value of the test case

Timestamp indicates the time when the test case was executed. Test group describes the anomaly location within the message structure. In this case the anomaly is in the length field of a BER encoded structure containing the message ID of an LDAP search request. Test case index is used to identify a test case location in the test set with the current test suite configuration, and this index is later used in this work to match error log lines to specific test cases.

Status code is the status code returned by the SUT, if any. In this case, the SUT did not respond to the anomaly message, so this field remains empty, as well as the next two

fields indicating the number of received messages and the number of received octets respectively. The next two fields indicate the number of messages and the number of octets sent by the test suite during the test case execution respectively.

Diagnosis is the verdict given to whether the SUT passed the test case. The verdict is made by the test suite and is based on the instrumentation method used to monitor the health of the SUT. The verdict may be pass, fail or inconclusive. Execution time indicates the time in seconds spent on executing the test case, and CVSS score represents the Common Vulnerability Scoring System value assigned to this test case.

Instrumentation rounds describes the number of valid case instrumentation rounds spent on establishing SUT condition after the test case if valid case instrumentation was used. The last field is the hash value calculated for the test case. The hash is a unique test case identifier independent of test suite configuration.

5.1.3. Code coverage measurements

Code coverage was selected as the reference metric to study the validity of the log-based coverage metric. The theory of code coverage is more thoroughly explored in section 2.2. This section describes the method and the tools used for measuring code coverage.

GNU Compiler Collection, or GCC, is an integrated collection of compilers supporting multiple programming languages, such as C, C++, Java, etc. [36 p. 3] Historically the abbreviation stood for GNU C compiler, which refers to the compiler collection's original, much more narrow scope. Currently GCC is widely used and it is the default compiler of Linux.

GCC contains a coverage and profiling tool called gcov [36 p. 681]. When GCC is used to compile the SUT binary from source code, it can be configured to add additional data for code coverage measurements. After a program compiled in this fashion is executed, code coverage data be generated. The coverage data can be used to analyze line coverage, function coverage and branch coverage as described in section 2.2.

An additional tool used in this study is the Linux Testing Project (LTP) GCOV Extension, or lcov [37]. Originally designed for Linux kernel coverage measurements, but also applicable to standard user space applications, lcov formats the output of gcov into a collection of HTML documents. These HTML documents provide a visualization of gcov results, complete with a directory tree of analyzed source code files and bar graphs representing coverage information.

A factor considered for the selection of the test environment and the case study target software was that the same method of code coverage measurements could be used for each SUT. Every SUT was implemented in C/C++ programming languages, compiled using GCC with gcov options enabled, and the SUT compilation, execution and coverage measurements were performed on the same Linux platform.

5.2. Log analysis

The log analysis consists of three phases. Together they implement the segmentation, feature extraction, classification and postprocessing steps of the more generalized process described in chapter 3.1.

1. Preprocessing: Identification of logged events, or error conditions. These events are indicated by the presence of a line or lines in the log. This step of the processing must by its very nature be somewhat target-specific. The log is also trimmed of unnecessary data, to speed up and improve the later steps.

2. Classification: Classification of events identified in the first step. Same events often produce different log data due to e.g. time stamps, index numbers, anomaly data being stored in the SUT log, etc. The classification aims to eliminate these variables, and to label the events consistently.

3. Postprocessing: In the final phase, the classification results are presented to the user who can verify the classification, and manually fix obvious labeling errors. Finally the labeled log data is combined with the statistics file (see 3.2.2) into a single, coherent representation of the data logged by the suite and the processed and classified error log data.

5.2.1. Preprocessing

To illustrate the phases of processing the log data, excerpts from the SUT log at different points of the processing are presented in the remainder of this chapter. Unless otherwise stated, all sample log excerpts are from OpenLDAP, as it was the most technically challenging log format to process out of all the case studies.

The preprocessing is a simple process. The Python code developed for the purposes of this study is presented in Appendix 1.

```
1 55feaf7f conn=5571 op=0 do_search: invalid dn: "(|=thisis,DC=anoma
2 ly"
3 55feaf7f conn=5572 op=0 do_search: invalid dn: "((x|=thisis,DC=anom
4 aly"
5 55feaf80 conn=5573 op=0 do_search: invalid dn: "(.*.){16}|=thisis,DC=a
6 nomaly"
```

Listing 11. Similar events in OpenLDAP log

Listing 11 illustrates a core problem of the analysis. The lines each represent an error event, and it is clear that each line represents the same event. However, the log output is different each time, making simple string comparison useless. However, we can see a few commonalities between the log lines.

- Each line is preceded by an 8-digit hexadecimal number, which is a timestamp
- With the exception of the running index number named conn, the beginning of the line after the timestamp is consistent

- The string is terminated by the anomaly sent by the suite. This part of the string varies wildly, and should not have an effect on the classifcation.

The anomaly can also contain any number of line break characters, therefore the assumption that a log line equals an event is false. The characteristics listed above are fairly consistent over all events encountered in the OpenLDAP error log. Based on these characteristics, the log preprocessing is done based on two assumptions.

- A hexadecimal timestamp identifies an event
- The first 40 characters contain enough information for event classification

It should be noted that if these assumptions are false, we irreversibly lose valuable data in this step. For example, an event not preceded by a timestamp would simply be filtered out and lost.

During the preprocessing, the events are matched using a Python regular expression. The regular expression is presented in Listing 12.

```
1 '^[0-9a-f]{8} '
```

Listing 12. Python regular expression to match OpenLDAP event

In other words, the event must begin with 8 repetitions of valid hexadecimal digits (0-9, a-f), followed by a space. When an event is identified, it is truncated to the first 40 characters immediately after the space character. Naturally also the test case index indicators are preserved.

After preprocessing, the log excerpt from Listing 3 becomes that of Listing 13.

```
1 !TESTCASE: 1939
2 connection_operation: conn 2910 unknown
3 !TESTCASE: 1940
4 ber_get_int returns 0xffffffffffffffff
5 !TESTCASE: 1941
6 connection_operation: conn 2912 unknown
```

Listing 13. Preprocessed OpenLDAP log

As can be seen, the timestamps have been discarded, and the log lines have been truncated to first 40 characters. This log format is suitable for classification.

5.2.2. Classification

The preprocessed log still contains variables, such as index numbers, and possibly parts of the anomaly, which means classifying different events based on simple string comparison will fail. Ignoring numerical data in classifications is one option, but there may be relevant numerical data and non-relevant non-numerical data. A better alternative seems to be string distance calculation. Levenshtein string distance is used in the following fashion.

Appendix 2 presents a Python implementation of the classification process developed and utilized for the purposes of this work. A threshold value for normalized

string distance is required for correct classification, which was discovered by experimentation. A too large value causes loss of data, while a too small value increases the processing time and memory requirements, as well as the work required in the postprocessing step as more redundant labels are assigned.

1. When an event is encountered in the log, its distance is calculated to every other distinct event already encountered
2. If no match is found, meaning that no distance is under an agreed-upon threshold, the event is stored as a new label
3. If a match is found in the dictionary of already established labels, the event is replaced with the matching label. If multiple matches are found, the closest matching label is assigned.

Listing 14 displays the log excerpt of Listing 13 after classification.

```
1 !TESTCASE: 1939
2 connection_operation: conn 2910 unknown
3 !TESTCASE: 1940
4 ber_get_int returns 0xffffffffffffffff
5 !TESTCASE: 1941
6 connection_operation: conn 2910 unknown
```

Listing 14. Classified OpenLDAP log

The first two events are previously encountered, and are added into the dictionary of labels. The third event is matched to the first, and is replaced with that label.

As part of the classification step, the log data is re-formatted and combined with the statistics file produced by the test suite. The labels encountered in a single test case are combined into the format illustrated by Listing 15.

```
1 {label1;label2 ... labeln}
```

Listing 15. Format for storing event class labels

These are then added as an extra column into CSV formatted statistics file, to create a coherent, easily parseable format containing all of the relevant data in a single text file. The format of the appended CSV file is displayed in Listing 16.

```
1 2015-09-20 16:06:11.377,LDAPv3.TCP.ldap-search.ldap-search-req.
2 search-req.V.Message.messageID.L,1939, , , ,1,631,pass,0.023,9.3,
3  , ,471405ee1a1ff95c,{connection_operation: conn 2910 unknown}
4 2015-09-20 16:06:11.400,LDAPv3.TCP.ldap-search.ldap-search-req.
5 search-req.V.Message.messageID.L,1940, , , ,1,846,pass,0.022,9.3,
6  , , 622d63ce953afb53,{ber_get_int returns 0xffffffffffffffff}
7 2015-09-20 16:06:11.422,LDAPv3.TCP.ldap-search.ldap-search-req.
8 search-req.V.Message.messageID.L,1941, , , ,1,1062,pass,0.023,9.3,
9  , ,eabfa0313dbf7d90,{connection_operation: conn 2910 unknown}
```

Listing 16. Classified OpenLDAP log after reformatting

5.2.3. Postprocessing

Unless the number of labels is excessively large, they can be presented to the user, who can then select obviously similar but separate labels to be combined. The postprocessing script then parses through the classified log, and replaces each occurrence of the duplicate label with the correct one.

For this reason, when selecting the threshold value for the distance metric in the classification step, one should err on the side of false negatives in favor of false positives. False negatives in matching results in duplicate labels, which are easily rectified in the postprocessing step, whereas false positives lead to permanently mislabeled events which can not be recovered.

Appendix 3 presents a Python script created for the purpose of postprocessing. The labels in the classified log file are presented to the user, who can combine them as deemed fit to correct apparent classification errors.

6. RESULTS

The classification implementation described in Chapter 5 was used to perform a number of measurements performed on each case study SUT described in Chapter 4. This chapter describes the measurements, and both the expected and the actual results. A further discussion on the measured results as well as their implications can be found in Chapter 6.

For each test target, four test runs were executed in order to compare the log analysis and code coverage results and to observe possible correlations between the measured metrics.

1. Initial test run of about 1 million test cases
2. Re-run of 10000 test cases which produced no events in the event log of the SUT during test run 1
3. Re-run of 10000 test cases sampled from every event category observed during test run 1
4. Re-run of 10000 test cases randomly selected from the test cases executed during test run 1

According to the hypothesis, test run 2 should indicate no log activity and thus low code coverage values. Test run 3 should indicate high level of log activity and thus high code coverage, ideally equal to the levels measured for test run 1.

The test cases of test run 4 represent a control group. The code coverage values measured from the control are used to verify that selecting test cases based on log output provides higher code coverage measurements than a similar number of randomly selected test cases.

Essentially the goal is to perform corpus distillation based on event log output. If the significantly reduced test case count of test run 3 can achieve the same levels of code coverage as test run 1, it would indicate that the event log activity is a valid way of measuring test coverage.

The size of 10000 test cases was selected based on preliminary measurements. Based on the number of observed log events, 10000 test cases was considered more than adequate to cover all of the events for the re-runs, while still reducing the test case count from the initial one million by a factor of 100.

6.1. OpenLDAP results

For the initial test run, Defensics was configured to generate a maximum number of test cases by inserting anomalies into the LDAP search operation. The test material for this test run consisted of one million test cases, and the achieved coverage measurements are assumed to be the maximum coverage achievable with the Defensics LDAPv3 server test suite.

Table 3. Test run 1 log analysis results for slapd

Event label	Time encountered
No lines	166823
ber_get_int returns 0xffffffffffffffff	736
ber_peek_tag returns 0xffffffffffffffff	399
conn=110684 op=0 do_search: get_ctrls fa	43604
conn=2082 op=0 do_bind: unknown version=	1
conn=3261 op=0 do_delete: invalid dn (##	1
conn=3267 op=0 do_abandon: ber_scanf fai	1
conn=3283 op=0 do_bind: ber_scanf failed	1
conn=3288 op=0 do_modify: slap_parse_mod	1
conn=3294 op=0 do_modrdn: invalid newrdn	1
conn=3305 op=0 do_extended: unsupported	1
conn=3720 op=0 do_search: invalid dn: "#	6094
connection_input: conn=1048 deferring op	16
connection_operation: conn 2094 unknown	407
get_ava ber_scanf	4940
get_filter: conn 20920 unknown attribute	728779
get_filter: unknown filter type=4	10396
get_mra ber_scanf missing value	994
get_ssa: conn 157293 unknown attribute t	17115
slap_global_control: unrecognized contro	6983
Unique log events	19

Table 3 presents the results of log analysis performed on the log produced during test run 1. 166823 test cases produced no log output, while the remaining test cases produced at least one of the event types listed in the table. During the test run 19 different log events were measured for 981308 test cases.

Table 4. Test run 1 code coverage for slapd

	Hit	Total	Coverage	Normalized
Lines	12487	75863	16.5%	100%
Functions	780	2704	28.8%	100%
Branches	6897	60608	11.4%	100%

Table 4 presents code coverage measurements performed during the initial test run. As these measurements are thought the represent the maximum coverage achievable with the selected testing method, these values are used to normalize the remaining code coverage measurements within a range of 0-100%.

Table 5. Test run 2 log analysis results for slapd

Event label	Time encountered
No lines	9953
Unique log events	0

Table 5 displays results for test run 2. 10000 test cases which produced no log events were selected from those executed in test run 1. If the hypothesis is correct that new log events indicate new areas of code being reached, a log which shows no activity should indicate a low code coverage.

Table 6. Test run 2 code coverage for slapd

	Hit	Total	Coverage	Normalized
Lines	11424	75863	15.1%	91.49%
Functions	750	2704	27.5%	96.15%
Branches	6001	60608	9.9%	87.01%

Table 6 presents code coverage measurements for test run 2. Coverage is noticeably lower for test run 2, which is to be expected because the number of test cases is only 1% of the number of test cases executed in test run 1.

For test run 3, 10000 test cases were again selected from those executed in test run 1, but this time the goal was to select a sampling of every log event observed. Essentially this means that corpus distillation was performed based on the logged events.

Table 7. Test run 3 log analysis results for slapd

Event label	Time encountered
No lines	6
ber_get_int returns 0x0	736
ber_peek_tag returns 0xffffffffffffffff	399
conn=1083 op=0 do_search: get_ctrls fail	2
conn=1148 op=0 do_bind: unknown version=	1
conn=2060 op=0 do_delete: invalid dn (	1
conn=2066 op=0 do_abandon: ber_scanf fai	1
conn=2082 op=0 do_bind: ber_scanf failed	1
conn=2087 op=0 do_modify: slap_parse_mod	1
conn=2092 op=0 do_modrdn: invalid newrdn	1
conn=2102 op=0 do_extended: unsupported	1
conn=2445 op=0 do_search: invalid dn: "	1063
conn=3987 op=0 do_search: get_ctrls fail	2582
connection_input: conn=1002 deferring op	76
connection_operation: conn 1160 unknown	407
get_ava ber_scanf	919
get_filter: conn 3759 unknown attribute	1102
get_filter: unknown filter type=0	1065
get_mra ber_scanf	1185
get_ssa: conn 4209 unknown attribute typ	1063
slap_global_control: unrecognized contro	18249
Unique log events	20

Table 7 presents the results from test run 3. Interestingly one extra event was observed compared to test run 1. Most likely the extra event was present in the test run 1 log as well, but was lost due to misclassification. A total of 20 events were observed.

Table 8. Test run 3 code coverage for slapd

	Hit	Total	Coverage	Normalized
Lines	11399	75863	15.0%	91.29%
Functions	737	2704	27.3%	94.49%
Branches	6109	60608	10.0%	88.57%

Table 8 presents code coverage measurements for test run 3. Based on the initial hypothesis, these values should be much higher than the ones measured for test run 2. Clearly this is not the case, as the table indicates slightly higher branch coverage values but also slightly lower line and function coverage values.

Test run 4 represents a control sample. A 10000 test case subset was randomly selected from the set of test cases executed in test run 1.

Table 9. Test run 4 log analysis results for slapd

Event label	Time encountered
No lines	1766
ber_get_int returns 0x1	8
conn=10003 op=0 do_search: get_ctrls fai	424
conn=1044 op=0 do_search: invalid dn: ¨	56
connection_input: conn=1001 deferring op	2
connection_operation: conn 1020 unknown	6
get_ava ber_scanf	46
get_filter: conn 1254 unknown attribute	7424
get_filter: unknown filter type=0	100
get_mra ber_scanf missing value	10
get_ssa: conn 10211 unknown attribute ty	181
slap_global_control: unrecognized contro	1101
Unique log events	11

Table 9 presents log analysis results for test run 4. As expected, the number of observed log events is considerably lower than for test run 1. A total of 11 unique log events were observed for test run 4, which is about half of the total number of events observed for test run 1.

Table 10. Test run 4 code coverage for slapd

	Hit	Total	Coverage	Normalized
Lines	11500	75863	15.2%	92.10%
Functions	735	2704	27.2%	94.23%
Branches	6065	60608	10.0%	87.94%

Table 10 presents the code coverage measurements for test run 4. The observed coverage is close to the coverage measurements observed for test runs 2 and 3, and considerably lower than the code coverage observed for test run 1. This is contrary to the initial hypothesis, according to which the measurements for test run 2 should show a considerably lower coverage than for test run 4, and test runs 1 and 3 should show a coverage which is considerably higher than the one measured for test run 4.

6.2. OpenSSL results

This chapter describes the results for the test runs performed on OpenSSL. Chapter 5.1.3 describes in detail which operations and elements of the SSL/TLS protocol were tested.

Table 11. Test run 1 log analysis results for OpenSSL

Event label	Times encountered	Event label	Times encountered
No lines	384506	140890B0	30
0306E06C	1	140890E9	1047
04067072	466	14089106	4000
04067084	2	1408A09F	1734
0407006A	465	1408A0A0	8546
0408B004	53021	1408A0B7	1588
04091068	1965	1408A0BB	961
04091077	1568	1408A0C1	16040
0B07707D	53024	1408A0D7	3556
0D068066	228127	1408A0E3	162593
0D0680A8	11913	1408A10B	120
0D06A03A	990	1408B010	360
0D06C03A	61806	1408B094	210
0D07207B	16135	1408B0EA	12356
0D07209B	212002	1408C06F	5486
0D07803A	162724	1408C095	940
0D078079	2888	1408C09A	840
0D07808B	4	1408E098	14931
0D07808C	2161	1408E0F4	35434
0D078094	1000	1408F081	23420
0D07809F	1170	1408F0A0	3613
0D08303A	449506	1408F0C6	10800
0D08309F	180	1408F10B	89992
0D0BD098	180	1408F119	12857
0D0BD0DC	3430	14094067	6129
0D0C40D8	4194	14094085	3808
10067066	2	14094091	2077
140760D5	1317	140940E5	8491
140760D6	110	140940F5	1616
140760FC	11790	140940F6	2824
1407612A	145	14094153	216
1408807A	4002	140943E8	216
1408809F	2500	140A1097	388
140880DC	26507	140A1159	19
14088109	723	1412C150	3278
1408900D	201805	1412C151	966
14089087	1930	14140152	83
1408909F	4150		
Unique log events	74		

Table 11 presents log analysis results for test run 1. A total of 74 distinct log events were observed for the 1067930 executed test cases. This should represent the maximum number of log events and code coverage possible for the SUT.

Table 12. Test run 1 code coverage for OpenSSL

	Hit	Total	Coverage	Normalized
Lines	18683	105309	17.7%	100.0%
Functions	1561	6316	24.7%	100.0%
Branches	8533	65338	13.1%	100.0%

Table 12 presents code coverage measurements for test run 1. For the purposes of this study, this is considered the maximum coverage that can be achieved with the chosen fuzzing method.

Table 13. Test run 2 log analysis results for OpenSSL

Event label	Time encountered
No lines	9965
1408A0A0	1
1408F081	3
1408F0C6	1
1408F119	1
Unique log events	4

Table 13 presents the log analysis results test run 2. For this test run a subset of about 10000 test cases was selected, and the goal was to produce a log with no events. 4 log events were still observed, but the number is considered an acceptably small fraction of the 74 observed for test run 1. These events were not studied in depth, but they may be caused by transient conditions, or possibly by the stress caused by the number of connections and test cases introduced by the test suite.

Table 14. Test run 2 code coverage for OpenSSL

	Hit	Total	Coverage	Normalized
Lines	17309	105309	16.4%	92.6%
Functions	1496	6316	23.7%	95.8%
Branches	7745	65338	11.9%	90.8%

Table 14 presents code coverage measurements for test run 2. As expected the subset of 10000 cases produced considerably lower code coverage compared to test run 1. According to the hypothesis, this should be the minimum code coverage measured for the SUT.

Table 15. Test run 3 log analysis results for OpenSSL

Event label	Times encountered	Event label	Times encountered
No lines	2113	140890B0	30
0306E06C	1	140890E9	120
04067072	157	14089106	120
04067084	2	1408A09F	116
0407006A	157	1408A0A0	126
0408B004	874	1408A0B7	119
04091068	180	1408A0BB	120
04091077	163	1408A0C1	120
0B07707D	874	1408A0D7	120
0D068066	1608	1408A0E3	443
0D0680A8	139	1408A10B	120
0D06A03A	121	1408B010	122
0D06C03A	344	1408B094	120
0D07207B	331	1408B0EA	119
0D07209B	1278	1408C06F	133
0D07803A	1263	1408C095	120
0D078079	125	1408C09A	120
0D07808B	4	1408E098	119
0D07808C	120	1408E0F4	121
0D078094	120	1408F081	118
0D07809F	117	1408F0A0	120
0D08303A	6140	1408F0C6	119
0D08309F	119	1408F10B	120
0D0BD098	119	1408F119	121
0D0BD0DC	151	14094067	119
0D0C40D8	141	14094085	120
140760D5	120	14094091	119
140760D6	110	140940E5	383
140760FC	120	140940F5	120
1407612A	120	140940F6	119
1408807A	503	14094153	119
1408809F	131	140943E8	119
140880DC	437	140A1097	119
14088109	121	140A1159	19
1408900D	1890	1412C150	120
14089087	120	1412C151	123
1408909F	120	14140152	83
Unique log events	74		

Table 15 presents the log analysis results for test run 3. The goal was to select a subset of about 10000 test cases in such a fashion that every log event observed for test run 1 would be observed. The number of unique log events observed is identical to test run 1.

Table 16. Test run 3 code coverage for OpenSSL

	Hit	Total	Coverage	Normalized
Lines	18380	105309	17.5%	98.4%
Functions	1542	6316	24.4%	98.8%
Branches	8358	65338	12.8%	97.9%

Table 16 presents code coverage measurements for test run 3. The results show clearly that the measured code coverage is noticeably higher than it is for test run 2, though it does not quite reach the levels of test run 1. Every code coverage metric shows an increase, with the increase of branch coverage being the most significant.

Table 17. Test run 4 log analysis results for OpenSSL

Event label	Times encountered	Event label	Times encountered
No Lines	3567	140890E9	10
04067072	3	14089106	42
0407006A	3	1408A09F	24
0408B004	510	1408A0A0	78
04091068	22	1408A0B7	11
04091077	20	1408A0BB	16
0B07707D	510	1408A0C1	152
0D068066	2153	1408A0D7	30
0D0680A8	98	1408A0E3	1530
0D06A03A	10	1408A10B	1
0D06C03A	538	1408B010	6
0D07207B	185	1408B094	3
0D07209B	1969	1408B0EA	107
0D07803A	1554	1408C06F	46
0D078079	19	1408C095	7
0D07808C	18	1408C09A	6
0D078094	12	1408E098	147
0D07809F	15	1408E0F4	337
0D08303A	4180	1408F081	224
0D08309F	4	1408F0A0	35
0D0BD098	2	1408F0C6	104
0D0BD0DC	35	1408F10B	830
0D0C40D8	26	1408F119	131
140760D5	17	14094067	54
140760D6	1	14094085	30
140760FC	113	14094091	25
1407612A	3	140940E5	88
1408807A	45	140940F5	15
1408809F	29	140940F6	18
140880DC	255	14094153	3
14088109	7	140943E8	3
1408900D	1873	140A1097	7
14089087	18	1412C150	35
1408909F	26	1412C151	15
Unique log events	67		

Table 17 presents log analysis results for test run 4. 10000 test cases were selected randomly from the pool of test cases executed in test run 1, with the analysis showing 67 observed log events, which represents a 9.5% drop from the 74 events observed in test run 1.

Table 18. Test run 4 code coverage for OpenSSL

	Hit	Total	Coverage	Normalized
Lines	17910	105309	17.0%	95.9%
Functions	1517	6316	24.0%	97.2%
Branches	8145	65338	12.5%	95.5%

Table 18 shows code coverage measurements for test run 4. The coverage results indicate a coverage that is higher than the coverage measured for test run 2, but lower than the coverage measured for test run 3. This conforms to the initial hypothesis.

6.3. Apache HTTP Server

This section describes the measurement results for HTTP daemon testing. Chapter 4.3.1 contains more information on httpd and what elements of the HTTP protocol were tested.

Table 19. Test run 1 log analysis results for httpd

Event label	Time encountered
No lines	1071129
AH00036	343
AH00126	1685
AH00127	81
AH00135	1392
Unique log events	4

Table 19 present log analysis results for test run 1. A total of 7 distinct log events were observed. The test run consisted of 1074630 test cases. This should be the maximum number of unique events observed for the SUT, and it should also represent the highest code coverage measured.

Table 20. Test run 1 code coverage for httpd

	Hit	Total	Coverage	Normalized
Lines	7437	21831	34.1%	100.0%
Functions	662	1514	43.7%	100.0%
Branches	3765	18460	20.4%	100.0%

Table 20 present code coverage measurements for test run 1. These are assumed to be the highest code coverage results achievable with the chosen fuzzing method for

the purposes of this study, and as in the previous chapters these results are used for normalization of the remaining code coverage measurements.

Table 21. Test run 2 log analysis results for httpd

Event label	Time encountered
No lines	10000
Unique log events	0

Table 21 presents log analysis results for test run 2. A subset of 10000 test cases were selected from those executed in test run 1. The goal was to produce an event log with no events, and this goal was met.

Table 22. Test run 2 code coverage for httpd

	Hit	Total	Coverage	Normalized
Lines	6896	21831	31.6%	92.7%
Functions	641	1514	42.3%	96.8%
Branches	3309	18460	17.9%	87.9%

Table 22 presents the code coverage measured for test run 2. As expected, the coverage measurements indicate a considerably lower coverage than for test run 1. The code coverage measurements for test runs 3 and 4 should be considerably higher.

Table 23. Test run 3 log analysis results for httpd

Event label	Time encountered
No lines	6705
AH00036	343
AH00126	1684
AH00127	81
AH00135	1389
Unique log events	4

Table 23 shows the log events observed for test run 3. About 10000 test cases were executed, and the goal was to reproduce the same error conditions that were observed for test run 1. The number of log events is so small, that it is trivial to observe that the same events are present in both logs.

Table 24. Test run 3 code coverage for httpd

	Hit	Total	Coverage	Normalized
Lines	7209	21831	33.0%	96.9%
Functions	656	1514	43.4%	99.1%
Branches	3519	18460	19.1%	93.4%

Table 24 represents code coverage measurements for test run 3. Higher coverage ratings were observed than the ones measured for test run 2 as per the initial hypothesis. The code coverage measurements for test run 4 should be lower.

Table 25. Test run 4 log analysis results for httpd

Event label	Time encountered
No lines	10090
AH00036	1
AH00126	16
AH00135	17
Unique log events	3

Table 25 represents log analysis results for test run 4. The subset of about 10000 test cases was selected randomly. The randomly selected test cases were able to reproduce only three of the observed four log events.

Table 26. Test run 4 code coverage for httpd

	Hit	Total	Coverage	Normalized
Lines	7188	21831	32.9%	96.7%
Functions	658	1514	43.5%	99.4%
Branches	3462	18460	18.8%	92.0%

Table 26 shows code coverage measurements for test run 4. The differences from coverage metrics measured for test run 3 were negligible. The code and function coverages were slightly elevated, while branch coverage was slightly reduced. However, the difference in the number of log events was also small between test runs 1, 3 and 4, and therefore the interesting result is that the only one to show a signifant difference in either log analysis result or code coverage measurement was test run 2.

7. DISCUSSION

This chapter presents a discussion of the test results presented in the previous chapter. The chapter begins with general observations made during the implementation phase of this work, proceeds to verify whether results confirmed the initial research question of this study, and finishes with the prospects of future development on this topic.

7.1. General observations

Initially it seems clear that the implementation of the event logging facilities varies greatly between the SUTs. All three SUTs use an entirely different default formatting for the log messages, as well as different configuration options for the data that gets logged. Therefore any design of an automatic log analysis solution must take into account the lack of uniform standards in event logging.

It was also noted that simple, machine readable event identifiers simplified the log analysis process significantly. The analysis of the OpenLDAP log, the only case study which did not utilize such identifiers, required the use of a rudimentary pattern recognition algorithm which significantly increased the complexity of the problem. The risk of event misclassification is also significantly higher compared to simple string comparison which can be utilized when event identifiers are present.

It was also typical for the SUTs to store the anomaly data sent in the event log messages apparently unsanitised. In the case of overflow anomalies and special characters such as line breaks the logs grew large and difficult to read. This could be used by an attacker to sabotage the system logs or exhaust the hard drive space of a poorly configured system.

The logging facilities for each of the case studies were also poorly documented. For example, httpd includes error codes starting with characters "AH" in its logs, but the official documents do not list the codes or their meanings. For example, one could study how many of the error conditions defined in the documentation can actually be triggered by fuzzing.

When discussing these shortcomings of implementation and documentation, it should be remembered that one criterion for selecting the SUTs in this study was that they are mature and widely used software. Considering the logging facilities, which can be described as ad-hoc and lacking standardization, it can perhaps be assumed that the quality of logging in general is not high.

7.2. Research question

This study set out to investigate whether a general purpose method for estimating software testing coverage based on test target event logs can be established. In order to do this, a corpus distillation of the test material was done based on event logs, and a comparison of code coverage metrics was performed after test runs executed with the full set of test cases and the reduced set.

Initially it should be noted that for the reasons listed earlier in this chapter, creating a fully general purpose analysis solution is extremely difficult. As there is no uniform

standard of logging, efficient and accurate event classification will require some degree of modification for each SUT, or a much more sophisticated pattern recognition algorithm than the one utilized for this work.

Looking at the number of different log events, a large variance in numbers can be observed. OpenSSL produces 74 different log events (see table 10), while httpd produces just 4 (see table 18). Differences can be attributed to differences in logging quality, but it should also be noted that the SSL/TLS handshake is a significantly more complicated operation than an HTTP Request.

When comparing the code coverage results for the reduced test sets, it seems clear that OpenSSL results show the most promise. The most significant drop in normalized code coverage metrics is 2.1% for branch coverage when high log event variance is sought (see table 3), while a low variance results in drops as a high as 9.2% (see table 13). Comparing results across all reduced test sets, the code coverage metrics seem to display direct correlation with log event variance.

The results for OpenLDAP do not show similar correlation. OpenLDAP code coverage results for the reduced test sets seem to show little difference, and code coverage seems to correlate more strongly with test case count than the number of observed log events. It must be noted that OpenLDAP was the only case study for which a pattern recognition algorithm was applied for string matching, which may introduce an additional source of measurement errors.

Using default configuration, httpd logging seems the most rudimentary. Only 4 different error conditions were observed, and a vast majority of test cases triggered no logged error conditions. The code coverage measurements however displayed behaviour similar to OpenSSL, with high variance in log events correlating with higher code coverage measurements.

Concerning the research question, the OpenSSL and httpd results indicate that it is possible to estimate fuzzing code coverage based on log events. However, the failure of the method for OpenLDAP also indicates that the method is either not reliable, or it is applicable only for a certain subset of SUTs. The method is likely limited both by the complexity of the protocol and the quality of the logging implemented in the SUT. The latter of the two may be difficult to estimate without access to source code.

7.3. Future development

Code coverage was selected as the method for a reference coverage metric. It may be of interest to study the correlation of the log analysis measurement to other reference metrics, such as specification coverage and input space coverage. Log analysis may also offer more advanced metrics than the number of unique log events used for this study.

As stated earlier in this chapter, the results were not consistent across the different SUTs. Further investigation is necessary to determine what factors have the most detrimental effect on the quality of the measurements. The positive results achieved with OpenSSL and httpd should be verified with other implementations, such as GnuTLS and Nginx.

The file format described in Chapter 4 lends itself for further statistical analysis. For example, studying a possible correlation between the type and density of log events with the test group reported by the test suite might yield interesting results.

8. CONCLUSION

When a program is subjected to fuzz testing, it produces an event log detailing the different error conditions triggered during the course of the testing. The purpose of this study was to investigate whether it is possible to develop a testing coverage metric based on the event log data. The motivation for such a metric stems from the fact that fuzz testing is a black box testing method, which implies lack of knowledge of the inner workings of the SUT. Event log data is available for analysis even when the source code is not.

A set of criteria was established in order to measure how well a log-based coverage metric would satisfy the purposes of the study: the metric would have to correlate with a well-known and established metric for coverage, the calculation of the metric should not require knowledge of the inner workings of the SUT, and the metric should be general purpose and applicable to a variety of test targets.

The metric developed in the course of the study was based on automatically classifying the different error conditions appearing in the log. Based on the hypothesis that a new type of an event observed in the log indicates a new area of program code having been reached, a correlation between the number of distinct events and code coverage was studied.

A problem encountered in the implementation of the event classification algorithm was the lack of uniform standards in event logging and the lack of documentation of log output. The problem of event classification ranged from trivial to challenging, depending on the format of the particular log. A simple string distance algorithm was implemented to measure similarity of log events for the purpose of event classification.

In order to measure the correlation between the developed metric and code coverage a set of measurements were performed. A commercial fuzz testing tool was used to establish a baseline of testing coverage by executing a full set of test cases. Following this, several condensed sets of test cases were executed based on the log output they generated during the initial test run. The goal was to measure whether code coverage increased with the number of distinct log events when the total number of test cases remained the same. A control set of randomly selected test cases was also executed in order to study whether test case selection based on maximizing log event variance showed better results than random selection.

The measurements were repeated for three different case studies. For each of the case studies an open source SUT was selected in such a way that they represented a selection of varied but widely used and mature protocols and software products. Despite the variance, it was considered important that each case study was performed in the same test environment using the same coverage measurement tools. The selected protocols were LDAP, HTTP and TLS, and the SUTs were OpenLDAP, Apache HTTP Daemon and OpenSSL respectively.

The results of the case studies indicated a significant degree of correlation between the coverage measurements in two of the three case studies. In the cases of HTTP and TLS, test case selection based on maximizing log event variance showed improved code coverage results compared to test case selection based on minimizing log event variance or random test case selection. A similar correlation was not observed for LDAP.

As the final conclusion, the results of this study clearly indicate that the log analysis method investigated is a viable method of estimating testing coverage. While some questions about the general applicability of the method remain due to the results of the LDAP case study, the issue may be resolved with a more advanced method of log event classification.

9. REFERENCES

[1] Kaner C., Falk J. & Nguyen H.Q. (1999) Testing Computer Software. Wiley Computer Publishing.

[2] Takanen A., DeMott J. & Miller C. (2008) Fuzzing for Software Security Testing and Quality Assurance. Artech House, Inc., Norwood, MA, 1. ed.

[3] Miller C. & Peterson Z.N. (2007) Analysis of Mutation and Generation-Based Fuzzing. Independent Security Evaluators.

[4] Tang L. & Li T. (2010) LogTree: A Framework for Generating System Events from Raw Textual Logs. In: 2010 IEEE 10th International Conference on Data Mining (ICDM). Sydney, NSW, pp. 491–500.

[5] Pressman R.S. (2010) Software Engineering: A Practitioner's Approach. McGraw-Hill, 7. ed.

[6] Myers G.J. (1979) The Art of Software Testing. John Wiley & Sons, Inc.

[7] Royce W.W. (1970) Managing the Development of Large Software Systems. Proceedings of IEEE WESCON 26, pp. 1–9.

[8] Fowler M., Continuous Integration (accessed on 23.2.2016). URL: `http://www.martinfowler.com/articles/continuousIntegration.html`.

[9] Beizer B. (1995) Black-Box Testing. John Wiley & Sons, Inc.

[10] Sutton M., Greene A. & Amini P. (2007) Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley.

[11] Kaksonen R. (2001) A Functional Method for Assessing Protocol Implementation Security. Licentiate thesis, VTT Technical Research Centre of Finland, VTT Publications 448, Espoo, Finland , p. 128.

[12] Miller B.P., Fredriksen L. & So B. (1989) An Empirical Study of the Reliability of UNIX Utilities. Tech. rep., University of Wisconsin, Madison.

[13] Beizer B. (1990) Software Testing Techniques. Van Nostrand Reinhold, 2. ed.

[14] Codenomicon Company Backgrounder (accessed on 10.2.2016). URL: `http://www.codenomicon.com/files/pdf/Codenomicon-Company-Backgrounder.pdf`.

[15] Kaner C. (1996) Software Negligence and Testing Coverage. Proceedings of STAR 96, p. 313.

[16] Rontti T. (2004) Robustness Testing Code Coverage Analysis. Master's thesis, University of Oulu, Department of Electrical and Information Engineering.

[17] Marick B., How to Misuse Code Coverage (accessed on 22.2.2016). URL: `http://www.exampler.com/testing-com/writings/coverage.pdf`.

[18] Andrews J.H. (1998) Theory and Practice of Log File Analysis. Tech. rep., Department of Computer Science, University of Western Ontario.

[19] Jayathilake D. (2012) Towards Structured Log Analysis. In: 2012 International Joint Conference on Computer Science and Software Engineering (JCSSE), Bangkok, pp. 259 – 264.

[20] Universal Format for Logger Messages (accessed on 22.11.2015). URL: `https://tools.ietf.org/html/draft-abela-ulm-05`.

[21] Duda R.O., Hart P.E. & Stork D.G. (2001) Pattern Classification. John Wiley & Sons, Inc., 2. ed.

[22] Marzal A. & Vidal E. (1993) Computation of Normalized Edit Distance and Applications. IEEE Transactions on Pattern Analysis and Machine Intelligence 15, pp. 926–932.

[23] Jurafsky D. & Martin J.H. (2000) Speech and Language Processing. Prentice Hall, Inc., 1. ed.

[24] Weigel A. & Fein F. (1994) Normalizing the Weighted Edit Distance. In: Proceedings of the 12th IAPR International Conference on Pattern Recognition, Jerusalem, pp. 399–402.

[25] Yujian L. & Bo L. (2007) A Normalized Levenshtein Distance Metric. IEEE Transaction on Pattern Analysis and Machine Intelligence 29, pp. 1091–1095.

[26] Sermersheim J. (2006), Lightweight Directory Access Protocol (LDAP): The Protocol (accessed on 22.2.2016). URL: `https://tools.ietf.org/html/rfc4511`.

[27] Butcher M. (2007), Mastering OpenLDAP: Configuring, Securing and Integrating Directory Services.

[28] Zeilenga K.D. (2016), Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map (accessed on 22.2.2016). URL: `https://tools.ietf.org/html/rfc4510`.

[29] OpenLDAP Software 2.4 Administrator's Guide (accessed on 4.12.2015). URL: `http://www.openldap.org/doc/admin24/`.

[30] Kurose J.F. & Ross K.W. (2013) Computer Networking: A Top-Down Approach. Pearson Education Limited, 6. ed.

[31] Oppliger R. (2009) SSL and TLS: Theory and Practice. Artech House, Inc.

[32] Dierks T. & Allen C. The TLS Protocol version 1.0 (accessed on 8.1.2016). URL: `https://tools.ietf.org/html/rfc2246`.

[33] Viega J., Messier M. & Chandra P. (2002) Network Security with OpenSSL. O'Reilly & Associates, Inc., 1. ed.

[34] Apache HTTP Server Project (accessed on 13.1.2016). URL: https://httpd.apache.org/ABOUT_APACHE.html.

[35] Apache HTTP Server Version 2.4 Documentation (Accessed 6.3.2016). URL: https://httpd.apache.org/docs/2.4/.

[36] Stallman R.M. & the GCC Developer Community, Using the GNU Compiler Collection (accessed on 22.2.2016). URL: https://gcc.gnu.org/onlinedocs/gcc/.

[37] Readme file for the LTP GCOV extension (LCOV) (accessed on 22.2.2016). URL: http://ltp.sourceforge.net/coverage/lcov/readme.php.

10. APPENDICES

Appendix 1: A Python script for preprocessing event log data
Appendix 2: A Python script for performing string classification
Appendix 3: A Python script for postprocessing classification results

Appendix 1: A Python script for preprocessing event log data

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
'''
@author : Tuomas Haanpää @copyright : 2016 Tuomas Haanpää
@license :
Copyright All rights
3 Clause BSD License
( c ) 2016 Tuomas Haanpää
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 re
import logutil
import sys

from optparse import OptionParser, make_option

def filterOpenldap(line):
'''
Filter logline by matching it to OpenLDAP log line format, then
truncatingto 40 characters
parameters:
line - the logline
'''
  if re.match('[0-9a-f]{8} ', line):
    result = line[9:49].strip()
    if(result.endswith("\n") == False):
      result = result + "\n"
    return result
  else:
```

```
56        return None
57
58  def filterHttpd(line):
59  '''
60  Filter logline by matching it to Apache HTTP Server log line format
61  parameters:
62  line - the logline
63  '''
64      match = re.search('AH[0-9]{5}:', line)
65      if match:
66          result = line[match.start(0):match.start(0)+7].strip()
67          result = result + "\n"
68          return result
69      else:
70          return None
71
72  def filterSsl(line):
73  '''
74  Filter logline by matching it to OpenSSL s_server log line format
75  parameters:
76  line - the logline
77  '''
78      match = re.search('error:[0-9A-F]{8}:', line)
79      if match:
80          result = line.split(' ')[0].split(':')[2] + '\n'
81          return result
82      else:
83          return None
84
85  def filterLog(logFile, outputFile,filterFunction):
86  '''
87  Filter logFile
88  The log lines are matched with a filter function, non-matching are
89  rejected, matching are truncated and stored to outputFile
90
91  parameters:
92  logfile - input file containing the log data
93  outputfile - output file
94  filterFunction - function used to filter the loglines
95  '''
96      print "Pre-processing logs..."
97      skip = True
98      loglinecount = 0
99      testcasecount = 0
100     for line in logFile:
101         #Skip lines until first testcase is encountered
102         if skip == True:
103             if logutil.getTestcaseIndex(line) > 0:
104                 #First test case encountered
105                 lines = [line]
106                 testcasecount = 1
107                 skip = False
108                 caseindex = int(line.split(" ")[1])
109             else:
110                 pass
111         else:
```

```
112         if logutil.getTestcaseIndex(line) > 0:
113           #Output the line array of the last test case
114           if (len(lines) > 0):
115               for logline in lines:
116                 outputFile.write(logline)
117           #create new line array for this test case
118           lines = [line]
119           testcasecount += 1
120           caseindex = int(line.split(" ")[1])
121         else:
122           #Apply the chosen filter to the line
123           filteredLine = filterFunction(line)
124           if filteredLine != None:
125             loglinecount = loglinecount + 1
126             if line not in lines:
127               lines.append(filteredLine)
128 
129     if len(lines) > 1 and caseindex <= lastCase:
130       for logline in lines:
131         outputFile.write(logline)
132 
133     print "Pre-processing complete"
134     return (testcasecount, loglinecount)
135 
136 def parseOptions():
137       global optionParser
138       option_list = [
139       make_option('-l', '--log',
140         action='store',
141         dest='log_file',
142         help='log file to analyze'),
143       make_option('-o', '--output',
144         action='store',
145         dest='output_file',
146         help='CSV file containing results'),
147       make_option('-f', '--filter',
148         action='store',
149         dest='filter',
150         help='Log filter to use (httpd, openldap, openssl)')
151       ]
152       usage = "\t%prog [options]"
153       optionParser = OptionParser(
154         option_list = option_list, usage=usage)
155       return optionParser.parse_args()
156 
157 if __name__=="__main__":
158     (opts, rest) = parseOptions()
159 
160     if opts.filter == "openldap":
161       filterFunction = filterOpenldap
162     if opts.filter == "httpd":
163       filterFunction = filterHttpd
164     if opts.filter == "openssl":
165       filterFunction = filterSsl
166 
167     inputf = open(opts.log_file, 'r')
```

```
168    outputf = open(opts.output_file, 'w')
169    (cases, lines) = filterLog(
170      inputf, outputf, filterFunction)
171    inputf.close()
172    outputf.close()
```

Appendix 2: A Python script for performing string clasiffication

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

'''
@author : Tuomas Haanpää @copyright : 2016 Tuomas Haanpää
@license :
Copyright All rights
3 Clause BSD License
( c ) 2016 Tuomas Haanpää
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.
'''

from optparse import OptionParser, make_option
import logutil
import time
import pdb
import multiprocessing

THRESHOLD = '0.15'
BLOCKSIZE = 5000
PROCLIMIT = 4
PROGRESSINTERVAL = 25000

def classify(cases, threshold):
'''
Classify loglines.
Parameters:
cases - list of case tuples, containing test case index, log
lines and the suite statistics line
```

```
56 threshold - classification threshold
57
58 returns:
59 labels - a dictionary with encountered labels and the test
60 case indices associated for every lable
61 '''
62   labels = dict()
63   cls = classifier(normalizedDistance)
64   for testcase in cases:
65     index = testcase[0]
66     loglines = testcase[1]
67     csvline = testcase[2]
68
69     for i in range(len(loglines)):
70       label = cls.label(loglines[i], threshold)
71       loglines[i] = label
72       if label in labels:
73         labels[label].append(index)
74       else:
75         labels[label] = [index]
76   return labels
77
78 def csvRead(statlist, index):
79 '''
80 extract a statistics line matching a testcase index from
81 a list
82
83 parameters:
84 statlist - a list with the contents of the statistics file
85 index  -  the index of the testcase
86 '''
87   i = 0
88   for i in range(len(statlist)):
89     if int(statlist[i].split(',')[2]) == index:
90       csvline = statlist.pop(i)
91       return csvline
92   raise Exception("No match in statistics.csv")
93
94 def formatCasecontentsToCsv(casecontent):
95   '''
96   Convert casecontent tuple into a csv line
97
98   parameters:
99   casecontent - tuple containing
100  {testcaseindex, labels, statistics}
101  '''
102  statline = casecontent[2]
103  labels = casecontent[1]
104  if len(labels) == 0:
105    csv = statline + ",{}\n"
106  else:
107    labelLine = ""
108    for label in labels:
109      labelLine += label.strip() + ";"
110    labelLine = ",{" + labelLine[:-1] + "}\n"
111    csv = statline + labelLine
```

```
112
113     return csv
114
115 class classifier:
116 """
117 Object which handles the classification and labels, and
118 stores the labels
119 """
120     def __init__(self, distanceFunction):
121         #array containing the lines already encountered
122         self.labels = []
123         self.distance = distanceFunction
124         self.cache = []
125
126     def label(self, line, threshold):
127 """
128 Label the log line, return the label
129 parameters:
130 line - line to be classified
131 threshold - threshold value for distance function
132 """
133         bestmatch = None
134
135         #check if an exact match can be found
136         for label in self.labels:
137             if label == line:
138                 return line
139
140         #Find the best matching label
141         for label in self.labels:
142             dist = self.distance(label, line)
143             if bestmatch == None:
144                 bestmatch = (label, dist)
145             else:
146                 if dist < bestmatch[1]:
147                     bestmatch = (label, dist)
148
149         #Check if the best match is below threshold
150         if bestmatch != None:
151             if bestmatch[1] < threshold:
152                 label = bestmatch[0]
153                 return label
154
155         #No match found. Create new label
156         self.labels.append(line)
157         return line
158
159 def normalizedDistance(line1, line2):
160 '''
161 Calculate normalized levenshtein distance between two strings
162
163 parameters:
164 line1 - First string
165 line2 - Second string
166 '''
167     distance = levDistance(line1, line2, dict())
```

```
168     totalLength = len(line1) + len(line2)
169     result = float(distance)/float(totalLength)
170     return result
171
172   def levDistance(line1, line2, distances):
173   """
174   Recursively calculate levenshtein distance between two lines
175
176   parameters:
177   line1 - first string
178   line2 - Second string
179   distances - Store matches as they occur, to minimize unnecessary
180   recursion
181
182   """
183     #Test if lines are equal, or if one line is empty
184     if line1 == line2:
185       return 0
186     elif len(line1) == 0 or len(line2) == 0:
187       return max(len(line1), len(line2))
188
189     #Arrange lines
190     if line2 < line1:
191       line1, line2 = line2, line1
192
193     #Check if lines have already been matched
194     if (line1,line2) in distances:
195       return distances[(line1,line2)]
196
197     #Operation costs according to Levenshtein #2
198     inscost = 1
199     delcost = 1
200     substcost = 2
201
202     #Check if last characters of the strings are equal
203     if line1[-1] == line2[-1]:
204       substcost = 0
205
206     #Recursive distance calculation
207     distance = min([
208       levDistance(line1[:-1], line2, distances) + inscost,
209       levDistance(line1, line2[:-1], distances) + delcost,
210       levDistance(line1[:-1],line2[:-1], distances) + substcost
211     ])
212
213     #Store match
214     distances[(line1,line2)] = distance
215     return distance
216
217   def combineLoglists(logarray, statarray):
218   '''
219   combine log and statistics arrays into a list of casecontent tuples
220
221   parameters:
222   logarray - list of log lines
223   statarry - list of suite statistics
```

```
224 caseconts - list of casecontent tuples, one for each test case
225 '''
226   resultarray = []
227   caseContents = None
228   for line in logarray:
229     index = logutil.getTestcaseIndex(line)
230     #index > 0 means that the next test case in the log has been
231     #encountered
232     if index > 0:
233       #Store previous testcase contents
234       if caseContents != None:
235         resultarray.append(caseContents)
236       #Try to get statistics line for the current test case
237       try:
238         caseContents = [
239           index,[], csvRead(statarray, index)]
240       except Exception as detail:
241         print "Exception at " + str(index)
242         print detail
243         exit(-1)
244     if index < 0:
245       if caseContents== None:
246         #First case not yet encountered, skip line
247         pass
248       else:
249         #Store log line as contents of the current case
250         caseContents[1].append(line)
251   #Store last test case
252   resultarray.append(caseContents)
253   return resultarray
254
255 def readBlock(logfile, statfile, logarray, statarray, blocksize):
256 '''
257 Read a block of input data to respective input arrays.
258
259 Parameters:
260 logfile - Log file
261 statfile - Suite statistics file
262 logarray - A list for log data
263 statarray - A list for statistics data
264 blocksize - Number of test cases to be read
265 '''
266   readFinished = False
267   logline = None
268   for i in range(blocksize):
269     statline = statfile.readline().strip()
270     if not statline:
271       #End of file reached
272       readFinished = True
273       break
274     index = int(statline.split(',')[2])
275     statarray.append(statline)
276     if logline == None:
277       logline = logfile.readline()
278     while logutil.getTestcaseIndex(logline) <= index:
279       logarray.append(logline)
```

```
280         last_pos = logfile.tell()
281         logline = logfile.readline().strip()
282         if not logline:
283           #End of file reached
284           readFinished = True
285           break
286     if logline:
287       #Move file pointer back one line
288       logfile.seek(last_pos)
289 
290     return readFinished
291 
292 def classifyProc(inputqueue, resultqueue, threshold, procID):
293 '''
294 process for handling log classification
295 
296 parameters:
297 inputqueue - Queue holding input data
298 resultqueue - Queue holding processed data ready for output
299 threshold - Classifcation threshold
300 procID - Internal process ID
301 '''
302     while True:
303       try:
304         block = inputqueue.get()
305       except EOFError:
306         break
307       blockID = block[0]
308       print "Process " + str(procID) + ": Classifying"
309       resultlog = combineLoglists(block[1], block[2])
310       classify(resultlog, threshold)
311       resultqueue.put((blockID, resultlog))
312       print "Process " + str(procID) + ": Finished"
313     return
314 
315 def parseOptions():
316       global optionParser
317       option_list = [
318         make_option('-l', '--log',
319           action='store',
320           dest='log_file',
321           help='preprocessed log file to analyze'),
322         make_option('-o', '--output',
323           action='store',
324           dest='output_file',
325           help='CSV file containing results'),
326         make_option('-s', '--statistics',
327           action='store',
328           dest='statistics_file',
329           help='file containing suite statistics'),
330         make_option('-t', '--threshold',
331           action='store',
332           dest='threshold',
333           help='threshold value for classifying, 0-1',
334           default=THRESHOLD)
335         ]
```

```
336      usage = "\t%prog [options]"
337      optionParser = OptionParser(
338        option_list = option_list, usage=usage)
339      return optionParser.parse_args()
340
341  if __name__=="__main__":
342    (opts, rest) = parseOptions()
343
344    threshold = float(opts.threshold)
345
346    #Open files for input and output
347    logfile = open(opts.log_file, 'r')
348    statfile = open(opts.statistics_file,'r')
349    outputfile = open(opts.output_file, 'w')
350
351    #Manager for classifier processes
352    manager = multiprocessing.Manager()
353    processpool = []
354    procID = 0
355
356    #The input queue length limit
357    blocklimit = 5 * PROCLIMIT
358
359    #Flag to indicate that the end of input has been reached
360    readFinished = False
361
362    #Read the header line of statistics file
363    statfile.readline()
364
365    logline = None
366    labels = []
367
368    #queues holding input and output data
369    inputqueue = manager.Queue()
370    resultqueue = manager.Queue()
371
372    #Initiate classifier processes
373    while len(processpool) < PROCLIMIT:
374      procID += 1
375      proc = multiprocessing.Process(
376        target=classifyProc,
377        args=(inputqueue, resultqueue, threshold, procID))
378      proc.start()
379      processpool.append(proc)
380
381    #The number of blocks processed
382    lastOutput = 0
383    #The number of blocks put into the input queue
384    lastInput = 0
385
386    #holds processed blocks ready for output, indexed
387    #by their sequence number so they can be printed to
388    #output in order
389    outputblocks = dict()
390
391    #Read data blocks from input files and enter them
```

```
392    #into input queue, from where the classifier processes
393    #get them
394    while readFinished == False:
395      logarray = []
396      statarray = []
397
398      #Input buffer is full, wait until data is ready for output
399      while lastInput - lastOutput >= blocklimit:
400        #If next block has been processed, print to output
401        if (lastOutput + 1 ) in outputblocks:
402          block = outputblocks.pop(lastOutput + 1)
403          #Print block contents to output
404          for line in block:
405            outputfile.write(
406              formatCasecontentsToCsv(line))
407          lastOutput += 1
408        #Wait until next block is processed by the classifier
409        #processes
410        else:
411          block = resultqueue.get()
412          outputblocks[block[0]] = block[1]
413
414      #read next input block, place into buffer for processing
415      readFinished = readBlock(
416        logfile, statfile, logarray, statarray, BLOCKSIZE)
417      lastInput += 1
418      inputqueue.put((lastInput,logarray,statarray))
419
420    #End of input reached, remaining output is handled
421    while lastOutput < lastInput:
422      if lastOutput + 1 in outputblocks:
423        block = outputblocks.pop(lastOutput + 1)
424        for line in block:
425          outputfile.write(formatCasecontentsToCsv(line))
426        lastOutput += 1
427
428      else:
429        block = resultqueue.get()
430        outputblocks[block[0]] = block[1]
431
432    logfile.close()
433    statfile.close()
434    outputfile.close()
```

Appendix 3: A Python script for postprocessing classification results

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  '''
4  @author : Tuomas Haanpää @copyright : 2016 Tuomas Haanpää
5  @license :
6  Copyright All rights
7  3 Clause BSD License
8  ( c ) 2016 Tuomas Haanpää
9  reserved.
10 Redistribution and use in source and binary forms, with or
11 without modification , are permitted provided that the
12 following conditions are met:
13
14 1. Redistributions of source code must retain the above
15 copyright notice, this list of conditions and the following
16 disclaimer.
17 2. Redistributions in binary form must reproduce the above
18 copyright notice, this list of conditions and the following
19 disclaimer in the documentation and/or other materials
20 provided with the distribution.
21 3. The name of the author may not be used to endorse or promote
22 products derived from this software without specific prior
23 written permission.
24 THIS SOFTWARE IS PROVIDED BY THE AUTHOR 'AS IS' AND ANY
25 EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
26 THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
27 PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR
28 BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
29 EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
30 TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
31 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
32 ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
33 OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
34 OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
35 POSSIBILITY OF SUCH DAMAGE.
36 '''
37 from optparse import OptionParser, make_option
38 import logutil
39 import shutil
40 import pdb
41 import os
42 import tempfile
43
44 from optparse import OptionParser, make_option
45
46 optionParser = None
47
48 def postProcess(inputFile, outputFile, mergedLabels):
49 '''
50 Combine two labels. Search for each occurrence of label 2, replace
51 with label 1.
52 parameters:
53 inputFile - input file
54 outputFile - output file
55 mergedLabels - tuple containing the assigned label, and the list of labels
```

```
56  to be replaced
57  '''
58    for line in inputFile:
59      elements = line.split(",")
60      #decode label string to a list of labels
61      caseLabels = logutil.
62      decodeLabelstringToList(elements[14].strip())
63      #search and replace labels
64      for i in range(0, len(caseLabels)):
65        if caseLabels[i] in mergedLabels[1]:
66          caseLabels[i] = mergedLabels[0].strip()
67      #Re-encode the new label list to string
68      elements[14] = logutil.encodeLabelsToString(caseLabels)
69      #Re-encode the CSV line with the new label string
70      csvLine = ""
71      for ele in elements:
72        csvLine += ele + ","
73      outputFile.write(csvLine[:-1])
74
75  def parseOptions():
76      global optionParser
77      option_list = [
78          make_option('-i', '--input',
79                      action='store',
80                      dest='input_file',
81                      help='serialized log file to process'),
82          make_option('-o', '--output',
83                      action='store',
84                      dest='output_file',
85                      help='Post-processed log file')
86          ]
87      usage = "\t%prog [options]"
88      optionParser =
89      OptionParser(option_list = option_list, usage=usage)
90      return optionParser.parse_args()
91
92
93  def postProcessLog(inputFilePath, outputFilePath):
94  '''
95  check for classification errors by having the user
96  manually combine labels, that should be the same but
97  have been classified differently
98
99  parameters:
100 inputFilePath - path to input file
101 outputFilePath - path to output file
102 '''
103   inpf = open(inputFilePath, 'r')
104
105   while True:
106     #Get the labels with their associated case indexes
107     indexed = logutil.getIndexedLabels(inpf)
108
109     keys = dict()
110     selector = 1
111     #Print out selection menu
```

```
112         for key in sorted(indexed):
113           keys[str(selector)] = key
114           print (str(selector) +
115             ") " +
116             key +
117             " (" +
118             str(len(indexed[key])) +
119             " instances)")
120           selector += 1
121
122         label =
123           str(raw_input("Select label (q to quite): ")).strip()
124         if label == "Q" or label == "q":
125           break
126
127         mergedLabels =
128           str(raw_input("Select labels to merge: ")).split(",")
129         parsedLabels = []
130         for mlabel in mergedLabels:
131           #Check for range selection, e.g. 2-4
132           parsedlabel = mlabel.split("-")
133           if len(parsedlabel) == 2:
134             rangeStart= int(parsedlabel[0])
135             rangeEnd = int(parsedlabel[1])
136             if rangeEnd < rangeStart:
137               rangeStart,rangeEnd = rangeEnd, rangeStart
138             for j in range(rangeStart,rangeEnd+1):
139               parsedLabels.append(str(j))
140           #Check for single input
141           if len(parsedlabel) == 1:
142             parsedLabels.append(parsedlabel[0])
143         #Convert from numeric choice to dictionary key
144         for i in range(len(parsedLabels)):
145           parsedLabels[i] = keys[parsedLabels[i]]
146
147         #output to temporary file
148         outpf = tempfile.TemporaryFile()
149         inpf.seek(0)
150         postProcess(inpf, outpf, (keys[str(label)],parsedLabels))
151         inpf.close()
152         #use output of this loop as input for the next one
153         inpf = outpf
154         inpf.seek(0)
155
156       #print the results to output file
157       inpf.seek(0)
158       outpf = open(outputFilePath, 'w')
159       for line in inpf:
160         outpf.write(line)
161
162       inpf.close()
163       outpf.close()
164
165
166
167     if __name__=="__main__":
```

```
168    (opts, rest) = parseOptions()
169    postProcessLog(opts.input_file, opts.output_file)
```