FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Ville Heikkilä

OPTIMIZING CONTINUOUS INTEGRATION TESTING USING DYNAMIC SOURCE CODE ANALYSIS

Master's Thesis
Degree Programme in Computer Science and Engineering
June 2017

Heikkilä V. (2017) Optimizing continuous integration testing by using dynamic source code analysis. University of Oulu, Degree Programme in Computer Science and Engineering. Master's thesis, 49 p.

ABSTRACT

The amount of different tools and methods available to perform software testing is massive. Exhaustive testing of a project can easily take days, weeks or even months. Therefore, it is generally advisable to prioritize and optimize the performed tests.

The optimization method chosen to be studied in this thesis was Dynamic Source Code Analysis (DSCA). In DSCA a piece of target software is monitored during testing to find out what parts of the target code get executed. By finding and storing this information, further code changes can be triggered to execute the stored test cases that caused execution in the modified parts of code.

The test setup for this project consisted of three open-source software targets, three fuzz testing test suites, and the DSCA software. Test runs of different lengths were triggered by code changes of varying size. The durations of these test runs and the sizes of the code changes were stored for further analysis.

The purpose of this thesis was to create a method for the fuzz testing suite to reliably communicate with the DSCA software. This was done to find out how much time can be saved if CI-testing is optimized by scanning every source code change to obtain a targeted test set as opposed to running a comprehensive set of tests after each change.

The data analysis demonstrates with certainty that using DSCA reduces the average run-time of a test by up to 50%.

Keywords: Fuzz Testing, Regression Testing

Heikkilä V. (2017) Jatkuvan integraation järjestelmien testauksen optimointi dynaamisen lähdekoodianalyysin avulla. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 49 s.

TIIVISTELMÄ

Ohjelmistotestauksessa käytettävien työkalujen ja metodien määrä on massiivinen. Ohjelmistoprojektin läpikotainen testaus saattaa kestää päiviä, viikkoja tai jopa kuukausia. Tämän takia on yleisesti suositeltavaa priorisoida ja optimoida suoritetut testit.

Tässä opinnäytetyössä tarkasteltavaksi optimointimetodiksi valittiin dynaaminen lähdekoodianalyysi (DSCA), jossa ohjelmistoa monitoroidaan ajonaikaisesti, jotta saadaan selville mitä osia lähdekoodista mikäkin testi suorittaa.

Tämä projekti koostui kolmesta avoimen lähdekoodin ohjelmistoprojektista, kolmesta fuzz-testaustyökalusta sekä DSCA-ohjelmistosta. Erikokoisilla lähdekoodin muutoksilla saatiin aikaan erikokoisia testimääriä uudelleenajettaviksi. Näiden ajojen suuruudet ja kestot tallennetiin, ja niitä vertailtiin.

Tämän opinnäytetyön tarkoituksena oli löytää keino saada fuzz-testaustyökalu keskustelemaan DSCA-ohjelmiston kanssa luotettavasti, sekä selvittää kuinka paljon aikaa pystytään säästämään optimoimalla CI-testausta skannaamalla jokainen lähdekoodimuutos kohdennettujen testien saamiseksi verrattuna siihen että jokainen lähdekoodimuutos aiheuttaisi kokonaisvaltaisen testiajon.

DSCA-ohjelmistoja käyttämällä saatiin varmuus siitä, että CI-järjestelmien testiajojen pituutta pystytään pienentämään huomattavasti. Keskimääräisen testiajon pituus pystyttiin testeissä jopa puolittamaan.

Avainsanat: Fuzz-testaus, Regressiotestaus

TABLE OF CONTENTS

FOREWORD

The research performed in this thesis was conducted while working for Synopsys Finland Oy. The idea for the topic was first introduced to me by Lauri Piikivi, and later refined to its final state by Rauli Kaksonen, Marko Laakso, Antti Kiiveri, and Jouni Knuutinen.

I would like to give my heartfelt thanks to everyone who supported me during my studies, including my friends, relatives and coworkers. I would especially like to thank Margit Vakkuri for all the support I still believe I did not deserve. I would also like to give my honest thanks to Christian Wieser from the University of Oulu for kicking my posterior every week to achieve important milestones in my thesis.

Finally, I would also like to thank Mr. Juha Röning for his guidance and important feedback, and Essi Ranta for making sure that what I wrote makes sense.

Oulu, Finland December 3, 2017

Ville Heikkilä

ABBREVIATIONS

ACK	Acknowledgment
AFL	American Fuzzy Lop
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CI	Continuous Integration
CSV	Comma Separated Values
CVS	Concurrent Versions System
DSCA	Dynamic Source Code Analysis
DTP	Data Transfer Process
EBCDIC	Extended Binary Coded Decimal Interchange Code
EOL	End-of-Line
FTP	File Transfer Protocol
FTPS	FTP Secure
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
ID	Identifier
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IETF	Internet Engineering Task Force
IM	Instant Messaging
JID	Jabber Identifier
MUC	Multi-User Chat
OPC	Object Linking and Embedding for Process Control
OSS	Open Source Software
OUSPG	Oulu University Secure Programming Group
POSIX	Portable Operating System Interface
RFC	Request For Comments
RTS	Regression Test Selection
SASL	Simple Authentication and Security Layer
SCA	Source Code Analysis
SCM	Source Code Management
SCRAM	Salted Challenge Response Authentication Mechanism
SDK	Software Development Kit
SDLC	Systems Development Life Cycle
SDM	Software Development Methodology
SDN	Software-Defined Networking
SFTP	Simple File Transfer Protocol
SFTP	SSH File Transfer Protocol
SHA1	US Secure Hash Algorithm 1
SSCA	Static Source Code Analysis
SSL	Secure Sockets Layer
SUT	System Under Testing
SVN	Apache Subversion
TCP	Transmission Control Protocol
TDD	Test Driven Development

TLS	Transport Layer Security
UA	Unified Architecture
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
WSS	Web Services Security
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XP	Extreme Programming
XSF	XMPP Standards Foundation

1. INTRODUCTION

Continuous Integration (CI) is a popular software development process based on the concept of developers regularly sharing their code changes into a shared repository. The changes should be shared at least daily, preferably multiple times per day.

CI systems necessitate the implementation of optimized testing solutions that do not take hours or days to complete as is often the case with fuzz testing. Ideally, synchronizing code changes would trigger a subset of tests aimed to test only the changed parts of the code. Some researches have already begun to implement regression testing in CI systems [1].

Fuzz testing, or fuzzing, is a type of robustness testing in which a piece of target software is bombarded with a plethora of invalid and unexpected inputs. The purpose of this is to find undesired behavior and flaws in the tested systems.

Source Code Analysis (SCA) is a software testing method that focuses on examining the actual code of the tested system. Static Source Code Analysis (SSCA) focuses on examining the code for defects without executing it, whereas Dynamic Source Code Analysis (DSCA) monitors an executed piece of software.

The effectiveness of DSCA relies heavily on a sufficient set of inputs to perform on the monitored software. If the test set is too small, the achieved code coverage will decrease and the reliability of the test results will suffer.

Automating fuzzing-based regression testing with DSCA will require some investments on behalf of the implementing party. As fuzzing tools can easily run tens or even hundreds of test cases per second, monitoring them can prove problematic, especially when each test case needs to be documented well enough so that they can be reliably identified and re-executed.

By using DSCA software to optimize fuzz testing in a CI environment, the purpose of this thesis is to answer the following questions:

1. How many commits in a software project contain changes that are not covered by testing and therefore do not require a test run?

2. How much smaller is the average triggered test run compared to the base set?

2. STATE OF THE ART

2.1. Software Development Methodologies

Ever since the 1960s, a number of Software Development Methodologies (SDM) have been produced to fit various different software development environments and needs. These SDM's started off with the still-in-use waterfall model [2], where each stage of the Systems Development Life Cycle (SDLC) had to be finished and approved before moving on to the next stage of development (Figure 1)[3].

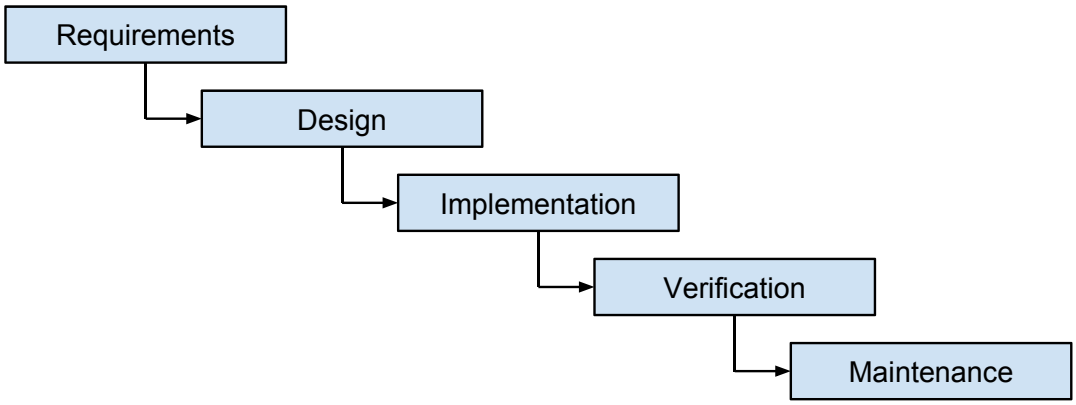

Figure 1. Waterfall model graph

As the waterfall model is quite rigid in adapting to changing customer needs, other SDM's were needed. New conventions rose through modified versions of the waterfall model like Sashimi ("Waterfall with Overlapping Phases") [4]. These varied from structured programming in the 1970s [5] to Extreme Programming (XP) and Scrum in the 1990s [6, 7], finally leading up to the Agile Manifesto in the 2000s [8].

This thesis focuses on software projects that utilize Continuous Integration (CI), an agile software development method coined in the 1990s by Grady Booch [9] and later redefined and advocated by XP [6].

2.1.1. Continuous Integration

The core principle behind CI is merging code to a shared repository multiple times per day to ensure that various components of the software work together [10]. In practice, this usually means using a shared code repository, and automated building and testing environment (Figure 2), optionally accompanied by additional manual testing [11].

For CI to work reliably, developers should commit their code changes daily, and each commit to the trunk (the base of the project) should trigger a new build of the software accompanied by the related automated tests [10]. The benefits of this include saving time due to a decreased amount of human involvement, quick identification of problems, and the ease of obtaining up-to-date deliverables [11].

If a clash occurs between two developers, the developer pushing their code last should get a warning. If no warning is issued, the integration should fail. Failed builds can and will happen, but should be resolved as quickly as possible. Following these guidelines will result in each stable build of the software working and the time spent on fixing bugs being minimized as they will show up much earlier in the development [12].

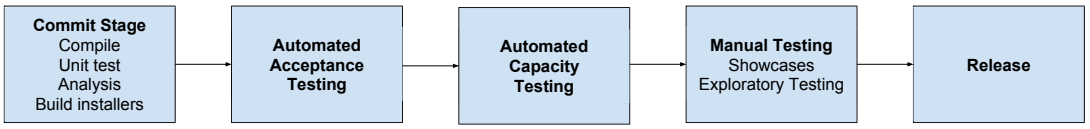

Figure 2. Continuous deployment pipeline

The second important step in the CI pipeline is automation. Building a running system should require a minimal amount of user input. This is usually achieved via automated build environments like make [13] for Unix, Apache Ant [14] for Java and Nant [15] or Microsoft Build Tools (MSBuild) [16] for .NET. Additionally, a good build tool analyzes what needs to be changed and builds only that. It is also important to note that building on a schedule, such as every night, is not CI. Scheduled builds make it no longer possible to find bugs immediately after merging code [12].

As a result of the aforementioned automation, everything related to building the system should be included in the code repository. One practical example is that a new user on a "virgin" machine should be able to checkout and fully build the running system with a single command [12].

In addition to building, the testing should also be automated. XP and Test Driven Development (TDD) [17] favored the method of writing tests before writing the actual code, which is not necessary in CI [12]. For CI purposes, only a suite of automated tests with a large enough code coverage is needed. The tests need to have the same ease of launch as the automated build has, and the results should simply indicate whether any of the tests failed. If any test should fail, the build should also halt [12].

The testing environment that is used should ideally be an exact copy of the production environment, using the same operating system, database software, version numbers, and the same libraries, even if they are not used. Testing in a different environment may make the development team blind to bugs that will only appear in the production environment [12].

2.2. Source Code Management

Using a single shared source code repository is an important aspect in CI. Without one, keeping track of a large software project and allowing multiple teams to access it is borderline impossible [10, 11, 12].

The amount of tools provided for Source Code Management (SCM) is plentiful, including open source software such as Concurrent Versions System (CVS) [18], Apache Subversion (SVN) [19] and Git [20], along with a large amount of proprietary solutions.

2.2.1. Git

In this thesis, the software projects utilize Git as their source code management system. This was a conscious choice, as Git is a popular choice for open source software, and it offers the checkout command [21] that is crucial in the research performed in this thesis.

Git checkout can be used to update files in a software project to an earlier state. Furthermore, it can also be used to replicate the development process. A user can first check out to an early stage of development, then start to check out individual code commits one at a time (Figure 3). This negates the need to do the research required for this thesis in "real-time" (at the time of development).

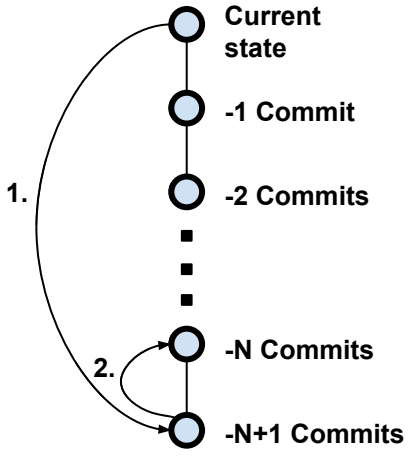

Figure 3. Using the git checkout command to reproduce software development.

SCM tools make it easier to create multiple branches to split the development process, for example experimenting and fixing bugs from previously released features. However, CI systems should keep their use to a minimum [11, 12]. Creating entirely new features in branches goes against the very term "Continuous Integration", as integration in branch situations would only happen when the branch is merged back into the mainline.

2.3. Software Testing

Having a mindset that the point of software testing is to confirm that there are no bugs in the software is dangerous and may lead to a subconscious psychological steering towards that goal. Therefore, it is more important to think that "Testing is the process of executing a program with the intent of finding errors" [22].

Even though modern software development tools and methods offer programmers great ways of finding bugs early in the development, the bugs still tend to find their way into the software. Usually, the bugs develop early on in the specification phase or later on in the design phase, but relatively rarely in the coding phase [23].

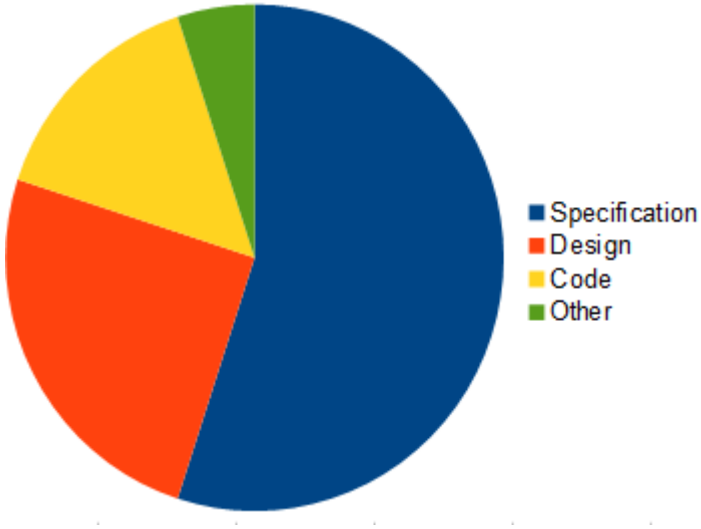

Figure 4. Distribution of the causes of bugs.

It is extremely important to find these bugs as early as possible in the development cycle, because the cost of fixing bugs gets higher and higher the later they are found [23]. As seen in Figure 5, the cost of fixing a bug increases logarithmically between the phases of development. Some estimates say that the total cost of software bugs in the US is almost $60 billion [24]. The software bugs do not only cause financial expenses but can also cost human lives [25], or even cause a large blackout [26].

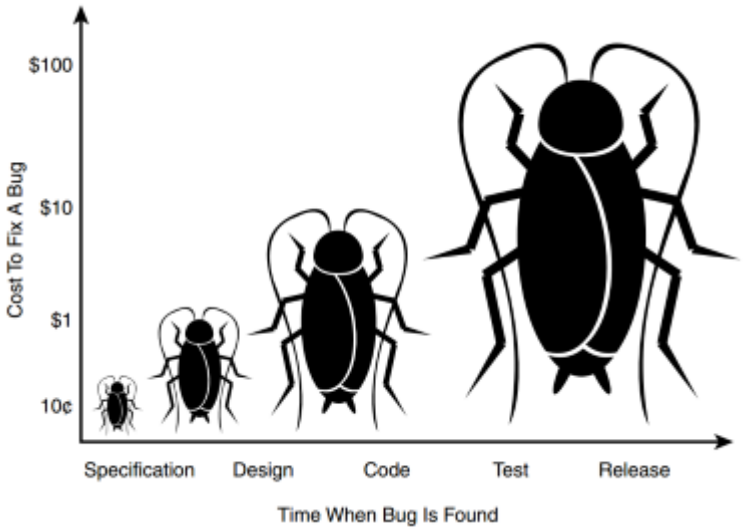

Figure 5. The cost of fixing a bug in different phases of software development.

2.3.1. Static vs. Dynamic

Software testing can be split into dynamic and static testing. Static testing does not involve executing the tested code, but it focuses on reading through the documentation to verify that it is sufficient and comprehensible, and checking the actual code [27]. This can involve scanning the code for a bad behavior, such as not releasing memory, or just finding parts of the code that never get executed.

Dynamic software testing involves actually executing the code and checking it for a functional behavior of the tested software. This can take the form of a simple positive testing to verify that the software works when it uses the parameters that were intended to be used. That is, to make sure that the software can perform the tasks it was meant for. Negative software testing is a more involved method of dynamic software testing, where the software is actually tested for error handling. This can be done by prodding the system with an input that the system is not supposed to encounter in valid usage, and thus verifying that the system continues to respond comprehensibly and does not break.

2.3.2. White-Box vs. Black-Box

Dynamic software testing can further be divided into black-box testing and white-box (also known as clear-box testing). The black-box testing tests the functionality of the software without knowing about it's inner workings, where as in white-box testing the tester has access to the target source code and can use that to tailor the performed tests to target specific parts of it [23]. Gray-box testing, as the name suggests, is a mix of black-box and white-box testing (Figure 6). The grey-box testing is performed in the same way as the black-box, but the tester can also glance at the software to see what makes it work [23].

Figure 6. The differences between the three different types of box testing.

2.3.3. Fuzz Testing

Fuzz-testing, or fuzzing, is a widely used form of negative black-box testing where the input of a software program is tested for exceptions, such as crashes or memory leaks, by feeding it pseudo-randomly generated anomalous data [28].

The word "fuzz" was first coined by Barton Miller in 1988 as a term to describe random, unstructured data in a university course project topic [29]. This project resulted in the creation of "fuzz", a simple program that generates a continuous string of random characters [30]. This wasn't necessarily the first fuzzing program in existence. One earlier example of fuzzers or fuzzer-like programs is "The Monkey", a small desktop accessory written by Steve Capps for Macintosh [29, 31]. It was used to simulate "–an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon." [31] to test for bugs in MacPaint and MacWrite.

From these humble beginnings, the concept of fuzzing has since resulted in the creation of a large variety of different fuzzing tools, both non-commercial like american fuzzy lop (AFL) [32] or radamsa [33], and commercial like Peach [34] or Defensics [35].

While fuzzing can be a simple and inexpensive tool in the software testing toolbox, it should not be used to replace the more traditional software testing methodologies. Rather, it should be used to complement these more sophisticated tools. Though a simple method, fuzzing can still be used to enhance testing, as it can often find bugs and development oversights that humans might never even think to test [30].

2.4. Regression Testing

Regression testing is a form of software testing that is performed after a piece of the developed software has been changed. Regression testing aims to verify that the performed changes have not introduced any new faults into the software or recreated a previously fixed bug. Therefore, it is generally considered a good coding practice to develop test cases to reproduce bugs after they have been fixed. By doing this, the developer or tester can always re-run the test case after code changes to see if the bug has re-emerged [22].

In practice, regression testing requires some type of validation testing to be performed on the software's new features. As long as these validation tests are labeled properly, the tester should be able to decipher which functionalities each test case corresponds to. This is usually done automatically after each compile, nightly or weekly, as the amount of tests performed can be unfeasible to go through by hand.

The various regression testing techniques can be coarsely divided into three different categories: Retest all, regression test selection, and test case prioritization, along with hybrid techniques that blend various aspects of these three [36].

Retest all is a slightly obsolete, resource intensive technique that simply consists of repeating every test case and retesting every feature [37]. As running each and every test case again is expensive, sometimes prohibitively so, more sophisticated methods are needed.

2.4.1. Regression Test Selection

Regression Test Selection (RTS) is a natural progression of retest all, aimed to combat its excessive costs [38]. RTS selects only a part of the performed tests to run, if the cost of selecting the tests is less than the cost of retest all. In RTS, test cases are divided into reusable, retestable and obsolete cases. To cover areas that the existing cases do not cover, additional cases can be created. [36]. The techniques for this are coarsely split into coverage, minimization, and safe techniques. Coverage techniques, as the name implies, find parts of the program that have been modified and are covered by tests, and rerun those. Minimization techniques are similar, but they only run a minimal set of test cases. Safe techniques focus on running test cases that produce different output based on the program version [39].

2.4.2. Test Case Prioritization

One of the key problems with regression testing has always been test case prioritization. The initial validation testing for new features can take hours, days or even weeks. Trying to run every test again after every change in the features would be extremely time consuming to the point of absurdity. Therefore, some kind of test case prioritization, selection, optimization, or minimization is needed.

Test case prioritization aims to further optimize RTS by ordering the test cases to be performed by a predetermined metric, like their code coverage or their estimated ability to find faults [40]. This is done to increase the fault detection rate. After the test cases have been ordered via test case prioritization, testing can be stopped when no faults appeared in a set amount of time [41].

2.5. Source Code Analysis

Source Code Analysis (SCA) is the act of analyzing either the source code (Static Source Code Analysis) or the compiled version (Dynamic Source Code Analysis) of a program in order to find faults with it. This is a largely automated procedure that can, in the case of Static Source Code Analysis (SSCA), be integrated to the developers' Integrated Development Environment (IDE) to give feedback during the development phase of a program.

These tools, and the rapidly developing computational power available, offer software engineers increasingly well optimized and automated methods to validate the software they are testing. However, this does not mean that there is no work to be done. SCA tools merely allow the software engineers to cover more ground with the same effort as they could without using them. Both SSCA and DSCA still require configuration and maintenance, along with the generation of test cases.

2.5.1. Static Source Code Analysis

SSCA aims to find faults in the code without executing it. This can include scanning for known bad behavior such as unterminated loops, memory leakage or referencing non-existing variables. This can theoretically be performed via manual code auditing, but due to time constraints, automatic evaluation programs are often used instead [42].

One way of visualizing the behavior of an SSCA tool is a tree graph. The SSCA tool simulates all possible program executions to generate the different states of the program, along with their inputs and outputs that lead to other states (Figure 7). However, this approach will not work if the program contains statements outside its scope of reasoning, like the output of a function it does not have access to [43].

SSCA tools have their downsides as well. The tools usually scan for a predefined set of patterns or rules in the code, so any problem not defined beforehand will not be found. SSCA tools also tend to have a sizeable false positive rate which needs to be sifted through manually [42].

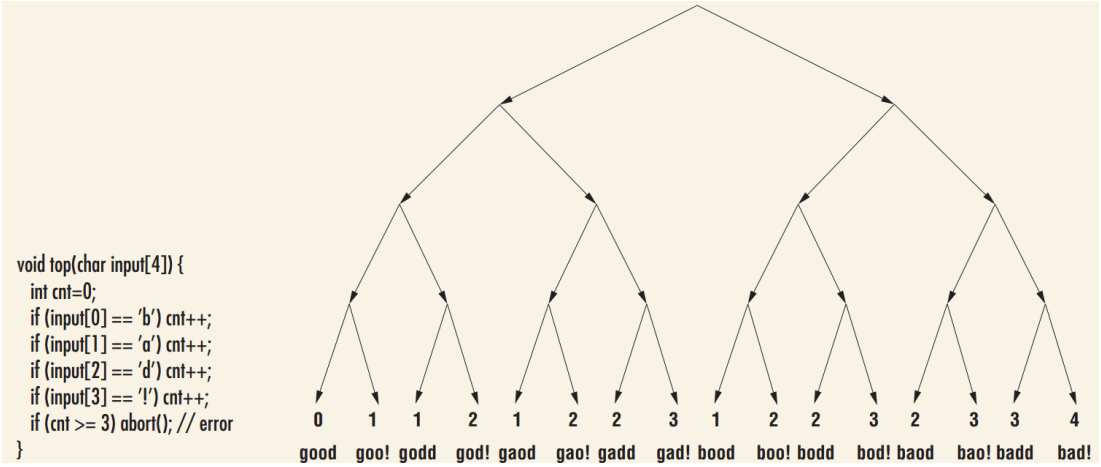

Figure 7. A short example program and the tree it generates.

2.5.2. Dynamic Source Code Analysis

Dynamic Source Code Analysis (DSCA) tools test the software by running it and prodding it with interesting inputs. This is done so that the software can be monitored during the processing of unwanted or invalid data to reveal problems in the code.

Because DSCA tools require running the tested code, they also require some added manual configuration for each piece of software they are used to test [42]. This, coupled with the fact that the program executions triggered may not apply to other program executions, makes DSCA time consuming [44], especially when multiple versions of the software are tested in parallel.

3. TESTING SETUP

3.1. Components

The testing setup for this thesis consists of a Virtual Machine (VM) running in a laptop and a centralized analysis server (Cockpit) (Figure 8). The VM is running the fuzz testing suite, the System Under Testing (SUT), a "Rerunner" script and the DSCA software that consists of three parts: a Scanner, a Controller and an Agent.

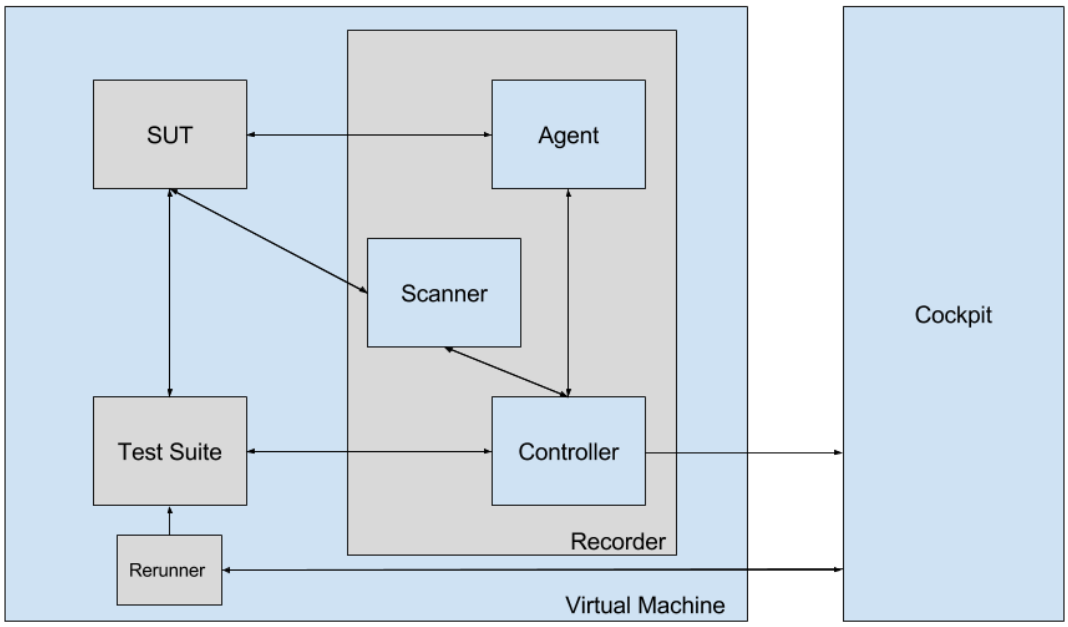

Figure 8. Testing setup architecture

3.1.1. DSCA Software

The Controller is the local main User Interface (UI) of the Recorder as it is in charge of the functionality of all other components. This includes receiving and forwarding the software blueprints from the scanner and the test case footprints from the Agent to the Cockpit. The controller also provides an HTTP Application Programming Interface (API) that provides a way to manage test case execution.

The Scanner is the part of the DSCA software that scans the target software to generate a blueprint of it. The blueprint is static and contains the checksums of all files the target software uses at runtime. After the scan is done, the blueprint is uploaded to the controller where it can then be uploaded to the Cockpit. These blueprints are then used to identify changes between different versions of the target software.

The Agent is a lightweight piece of monitoring software, which – in the case of running Java software – is started by supplying the tested application with a running option at startup. This will cause the Agent to start to run in the background. This attaches it to the target software, as it starts to record the footprints of the test case execution that are triggered by the Controller that signals the beginnings and ends of the test cases. When testing a Java software, the test case execution is recorded on the instruction level as Java software has built-in debug details by default. If the debug details of the Java software are intentionally disabled, the recording is done on the method level. After each test case the Agent sends the recorded test case footprint to the Controller so that the footprints can be uploaded into the Cockpit for storage and analysis.

3.1.2. Cockpit

The Cockpit is a centralized server that does the heavy lifting in the DSCA process. This means storing and processing all the software blueprints to find differences between versions, and filtering the list of all the test case footprints to find the cases that tested parts of the code that were changed between different versions of the software. For usage, the Cockpit provides an HTTP UI along with an HTTP API through which this data can be obtained.

In the case of the experiments performed in this thesis, the list of test cases to run after performing changes to the target software was requested by the Rerunner from the Cockpit by sending an HTTP GET request via cURL. The list of test cases to run was returned in a Comma Separated Values (CSV) along with additional information to be parsed by the Rerunner.

3.1.3. Test Suite

Without the addition of DSCA, the test suite would simply send anomalized messages to the SUT and possibly wait for some type of reply messages. When the DSCA software is added to this setup, some additional configuration becomes necessary.

Because the aim of this setup is to optimize the amount of test cases needed to run between different versions of the software, the test cases need to be labeled for identification. This stands for both human-readability and machine-readability. It is useful to an engineer so they can identify test cases on a glance, but mandatory for a machine to be able to process them.

The used fuzzing platform for the test suites supports external instrumentation of targets. In practice, this means running user-defined commands before or after each test case or each test run. In this setup the Controller needs to be informed when a test case is starting or ending. The external instrumentation also supports a variety of environment variables generated by the test suite that can be used in external instrumentation. These variables include data, such as test case indices, test case verdicts, and test suite names and version numbers.

None of the pre-existing variables were deemed suitable for the needs described above. The test case indices were at first considered as an option, but they posed some problems with reliability as they can and will change based on the configuration used and the version of the test suite. Due to these reliability problems, two new external instrumentation environment variables were developed specifically for the purposes of this thesis.

Both of the new variables were information that already existed in the test suite, so getting them to function as environment variables was not a particularly difficult task. The first new variable was the test group, chosen to make the test case information more human-readable. As shown in Figure 9, the test group tells the end user which sequence of messages is being tested, the part of the message that is being anomalized, and a rough idea of what the anomaly is like.

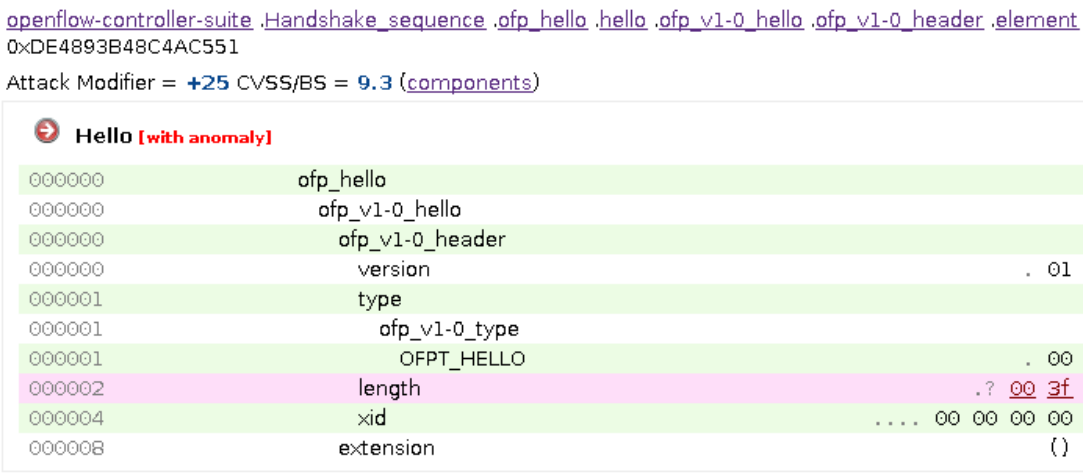

Figure 9. Test case information in the test suite

The second environment variable was picked to make the test case information machine-readable, and make the actual automation of running test cases possible. This was achieved by using the test case hashes, 16-character hexadecimal codes generated by the test suite to internally identify each individual test case. Both of these new variables proved to be a useful addition to the existing variables and were introduced to the released versions of the test suites to be used by end customers. The combination of the hash code and test group can be seen in Figure 10.

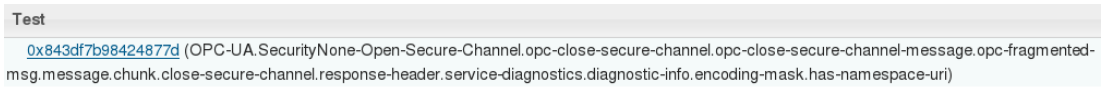

Figure 10. Test case hash and test group as seen in the Controller

By using these newly added environment variables, the test suite was able to use the external instrumentation feature to communicate with the Controller. The communication was done by utilizing the Controller HTTP API as seen in Figure 12 to signal when a test case was about to start and when it ended. This was performed by first sending an HTTP GET request using cURL[45] with the test case group and hash code to the Controller that then sends it to the Agent. The Agent then replies with an HTTP

cookie that is sent through the Controller to the test suite. This cookie is then stored as it is needed to inform the Controller that a test case has ended.

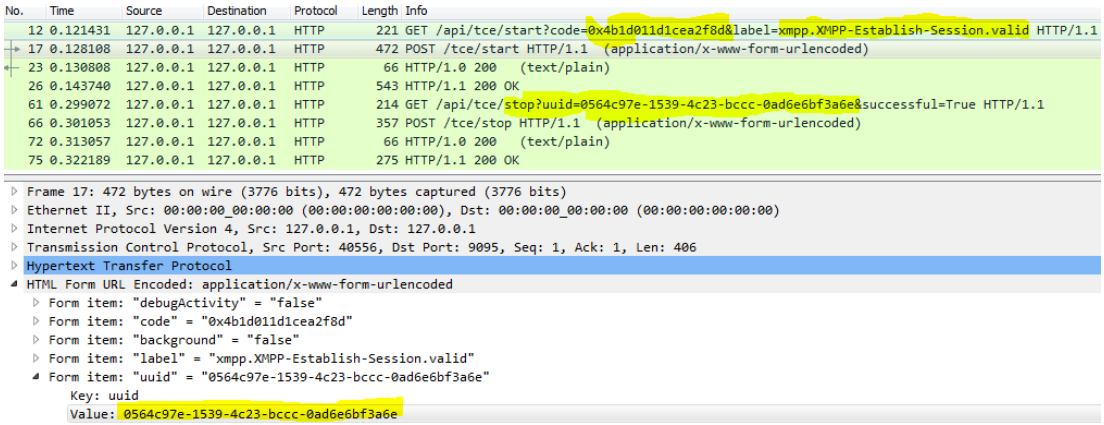

Figure 11. Traffic capture showing the start and stop messages between the test suite, Agent and Controller

Figure 11 shows a traffic capture with message No. 12 signaling the Controller that a test case is about to start with the test case hash as an identification code and the test group as a label. This information is then forwarded to the Agent in message No. 17 which replies with a cookie or Universally Unique Identifier (UUID) in message No. 23. The Controller then forwards that information to the test suite in message No. 26.

Stopping the test case is initiated in message No. 61 by using the UUID received previously from the Agent. The Controller relays the stopping message to the Agent in message No. 66, gets the reply in message No. 72 and finally notifies the test suite about the successful stopping of test case recording in message No. 75.

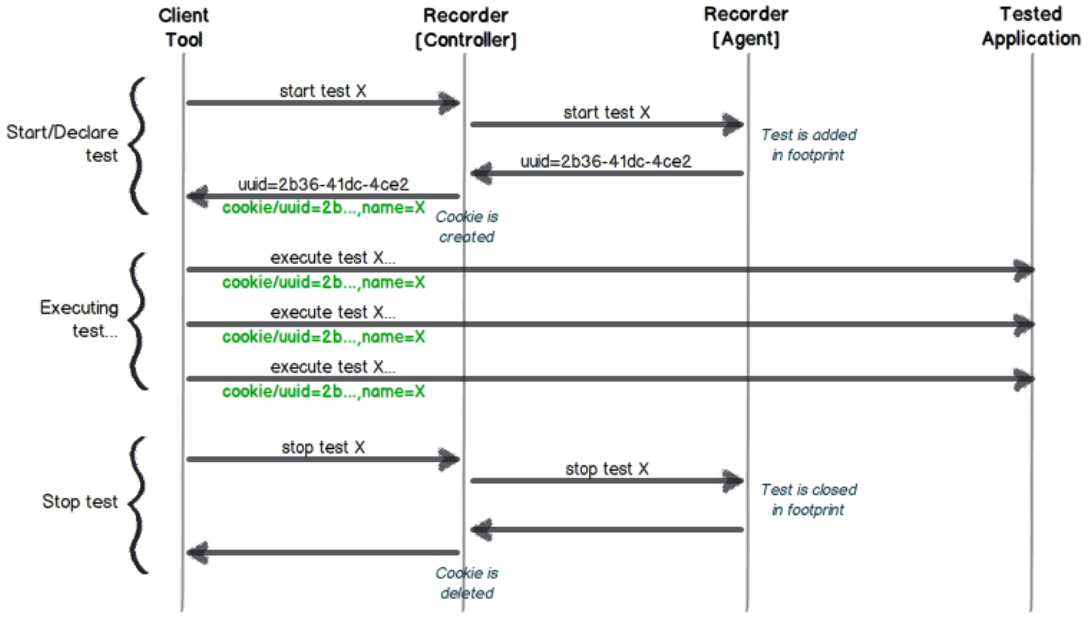

Figure 12. Communication between the Test Suite, Recorder, Agent and the SUT

3.1.4. Rerunner

The Rerunner is not a separate entity per se, but an executable script comprises multiple components (Figure 13) located in the VM running the test suite, SUT and DSCA software. The DSCA software used in these case studies is normally used for a significantly smaller amount of test cases that are managed either by hand or via a handful of plug-ins for certain specific testing tools. If the tested software was a website and the test cases were manually recorded customer behaviors, running them again by hand would be simple. In the case of thousands of fuzz testing cases, the process gets more involved.

When a new version of the target software has been scanned and the scan has been analyzed, the Cockpit will reply with an email describing the results. If the new version has caused the need to rerun test cases, this will be the trigger for it. After receiving the email, the Rerunner will send an HTTP GET request to the Cockpit to request a list of affected test cases. If there are tests to rerun, the Cockpit will reply with a CSV file containing the code (hash) and label (test group) along with possible auxiliary data for each test case that needs to be performed again.

A problem with the CSV file is that it needs to be trimmed down to useable data that the test suite can use. In our case, this would be the hash code. The Rerunner removes all the excess data from the CSV file by first removing every other column except for the hash code. After this, the title row is removed and each line break is replaced with a comma. This gives us a list of comma-separated hash codes that the test suite can use.

The test suite can be used from the command prompt by supplying the used settings as a file, and the list of test cases to run can be appended to that. This should make running the test cases in the comma-separated list easy, except for one technical limitation. Various Portable Operating System Interfaces (POSIX) limit the length of command line inputs. This means that a list of thousands of 18 character hexadecimal codes will simply refuse to run.

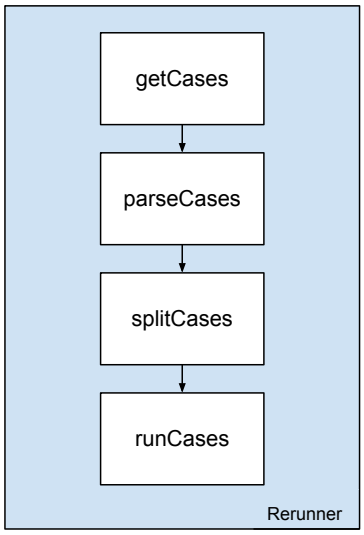

Figure 13. Different components of the rerunner script

To combat this length limit, the Rerunner was equipped with a splitter. The splitter splits the list of comma-separated hash codes into sets of 1000 test cases, and performs a separate test run for each of them. This can be seen in some of the test case results, where running a subset of the original set of test cases can take longer than the original set because the test suite needs to be started separately for each set of 1000 test cases.

After all the test cases have been performed again, the results will be uploaded from the Controller to the Cockpit to return the code coverage of the project back to where it was before the changes that required retesting were introduced. This means that process of upgrading the software and scanning the blueprint for changes can continue until more cases need to be performed again.

4. CASE STUDIES

The data in the case studies was gathered by first returning the software project back in time with git checkout and performing a base set of test cases. The base set was generated by the fuzzing tool, and it consisted of test cases configured to test as many features as possible and to place anomalies evenly around the tested messages.

The DSCA software posed some technical limitations to this test case generation. Due to the architecture of the Cockpit, only 32 thousand unique test cases could be stored for each test subject. This in itself was not a troubling limit. More demanding was the limit of the local controller that could only hold ten thousand cases. In some of the case studies the running time was the most demanding limitation, as keeping the running time sensible while still running enough test cases to generate interesting data was a difficult balance to achieve.

The test case reduction necessitated by these limits was achieved by reducing the amount of anomalies in each test run. The fuzz testing software that was used contained a built-in method for configuring the amount of anomalies (Figure 14) such as overflows, underflows and binary anomalies.

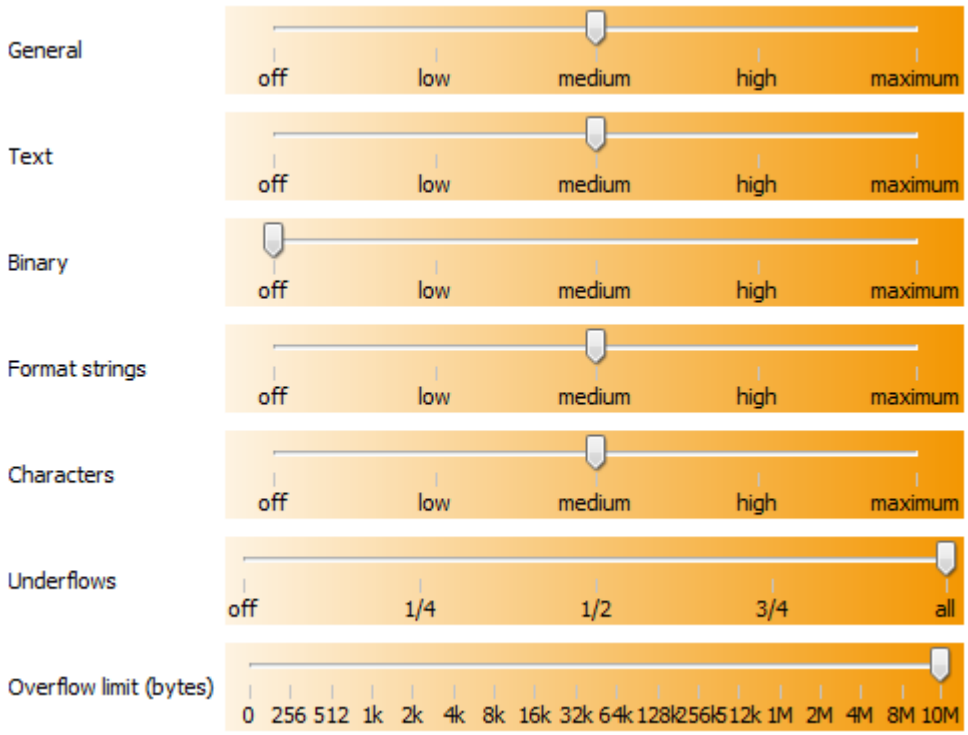

Figure 14. The fuzzing tool anomaly adjustments

The DSCA Recorder recorded the footprints of these test cases to see which Java classes got called during their execution. These footprint files were then uploaded to the Cockpit to be stored in a database, and to calculate a code coverage percentage for the performed tests.

After performing the base set of tests on the checked out version of the target software, the project was moved forward in time by one commit. This was done by first checking out the master of the branch, and then rewinding back one commit less than before. The checked out commit was then scanned for changes by the DSCA scanner and the scan results were uploaded into the Cockpit.

If the Cockpit noticed that there were stored test cases that caused calls in the commit-modified code, it produced a list of affected test cases that could then be obtained via the HTTP API. These cases were then executed again to maintain the same code coverage. If no affected test cases were found, the project was moved forward again and the process was repeated.

4.1. Case study 1: FTP Server

4.1.1. Protocol & Software

The setup for case study 1 consisted of a File Transfer Protocol (FTP) fuzzing test suite running against Apache FtpServer[46]. FTP is a network protocol based on a client-server model architecture that is used in transferring files between a client and a server. Authentication in FTP can be performed either in plaintext or anonymously. For more secure operation, FTP can also be secured with Secure Sockets Layer (SSL) or its successor Transport Layer Security (TLS), resulting in FTP Secure (FTPS). This is not to be confused with SSH File Transfer Protocol (SFTP), which in turn is not to be confused with Simple File Transfer Protocol (also SFTP).

The commands tested by the fuzzing test suite are shown in Table 11. The XMKD, XPWD and XRMD commands listed are "experimental"[47] aliases to the MKD, PWD and RMD commands. Some FTP implementations require that all commands are four characters long, so the extended commands were introduced. However, all implementations should recognize both versions of these commands [48].

The transmission modes mentioned in the explanation for the MODE command are stream mode, block mode and compressed mode. Stream mode transmits the data as a stream of bytes without restrictions on representation types. Block mode transmits the data by splitting the data into blocks with one or more header bytes. Compressed mode is used to compress a string of replicated data bytes [49].

The aforementioned representation types are the four data types defined to be used with FTP, American Standard Code for Information Interchange (ASCII), Extended Binary Coded Decimal Interchange Code (EBCDIC), Image and Local. In ASCII, files are represented as ASCII text files with End-of-Line (EOL) markers. EBCDIC is the same, but with International Business Machines Corporation's (IBM) EBCDIC character set. Image representation simply defines data as blocks without any internal structure, all data is transferred one byte at a time without processing. The local type is used to handle data with logical bytes that are not eight bits. This allows the data to be stored in the destination system in an identical way compared to its local representation [49].

Table 1. A list of messages tested in case study 1 in alphabetical order

Command	Explanation
APPE	Append data into a file on the server
CDUP	Change current working directory to parent directory
CWD	Change current working directory to another directory
EPSV	Allows using IPv6 addresses in active mode data transfer connections
FEAT	Request supported features from the server
HELP	Request information about a specified FTP command from the server
LANG	Set the language used by the server
LIST	List files in the current working directory
MKD	Create a directory in the current working directory
MLSD	List files and attributes of the specified directory
MODE	Set the transmission mode used in data transfer
NLST	List files and attributes in an unadorned format
NOOP	Do nothing. The server will respond if it is still running
PASV	Enter passive mode
PWD	Print the path of the current working directory
REIN	Reinitialize all user specific information used during a connection
REST	Mark data to know where to continue if disconnected
RMD	Remove a directory
SITE	Request a list of system specific commands from the server
STAT	Print the status of the connection
STRU	Set the data structure used during data transfer
SYST	Request information about the operating system of the server
XMKD	MKD command padded to 4 characters
XPWD	PWD command padded to 4 characters
XRMD	RMD command padded to 4 characters

The data structures mentioned in the explanation for the STRU command are file-structure, record-structure and page-structure. File-structure imposes no internal structure for the files, and they are considered as continuous sequences of data bytes. Record-structure defines files as consisting of sequential records, whereas page-structure defines files as consisting of independent indexed pages [49].

The active and passive modes in the explanations for the PASV and EPSV commands are the two possible Data Transfer Process (DTP) modes in FTP that establish and manage the data connection. In active mode, the client initiates the connection by sending a message from its command port to the server's command port, followed by the server responding from its command port to the client's command port and sending data from its data port to the client's data port. The client then responds to this by sending an acknowledgment (ACK) from its data port to the server's data port (Figure 15). In passive mode, the command port exchange stays the same, but the data transfer is initiated by a message that the client sends to a random port (specified by the server) and the server responding to the message by first sending an ACK and then the data [50].

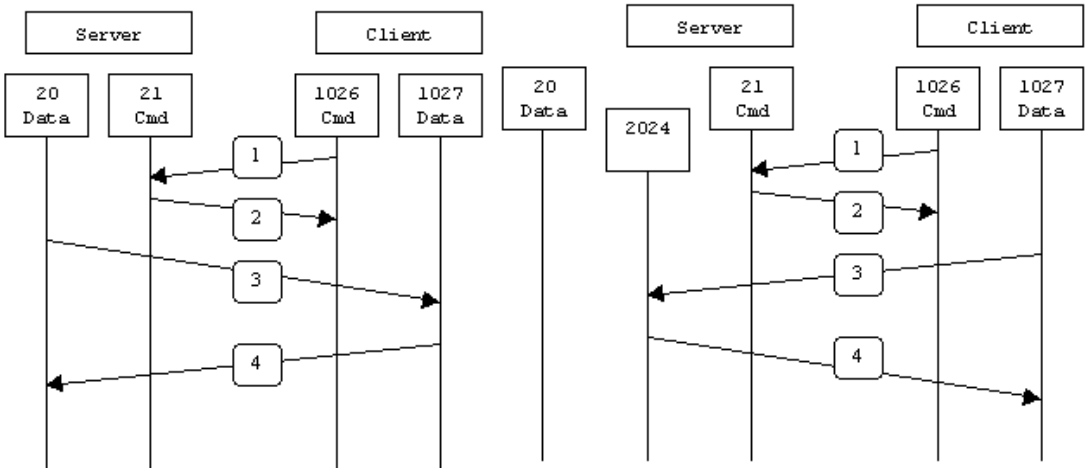

Figure 15. The difference between the active (left) and passive (right) modes in FTP

All of the tested messages are sent in the middle of a message flow. Each test case is preceded by a sequence of messages that sets up the connection for the exchange of messages (Figure 16) and followed by a sequence that closes the connection and makes sure that no connections are left hanging open in the server side (Figure 17).

Case study 1 was performed by running a preformed test set of 1065 test cases that took 18 minutes and 5 seconds to run. After this, the software project was updated by 50 commits, one commit at a time with scans and subsequent runs in-between.

Of the 50 scanned commits, 8 triggered new test runs of on average 266 test cases and lasting on average 4 minutes and 14 seconds, which was 23.41% of the initial test run. These results are shown in a simplified form in Table 2. The "#" symbol denotes the commit number, counting up from commit 0 that served as the earliest commit, against which the base set of test cases was performed.

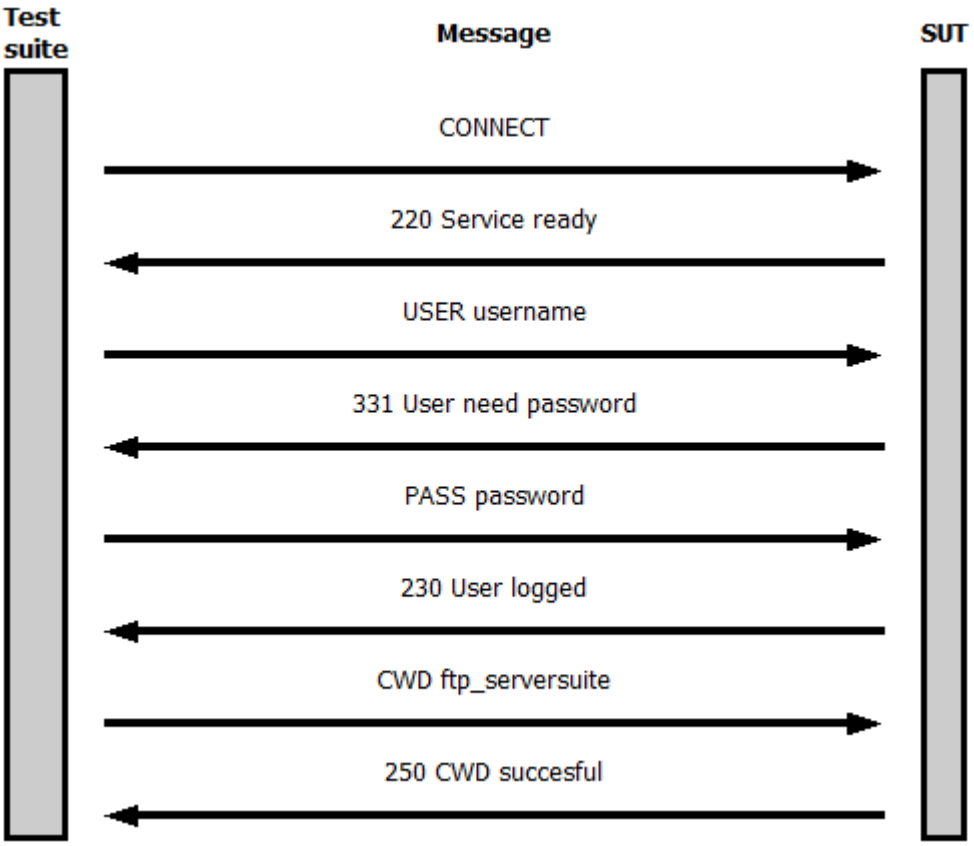

Figure 16. The FTP preamble sent before each test case

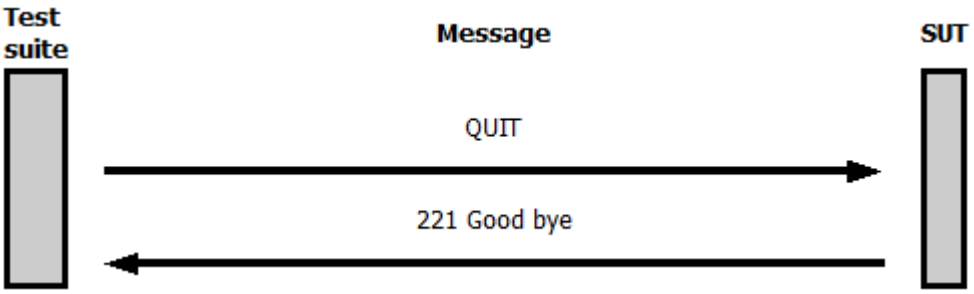

Figure 17. The FTP postamble sent after each test case

4.1.2. Data Analysis

As discussed earlier, 8 out of 50 commits caused a rerun of the test cases, and these results are shown in a simplified form in Table 2. The combined run time of these test runs was 33 minutes and 58 seconds. In comparison, running the base set of test cases (18 minutes and 5 seconds per run) after each commit would have taken a total of 15 hours, 4 minutes and 10 seconds.

This result can be interpreted as reducing the running time by 96.24% from the theoretical maximum. However, a large amount of commits had nothing to do with the networking functionality that was tested. These are the commits that modified areas of the software project, such as the documentation, code commenting, build environment, or any other non-source code part of the application. If these commits are removed from the table, the results will become more realistic.

Table 2. Simplified results from case study 1

#	Explanation	Test cases	Run time
0	Initial commit	1065	00:18:05
...	2 Commits without tests	-	-
3	-	10	00:00:37
...	4 Commits without tests	-	-
8	-	532	00:01:23
...	7 Commits without tests	-	-
16	-	532	00:04:07
...	2 Commits without tests	-	-
19	-	532	00:03:57
...	1 Commit without tests	-	-
21	-	10	00:00:39
22	-	10	00:00:36
...	1 Commit without tests	-	-
24	-	10	00:00:37
...	16 Commits without tests	-	-
41	-	489	00:03:57
...	9 Commits without tests	-	-

Table 3. Filtered results from case study 1

#	Explanation	Test cases	Run time
0	Initial commit	1065	00:18:05
...	2 Commits without tests	-	-
3	-	10	00:00:37
...	2 Commits without tests	-	-
6	-	532	00:01:23
...	1 Commit without tests	-	-
8	-	532	00:04:07
...	1 Commit without tests	-	-
10	-	532	00:03:57
...	1 Commit without tests	-	-
12	-	10	00:00:39
13	-	10	00:00:36
14	-	10	00:00:37
15	-	489	00:03:57

As seen in Table 3, the total amount of commits drops down to 15 of which 8 triggered new test runs. With this setup, the total theoretic maximum running time drops to 4 hours, 31 minutes and 15 seconds. Compared to the length of the performed test runs, this results in the run time dropping by 87.48%. Figure 18 helps in visualizing the difference between the various running times presented in this case study. Retest all is the theoretical total runtime of executing the initial set of test cases after each commit. Filtered retest all is the same, but with the non-source code commits removed. Finally, RTS is the actual time taken by the triggered test runs.

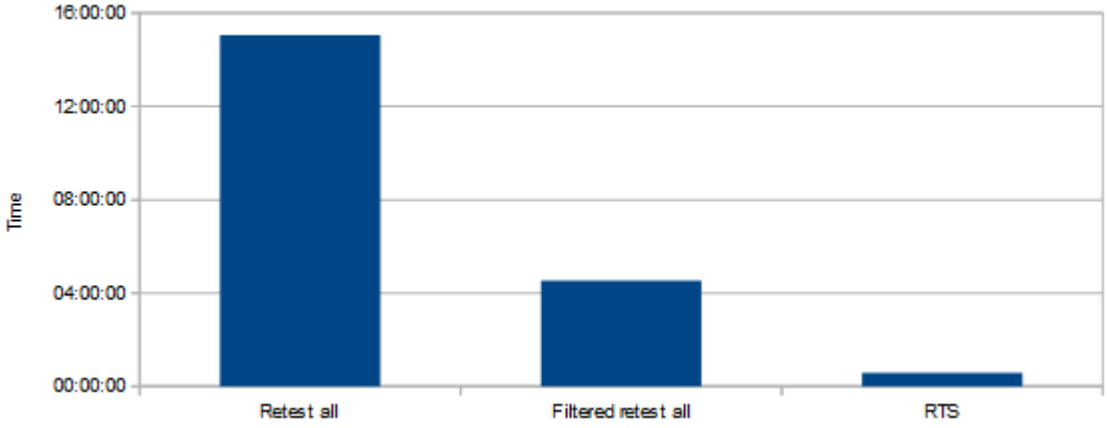

Figure 18. Test run length reduction

This is still a large reduction in runtime, but it should be noted that a large portion of the test cases test very basic networking functionalities of the application, such as sending and receiving messages. Foundations like these are rarely changed, so there will rarely be a need to rerun them. Another consideration is that even though the code coverage of this project was 16%, only 7.39% of the project's code was modified during the 50 commits.

4.2. Case study 2: XMPP Server

4.2.1. Protocol & Software

The setup for case study 2 consisted of an Extensible Messaging and Presence Protocol (XMPP) fuzzing test suite running against Openfire [51], an instant messaging (IM), presence and group chat server. XMPP is an Extensible Markup Language (XML) based communications protocol previously known as Jabber and originally developed by the open-source community. The core principles of the protocol were later standardized by the Internet Engineering Task Force (IETF) in a handful of Request For Comments (RFC) specifications [52, 53, 54] with additional extensions developed by the XMPP Standards Foundation (XSF) [55].

The messages and features tested in case study 2 are show in Table 4. In addition to the core XMPP specifications, the tested features include Service Discovery, specified in XEP-0030 [56], and Multi-User Chat messages, specified in XEP-0045 [57]. All authentication was performed using Simple Authentication and Security Layer (SASL) with Salted Challenge Response Authentication Mechanism (SCRAM) and the US Secure Hash Algorithm 1 (SHA1).

Table 4. Tested features from case study 2

Feature	Explanation
Establish Session	Authenticate user, bind the user to a JID and establish a session with the server
StartTLS	Perform explicit TLS negotiation using StartTLS
Get Roster	Get the roster from the server
Push Roster	Store a new, modified or deleted item on the server
Delete Roster	Delete an item from the roster
Presence	Notify the server of the user's presence
Presence Probe	Probe the server for information about a user's presence
Message	Send an XMPP message to the target JID
Discovery	Query the server for information and items
MUC Message	Join a MUC and send a message to the other occupants

The "Roster" mentioned in Table 4 is XMPP terminology for the contact list for a specified user that is stored on the server. The Jabber Identifier (JID) is a unique identifier for individual entities in XMPP networks. In practical terms, it serves as the identifier behind a username or Multi-User Chat (MUC) as a room-JID.

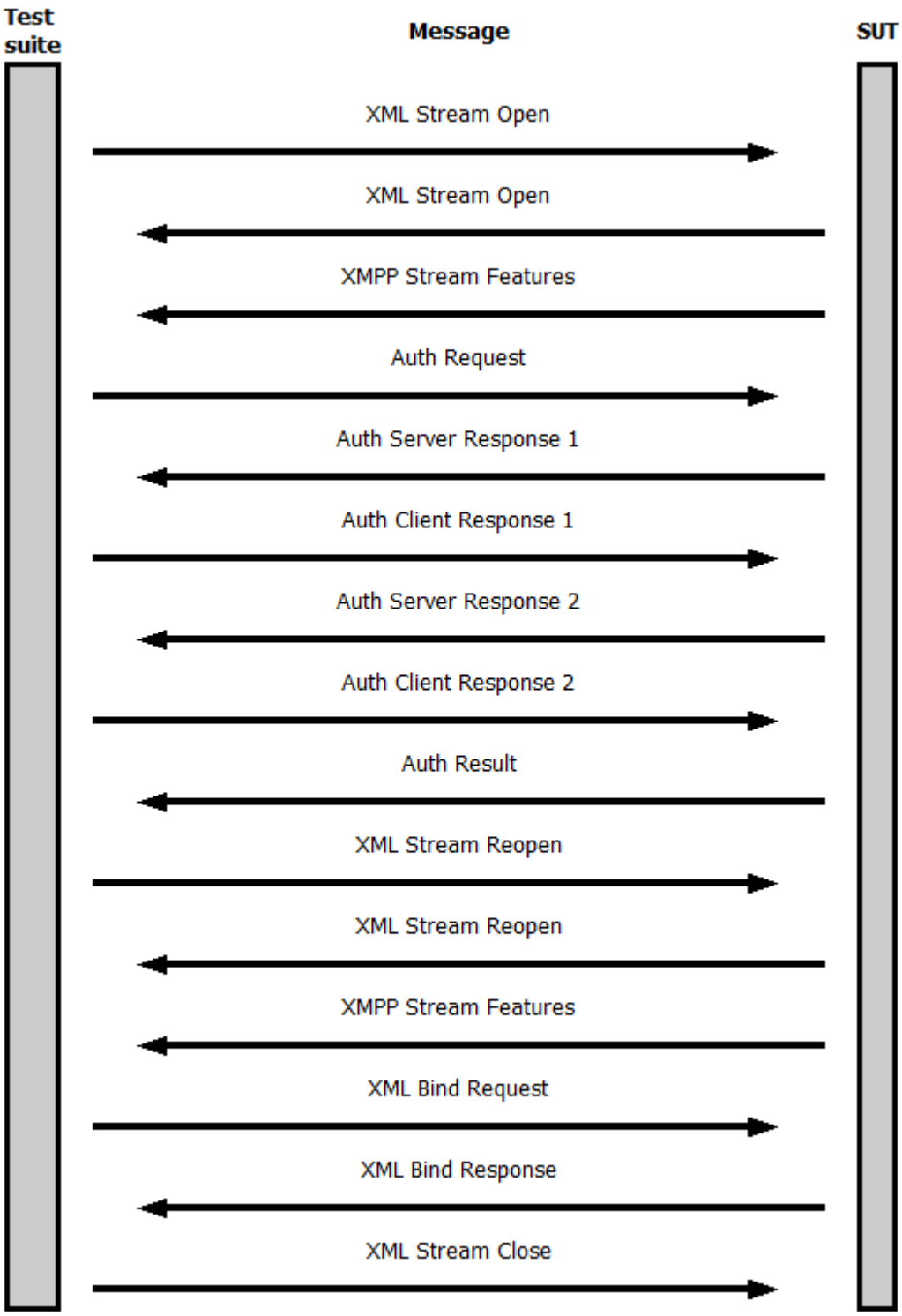

Figure 19. XMPP Establish Session sequence

The "Establish Session" -sequence is shown in Figure 19. The messages in this sequence are only anomalized in the actual sequence itself, but the messages in it are also contained in every other sequence, aside from StartTLS. The sequence is a generic setup that is used to prepare the SUT for messaging.

The "StartTLS" -sequence is otherwise the same, but the various "Auth"-messages are replaced with a simple StartTLS request-response-pair. All the Roster and Presence sequences are simple request-response-pairs preceded by the "Establish Session"-sequence, except for the XML Stream Close -message which is sent as the last message in all sequences to make sure all streams are closed on the server side. The Discovery sequence is otherwise similar, except it contains two request-response-pairs, one for discovering information of the server and one for discovering items on it. The Message and MUC Message sequences are one-way messages preceded by the Establish Session-sequence and finished with an XML Stream Close-message.

Case study 2 was performed by running a preformed test set of 9385 test cases that took 6 minutes and 32 seconds to run. Due to the test suite running such a large volume of test cases in such a short time, the rerunner script that split the test cases into sets caused a noticeable delay in the subsequent runs. Starting the test suite took 22 seconds for each set of 1000 test cases, compared to the average triggered test run executing 3204 test cases and lasting 5 minutes and 25 seconds. On average, starting the test suite took 27.08% of the runtime, in some test runs the starting accounted for over 50% of the test run's duration. This case study was also dotted with commits that broke the functionality of the application. These commits could have been scanned for changes, but actual testing was not possible. Therefore the changes introduced by these commits were scanned together with the next passing build.

Of the 50 commits scanned, 5 were broken commits and 15 triggered new test runs, these results are shown in a simplified form in Table 5. The "#" symbol denotes the commit number, counting up from commit 0 that served as the earliest commit, against which the base set of test cases was performed.

Table 5. Simplified results from case study 2

#	Explanation	Test cases	Run time
0	Initial commit	9385	00:06:32
...	5 Commits without test cases	-	-
6	-	5062	00:09:27
7	-	4729	00:09:09
8	Broken commit	-	-
9	-	4729	00:06:48
10	-	4734	00:06:46
11	-	4729	00:06:44
12	-	4729	00:08:46
13	-	4728	00:06:54
14	Broken commit	-	-
...	1 Commit without test cases	-	-
16	-	3643	00:05:28
...	18 Commits without test cases	-	-
35	-	3643	00:06:21
...	2 Commits without test cases	-	-
38	-	11	00:00:47
39	-	3643	00:06:16
40	Broken commit	-	-
41	-	3643	00:05:44
42	-	9	00:00:42
43	-	9	00:00:42
...	4 Commits without test cases	-	-
48	Broken commit	-	-
49	-	19	00:00:45
...	1 Commit without test cases	-	-

4.2.2. Data Analysis

As discussed earlier, 15 out of 50 commits caused a rerun of test cases. The combined run time of these test runs was 1 hour, 21 minutes and 19 seconds with each test run lasting on average 5 minutes and 25 seconds, which is 82.91% of the duration of the initial test run. In comparison, running the base set of test cases (6 minutes and 32 seconds per run) after each commit would have taken a total of 5 hours, 26 minutes and 40 seconds.

This results in a 75.11% reduction in runtime, a much more realistic value than the one obtained in case study 1, though it is still overly optimistic. Again, clearing away commits that did not have anything to do with the tested source code provides more realistic data.

Table 6. Filtered results from case study 2

#	Explanation	Test cases	Run time
0	Initial commit	9385	00:06:32
...	1 Commit without test cases	-	-
2	-	5062	00:09:27
3	-	4729	00:09:09
4	Broken commit	-	-
5	-	4729	00:06:48
6	-	4734	00:06:46
7	-	4729	00:06:44
8	-	4729	00:08:46
9	-	4728	00:06:54
10	Broken commit	-	-
...	1 Commit without test cases	-	-
12	-	3643	00:05:28
...	6 Commits without test cases	-	-
19	-	3643	00:06:21
...	2 Commits without test cases	-	-
22	-	11	00:00:47
23	-	3643	00:06:16
24	Broken commit	-	-
25	-	3643	00:05:44
26	-	9	00:00:42
27	-	9	00:00:42
...	2 Commits without test cases	-	-
30	Broken commit	-	-
31	-	19	00:00:45
...	1 Commit without test cases	-	-

If the aforementioned non-source code commits are stripped from the data, the results get more realistic. As seen in Table 6, the total number of commits drops down to 32, four of which were broken and could either not be built or executed, which prevents fuzz testing.

Of the 32 commits, 15 caused new test runs, the total runtime of these new test runs was 1 hour, 21 minutes and 19 seconds. If every one of these 32 commits triggered the complete test run that took 6 minutes and 32 seconds to run, this would have taken a total of 3 hours, 29 minutes and 4 seconds to run. This means a 61.10% reduction in runtime.

Figure 20 shows the differences between the various running times described in this case study. Retest all is the theoretical total runtime of executing the initial set of test cases after each commit. Filtered retest all is the same, but with the non-source code commits removed. Finally, RTS is the actual time taken by the triggered test runs.

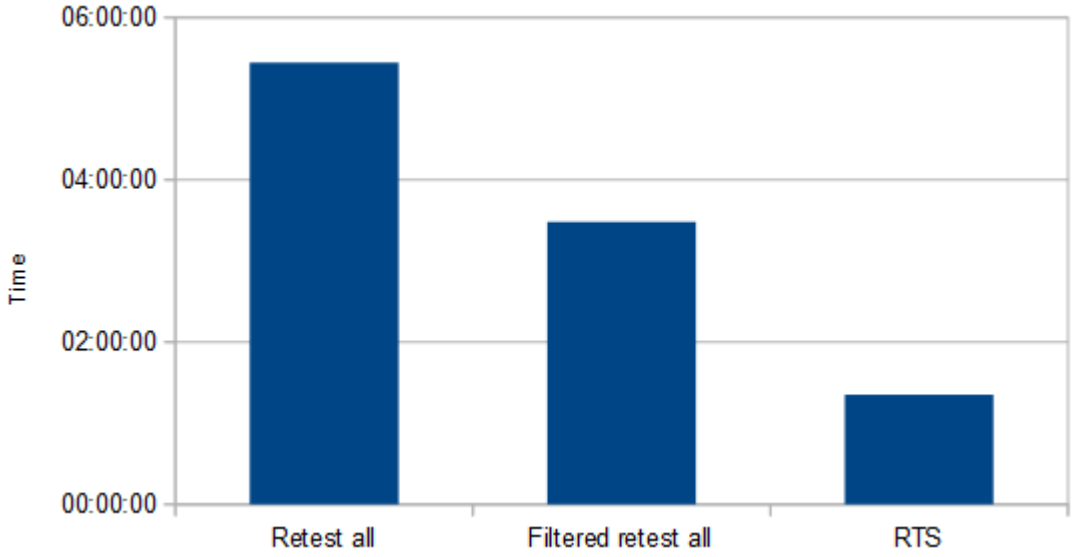

Figure 20. Case study 2 test run length reduction

4.3. Case study 3: OPC UA Server

4.3.1. Protocol Software

The setup for case study 3 consisted of an Object Linking and Embedding for Process Control (OPC) Unified Architecture (UA) fuzzing test suite running against Eclipse Milo [58], an open-source OPC UA implementation containing a protocol stack, client and server Software Development Kits (SDKs) along with example client and server implementations. The testing was performed against the supplied example server implementation.

OPC UA is an industrial automation machine to machine communication protocol and architecture solution developed by the OPC Foundation [59] that combines various OPC Classic specifications into a single framework. These specifications are combined in the International Electrotechnical Commission (IEC) [60] standard IEC-62541 (Table 7).

Of the listed specifications, parts 3, 4 and 6 are of particular interest. Part 3 of the specification describes how the OPC UA address space works. Part 4 describes the interactions that happen between OPC UA clients and servers. Part 6 maps the various services provided by OPC UA into messages and describes the various security mechanisms used.

Table 7. IEC 62541 Overview

ID	Title
IEC/TR 62541-1	OPC Unified Architecture - Part 1: Overview and Concepts
IEC/TR 62541-2	OPC Unified Architecture - Part 2: Security Model
IEC 62541-3	OPC Unified Architecture - Part 3: Address Space Model
IEC 62541-4	OPC Unified Architecture - Part 4: Services
IEC 62541-5	OPC Unified Architecture - Part 5: Information Model
IEC 62541-6	OPC Unified Architecture - Part 6: Mappings
IEC 62541-7	OPC Unified Architecture - Part 7: Profiles
IEC 62541-8	OPC Unified Architecture - Part 8: Data Access
IEC 62541-9	OPC Unified Architecture - Part 9: Alarms and Conditions
IEC 62541-10	OPC Unified Architecture - Part 10: Programs
IEC 62541-11	OPC Unified Architecture - Part 11: Historical Access
IEC 62541-12	OPC Unified Architecture - Part 12: Discovery
IEC 62541-13	OPC Unified Architecture - Part 13: Aggregates
IEC 62541-14	OPC Unified Architecture - Part 14: PubSub
IEC 62541-100	OPC Unified Architecture - Part 100: Device Interface

The standard specifies two protocols for transmitting data over a network, a Web service-oriented protocol that uses regular HTTP Uniform Resource Locators (URLs) such as `http://example:8080`, and a binary Transmission Control Protocol (TCP) protocol that uses proprietary OPC URLs such as `opc.tcp://example:8080`. The tests performed in this case study used the binary TCP protocol, because that was the method supported by the fuzzing test suite.

Table 8. Tested features in case study 3

Security Policy	Message
None	Open Secure Channel
None	Find Servers
None	Get Endpoints
None	Test Stack Request
Basic128Rsa15	Open Secure Channel

The features tested in this case study are listed in Table 8. "Open Secure Channel" forms the base for each subsequent test, and the messages it contains are shown in Figure 21. Each test is started with an OPC Hello-ACK exchange followed by an Open Secure Channel Request-Response exchange. After this, a test group specific request-response-pair is sent, followed by a Close Secure Channel-message sent by the test suite, making sure no sessions are left open on the server side.

The "Basic128Rsa15"' group is essentially the "Get Endpoints" group followed by an "Open Secure Channel" sequence, except that it's using the Basic128Rsa15 security policy, defined by the OPC foundation [61]. The security policies used in OPC UA are simple combinations of an encryption algorithm and a signature algorithm. For example, Basic128Rsa15 is a 128-bit basic encryption algorithm combined with the RSA15 key wrap algorithm. The other security policies supported by OPC UA are

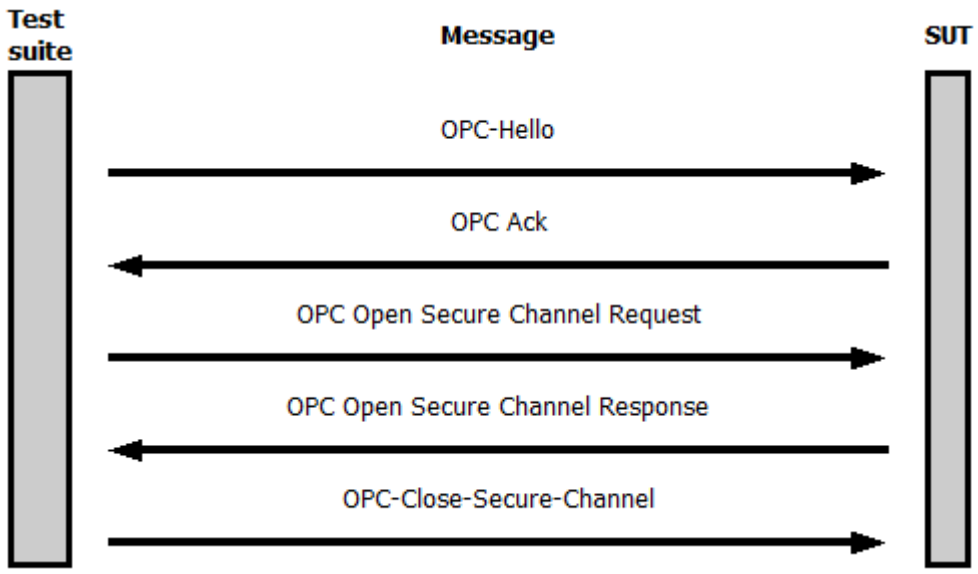

Figure 21. OPC UA Open Secure Channel -sequence

Basic256 and Basic256Sha256. The "None" security policy used in these tests disables all security settings and is generally not used in production environments.

Along with the various security policies, OPC UA also supports three different message security modes: None, Sign and Sign&Encrypt. None applies no security to the messages, Sign will sign the messages but not encrypt them and Sign&Encrypt both signs and encrypts the messages.

OPC UA also supports four different authentication token types. Anonymous Identity Token will not contain any user information. Username Identity Token will contain both a username and a password. X.509 Identity Token will identify the user via an X.509 certificate. Finally, an Issued Identity Token will identify the user with a Web Services Security (WSS) Token.

The "Find Servers" message queries the target for a list of servers it knows, whereas "Get Endpoints" queries the target server for all the endpoints it contains. One server may contain various endpoints that all have their own security policies, URLs, security modes and authentication token types they support (Figure 22). The client can use a discovery URL for getting the list of endpoints along with their security configurations without any security needed for the query message itself.

The test suite that was used contained support for multiple other security policies, as did the target software used in this test. However, the support for other security policies was enabled in the target software only partway through the list of commits processed in the case study. This means that the tests could not have been performed at the beginning of the study. In a normal software testing scenario, tests for new features would have been implemented when the new features were added. This, however, was not in the scope of this thesis, so adding more tests in the middle of the simulated software development was left out.

Case study 3 was performed by running a preformed test set of 7782 test cases that took 12 minutes and 44 seconds to run. After this, the software project was updated by 50 commits, one commit at a time with scans and subsequent runs in-between.

Figure 22. Example "Find Endpoints" response showing 4 endpoints. One of the endpoints displays it's URL, Security Mode, Security Policy and Authentication Token Types

Of the 50 commits, only 4 triggered new test runs, of on average 3992 test cases and lasting on average 6 minutes and 2 seconds, which was 47.38% of the initial test run. These results are shown in a simplified form in Table 9. The "#" symbol denotes the commit number, counting up from commit 0 that served as the earliest commit, against which the base set of test cases was performed.

4.3.2. Data Analysis

As mentioned earlier, 4 out of the 50 commits caused a rerun of test cases. The combined run time of these test runs was 24 minutes and 9 seconds. In comparison, running the whole base set of test cases (12 minutes and 44 seconds) after each commit would have taken a total of 10 hours, 36 minutes and 40 seconds.

This result can be interpreted as reducing the running time by 96.21% from the theoretical maximum. However, a portion of the commits scanned did not touch any code, or only touched code that was not relevant to the tested parts of the application.

Table 9. Simplified results from case study 3

#	Explanation	Test cases	Run time
0	Initial commit	7782	00:12:44
...	2 commits without test case	-	-
3	-	7782	00:12:31
...	5 commits without test cases	-	-
9	Broken commit	-	-
10	-	7744	00:09:11
11	-	45	00:01:33
...	11 commits without test cases	-	
23	Broken commit	-	
...	14 commits without test cases	-	
38	-	395	00:00:54
...	7 commits without test cases	-	
46	Broken commit	-	-
...	4 commits without test cases	-	

If commits that did not touch any code are removed from the table, the results can be shrunken to be more realistic.

As seen in Table 10, the total number of commits drops only to 42 of which 4 triggered new test runs. The reason for such a small drop in commits can be explained with the width of features offered by the project compared to the narrowness of features tested by the fuzzing test suite. The project offered a high-performance stack along with the client and server SDKs built on top of it.

With this filtered table of results, the total theoretic maximum running time drops down to 8 hours, 54 minutes and 48 seconds. Compared to the length of the performed test runs, this results in the run time dropping by 95.48%. Figure 23 helps to visualize the difference between the various running times presented in this case study.

As discussed earlier, the target chosen for case study 3 proved to be a difficult one. As the tested OPC UA Server implementation was only a small portion of the software project, most of the changes that happened during the project did not trigger any new test runs. A notable exception to this was commit number 3. This commit was a large restructuring of the code which necessitated running all test cases again.

Table 10. Filtered results from case study 3

#	Explanation	Test cases	Run time
0	Initial commit	7782	00:12:44
...	1 commits without test case	-	-
2	-	7782	00:12:31
...	3 commits without test cases	-	-
6	Broken commit	-	-
7	-	7744	00:09:11
8	-	45	00:01:33
...	11 commits without test cases	-	
20	Broken commit	-	
...	10 commits without test cases	-	
31	-	395	00:00:54
...	6 commits without test cases	-	
38	Broken commit	-	-
...	4 commits without test cases	-	

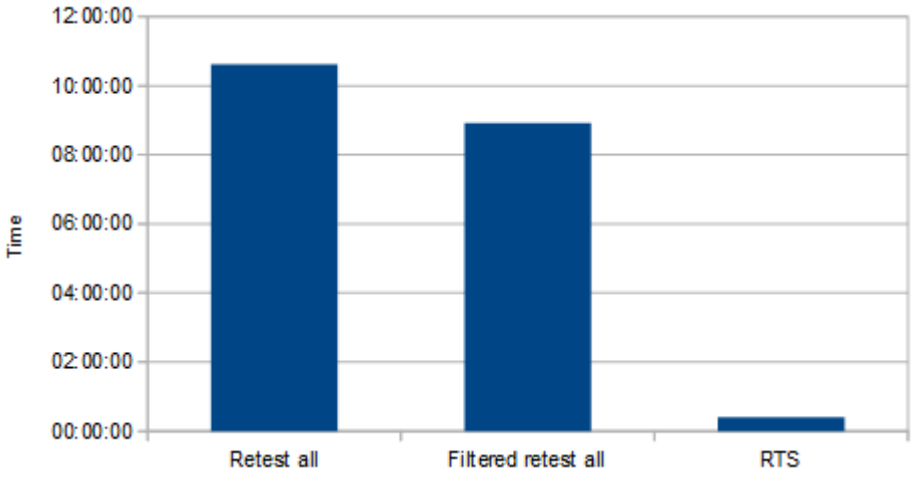

Figure 23. Case study 3 test run length reduction

5. DISCUSSION

The three test subjects chosen for the case studies in this thesis varied greatly in their scope, as did the achieved code coverage percentage of them. Due to this, three varying sets of data were gathered, but the results obtained from them all point to the same conclusions.

5.1. Answers to the Research Questions

At the beginning of this thesis, a two-part research question was posed:

1. How many commits in a software project contain changes that are not covered by testing and therefore do not require a test run?
2. How much smaller is the average triggered test run compared to the base set?

Based on the research data gathered in this thesis, the amount of commits in software project that did not trigger a test run was significant. 96.24% in case study 1, 75.11% in case study 2 and 96.21% in case study 3. This data is based on the presumption that every single commit would trigger a new test run. If commits not related to actual code are removed, the percentages drop down to 87.48%, 61.10% and 95.48%, respectively.

Table 11. A table combining all results from the case studies performed in this thesis

	Case Study 1	Case Study 2	Case Study 3
Theoretical maximum runtime	15:04:10	05:26:40	10:36:40
Filtered maximum runtime	04:31:15	03:29:04	08:54:48
Actual total runtime	00:33:58	01:21:19	00:24:09
Decrease from theoretical	96.24%	75.11%	96.21%
Decrease from filtered	87.48%	61.10%	95.48%
Base test run size	1065 cases	9385 cases	7782 cases
Base test run duration	00:18:05	00:06:32	00:12:44
Average triggered run size	266 cases	3204 cases	3992 cases
Average triggered run duration	00:04:51	00:05:25	00:06:02
Decrease in run size	75.02%	65.86%	56.41%
Decrease in run duration	41.41%	17.10%	52.62%

The filtered percentages result in an average of 81.35% reduction in runtime. While this is a remarkable amount, the variance in reduction time between the three case studies shows that the results obtained by this method can vary greatly based on the tested software and how well the testing tool is tailored to it.

However, the second part of the research question is more interesting. The base test set took 18 minutes and 5 seconds for case study 1, 6 minutes and 32 seconds for case study 2, and 12 minutes and 44 seconds for case study 3. The average length for the triggered test runs in these case studies were 4 minutes and 51 seconds, 5 minutes and 25 seconds and 6 minutes and 2 seconds respectively.

These time reductions equate to 41.41% for case study 1, 17.10% for case study 2 and 52.62% for case study 3, and an average reduction of 37.04%. The results again display the variance between the various software targets. Case study 2 seems to have the smallest reduction in runtime, even though it had a 65.86% reduction of actual test cases, as compared to 75.02% for case study 1 and 56.41% for case study 3. This is explained by technical obstacles in the setup that was used to conduct the experiments and gather the data. Each test set had to be split into groups of 1000 test cases, and each group required the restart of the fuzz testing software. If all test cases could have been performed in a single set, the reduction in runtime would have been greater.

5.2. Assessment of the Used Methods

Using DSCA for optimizing regression testing proved to be a viable solution. In the case of the research performed in this thesis, having to manually update the software and perform the software scans proved to be a lot of manual work. This would not be an issue in a typical CI environment, where changes to software can easily be made to trigger various operations, like the aforementioned software scans.

The usage of git to simulate the software development process was a surprising success. Using git checkout to move to earlier commits to perform scans on the state of the software worked without any major issues. Although a few projects that used automated build environments required a cleaning operation, most of the problems were caused by the operator accidentally running the required commands in the wrong order, or completely forgetting to type one of the commands.

Fuzzing itself may be an unorthodox choice for a regression testing method due to the inherent time requirements of generating, sending and receiving messages and message sequences. Nevertheless, it's a tool among others in the toolbox for software testing, and finding new uses for existing tools is an exiting part in the life of a test engineer.

5.3. Future Work

As the work in this thesis was a performed with both limited starting knowledge and limited amount of time, some aspects of it left clear avenues for future development.

Firstly, the test target selection could be improved by not using open source targets and simulating the process of software development using git. With actual time and resources, it would be better to use actual pieces of software during their development. These would preferably be pieces of software where the achieved code coverage is as large as possible to gather interesting data. In a situation like this, scans for changes in the software could only be started when it was known beforehand that actual tested functionality had been changed, and commits that only changed documentation could be left without a scan.

Another piece of further development would be the actual execution of triggered test runs. As the work performed in this thesis simply launched the fuzzing test suite from the command line supplied with the list of test cases to run, the input limit of the bash command line was quickly ran into. The fuzzing test suite does also support running in

an HTTP server mode, where commands are sent via API calls. This might not remove the need for running the test cases in batches of 1000 cases, but it would allow running all batches without shutting down and restarting the program between them.

Finally, other clear avenues for more work would be additional tools. There are a large number of DSCA tools that support programs written in other languages. Fuzz testing could also be easily replaced by other testing tools and methods.

6. CONCLUSIONS

The objective of this thesis was to create a method for the fuzz testing tool to reliably negotiate with the used DSCA tool and to assess how much fuzz-based regression testing can be optimized using DSCA. This was achieved by creating a set of automation scripts that negotiated the start and end of each test case and by devising a method of replaying past software development with open source software projects.

The software development replay consisted of checking out individual code commits one at a time using the git checkout command and comparing the differences between subsequent commits. These differences could then be used to trigger a test run targeted to only test changed parts of the code.

These methods were used on three separate open source software targets to gather information on how many commits in a software project do not contain changes covered by testing, and to find out how much smaller the average triggered test run was compared to running the base set of test cases again after each commit.

The results of the performed tests show that the amount of commits that are not covered by testing varies greatly between software targets and the code coverage percentage achieved by the testing. The amount of time saved by DSCA was on average 81.35%. The runtime reduction achieved by DSCA was on average 37.04%.

These results show that testing can be optimized by a great margin, at least in the case of fuzz testing. This is partly due to the inherent nature of fuzz testing, as the base set of test cases being performed on the software targets is measured in thousands of cases.

7. REFERENCES

[1] Marijan D., Gotlieb A. & Sen S. (2013) Test case priorization for continuous regression testing: An industrial case study. In: 2013 IEEE International Conference on Software Maintenance, pp. 540–543.

[2] Laplante P.A. & Neill C.J. (2004) "The Demise of the Waterfall Model is Imminent" and other urban myths. amcqueue 1, pp. 10–15.

[3] Elliot G. (2004) Global Business Information Technology: An Integrated Systems Approach. Addison-Wesley.

[4] Microsoft Press & McConnell S. (2002) Rapid Development. Microsoft Press.

[5] Dahl O.J., Dijkstra E.W. & Hoare C.A.R. (eds.) (1972) Structured Programming. Academic Press Ltd., London, UK.

[6] Wells D. (accessed 06.12.2016), Extreme programming: A gentle introduction. URL: `http://www.extremeprogramming.org`.

[7] Takeuchi H. & Nonaka I. (1986) The new new product development game. Harvard business review 64, pp. 137–146.

[8] Fowler M. & Highsmith J. (2001) The agile manifesto. Software Development 9, pp. 28–35.

[9] Booch G. (1990) Object Oriented Design: With Applications. Benjamin-Cummings Publishing.

[10] Duvall P.M., Matyas S. & Glover A. (2007) Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley.

[11] Humble J. & Farley D. (2011) Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation. Addison-Wesley.

[12] Fowler M. (accessed 06.12.2016), Continuous integration. URL: `http://www.martinfowler.com/articles/continuousIntegration.html`.

[13] Feldman S. & Smith P. (accessed 29.01.2017), Make - gnu project - free software foundation. URL: `http://www.gnu.org/software/make/`.

[14] Apache Software Foundation (accessed 29.01.2017), Apache ant - welcome. URL: `http://ant.apache.org/`.

[15] Shaw G., MacLean I., Hernandez S. & Driesen G. (accessed 29.01.2017), Nant - a .net build tool. URL: `http://nant.sourceforge.net/`.

[16] Microsoft Build Engine (accessed 29.01.2017), Github - microsoft/msbuild: The microsoft build engine (msbuild) is the build platform for .net and visual studio. URL: `https://github.com/microsoft/msbuild`.

[17] Beck K. (2002) Test-Driven Development By Example. Addison Wesley.

[18] The CVS Team (accessed 29.01.2017), Concurrent versions system - summary [savannah]. URL: http://savannah.nongnu.org/projects/cvs.

[19] Apache Software Foundation (accessed 29.01.2017), Apache subversion. URL: https://subversion.apache.org/.

[20] Torvalds L. & Hamano J. (accessed 13.10.2016), Git. URL: https://git-scm.com/.

[21] Git (accessed 29.01.2017), Git - git-checkout documentation. URL: https://git-scm.com/docs/git-checkout.

[22] Myers G.J., Sandler C. & Badgett T. (2011) The art of software testing. John Wiley & Sons.

[23] Patton R. (2005) Software testing. Sams.

[24] Tassey G. (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project 7007.

[25] Blair M., Obenski S. & Bridickas P. (1992) Patriot missile defense: Software problem led to system failure at dhahran. Report GAO/IMTEC-92-26 .

[26] Anderson C.W., Santos J.R. & Haimes Y.Y. (2007) A risk-based input–output methodology for measuring the effects of the august 2003 northeast blackout. Economic Systems Research 19, pp. 183–204.

[27] Everett G.D. & McLeod Jr R. (2007) Software testing: testing across the entire software development life cycle. John Wiley & Sons.

[28] Sutton M., Greene A. & Amini P. (2007) Fuzzing: brute force vulnerability discovery. Pearson Education.

[29] Takanen A., Demott J.D. & Miller C. (2008) Fuzzing for software security testing and quality assurance. Artech House.

[30] Miller B.P., Fredriksen L. & So B. (1990) An empirical study of the reliability of unix utilities. Communications of the ACM 33, pp. 32–44.

[31] Hertzfeld A. (accessed 31.12.2016), Monkey lives. URL: http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt.

[32] Zalewski M. (accessed 30.12.2016), american fuzzy lop. URL: http://lcamtuf.coredump.cx/afl/.

[33] OUSPG Oulu University Secure Programming Group (accessed 30.12.2016), Radamsa - ouspg. URL: https://www.ee.oulu.fi/research/ouspg/Radamsa.

[34] Peach Fuzzer (accessed 30.12.2016), Peach fuzzer: Discover unknown vulnerabilities. URL: http://www.peachfuzzer.com/.

[35] Synopsys Inc. (accessed 30.12.2016), Defensics - intelligent fuzz testing. URL: https://www.synopsys.com/software-integrity/products/intelligent-fuzz-testing.html.

[36] Duggal G. & Suri B. (2008) Understanding regression testing techniques. In: Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology, Citeseer.

[37] Leung H.K. & White L. (1989) Insights into regression testing [software testing]. In: Software Maintenance, 1989., Proceedings., Conference on, IEEE, pp. 60–69.

[38] Rothermel G. & Harrold M.J. (1997) A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology (TOSEM) 6, pp. 173–210.

[39] Rothermel G. & Harrold M.J. (1996) Analyzing regression test selection techniques. IEEE Transactions on software engineering 22, pp. 529–551.

[40] Elbaum S., Malishevsky A.G. & Rothermel G. (2002) Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering 28, pp. 159–182.

[41] Rothermel G., Untch R.H., Chu C. & Harrold M.J. (2001) Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27, pp. 929–948.

[42] McGraw G. (2006) Software security: building security in, vol. 1. Addison-Wesley Professional.

[43] Godefroid P., de Halleux P., Nori A.V., Rajamani S.K., Schulte W., Tillmann N. & Levin M.Y. (2008) Automating software testing using program analysis. IEEE Software 25, pp. 30–37.

[44] Jarzabek S. (2007) Effective software maintenance and evolution: A reuse-based approach. CRC Press.

[45] Daniel Stenberg (accessed 12.10.2016), curl. URL: https://curl.haxx.se/.

[46] Apache FtpServer (accessed 11.01.2017), Ftpserver home – apache mina. URL: https://mina.apache.org/ftpserver-project/.

[47] IANA I.A.N.A. (accessed 05.03.2017), Ftp commands and extensions. URL: http://www.iana.org/assignments/ftp-commands-extensions/ftp-commands-extensions.xhtml.

[48] Braden R. (1989) Requirements for internet hosts – application and support URL: https://tools.ietf.org/html/rfc1123.

[49] Postel J. & Reynolds J. (1986) file transfer protocol (ftp) URL: https://tools.ietf.org/html/rfc959.

[50] Ribak J. (accessed 05.03.2017), active ftp vs. passive ftp, a definitive explanation. URL: http://www.slacksite.com/other/ftp.html.

[51] Ignite Realtime (accessed 23.02.2017), Ignite realtime: Openfire server. URL: https://www.igniterealtime.org/projects/openfire/index.jsp.

[52] Saint-Andre P. (2011) Extensible messaging and presence protocol (xmpp): Core URL: http://tools.ietf.org/html/rfc6120.

[53] Saint-Andre P. (2011) Extensible messaging and presence protocol (xmpp): Instant messaging and presence URL: https://tools.ietf.org/html/rfc6121.

[54] Saint-Andre P. (2015) Extensible messaging and presence protocol (xmpp): Address format URL: http://tools.ietf.org/html/rfc7622.

[55] XMPP Standards Foundation (accessed 23.02.2017), Xmpp | specifications. URL: https://xmpp.org/extensions/index.html.

[56] Saint-Andre P. (2008) Xep-0030: Service discovery URL: https://xmpp.org/extensions/xep-0030.html.

[57] Saint-Andre P. (2016) Xep-0045: Multi-user chat URL: https://xmpp.org/extensions/xep-0045.html.

[58] Eclipse Foundation (accessed 08.04.2017), Eclipse milo | projects.eclipse.org. URL: https://projects.eclipse.org/projects/iot.milo.

[59] OPC Foundation (accessed 08.04.2017), Home page - opc foundation. URL: https://opcfoundation.org/.

[60] IEC (accessed 08.04.2017), Welcome to the iec - international electrotechnical commission. URL: http://www.iec.ch/.

[61] OPC Foundation (accessed 08.04.2017), Corrected profile reporting links. URL: https://opcfoundation.org/UA/SecurityPolicy/.