Incorporating Fuzz Testing to Unit Testing Regime
Abstract

Software defects are a common problem, despite of decades of research on how to seek and destroy bugs. Software defects are not a mere nuisance, since they come with a very real cost to the industry and the users of software, leading to loss of millions of dollars, countless hours of work and even human lives. Thus there is a very real need to invent new ways to hunt down software defects.

This thesis aims to answer questions concerning integration of modern software development pipeline and fuzzing, an effective fault-based testing technique with strong background in security and robustness testing. More specifically, this thesis seeks to find out how to integrate fuzz testing with continuous integration frameworks to lessen the redundancy in testing: fuzzing usually has its own, separate testing pipeline. Additionally this thesis looks into the possibility of automating generation of fuzzed unit tests using a tool that would use existing unit tests as the raw material for creating the tests to determine, if such approach could be feasible.

This study consists of theoretical and empirical parts. The literature part explores software testing research for results relevant to this thesis, empirical part describes a prototype of unit test fuzzer developed for unit tests written in Python, and observations of relevant issues made during the development process while also describing experiences of how well test cases generated by the tool or manually could be introduced to the existing continuous integration workflow. Research method applied is design science.

The findings show that creating the tool described is not as easy as it would first seem, listing issues large enough to motivate discontinuing the prototyping after first initial version. On the other hand, integrating fuzzing to a continuous integration based workflow seems to be a feasible idea, and automated test case generation is not the only way to create fault-based unit tests.
Foreword

I would like to thank my thesis supervisor Ari Vesanen for his guidance during the writing process, and for his patience during the writing process. I would also like to thank my colleagues at Oulu Secure Programming Group, my family, friends and Johanna, my significant other for their invaluable support, encouragement and advice. Writing this thesis has been immensely teaching experience for me.

This thesis has been created as a part of the Cloud Software Program.

In Oulu, 10.11.2013

Juho Myllylahti
Abbreviations

AST Abstract syntax tree
CI Continuous integration
GPL General Public License
IEEE Institute of Electrical and Electronics Engineers
JSON JavaScript Object Notation
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
OS Operating System
SUT Software under test or system under test
UUID Universally unique identifier
Contents

Abstract ........................................................................................................................................... 2
Foreword .......................................................................................................................................... 3
Abbreviations ................................................................................................................................. 4
Contents .......................................................................................................................................... 5

1 Introduction .................................................................................................................................... 6
  1.1 Purpose of This Thesis ............................................................................................................. 7
  1.2 Research Question .................................................................................................................. 7
  1.3 Research Methods .................................................................................................................. 8

2 Theory and Prior Research ........................................................................................................... 9
  2.1 Software quality ...................................................................................................................... 9
  2.2 Software Testing .................................................................................................................... 10
  2.3 Why is Software Testing Important? ..................................................................................... 11
  2.4 Test Automation .................................................................................................................... 12
  2.5 Unit Testing ........................................................................................................................... 12
  2.6 Build Automation .................................................................................................................. 14
  2.7 Testing Oracle ....................................................................................................................... 16
    2.7.1 Oracle Problem ................................................................................................................. 16
  2.8 Model-Based Testing .............................................................................................................. 17
  2.9 Fault-Based Testing Methods ................................................................................................. 18
    2.9.1 Fuzzing ............................................................................................................................. 18
    2.9.2 Using Fuzzing as a Metric For Software Quality ............................................................... 22
    2.9.3 Random Testing ................................................................................................................ 23
    2.9.4 Differential Testing .......................................................................................................... 24
    2.9.5 Mutation Testing .............................................................................................................. 25
    2.9.6 Parametrised Unit Testing ............................................................................................... 25
    2.9.7 Random Testing ................................................................................................................ 26
    2.9.8 Other Related Work .......................................................................................................... 26

3 Overview of the Software Tool Created ...................................................................................... 27
  3.1 Existing Software Components Used ..................................................................................... 27
  3.2 Ideas Behind the Creation of the Tool .................................................................................... 27
  3.3 Method of Fuzzing the Unit Tests ......................................................................................... 29
  3.4 Internal Structure of the Tool ................................................................................................ 30
  3.5 Restrictions Imposed by the Tool .......................................................................................... 31

4 Results .......................................................................................................................................... 33
  4.1 Flexibility of Python ............................................................................................................... 33
    4.1.1 Difficulty of Pinpointing Testable Functions and Weeding Out Internal Functions ....... 34
  4.2 Development Toolchain Integration ...................................................................................... 35
  4.3 Lack of Uniformity of Unit Testing ......................................................................................... 35
  4.4 Difficulty of Conceiving a Test Oracle .................................................................................. 36
  4.5 Bug Triaging ........................................................................................................................... 37
  4.6 Integrating Fuzzing Into the Testing Toolchain ................................................................. 37

5 Conclusions ................................................................................................................................. 40

References ....................................................................................................................................... 42
1 Introduction

The idea behind this thesis first began to form when I was getting into fuzzing, a form of software testing method especially popular amongst people who try to find previously uncovered, security-related bugs from existing software while working on limited budgets (fuzzing is really effective in this regard). Unfortunately fuzzing—which has been around for quite a while (B. P. Miller, Fredriksen, & So, 1990) – still has limited popularity outside of security testers. This seemed like a shame, since many software projects could really reap the benefits of fuzzing without spending a lot of effort and everyone would come out as winners, apart from the "bad guys".

This lead into the logical question of "why". The reasons might be numerous and it is easy to come up with some probable causes. For one, the industry does – in some contexts – move slowly: most the developers have never even heard of fuzzing. There might be a notable population of people working with software that regard software testing as Somebody Else’s Problem, a stance that does not exactly help in spreading interest about new and wonderful testing technologies, when even unit tests are seen as a chore hindering “proper work”. Especially when fuzzing usually has the inconvenient feature of finding inputs that break code that works quite nicely under normal conditions; why go begging for a bloody nose? This might be the second big reason, people fear that testing that is "too effective"; bugs found should be fixed, but fixing them requires developer time. Time that could be spent on developing new exiting features which are usually much better selling points of a team doing a good job compared to a vague notion of "software is now more robust" (unless the bugs fixed actually hinder the everyday usage, at which point one does not really need a fuzzer to become sentient about their existence). Thirdly people might be reluctant to complicate their existing testing toolchain.

While pondering on this, I began to wonder how difficult it would be to "smuggle" fuzzing into typical, existing testing infrastructure that self-respecting software projects probably have. Could it be possible to create some form of a plug-in that takes existing unit test suites and begins to fuzz them while fitting snugly into the picture, meaning that it would take minimal effort to try it and that there should be a minimal risk of causing trouble to the existing software development process? If such a component was feasible, it might encourage people into trying fuzzing which could make the world a better place – to paraphrase: "How hard can it be?"

Quite difficult, it turned out. The original idea was to create an initial version of a tool that would generate fuzzed test cases from existing unit tests, test its viability, evolve the design based on those results and optionally iterate this process if needed. Unfortunately, after creating the first prototype of the tool and testing it to some real unit test suites it became quickly apparent that the approach taken would not bear fruit: although the tool worked well for some dummy unit tests written to aid the development, the results were grim when real test suites were tested, resulting in virtually no unit tests being mutated in a way that could have any hope in working as intended, due to matters discussed later in this thesis.

Relating to the original idea it also seemed necessary to investigate how well these test cases could be run in the existing testing infrastructure without affecting the existing development processes; some software teams strive to keep their build "green", and it would be undesirable to include a component that could randomly break the build, caus-
ing havoc when numerous persons would be automatically notified in bold red letters – or with traffic lights, lava lamps or in extreme cases, with foam missiles (Dance, 2012) – that the build is broken and requires immediate attention.

1.1 Purpose of This Thesis

Purpose of this thesis was to look into the possibility of creating an easy-to-use unit test fuzzing tool and to investigate the feasibility of integrating the said tool – or other solution which creates fault-based unit tests – into an existing testing infrastructure in order to study if fuzzing could be made more mainstream using an approach like this. Since fault-based testing is (seemingly) seldom used in software development although its effectiveness is well proven, the problem might be the barrier of entry: fuzz testing is still restricted largely to security-related bug hunting and it a bit of an oddity to the mainstream software testers. It might be easier to persuade developers into trying fuzzing, if it could be easily integrated into their existing testing workflow instead of requiring setting up and maintaining a separate toolchain. If fuzzing one day becomes a common tool for the majority of software testers, it would probably lead to more robust and higher quality software for all of us to use in our daily lives.

1.2 Research Question

The research question consists of two parts. Firstly, part of the research question looked into the possibility of integrating fault-based testing – in form of fuzzed unit tests – into existing testing infrastructure; secondly it was examined how well these fault-based tests could be generated automatically from existing ”normal” unit tests. For the automated test generation, the programming language selected was Python, both as the language of the unit tests as well as the language the unit test mutator was implemented in. This naturally limits the results to be mostly applicable to Python-based solutions. The continuous integration (CI) framework inspected was Buildbot, but naturally results apply to other CI solutions if they have support for similar features.
1.3 Research Methods

The research method of this thesis is design science. Hevner, March, Park, and Ram (2004) list seven guidelines for design science research; they are presented in the following table and accompanied with comments on how this thesis strives to fill those guidelines:

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Design as an Artifact</td>
<td>The artifact produced is the prototype of the software tool, described in section 3.</td>
</tr>
<tr>
<td>2. Problem relevance</td>
<td>Relevance of the problem should be attended in this section and in section 2.</td>
</tr>
<tr>
<td>3. Design evaluation</td>
<td>Evaluation of the successfulness of the design is debated in section 4.</td>
</tr>
<tr>
<td>4. Research contributions</td>
<td>Research contributions are discussed in section 4.</td>
</tr>
<tr>
<td>5. Research rigor</td>
<td>Questions related to the rigourousness of used methods present them in sections 3 and 4.</td>
</tr>
<tr>
<td>6. Design as a search process</td>
<td>As the artifact created did not fulfill the required goals this search is still ongoing; applicable findings so far are described in sections 4 and 5.</td>
</tr>
<tr>
<td>7. Communication of research</td>
<td>Naturally the communication of this research is mostly the thesis itself.</td>
</tr>
</tbody>
</table>

As the artifact created in this research was not satisfactory in view of the goals set, the research should be seen as preliminary, not as rigorous proof of the feasibility of the idea. However it hopefully gives valuable insight for the next iteration of combining the two differing worlds of software testing.
2 Theory and Prior Research

In this section relevant concepts and existing research is exhibited. Firstly relevant, general software testing topics are covered. Secondly different fault-based software testing methods are discussed.

2.1 Software quality

In general, software quality is a vast subject and elusive to define. The definitions here should be seen as bare scratching of the surface.

The quality of software can be measured from many different viewpoints and these viewpoints can be weighted differently; software with elegant internal structure with well-defined functionality can perform a time-consuming computing task adequately when software with incoherent structure and poor user interface can clear out the same task noticeably faster. Even more fundamental questions are the differences of opinions on whether more features equals more quality; for example, some like the UNIX approach that the software should do only one thing and do it well, where others see the same software as lacking in usability and missing several features. Kitchenham and Pfleeger (1996) mention in their article several quality views, and it is easy to see those different viewpoints conflict: if the user is presented with an application built using the UNIX principles, and criticises the lack of features, the value-based view – which sees the customer satisfaction as the metric of quality – is more inclined to see adding bells and whistles to the software as improving the quality, compared with the transcendental view if it is improbable that the added features can be implemented properly instead of adding them hastily, as an afterthought.

Mathur (2008) divides software quality attributes into static and dynamic quality attributes. Static quality attributes include things such as the quality of the code and the documentation, dynamic quality attributes relate to the behavioural quality of the software, when it is used. Additionally Mathur sees completeness, consistency, usability, performance and reliability as important aspects of software quality. In this thesis, the main points of interest are the quality of the code and the behavioural quality of the software, since the testing method applied tests the actual behaviour of the system.

Naturally there are also international standards covering software quality and software quality requirements, such as ISO/IEC 25030, which is a part of the SQuaRE series of software product quality standards. The quality model they used is based on the quality model described in ISO/IEC 9126 standard, containing categories such as functionality, reliability, usability, efficiency, maintainability, portability – these are parts of the internal and external quality – and effectiveness, productivity, safety and satisfaction (dubbed under "quality in use"). The most interesting categories in this thesis are functionality (which includes security), reliability and maintainability. (Bøegh, 2008)
2.2 Software Testing

Software testing is the common name for practices which aim to gain proof that the Software Under Test or System Under Test (SUT) is robust and will not fail when deployed; Mathur (2008) defines that "primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is they conform to the requirements". Usually, it is not feasible to try to aim for "mathematical proof" of correctness since the input space of even the simpler programs is vast and it is not possible to cover all the edge cases – this problem is called the path explosion problem (Godefroid, Levin, & Molnar, 2008; Charette, 2005), another software verification problem that resembles closely the path explosion is the state explosion problem (Valmari, 1998). Due to these problems, one tries to convince oneself that every test permutation executed increases the probability that the software indeed works as intended and that at some point of testing one can deem the program robust enough for use. McKeeman (1998) makes the brilliant notion that software testing is the last chance for the development organization to reduce the number of bugs delivered to customers.

Another problem with the proof of correctness is that the program can be proved to work correctly, but it still might not conform to the requirements set to it since the dynamic quality attributes can be subjective and difficult to define into clear-cut rules. For example, it is obviously difficult to define requirements for usability in a way that the program could be proved – in a mathematical sense – to have good usability. Another issue is the performance: software can, for example, use an algorithm that yields correct results but lacks in performance: these aspects can be difficult to prove but can easily be spotted by testing the software and analysing the performance metrics.

Software testing converges around software defects. There are many different terms and varying terminology used when describing software faults; programmer errors lead to software defects, also known as bugs or faults (Mathur, 2008). In this thesis, the preferred term is defect although some bugs might slip through.

There are numerous different techniques to implement software testing – and they will be discussed briefly later – but concerning the software correctness, the testing can be divided into two different categories: negative (fault-based) and positive testing. There are two distinct things the software must fulfill in order to be seen as robust. Firstly, it must react correctly to valid input. This means that if the SUT is – for example – a calculator it must return correct answers to calculations given as input. Secondly, it must handle incorrect input sensibly. A good example of this is a server, which must not become unstable even though it receives an incorrect input due to temporary malfunction of some other device in the network. In here, the first clause falls under positive testing and the second under negative testing. Positive testing tries to ensure the program works as intended when given valid starting values and negative testing tries to ensure the program reacts correctly to improper input. (Jain, 2011)

When creating software, the developers are primarily concerned with the domain of positive testing since positive testing resonates with our need to implement the required features of the program, a phenomenon called positive test bias (Leventhal, Teasley, Rohlman, & Instone, 1993). Thus – especially if the tester is the same person as the implementer – the test cases written are skewed towards positive testing. This being the case, we should add some testing that the component can also handle improper input,
but at that point we are not adding features per se, we are ensuring that the feature of robustness is there. At this point, many can feel the temptation to just move on instead of polishing the module, which is already "ready" and "working" – especially if the software project is running late, which unfortunately is often the case. In those situations testing is at the risk of being seen as a money-hogging expense not producing any added value – even by the customer. Additionally, if the negative tests are automatically generated the software engineers might argue that no ordinary user would be able to stumble into those bugs and they are just useless noise that prevents them from concentrating on fixing the "real" errors (McKeeman, 1998). McKeeman argues that negative testing – in his instance "differential testing" – is most easily applicable to software, which has already matured to the point that it exhibits a small number of known bugs. He sees that this type of testing becomes an attractive alternative at that point, because it works as a morale-builder for the software team as their software can pass millions of test cases without issues and at the same time they gain assurance that their software is now tested even better.

The most important testing concepts concerning this thesis are unit testing and random testing.

2.3 Why is Software Testing Important?

Software testing is an integral part of software developing process. One of the best indicators of this are the development costs; it is estimated that 39 percent of total software development costs are spent on software testing (Pacheco, 2009). Still, despite this enormous effort software defects are alive and well, causing the loss of money, property and even human lives. Software defects also hinder software development processes, causing delays, raised costs and even reluctance to engage in software projects; even if some task or processes could potentially be performed much more efficiently and easily on software, the fear and concern of faulty and expensive software stalls the transition. Unfortunately, considering the current state of affairs, the choice not to use the software can often be the correct one. The robustness of the software is and should be one of the things to consider when decisions about software are made. Thus, taking these factors into consideration, the resources allocated to software testing should be seen as a sound investment as it will aid risk management, making the software projects less prone to fail or exceed their budgets. (Charette, 2005)

Jones and Bonsignour (2011) have written an excellent book about the economics of software quality where the effects of bad and good software quality are rigorously analysed. They summarise: "The essential message that will be demonstrated later in the book is that a high level of software quality will raise the economic value of software for the producers, financiers, and the consumers of software applications. Conversely, low software quality levels will degrade the economic value of software for both the producers and consumers of software applications." Although they note that software testing itself does not guarantee the high quality of the software, they see it as one of the requirements to create high-quality – and thus more profitable – software.
2.4 Test Automation

Manual testing can be a laborious and human intensive task. This also means that if software is tested only manually, either the testing costs are large or that the test coverage is spotty. Additionally humans are error-prone and subject to fatigue. Conversely: if testing is automated, the cost of testing can be lowered and/or the quality of the testing can be improved due to the more efficient usage of the testing budget (Clark, Dan, & Hierons, 2013). Thus it is easy to see that the need for testing automation is tremendous. Unfortunately, many of the testing solutions are often difficult to generalise, which forces – to an extent – the software development teams into developing their own ones, taking some components that are universal and then applying some domain-specific parts to create a suitable automation gestalt. (Mathur, 2008)

Usually the test automation is combined with the build automation: each new software revision is first automatically tested and after the necessary tests have passed, a new build is created. Software doing this work is often called continuous integration software. (Fowler & Foemmel, 2006)

The biggest drawback of test automation is the potential labouredness of setup. It seems that it is difficult to conceive universal solutions for testing tools and thus some configuration and customisation steps are required in order to embed the testing components to the existing software development process (Buildbot, 2013c; Graham & Fewster, 2012). One big problem of test automation is also the so-called oracle problem (discussed closer in subsection 2.7).

2.5 Unit Testing

The IEEE standard 1008-1987 defines unit testing as follows: ”Software unit testing is a process that includes the performance of test planning, the acquisition of a test set, and the measurement of a test unit against its requirements. Measuring entails the use of sample data to exercise the unit and the comparison of the unit’s actual behaviour with its required behavior as specified in the unit’s requirements documentation.” (ANSI/IEEE, 1993) In other words, unit tests seek to isolate and test individually all the smallest, separate modules the SUT is constructed of; sometimes this requires separate mock code or throwaway drivers to be written so that the dependencies the modules have can be eliminated. Unit tests are usually technical in their nature, defining exact inputs and the corresponding outputs expected of the software when the test case is executed. (Runeson, 2006)

So, unit testing isolates units from the SUT and tests them separately, comparing pre-defined, expected results with the results actually returned by the unit. For example, a naive test case written in pseudocode to test calculator software might look similar to this:

```python
function addition_test():
    # Create new calculator object, use it to add
    # 2 + 2 and assert it matches with the expected
    # result (4).
    calculator = new Calculator()
```
Naturally unit tests are not usually as simple and straightforward as pictured here — although they should be as simple as possible! — since when testing complex software systems consisting of many parts, the requirement of isolation does usually mean set-up overhead. There is often the need to mock certain parts of the system in order to limit the testing to the singular unit under test. For example, if we are testing management software for industrial hardware, we could create a mock object to communicate with instead of sending commands via serial interface to the actual hardware. This prevents software tests from failing if the motor malfunctions for some reason and as an added bonus we do not need to connect the build system with potentially complex and expensive hardware. Naturally the software needs to be tested with the actual hardware also, but it is done elsewhere, not in this specific test case.

Much of the power of the unit tests lay in the prevention of software regressions — a class of software bugs, where a feature of the software ceases to function as intended in a subsequent version (Rothermel, Untch, Chu, & Harrold, 2001). If unit testing covers the functionality, the regression is caught as soon as the unit test suite is executed after the breaking change. To help ensure that the unit tests are systematically run after each change, a continuous integration system — component which automatically builds and runs unit tests to each new software revision committed to the version control system; explained in greater detail shortly — can (and should) be added to the software pipeline.

Another problem with the unit tests is how good the coverage of it actually is. There are tools available to measure unit test coverage, but their usage is infrequent. This poses an obvious problem since the unit test suites are only as good as their coverage is. An untested code path does not fail the test suite even if a bug is introduced. Often these paths are some parts of the programs that are difficult to unit test, such a GUI interactions, or functionality that would require extensive creation of mock objects or simulation of complex system states. Similar blind spots can be "obvious" system features; the test suites are often written by the developers themselves, which can affect what kinds of tests are written. Thus, the coverage of the test suite needs to be measured if the goal is to do effective unit testing. (Runeson, 2006; Williams, Kudrjavets, & Nagappan, 2009)

It can also be difficult to measure when the SUT has been subjected to adequate testing, additionally the developers can grow tired on writing unit tests, which can affect the quality of the tests created (Runeson, 2006). There can also be too many unit tests: it should be possible to run the test suite in a reasonable amount of time, since making the test suite too cumbersome to run eats away the motivation to use it. (Williams et al., 2009)

Due to the principle how the unit tests work — isolate an indivisible unit from the dependencies and test it individually — it can be difficult to add unit testing as an afterthought. Committing to effective unit testing also affects the whole software development process, since the time required by the writing of the tests needs to be factored into the development schedule. It also requires the managers to support and require the unit testing to be there, since the opposite usually means that the effort for the most part will run dry after a few months. Coordination between teams is also required in order to unify the testing tools and conventions and to allow all actors — not just developers — to add unit tests to the repository. (Williams et al., 2009)
2.6 Build Automation

Term build automation contains all the steps taken to automate the software build process in order to diminish the need for human interaction (for an example, see Figure 1 which describes the build steps of the Chromium browser project). In modern software development build automation ensures that the software build stays free of compilation-fatal bugs and that building the software from the source files takes minimal effort. One of the pieces of this puzzle are continuous integration servers, which compile the software – also integrating external dependencies, if they exist – and run the test suites for it after each new revision is committed into the software repository or after some other event triggers. They also handle sharing out the information about the build’s status: they might send an e-mail alert to the developers when the software build fails so that the problem can be isolated and fixed as soon as possible. Another option is to visit the web interface of the continuous integration server to find out information about, for example, build’s status and the build slaves (see Figures 2 and 3).

<table>
<thead>
<tr>
<th>Step name</th>
<th>Script</th>
<th>Description</th>
<th>When is it orange</th>
<th>When it is red</th>
</tr>
</thead>
<tbody>
<tr>
<td>svntk</td>
<td></td>
<td>Kill all leftover svn processes before starting a new test cycle.</td>
<td>NA</td>
<td>NA, contact trooper</td>
</tr>
<tr>
<td>update</td>
<td></td>
<td>Update the internal build scripts on the slave.</td>
<td>NA</td>
<td>SVN server failure, contact trooper</td>
</tr>
<tr>
<td>update</td>
<td></td>
<td>Update the checkout with gclient. This also runs the hooks like gyp to generate the make/cksum/keys/cksum files.</td>
<td>NA</td>
<td>SVN server failure or DEPS breakage; contact trooper</td>
</tr>
<tr>
<td>taskkill</td>
<td></td>
<td>Kill a bunch of other possible leftover processes (test_shell.exe, ui_tests.exe, etc.) that would interfere with a clean run.</td>
<td>NA</td>
<td>NA; contact trooper</td>
</tr>
<tr>
<td>check deps</td>
<td>checksdp.py</td>
<td>Ensure that source dependencies stay clean. It’s done by parsing and .files according to rules to DEPS files.</td>
<td>NA</td>
<td>A bad change; revert foundBuild failure; rebuild and retry</td>
</tr>
<tr>
<td>compile</td>
<td></td>
<td>Compile the executable</td>
<td>NA</td>
<td>A bad change; revert</td>
</tr>
<tr>
<td>archive build</td>
<td></td>
<td>Archive the executable and symbols into &quot;snapshots&quot;</td>
<td>Failed to fetch the ut requested. The last archived build is used instead.</td>
<td>Failed to fetch any build, the slave probably needs to be restarted, contact a trooper</td>
</tr>
<tr>
<td>extract build</td>
<td></td>
<td>Extracts an archive build on a faster from the corresponding &quot;builder&quot;</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>various tests</td>
<td></td>
<td>Run unit tests, debugger_unittests.py, chrome_tests.py, etc.</td>
<td>See testing information</td>
<td>Only FLAKY_tests failed</td>
</tr>
<tr>
<td>layout tests</td>
<td></td>
<td>Run html based tests from wakatip</td>
<td>Tests marked as FAIL passed, no test unexpectedly failed. See test expectations in layout test doc.</td>
<td>Unexpected layout test failure. It’s usually related to a Wakatip fail.</td>
</tr>
<tr>
<td>BVT tests</td>
<td></td>
<td>Run tests on actual ChromiumOS hardware</td>
<td>NA</td>
<td>Tests failed or machine broke.</td>
</tr>
<tr>
<td>Reliability tests</td>
<td></td>
<td>Run distributed tests to find non-deterministic crashes. It is green when only &quot;known crashes&quot; happens</td>
<td>Fails to grab the summary of the test run for the expected build.</td>
<td>New stack traces appeared in crashes.</td>
</tr>
</tbody>
</table>

Figure 1: Chromium project’s build steps (Chromium, 2013b).

It is important to note that since the information about the build chain is used like this, the test cases generated by the tool described in this thesis must be somehow distinguished from the normal unit tests, since the defects found might not be introduced by the latest change to the code base. If they cannot be told apart, the tool has a serious flaw since it interferes with the normal work flow of the software development team by making the information on the build’s status a less trustworthy indicator of the commit quality, since we add a random variable to the equation. Thus we need a way to tell the different test suites apart, and bug found by the testing generated by the tool must not break the build. Naturally we need to be able to save the failing test case so we can fix the bug at a later phase, when the development process used allows us to do so.

Often build automation has the side effect of making the dependency management more formal. When a new library is added as a dependency, build scripts must automate the
fetching or the library must be added to the build slaves manually. This lessens the chance that a new dependency is added to the project by accident. Note that this also includes platform-specific dependencies if the project in question is of a multiplatform variant.

**Figure 2:** Screenshot of the Chromium project’s build status timeline (Chromium, 2013a).

One interesting aspect concerning build automation is that if doing multiplatform development, the continuous integration server can be set up with multiple build slaves, each running different operating systems. This eases the multiplatform development by ironing out some of the interoperability problems at an early phase. For example, if developer uses an operating-system exclusive system command without providing an alternative execution path, the build will fail when unit test testing that code path will fail on a build slave running some other operating system flavour. Platform-specific software regressions are a common pest, so this is a valuable feature when developing multiplatform software.

**Figure 3:** Screenshot of the Chromium project’s build slave information page (Chromium, 2013a).
2.7 Testing Oracle

(Abran & Bourque, 2004) define the testing oracle as follows: "An oracle is any (human or mechanical) agent which decides whether a program behaved correctly in a given test and accordingly produces a verdict of ‘pass’ or ‘fail.’ There are many different kinds of oracles, and oracle automation can be very difficult and expensive." Thus, the testing oracle is an integral part of software testing, because it is impossible to test software if we cannot determine if tests passed or failed. In manual testing, the oracle is usually the person herself, but when automating testing some other solution must be thought out. One of the simplest test oracles one can come up with is a component which checks if the software instance is still up – and thus not crashed – after the test case has been run or a component which polls that the web service is still responding after the test case has been sent (McKeeman (1998) considers this as one class of results that are easy to evaluate). Naturally one might wish for a testing oracle producing more fine-grained data, such as if the state of the SUT has become abnormal or if the test case caused a side effect in SUT, such as an unexpected system or URI call. Unfortunately automating the testing oracle is not always easy, or even feasible (Hoffman, 2001).

The purpose of an oracle in software testing is to be the indicator of the software state: is everything in order, was the input we offered handled correctly and without affecting the software state in an unexpected way, does the program still work as expected etc. Usually, an oracle is needed in order to automate the testing and reduce the human effort – it is far more efficient to have the state monitored by the testing environment than to have a person inspect the SUT state and output after each test case. (Utting, Pretschner, & Legeard, 2006)

A good example of a usable oracle could be the response headers of an HTTP server: if the server response code doesn’t start with 5, we can assume the input has been handled correctly – or at least for the HTTP server’s part; conversely, if the server responds with 5xx code, we know something has clashed internally and that the server has failed handling the test case correctly. Further, if the server responds in incomprehensible scribble, responds more slowly than expected or – in extreme cases – ceases to respond at all, we can safely assume that the server has failed, possibly in a detrimental way. Using these information tidbits, we can construct our testing oracle, which can be queried to determine if the test case was handled correctly or not.

In model-based testing, the oracle is partly the model itself: a certain transformation – test case – is performed both in the model and the SUT, and then the state of the SUT is compared with the expected state pointed out by the model (Utting et al., 2006). If these two states do not match, we presume the test has failed. Compared with the ad hoc test oracle presented earlier, the model-based oracle has the advantage of more fine-grained information on what output is correct: based on the model we could, for example, catch a bug where the HTTP server returns code 404 (Not found) instead of 403 (Forbidden); the model automates the generation of the test oracles (Utting et al., 2006).

2.7.1 Oracle Problem

In a nutshell, the oracle problem is the difficulty of attaining a test oracle – especially an automatable one – and it is also one of the biggest obstacles one might come across when
automating testing and/or engaging in fault-based, negative testing (Andrews, Menzies, & Li, 2011). McKeeman (1998) goes as far as describing the evaluation of the result of the test as "the ugliest problem in testing". Often an incomplete, "good enough" testing oracle can be created: this testing oracle variant is called a partial oracle (Weyuker, 1982). The partial testing oracle is not perfect and can – and will, by definition – give incorrect results, but is trustworthy to some degree and might be the best feasible solution. In this situation the test cases flagged by the partial oracle as failing ones need to be examined by hand. If the oracle is good enough and the amount of test cases to be investigated is acceptable, this solution is usually satisfactory. Unfortunately though it still makes the system less than optimal. We also need to note the more unfortunate situation, where we cannot create even a usable partial oracle. At this point it is possible that the idea of automatable testing needs to be surrendered. Another option – if possible – is to accept that the system is going to miss bugs and lower the sensitivity of the oracle, hoping that the testing catches at least some bugs to justify the effort spent on creating and maintaining it. All in all, fact still stands that some form of a feasible oracle is needed, and if the testing is automated that requirement must extend to the oracle as well.

2.8 Model-Based Testing

The foundational idea behind model-based testing is that a formal, abstract model of the SUT is created. This model is then used to generate test cases that are run against the SUT; if the state of the SUT does not match the state of the model, a bug is found – either in the model or in the software.

Good modeling formalism should be precise enough that all the main features of the SUT can be modeled and thus tested. On the other hand, it should strive not to become overly expressive, since this leads to slower adaptability, making the modeling potentially more tedious and the resulting model harder to understand. The balance should be measured carefully in order to be able to cut the correct corners. (Utting et al., 2006) Good formalism can also allow more reusability of model components; a bad one might not be reusable at all, leading into a much more complex model. If the model uses reusable components and is modular, it is easier to omit the unimportant components of the SUT, allowing the test case generation to be easier. It should also allow automated modification of the model. (Lamsweerde, 2000)

One important aspect of the model is that all the model states can be reached by the other states. If not, this could lead into situation where the test generation hangs when the desired end state is unreachable. It could also be seen as a smell that the model is not completely modeling the important parts of the SUT, and that the model completeness should be finished. (Utting et al., 2006)

SUT affects the feature requirements in many ways. One such thing is the complexity of the data used: for some SUTs all the data requirements can potentially be modeled with simple value ranges or even lists of acceptable values; some SUTs handle so complex data that need might arise to create a separate model for the data generation, since the Chomsky hierarchy of the data grammar simply doesn’t allow to model it as, for example, finite-state automaton (Puuperä, 2010).

For more complex problems on resolution is to build the so called programmatic model which gives more headroom in constructing the model, for example, data and behaviour
can be intermixed (Utting et al., 2006). The downside of this approach is that the result is potentially more ad hoc and less "theoretical", so many otherwise applicable methods and tools cannot be used. One could perhaps compare this type of a model building with creating solutions by shell scripting different software units together instead of going for the monolithic ivory tower solution.

The architecture of the SUT can also affect the model. For example, in concurrent systems the software code can be highly modularised and the communication paths and dependencies rigidly defined. Thus it makes sense to take advantage of this information when constructing the model; in the best cases it might allow to create a good model of the SUT quickly and efficiently. (Utting et al., 2006)

For example, software modules written in Erlang express quite easily if they hold any state and thus it can quickly be seen during model creation if we need the methodology to be able to model the state at all, or if we can trust the SUT and thus the model representing it to be stateless. To continue the example, the primary communication method of modules in Erlang is message passing, thus the model can usually omit other kind of interference with the model entities.

If the SUT modeled is nondeterministic or its structure changes during execution, the model generation would become more tedious. It also creates more pressure to check the requirements for completeness and can require planning beforehand how to handle the possible voids. (Utting et al., 2006)

2.9 Fault-Based Testing Methods

This subsection describes fuzzing and other fault-based software testing methods and work related to them.

2.9.1 Fuzzing

Fuzz testing or fuzzing is an effective form of fault-based random or semi-random testing with a strong background in security and robustness testing (J. DeMott, 2006; Godefroid et al., 2008; Bratus, Goodspeed, & Johnson, 2012). The simplest form of fuzz testing is generating random data out of nowhere and feeding it as an input to the SUT, practice akin to this is commonly known as black-box fuzzing or black-box random testing (B. Miller & Cooksey, 2006; Godefroid et al., 2008). Although this method of software testing might seem naive and too "easy" to yield any real findings, the results speak for themselves; Microsoft, for example, has had very encouraging results with fuzzing (Godefroid et al., 2008) and actually requires it in their Security Development Lifecycle (Howard & Lipner, 2009; Microsoft, 2013), see also Figure 4.

Sutton, Greene, and Amini (2007) mention that what fuzzing lacks in elegance, it makes up for in simplicity and effectiveness. Fuzzing is especially good in finding bugs related to memory management such as use-after-free bugs (J. D. DeMott, Enbody, & Punch, 2013). Although using languages that automate memory management by using garbage collectors – such as Java or Python – remove some (not all!) bug types, memory bugs are still a big problem, especially security-wise (Veen & Dutt-Sharma, 2012).
What is the Security Development Lifecycle?

The Security Development Lifecycle (SDL) is a software development process that helps developers build more secure software and address security compliance requirements while reducing development cost.

Verification Phase

**SDL Practice #11: Perform Dynamic Analysis**
Performing run-time verification checks software functionality using tools that monitor application behavior for memory corruption, user privilege issues, and other critical security problems.

**SDL Practice #12: Fuzz Testing**
Including program failure by deliberately introducing malformed or random data to an application helps reveal potential security issues prior to release while requiring modest resource investment.

**SDL Practice #13: Attack Surface Review**
Reviewing attack surface measurement upon code completion helps ensure that any design or implementation changes to an application or system have been taken into account, and that any new attack vectors created as a result of the changes have been reviewed and mitigated including threat models.

**Figure 4:** Verification phase of the Microsoft’s Security Development Lifecycle (Microsoft, 2013).

There are several different approaches to fuzzing. The black-box method described earlier is usually the simplest way to do fuzz testing (B. P. Miller et al., 1990; McNally, Yiu, Grove, & Gerhardy, 2012). The minimal example of this method could be demonstrated on UNIX-based systems as follows:

```
cat /dev/random | ./SUT
```

So, while this simple piping of random data to the SUT seems primitive and unlikely to yield any results, reality unfortunately is not as graceful – even though brief testing with method like this should be seen as bare minimum the software should be able to digest without unspecified behaviour if any kind of input validation has been implemented. Of course one might argue that given enough time this method will also inevitably exhaust any input space the SUT has. Pure black-box method like this was used by B. P. Miller et al. (1990) as they tested the utilities with random strings and used a very simple test oracle: application hang-ups or crashes were deemed a test-failing condition, every other outcome was passed as a test success (Sutton et al., 2007).

The downside of random, absolute black-box fuzzing is that software input is seldom completely unstructured. For example, if the software tested is an HTTP server, the probability that the random data generated even resembles a valid HTTP header is negligible. This means – provided the HTTP server is at least somewhat advanced – that
great deal of the test cases are simply rejected by the HTTP server and thus the testing effort is mostly wasted. (Viide, Helin, & Laakso, 2008) Thus in order to make testing more efficient in programs that take structured data as input, one needs to "guide" the fuzzer to generate test cases that carry more resemblance to the valid input data and thus get closer to the semantic core of the software, beyond the initial barrier of entry (Ganesh, Leek, & Rinard, 2009). There are a few ways to carry out this task. One is to generate a model of valid input data and then create new, mutated data generations based on that model (Viide et al., 2008). Other approach is to take one or more valid input files and then mutate them in different ways, treating them as raw binary data (OUSPG, 2012b). Sutton et al. (2007) note that all fuzzers fall into one of these two categories; mutation-based fuzzers, which apply mutations to existing input samples to create test cases or generation-based fuzzers, which use a model of the target protocol to create test cases from scratch. This taxonomy can be argued against, though, since the two techniques can be combined, resulting in a fuzzer that maps somewhere in the middle of the two categories (Pietikäinen et al., 2011).

It should be noted that there seems to be some flux on how the terms "black-box fuzzing" and "white-box fuzzing" are defined and used. For example Neystadt (2008) defines black-box fuzzing as "sending of malformed data without actual verification of which code paths were hit and which were not" and white-box fuzzing as "sending of malformed data with verification that all target code paths were hit – modifying software configuration and the fuzzed data to traverse all data validations in the tested code". Additionally he defines terms "dumb fuzzing" and "smart fuzzing" to distinguish between fuzzers that either care or do not care of the structure and schema of the data they fuzz. Roughly, these definitions could be mapped so that mutation-based fuzzers resemble more dumb fuzzing than generation-based fuzzers although they can also contain smart fuzzing-ish properties. The definition of Neystadt might also allow some analysis of the SUT – such as using utility akin to UNIX’s strings to scan SUT binary for ASCII strings in order to feed them to the fuzzer – while still remaining in the black-box category; others might argue that behaviour like that makes the fuzzer a white-box – or at least a grey-box – fuzzer. (Helin, Viide, Laakso, & Röning, 2006)

White-box fuzzing can also be done in a manner where the inner workings of the program are tediously observed to determine, for example, input constraints laid out by the program during it’s execution or to gain knowledge of test coverage. This information can then be used to guide the fuzzer to produce certain kinds of inputs which increase the test coverage or pinpoint the testing to certain parts of the software. (McNally et al., 2012; Bounimova, Godefroid, & Molnar, 2012)

Mutation-based fuzzing and the term "fuzzing" was introduced by Barton Miller and was successfully used against UNIX utilities to locate numerous defects (B. P. Miller et al., 1990). According to Miller, the original work was inspired by being logged on to a modem during a storm, which caused junk characters to emerge to the data streams due to line noise, causing programs to crash (Neystadt, 2008). Miller and his colleague repeated their test against UNIX utilities in 1995 (B. Miller, Koski, Lee, Maganty, & Murthy, 1995), against Windows NT in 2000 (Forrester & Miller, 2000) and against Mac OS applications in 2006 (B. Miller & Cooksey, 2006). The idea behind generation-based fuzzers can be traced farther: algorithm resembling generation-based fuzzing was described in 1972 by Paul Purdom in his paper "A Sentence Generator for Testing Parsers" (Purdom, 1972; Holler, Herzig, & Zeller, 2012).
Miller writes that they often met – at least used to meet – resistance from the testing and software engineering community, probably due to the lack of a formal model and methodology and undisciplined approach. He comments that fuzzing should not be seen as a contender for more formal software testing methods, more so as a useful and extremely easy addition to the software tester’s toolkit. (B. Miller, 2009)

Fuzz testing is often performed in an ad hoc manner; Sutton et al. (2007) describe fuzzing phases as follows: “Depending on various factors, the fuzzing approach chosen can vary greatly. There is no correct approach to fuzzing. It is entirely dependent on the target application, the skills of the researcher, and the format of the data being fuzzed.” They continue by defining a few phases, which are always performed according to their experience (it should be noted that their focus is on security testing). The phases are presented in the following list:

1. Identifying the target. The target determines which fuzzing tool or technique should be used. If the testing is done to an internally developed application, this step has already been fulfilled; if not, the tester should check vendor’s past track record on security vulnerabilities: poor history increases the chances of finding new defects. Additionally it might be required to concentrate on a subset of the target, such as a specific file format or software library.

2. Identifying inputs. Almost all exploitable vulnerabilities are caused by applications accepting user input and processing that data without proper sanitising or validation. Enumeration of input vectors is critical to the success of fuzzing. It should be noted that this includes every input vector: environment variables, registry keys, and so forth.

3. Generating fuzzed data. Based on the information gathered a decision should be made whether to use a mutation- or generation-based fuzzer. If mutation-based fuzzer is used, the samples are collected at this point. The data generation phase should be as automated as possible.

4. Executing fuzzed data. At this phase, the generated data is fed to the SUT. This phase, too, should be automated.

5. Monitor for exceptions. At this phase the SUT is observed for faulty behaviour. The testing should be constructed so that the crashes found can be easily pinpointed to a singular input.

6. Determining exploitability. Once a fault has been identified – depending on the goals of the testing – it can be necessary to determine if the defect found is exploitable by an attacker. This step is typically manual process and not easily automatable.

Additionally fuzzing is often an iterative process, similar to the exploratory testing. Usually, the tester writes scripts that automate generation of fuzz cases and the process of feeding the generated fuzz cases to the SUT and observing the test oracle. When using a mutation-based fuzzer, the script points the fuzzer to a folder of samples that it begins to process, outputting the test cases to another folder. Another script feeds the sample files iteratively from this folder to the SUT and might, for example, use UNIX utility grep to filter the console output of the program or operating system diagnostical data to
determine crashes. If a test case that causes a crash is found, it is saved for further inspection. Additionally tools such as AddressSanitizer (Serebryany, Bruening, Potapenko, & Vyukov, 2012) can be used to improve the test oracle. As the testing harness has been created, the fuzz testing should run automatically without the need for human intervention, apart from occasionally checking that the system runs normally and if any potential test cases have been caught. After a while it should be apparent if the testing does or does not yield results. There might be bugs that trigger constantly, ”heisenbugs” which are caused, for example, by race conditions, and are difficult to reproduce; it might also be that the fuzzing does not seem to produce any results. At this point there might be a need to improve the sample set or check the scripts that they work as intended and that the test oracle does not miss crashes – problems concerning the test oracle seem to be a problem of the exploratory testing as well (Itkonen, Mäntylä, & Lassenius, 2007) due to fact that the oracle used is often a partial oracle which is only ”probably right”; this is especially true with automated fuzzing, since filtering console output for strings that point to a software fault is not usually completely foolproof. It might also be that the input vector tested is actually robust and another file format or input method should be tested instead.

It should be noted that fuzzing does not give any guarantees of test assessment (e.g. how good testing coverage was achieved), so in that regard it is not nor should be the only method used to test SUT (Gerlich, Gerlich, & Boll, 2007); on the other hand there are results that it could provide sound reliability estimates (Duran & Ntafos, 1984). Recently there have also been developments that aim to increase the code coverage of black-box fuzzing by automating the process of grammar generation (Kim, Cha, & Bae, 2013).

Fuzzing – and other randomized testing techniques – can also be used as the first barrier of entry before engaging in more rigorous and formal testing methods such as model checking (Groce, Holzmann, & Joshi, 2007).

2.9.2 Using Fuzzing as a Metric For Software Quality

Although the research of this perspective of fuzzing is practically non-existent, apart from short comment by Godefroid and Molnar (2010), it is an interesting possibility to mention. As mentioned by Hamlet (1994), the random testing has the useful property that the test points are – at least on principle – statistically independent, ergo a statistical prediction can be made based on the observed results. As the software bugs are the bane of the industry (Charette, 2005) there is certainly a dire need for robustness metrics. Ideally, this kind of metric could be used as a pointer of component robustness, e.g. an embedded system used to control car brakes could be tested with some number of randomly generated tests and the results would give at least some assurance that the component’s software is fit for real-world use. Similarly, when ordering a software system this kind of metric could be used to set clear-cut robustness minimum for the software provider and the benchmarking could be set up from the day one. It would also help the software provider in the selection of the third-party components and ensuring the quality of the components developed by the subcontractors.

It should be noted though that this metric should not be the only one used as it does not give guarantees of testing coverage and would be very reliant on the competent programmer hypothesis (programmers are usually competent and the programs written are
often close to the correct ones, differing mostly because of simple faults) and the coupling effect (test data that catches simple faults is bound to also catch more complex ones) (Offutt, 1989). Irregardless of whether coupling effect is or is not there this testing metric could still have its place, perhaps as an indicator whether it is reasonable to venture further and measure the software quality using more laborious testing metrics.

2.9.3 Random Testing

There is some research done on software testing technique similar to fuzzing under the term random testing. It could be said that for practical implementations random testing mostly equals fuzzing, but it is reasonable to also cover research done under the name random testing. Additionally some define random testing so that it is questionable if all the flavours of fuzzing can be accommodated under it. For example, Gotlieb and Petit (2006) define random testing as follows: "Random Testing (RT) is the process of selecting test data at random according to an uniform probability distribution over the program’s input domain. Uniform means that every point of a domain has the same probability to be selected." A similar definition is also used by Arcuri in his papers (Arcuri, Iqbal, & Briand, 2010; Arcuri & Briand, 2011). This definition fits the best for the simplest fuzzers where the test cases are just random data. However, for mutation-based fuzzers the test cases are probably skewed to be something that resembles the original sample files and thus the probability distribution to the program’s input domain is not uniform.

As we recall from fuzzing the completely random test cases have a tendency to be less effective than test cases that resemble valid input data. Thus there has been quite some research on techniques to improve the effectiveness of random testing: techniques such as Adaptive Random Testing (T. Chen, Leung, & Mak, 2005), Mirror Adaptive Random Testing (T. Chen, Kuo, Merkel, & Ng, 2004) and Path-Oriented Random Testing (Gotlieb & Petit, 2006) have been developed and deemed to generate more effective test cases than plain random testing; an empirical analysis of the effectiveness of some of the different flavours was conducted by Mayer and Schneckenburger (2006) and they noted that they should be used if the execution of a single test case takes a lot of time. This is notable, since while these methods apparently created effective test cases, the metric that actually interests us is how many bugs can be caught in a given amount of time. Using this metric, these more "sophisticated" variants of random testing fall short as shown by the findings of Arcuri et al. in their research: they proved vanilla random testing to be more effective than previously thought and that adaptive random testing was highly ineffective even in trivial situations (Arcuri et al., 2010; Arcuri & Briand, 2011). This is probably the reason why these methods are seldom seen in everyday use.

One interesting question regarding the results of Arcuri et al. remains: why the methods used to smarten fuzzing actually make the testing more effective when the results are contrary in random testing? There does not seem to be any research on this but it could be postulated that the reason is probably this: the methods such as adaptive random testing use potentially expensive techniques – "distances" between different outputs, oracle output of previous test cases run, real-time branch coverage statistics – to steer the test case generation to output data that fits the envelope they have set. This causes these methods to use more time in the generation phase than it takes to execute the generated test case, and thus causing the generation phase to become a bottleneck in the testing process, leading to less efficient testing. In contrast, the "smarter" fuzzers are usually
still quite efficient in creating test cases, and thus do not become the bottleneck in the testing process.

One interesting thing related to adaptive random testing is that it might be beneficial to try to combine some of the ideas with fuzzing: in adaptive random testing the effectiveness of random testing is improved by trying to select test cases so that they map as evenly as possible to the input space. This is done by creating a pool of randomly generated cases and then selecting from it the cases that advance the evenness best (T. Chen et al., 2005). The idea behind this relied on observation by Chan, Chen, Mak, and Yu (1996) that fault-inducing regions form patterns onto the input space (as visualized in Figure 5); thus running certain number of random test cases should hit these patterns with greater probability if the test cases are scattered uniformly on the input space (T. Chen et al., 2005). In some cases it could be viable to take advantage of this method: create a pool of test cases with a fuzzer and then use the idea behind adaptive random testing to filter a subset of inputs to run against the SUT. One organic method that fuzz tester utilise is to feed earlier fault-inducing test cases as samples for the subsequent runs of fuzzing. These test cases can be found by the tester herself or they can be scavenged from regression test suites. CERT/CC has recently added a mechanism such as this to their fuzzing framework and hypothesise that this is due to better discovery of fault patterns and that the testing is better at mapping out these fault regions because of this mechanism (Householder, 2013).

Vanilla random testing or dumb fuzzing usually provides low code coverage due to the fact that the input is seldom unstructured (Godefroid, 2007).

2.9.4 Differential Testing

Differential testing is a form of random testing which can be used to test mature software systems. The idea of differential testing is to take two or more comparable systems and feed random test cases for them to see if they produce the same output. If the outputs of the systems differ or the SUT hangs while the other implementation does not, the input is flagged as potentially fault-inducing. Differential testing relates to model-based testing in a sense that an alternative implementation can be seen as the model of the SUT and the testing oracle is built based on this assumption. (McKeeman, 1998)
Due to the nature of the testing method, it is apparent that it requires at least one alternative implementation of the same nature that produces same outputs with same inputs. This might not always be easy, especially if the task the software is created to handle can be solved in several different ways. For example, web browsers can render a web page with small differences and both output can be seen as correct. (McKeeman, 1998)

Differential testing is advantageous in situations where there is a high premium on correctness – a good example of this would be astronautical software (Groce et al., 2007) – or where a party is developing a new version of an old, and/or existing software component and wishes to ensure that the versions work as identically as possible. (McKeeman, 1998)

2.9.5 Mutation Testing

Since fuzzing shares a few integral keywords – such as mutating and testing with mutations – with mutation testing or mutation analysis it is appropriate to mention it to avoid confusion between these testing techniques. Mutation testing is a testing method that could be described as testing the "goodness" of the tests. The principle of the mutation testing is to mutate the SUT to see, if the existing test suite can catch these mutations: failing to do so deems the test suite not satisfactory. Usually the changes done to the SUT are quite small, such as changing Boolean operators in conditional statements; it is argued that if these small mistakes are not caught by the test suite it results in bigger problems to be missing as well. (Vigna, Robertson, & Balzarotti, 2004; Clark et al., 2013)

The difference between mutation testing and random testing – or fuzzing – is that the former mutates the SUT while the latter mutates the input vectors of the SUT. Additionally, the mutation testing tests the tests, where random testing tests the SUT. (Smith & Williams, 2009)

2.9.6 Parametrised Unit Testing

The idea behind parametrised unit tests is that instead of writing traditional, "closed" unit tests which take no parameters, such as

```python
def test_add():
    assert(2, SUT.add(1,1))
```

the unit test is parametrised to accommodate testing with arbitrary values. Transforming the previous example into a parametrised unit test could yield something akin to:

```python
def test_add(a, b, result):
    assert(result, SUT.add(a,b))
```

This form of unit tests also allows to define sample bags from the input space of each parameter and to go through them every time a certain type of input is used. Additionally, if fault is found a "traditional" unit test can be extracted from the system simply by instantiating the parameters. (Tillmann, Schulte, & Grieskamp, 2005)
2.9.7 Randomised Unit Testing

Andrews et al. (2011) define randomised unit testing as "unit testing where there is some randomization in the selection of the target method call sequence and/or arguments to the method calls". As it is apparent, the unit test fuzzer done in this thesis fits nicely under that classification. What differs, however, is the approach to this problem: instead of using the existing unit tests as the raw material for new tests, the existing solutions offer tools which are used to create new unit tests and where the developer determines the testing oracle, the functions to be tested and what type of inputs are given to the SUT (Andrews, Haldar, Lei, & Li, 2006; Andrews et al., 2011). One existing solution is also to parse the similar information from SUT components such as the Java class headers (Csallner & Smaragdakis, 2004).

2.9.8 Other Related Work

Although non-academic, it might be interesting to note that Google’s Chromium project has done some degree of integration of fuzzing and a CI system (Chromium, 2013b). In Figure 1 the build step that includes fuzzing the user interface is the last one, "reliability tests". Apparently, this type of fuzzing did not bear too much results, and nowadays Chromium is fuzzed using specialised fuzzing cluster dubbed ClusterFuzz (Y. Chen et al., 2013; Arya & Neckar, 2012). ClusterFuzz is integrated differently: instead of being connected to the continuous integration system, it functions autonomously by regularly fetching the latest good revision of Chromium and fuzzing it in a traditional way. If a bug is found, it is submitted to the bug tracker for further examination and fixing. One interesting point concerning the integration of the CI system and the reliability tests is the fact that the fuzz tests were not detached from the rest of the tests and they could flunk a build if triaging did not identify the freshly found bugs as know issues.

Additionally there has been some research, which compares different flavours of automated unit test generation approaches. For further reading Wang and Offutt (2009) and Singh (2012) can be looked into.
3 Overview of the Software Tool Created

This section describes the software tool created for this thesis. The description begins by going through the third party software components used as the building blocks when creating the tool.

3.1 Existing Software Components Used

Python is a dynamically typed programming language, which seems to have gained adoption both in academic and the corporate world. Some features the official website of the language lists as distinguishing are readable syntax, intuitive object orientation, modularity, high level dynamic types and embeddability. Python seems to offer a good platform for rapid prototyping, and its standard library includes a library for working with abstract syntax trees so it seems to offer the metaprogramming capabilities needed for this thesis. (Python Software Foundation, 2012)

Radamsa is a fuzzer framework developed at the Oulu Secure Programming Group at the University of Oulu. It is classified as black-box fuzzer and it requires minimal information about the SUT, which makes both integrating it into the workflow easier as well as providing better chance that the end result will be easily portable and capable of being used as a general tool with minimal configuration effort. It is available for Linux, Unix, Windows and OS X. (OUSPG, 2012b)

Blab is a tool that generates data according to grammars, also developed at the Oulu Secure Programming Group. In this context, it can be used to generate strings and numeral inputs when such need arises. There is also an option to generate more complex data, such as HTML or JSON, by defining a grammar for it. (OUSPG, 2012a)

Buildbot is a GPL licensed continuous integration tool written in Python. It automates the test cycle, removing the need for human intervention to validate the build and thus offers the software development team valuable information by contributing information about the builds' health. Buildbot is built to match ideology that every software project is different and that the continuous integration tool needs to be flexible and customisable to accommodate differing needs the different projects might have. (Buildbot, 2013c)

3.2 Ideas Behind the Creation of the Tool

The basic idea behind the software tool was to generate unit test cases using existing unit test suites as the floor plan and adding mutations to existing function parameters by utilising existing fault-based testing tools (a fuzzer, Radamsa and a data generation tool, Blab) via python bindings. The test cases are bundled into separate test suites in order to separate between the success state of the “normal” unit tests and the success state of the generated random tests. This is done in order not to interfere with normal software development cycle where emerging unit test failure in previously successful build is seen as a clear sign that the latest change has broken something and must thus be reverted, then inspected and finally fixed. The test suite is then run by the continuous integration server and non-failing test cases are discarded. If a failing test case is found, it is saved
for bug fixing purposes. The ideal solution is seen as a drop-in into the existing test infrastructure needing minimal configuration and working as transparently as possible, since this is seen to make the barrier of entry as small as possible. It is expected that this approach is not going to be as effective as the more dedicated approaches but then again it is probably better than no random testing at all.

There are a few possibilities on how to attach the software tool to the CI framework: the first that comes in mind is to integrate the functionality into the CI tool itself. Unfortunately, this choice has the obvious drawback that the updates for the CI suite become tedious. A more acceptable solution is to create a plugin for the CI suite, if such an option is available. This option seems the most reasonable one, since it is the canonical way to do things similar to this – but due to the architecture of Buildbot there is another option: since the external libraries used in the building process can be built from external repositories, we can use a repository to contain the needed components for the tool. In this approach, the tool is contained in a repository – external or the same where the resources of the SUT are – and the tool is invoked by adding a ShellCommand line to the Buildbot configuration file. This means the tool is easy to try and it is easily disabled if it causes problems. The problem with this approach is that the source repository contains additional files if an external, isolated repository is not used; it may slow down the build process and it adds an additional – although optional – dependency to the build process. Too big hits on performance should be avoided, since as Williams et al. (2009) noted, too cumbersome unit test suite can discourage from using it.

The solution selected for the tool in this thesis is to create a software bundle that is pulled from the repository to the build slaves and the fuzz testing cycle is invoked using Buildbot’s ShellCommand functionality. This approach is easy to implement and it also makes it easy to test the tool in a random toolchain, since very little has to be presumed from the build process.
3.3 Method of Fuzzing the Unit Tests

The working logic of the component that mutates the unit tests is best explained with a diagram, see Figure 6.

![Diagram depicting the process used to transform the vanilla unit tests into fuzzed unit tests.](image)

**Figure 6**: Diagram depicting the process used to transform the vanilla unit tests into fuzzed unit tests.

Example of the change in the structure of a single unit test – fifth phase in Figure 6 – is best explained using an example, as follows:

Sample test case before transformation...

```python
def unit_test():
    x = SUT.func("foo")
    self.assertEquals("bar", x)
```

...and after transformation:

```python
def fuzzed_unit_test():
    try:
        x = SUT.func("foo#FUZZED")
```
cmp("bar", x)
except(Exception, e):
    self.assertEquals(type(e), StandardException)

3.4 Internal Structure of the Tool

The software created consists of approximately 1000 lines of Python code, the largest part being the general-purpose Python bindings for Radamsa and Blab, taking roughly 600 lines of code. The internal structure of the software tool is quite simple due to the choice of trying to build minimum viable solution first; after the initial testing and taking a glimpse of some real-world unit test suites of bigger projects, the development was halted due to issues described later. The architecture of the tool mimics a pipeline, where an object dubbed UnitTestDataContainer is created for each unit test file. The objects representing unit test files and stored into a list. The internal structure of the object is following:

class UnitTestDataContainer(object):
    def __init__(self):
        self.in_path
        self.uuid
        self.source_code
        self.ast
        self.fuzzed_ast
        self.fuzzed_source
        self.out_path
        self.out_filename

A more verbose description of the pipeline, roughly describing the working principles of the tool in pseudocode follows:

- Expand a path into a list of filenames.
- Map to the list of filenames a function that creates UnitTestDataContainer for each of the files to the list, generates a UUID and sets the source path.
- Map to the list of objects a function that reads in the source code of the unit test, parses it into an AST and sets a reference to it into the object.
- Map to the list of objects a function that creates a deep copy of the AST and sets a reference of it to the fuzzed_ast parameter.
- Map to the list of objects a function that transforms the unit tests in the copied AST into the desired form.
  - The function walks the AST and searches for function definitions.
  - When a function definition is found and it is determined to be a definition of a unit test, its structure is transformed to the form previously described.
  - The AST subtree is walked to locate function calls.
When a function call is found it is ensured that it is not a call to one of the built-in functions of Python or that it is not an assert.

Parameters of function calls are investigated to determine their type. If appropriate type is found (in this case, a number or a string) it is mutated using the mutation tools provided by the Python bindings.

If the function call was an assert, it is replaced with Python built-in function \( \text{cmp}(x, y) \) in order to disable the actual assert but to force evaluation of the parameters in case the original assert of being similar to `assert("bar", SUT.foo("baz"))`.

- Map to the list of objects a function that unparses the mutated AST back to source code form and stores a reference to it into the object.
- Map to the list of objects a function that generates an output path for each of the mutated unit tests and stores it into the object.
- Map to the list of objects a function that writes the mutated source code to the designated output path.

There are some apparent problems with this approach: for one, the variables used are not tracked. In practice, this means that if the function call uses a variable as one of the parameters, the parameter type checking becomes tedious. Naturally there are solutions to this problem. For example, while walking the AST, the type information can be stored each time a value is assigned to a variable. Another, more brute-force approach would be to walk the AST again to locate the assignment of the variable; probably not a good way to achieve this functionality, but a possibility nevertheless. During the development problems arose from the fact that AST does not offer possibility to walk the tree upwards. For example, if given a random node, the only option to locate parent of that node is to walk the AST from the beginning, comparing the node until a match is found. In general, the AST implementation offered by the Python standard library is obviously meant primarily for the compiler or the interpreter and is inadequately documented and seemingly lacking in metaprogramming capabilities. In hindsight, it might be better long-term solution to either write a custom AST implementation or to use a graph as the data structure to depict the source code.

### 3.5 Restrictions Imposed by the Tool

Ideally the restrictions with a tool like this would be minimal and it would just work regardless of the SUT. Unfortunately, this is not possible, or at least it requires too much effort in the scope of this thesis, considering the goal is to test the feasibility of an idea. Secondly, much of the idea of the tool is that it is a drop-in component, requiring minimal configuration; should the tool need extensive configuration before it works, it is probably better to invest that effort in building up proper fuzzing toolchain, since much of the gain from the easy explorability is lost.

Another difficulty is that Python enables to customise the unit testing extensively due to the languages dynamic nature. Many of the projects – especially bigger ones – have opted in for this feature. Reasons to make modifications can range from trying to get
different unit test suites in a multi-language project to function better together to trying to get the unit tests to follow the project-specific coding conventions. This leads to situations where the parsing of the unit tests becomes too complex and we cannot presume to get at least some cases right using the fairly simple logic in transforming the unit tests. Naturally, we could try to build more elegant constructions, but weighing the gains against the risk of stumbling into the territory of higher-order grammars and crashing into the Entscheidungsproblem – since the unit tests are written in full-fledged, unrestrained Turing complete programming language – it is hardly worth the effort; especially noting that larger projects with more resources available have a better chance in engaging "proper" fuzz testing.
4 Results

In this section the results found while building and testing the tool are discussed. The primary findings: firstly, the target language selected (Python) is not optimal as a target language for this type of tool due to the flexibility of the language; programming language which requires more rigidity – e.g. type information, function visibility information – from the developer might dampen these problems, but some fundamental issues might still stand, such as the oracle problem and the tendency of developers to make their testing too "smart" to be exploited by a tool exhibited in this thesis. Secondly, if fuzz tests are generated – perhaps manually – the continuous integration systems seem to be flexible enough to make the integration of the fuzz test suites possible. Only Buildbot was examined, but there does not seem to be a reason why this would not apply to most CI solutions as well due to the nature of the tools.

We begin by looking into the issues related to the usage of Python.

4.1 Flexibility of Python

One clear issue posed is the dynamic nature of Python language: compared with statically typed and more "conservative" languages, tracking of the object types the variables represent and what is happening in the program flow is much more tedious. In Java, for example, one can backtrack quite easily what is the type information of a variable, since it is explicitly pointed out in the source code. On the other hand, in Python the following might be possible:

```python
json = "A string"
print(type(json))
print(json)
import json
print(type(json))
```

The output of this code is:

```
<type 'str'>
A string
<type 'module'>
```

So, in the worst case it might be impossible to determine the type information of the object the variable points to without evaluating the code. A far-fetched example of this might be:

```python
import random

json = "A string"
if random.choice([True, False]):
    import json
print(json)
```
Success typing (Lindahl & Sagonas, 2006) – or similar type information sniffing techniques – might help a bit in this situation, but even them are not a foolproof solution, especially when concerning language as flexible as Python.

Unfortunately, it is not impossible for tests to have code paths like these. Sometimes, for example, there are differences in the Python standard libraries that may force the developer to use a third party library as a fallback if there is a need to support a legacy version of Python. In Java things like these are more formally handled, when in Python the flexibility of the language allows the developer to "code herself out" of the situation. As a side effect, this makes the code harder to parse.

If parametrised unit tests (Tillmann et al., 2005) were used, it could be of some help in overcoming these problems:

```python
FOO_IN = "foo"
BAR_OUT = "bar"

def unit_test(foo, bar):
    x = SUT.func(foo)
    self.assertEquals(x, bar)

def run_tests():
    unit_test(FOO_IN, BAR_OUT)
```

As we can see, they would greatly help in determining what the type of the parameters we are aiming to fuzz is, and where in the code they are located by making more of the information explicit. Unfortunately there are several reasons why writing tests in this form might be a bad idea: it makes the tests less readable, adds overhead for the test writers and removes some of the flexibility that might be useful when writing the tests. It is obvious that the test developer is the first class citizen here and should not be forced to change her working habits in order to allow the unit tests to be used for something else than originally meant for.

4.1.1 Difficulty of Pinpointing Testable Functions and Weeding Out Internal Functions

The unfortunate feature of Python in this context is that very little implicit information about the encapsulation is available – as a matter a fact Python offers very thin means to encapsulate code at all (Goldwasser & Letscher, 2008). As Python’s design philosophy seems to be very trusting that the programmer most often does the right thing, it only offers naming convention to designate whether class members are intended to be public or if they are internal. In this scheme, the names of the class members intended for internal use begin with an underscore, additionally members which have their name starting with double underscore are subjected to name mangling which makes it more difficult to use them by accident (but even this does not make accessing them impossible or especially difficult) (Rossum & Warsaw, 2013).

Languages, which require the programmer to explicitly express the accessibility of the code – such as Java and C – fare better in this regard, since the accessibility information
can be used as a hint whether the classes and methods or functions are meant to be internal or public. If those environments were tested, we could take the stance that all the public methods are tested and ones marked protected or private (or similar) are ignored, and it would probably be a good approximation of the intent the programmer had about the usage of the corresponding objects. The biggest obstacle on this regard could probably be methods marked public even though they are not, but in this case it can be presumed that the programmer should be willing to fix the accessibility to make the code more canonical in regard to the coding conventions of the corresponding language — provided the accessibility constructs are built into the language as a real feature meant to be used.

4.2 Development Toolchain Integration

The test suite is expected to work deterministically: the behaviour should be predictable and consistent so that the information is usable to gain information on regressions and the status of the build, whether the latest build passes the designated unit test suites or not. This contradicts with the idea of random testing, since random tests might pass or they might fail depending on luck. Thus, the behaviour of the fuzzed unit tests needs to be isolated from the static unit tests so that the bugs found are stored somewhere for inspection when the software development cycle allows it and they do not interfere with the normal development process. Usually, the problem should be solvable by designating a new test suite for the fuzz tests and configuring the system so that failed tests in that suite do not flag the build broken. However, there are potential risks if there are external dependencies to the SUT; namely there might be, for example, a database that is used which might be affected by a test case. Naturally the build system itself might contain bugs; on the other hand, these bugs might emerge during normal unit testing as well.

Another problem is the isolation of the system when performing the tests. Usually it is a good practice to isolate the machine running fuzz tests from communicating with other machines, since there can be incidents of the system becoming unstable and performing weirdly. For example, the fuzzer might flip a byte from an URL causing the system to send a single malformed request or a bombardment of them to an unsuspecting, random machine in the network. The isolation might be problematic, since the same build machines might be used by other projects.

4.3 Lack of Uniformity of Unit Testing

Nowadays it seems that more and more often different programming languages include a unit testing library with their standard distribution. From the viewpoint of this thesis, it is also unfortunately true that many projects exist to improve the capabilities of the vanilla unit testing framework. In the Python ecosystem, for example, there a few mainstream alternatives to the standard library exist: pytest and nose. Additionally Python standard distribution includes a library called doctest, which allows the developer to write unit tests straight into the documentation of the source code. Big projects might also have some customizations of their own, such as adjusting the unit tests to match the coding conventions of the project.

Another problem is that the unit tests can be performed in different ways, even if using
the default unit testing library. For example: it is possible that instead of sending the test
data in the function call, the unit test only supplies a file path or file handle to the SUT.
Naturally the file path can be fuzzed, but it probably only tests the standard library’s
ability to handle improper file paths instead of mutating the actual data that the SUT
reads from the file.

The lack of uniformity is somewhat expected, since there is a collision of interest between
developers of tools like this and the developers and testers writing the unit tests. It is
obvious that the testing of the SUT should be done as smartly as possible to minimize the
burden of testing. For example, it is preferable that the same script can be run regardless
of the underlying operating system. This means that instead of using just vanilla unit
tests, extra layer of scripts are written which activate or deactivate certain parts of the
unit test suite depending on the system, which the SUT is executed on. This leads into
the situation where the test developer will gladly utilize the flexibility the language in
use has to offer in order to reach her goal. This usually means that – from the viewpoint
of this thesis – the goal of automated unit testing becomes more difficult to achieve.

4.4 Difficulty of Conceiving a Test Oracle

It is imperative that it is possible to conceive an automated testing oracle, since the test-
ing needs to be automated for this testing method to be feasible (Andrews et al., 2011). At
a first glance it would seem easy to determine it since the unit tests have an in-built mech-
anism for it: the asserts. The problem arises when the tests are fuzzed and their structure
changed: the assert is no longer the test oracle, since the input might have been changed.
Similarly the goal of the test changes drastically: we are no longer (usually) testing that
the SUT behaves properly with a proper input but that it can handle improper input
properly, i.e. it does not crash or return improper values, e.g. authentication function
returning success when using abnormally long gibberish as the password.

Here might lie the actual caveat emptor of this method: when the programmer writes
the unit tests she determines by hand the testing oracle for the – usually – positive test
case, but not for the negative one. Additionally, even if some negative test cases have
been created and they could be used as a basis for the testing oracle of the negative test
cases, there is no easy or infallible way of distinguishing and extracting those testing
oracles from the testing oracles written for the positive test cases.

Thus, we need to come up with a different solution as our automated test oracle. In this
thesis, the solution selected is to wrap the mutated code in try-catch -structure to trap
certain flavours of exceptions and errors and determine them as a failed test. It is easy to
see that this method is far from satisfactory, but unfortunately alternative one-size-fits-
all options are even more lacking. The biggest problem for the selected method is the test
cases, which use assert that ensures that the SUT throws a certain flavour of an exception.
The following example is taken from the unit test suite of the Python language:

```python
self.assertRaises(MemoryError, list, xrange(sys.maxint // 2))
```

If we are to use the method earlier described and MemoryError is one of the triggering
error flavours, our only option is to omit the test cases where the method tested is men-
tioned in order to avoid false positives, since that error is deemed as correct behaviour for
that method and thus we should avoid reporting those incidents as faults. False positives should be kept at minimum in general because assigning false positives for developers as bugs-to-be-fixed is highly counterproductive (Smaragdakis & Csallner, 2007).

4.5 Bug Triaging

It is a known property of fuzzers that they find bugs indiscriminately and that they can find some bugs are more often than others. Unfortunately due to these properties triaging fuzzer bugs is not always an easy task: if a fuzzer is left running overnight, there might be thousands of faulting test cases waiting in the morning where almost all of them trigger one or two common bugs. Additionally amongst them might be one test case that triggers a critical bug. At this point it is necessary to triage the bugs to different categories to find the needle in the haystack; unfortunately doing triaging by hand can be laborious and by that, costly. (Y. Chen et al., 2013)

Fortunately there are some options for this problem: usually when fuzzing output from tools such as AddressSanitizer (Serebryany et al., 2012) can be exploited to triage bugs; additionally techniques such as delta debugging (Zeller & Hildebrandt, 2002; Y. Chen et al., 2013) can be used to minimize the test case and to help in the grouping of similar kinds of bugs.

The first thing that needs to be considered concerning triaging problems in context of fuzzing unit tests is that if the fuzz tests are not properly isolated from the normal unit tests it can cause large amounts of confusion, if the fuzz tests break the build. For example, if the build health is tracked via email alerts there is a risk that developers’ inboxes are flooded with reports on failed fuzz tests, and this is naturally not a desirable situation.

Additionally, the triaging problem could be very real when going through the collection of failed test cases that have been saved during the fuzz testing: since they are in unit test format, the methods called in the unit test can point in the right direction to hunt the bugs, but on the other hand it might make the triaging more difficult since "traditional" bug triaging tools might not be viable due to the unusual format of the test cases. Nevertheless, existence of proper automated triaging is imperative, since it is not viable for developers to start going through large amounts of failing unit tests due to the realities of software development; development time costs money and bug triaging can be a surmountable time sink. This problem also emerges with other testing techniques such as static analyzers, where the sensitivity of the tools has been forced to be lowered (Berger, 2012).

The bug triaging problem is connected with the oracle problem: if the testing oracle is partial, the triaging tools need to become part of the oracle to build a suitable solution to filter and sort the potential issues found so that the developer time is used as efficiently as possible.

4.6 Integrating Fuzzing Into the Testing Toolchain

While the automated unit test fuzzing was difficult, this task was actually quite easy. As it appears, the Buildbot project itself has a simple fuzzing test suite (Buildbot, 2013b).
They have opted to use a separate test suite for the fuzz tests to isolate them from the other tests, which seems like a reasonable solution for that problem. So, the fuzz tests can be isolated from the other tests. Test suites can also be configured not to fail the build even though a fault is found during the execution of the test suite. This guarantees that fuzz test suite does not cause false positives in this regard.

Buildbot also seems to have the option to run the fuzz tests for a long period of time, which does improve the effectiveness of the testing since when given more time more fuzzing iterations can be run, improving the probability of finding bugs. In general Buildbot enables a great deal of flexibility on what to do on each of the build steps and how to execute them. Since Buildbot allows its user to run arbitrary scripts – both on build master and build slaves – during the build steps (Buildbot, 2013a), there is very little to hinder the integration of more exotic testing features. Figure 7 presents one possible outline of how the continuous integration framework could be configured.

![Diagram showing one possible workflow of fuzz testing integration.](image)

**Figure 7:** Diagram showing one possible workflow of fuzz testing integration.

There might be need for an additional layer of protection in addition to the isolation provided by using a separate test suite. It is a known phenomenon that fuzzing can cause instability in the underlying operating system and this might pose problems since the instability of the continuous integration infrastructure is probably not desired. For this problem, there is at least one possible solution to solve it: separate build slaves for the fuzz tests that run on dedicated hardware.

All in all, initially it seems that integrating fuzzing into the testing toolchain should be
possible, based on the information outlined here. Naturally, for more concrete results a real-world experiment should be conducted: information presented here is based on proof-of-concept testing with very simple Buildbot configuration.
5 Conclusions

The problem this thesis began to tackle was the integration of modern software development and testing pipeline and fault-based testing, in form of fuzzing. The goal was to find out if fuzzed unit tests could be well-behaving citizens in a software project using continuous integration solution to automate building and testing the software developed and if would be possible to even automate the creation of fuzzed unit tests by using the existing unit test suites as the material for an automated tool, which would do the heavy lifting for us.

The conclusion of this thesis is that it indeed seems to be possible to integrate fuzzing into existing testing infrastructure that is using continuous integration solution. Modern continuous integration frameworks – such as Buildbot investigated in this thesis – allow a great deal of flexibility in general, since requirements posed by different software projects are heterogeneous and require the CI solution to be flexible to adapt the different needs; due to this integrating fuzzed unit tests should not be insurmountable.

Another conclusion is that while it might be plausible to create an automated unit test fuzzer, it is not as easy of a task that one might think at first – at least if the SUT and/or the tests are written in Python – and that for now it is probably better to give the developers tools that make the fuzzing easy, but let them decide on which functions and parameters to use it on and to build the testing oracle that best suits each situation. Additionally it might be useful, if unit tests were written in parametrised form (Tillmann et al., 2005), since that would allow easy instantiation of the fuzzed parameters and the file would be static and self-contained. It should be noted that these results are most relevant when the tests are written in Python and that the restrictions to the tool are in effect, namely that the automated unit test fuzzer should be quite simple. Nevertheless, some open questions, such as identifying public methods, seem difficult to solve programmatically so that the results would be adequate enough.

For future research there are numerous interesting openings available. Probably the most obvious one would be to test the integration of fuzzed unit test in vivo with a real-world software project and to formulate, which kinds of auxiliary tools and software libraries would make writing such tests as effortless as possible. It would also be interesting to find out details which kinds of paths to take to make the testing as practical as possible, such as whether to use all available idle time of the build slaves to run the fuzzed unit tests or if there would be practical or business reasons not to do so – and whether this type of fuzzing would be effective in finding bugs in the first place, optionally compared with the results of more traditional fuzzing approach. Naturally it would also be interesting see if unexpected complications would arise and the ideas to isolate the side effects of fuzzing from the main unit testing would need strengthening to ensure peaceful co-existence.

Another interesting venue for future research could be testing with a language that uses manual memory management and which has explicit visibility and parameter type information baked into the functions. Since manual memory management can subject the software to have bugs that are rare with languages like Python, it could be viable to use tools that catch bugs in memory management to form part of the testing oracle, thus lessening the problems identified with creating one. Additionally the explicit information of the functions could help in pinning down the functions to test and to make determining the type information of the parameters in order to tip the scales enough that automated
fuzzing of unit tests might just become viable enough to be useful.
References


