Transparently Improving Regression Testing Using Symbolic Execution

Paul Dan Marinescu

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College, June 2014
Declaration of Originality

I Paul Dan Marinescu declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Abstract

Software testing is an expensive and time-consuming process, often involving the manual creation of comprehensive regression test suites. Current testing methodologies, however, do not take full advantage of these tests. In this thesis, we present two techniques for amplifying the effect of existing test suites using a lightweight symbolic execution mechanism. We approach the problem from two complementary perspectives: first, we aim to execute the code that was never executed by the regression tests by combining the existing tests, symbolic execution and a set of heuristics based on program analysis. Second, we thoroughly check all sensitive operations (e.g., pointer dereferences) executed by the test suite for errors, and explore additional paths around sensitive operations.

We have implemented these approaches into two tools—KATCH and ZESTI—which we have used to test a large body of open-source code. We have applied KATCH to all the patches written in a combined period of approximately six years for nineteen mature programs from the popular GNU diffutils, GNU binutils and GNU findutils application suites, which are shipped with virtually all UNIX-based distributions. Our results show that KATCH can automatically synthesise inputs that significantly increase the patch coverage achieved by the existing manual test suites, and find bugs at the moment they are introduced.

We have applied ZESTI to three open-source code bases—GNU Coreutils, libdwarf and readelf—where it found 52 previously unknown bugs, many of which are out of reach of standard symbolic execution. Our technique works transparently to the tester, requiring no additional human effort or changes to source code or tests.

Furthermore, we have conducted a systematic empirical study to examine how code and tests co-evolve in six popular open-source systems and assess the applicability of KATCH and ZESTI to other systems.
Contents

Abstract 4

List of Figures 7

List of Tables 10

List of Publications 13

1 Introduction 14

2 Background 20

2.1 Symbolic Execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.2 Regression Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3 Program Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3 Empirical Analysis 27

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.2 COVRIG Infrastructure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.3 Code, Test and Coverage Evolution of Six Popular Codebases . . . . . . 34

3.3.1 Code and Test Evolution . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.3.2 Overall Code Coverage ........................................ 41
3.3.3 Patch Coverage .................................................. 46
3.3.4 Bug analysis ....................................................... 50
3.3.5 Threats to Validity ............................................... 51
3.4 Conclusion .......................................................... 52

4 KATCH ........................................................................ 55

4.1 Overview ............................................................... 55
4.2 KATCH ................................................................. 57
  4.2.1 Patch Preprocessing ............................................. 57
  4.2.2 Seed Selection With Weakest Preconditions ............. 59
  4.2.3 Greedy Exploration with Informed Path Regeneration 64
  4.2.4 Definition Switching ............................................. 68
4.3 Implementation ...................................................... 70
  4.3.1 Patch Preprocessing ............................................. 71
  4.3.2 Input Selection ................................................... 71
  4.3.3 Symbolic Exploration .......................................... 73
  4.3.4 Limitations ....................................................... 75
4.4 Experimental Evaluation .......................................... 76
  4.4.1 Coverage Improvement ........................................ 77
  4.4.2 Bugs Found ..................................................... 81
4.5 Discussion and Conclusion ....................................... 82
5 ZESTI

5.1 Overview ................................................................. 85
5.2 Zero-Effort Symbolic Test Improvement .................................. 86
  5.2.1 Thoroughly Testing Sensitive Operations .......................... 87
  5.2.2 Exploring Additional Paths Around Sensitive Operations ....... 89
  5.2.3 Improving Efficiency by Discarding Test Cases ................. 93
5.3 Implementation ............................................................ 94
5.4 Experimental Evaluation ................................................ 96
  5.4.1 Benchmarks .......................................................... 96
  5.4.2 Bugs Found ......................................................... 98
  5.4.3 Symbolic Bug Checks and Performance Overhead ............... 103
5.5 Conclusion ................................................................. 107

6 Related Work ............................................................... 108
  6.1 Symbolic Execution .................................................... 108
  6.2 Regression Testing ..................................................... 112
  6.3 Other Related Techniques ............................................ 113

7 Conclusion ................................................................. 115
  7.1 Summary of Thesis Achievements .................................... 115
  7.2 Applications ........................................................... 116
List of Figures

1.1 Example showing at a high-level the added benefits of KATCH and ZESTI over regular regression testing. .............................. 15

1.2 KATCH and ZESTI are integrated in the software development life cycle and automatically find bugs and generate inputs that increase program coverage. ......................................................... 16

2.1 Code snippet used to illustrate symbolic execution. .................. 21

2.2 Code snippet and associated interprocedural control flow graph. .... 25

3.1 COVRIG infrastructure. .................................................. 31

3.2 Evolution of executable lines of code. ............................... 40

3.3 Evolution of textual lines of test code. ................................ 41

3.4 Co-evolution of executable and test code. Each increment represents a change. ............................................................. 42

3.5 Evolution of overall line and branch coverage. ..................... 44

3.6 Patch coverage distribution. ........................................... 47

4.1 The main stages of the KATCH patch testing infrastructure. ........ 56

4.2 Execution of the patch preprocessing steps on an example patch adapted from the lighttpd web server. .......................................... 58
4.3 Code snippet with instructions annotated with the minimum distance to the target, computed using only control-flow analysis and control-flow analysis combined with weakest preconditions ........................................ 60

4.4 Execution of weakest preconditions-enabled minimum distance computation for input 50 on the code in Figure 4.3 ......................................................... 63

4.5 Execution of weakest preconditions-enabled minimum distance computation for input 150 on the code in Figure 4.3 ......................................................... 63

4.6 Example based on lighttpd patch 2660 used to illustrate the greedy exploration step. ................................................................. 66

4.7 Example based on lighttpd patch 2660 used to illustrate the informed path regeneration step. ................................................................. 67

4.8 Example from diffutils revision 8739d45f showcasing the need for definition switching. ................................................................. 68

4.9 Configuration file used to test diffutils. The file specifies the repository address, the folders which may contain relevant changes, the programs to test and the libraries required to build the system. ................................. 70

4.10 Example from diffutils revision cd10d2c3 (edited for clarity) where the strcmp model is used to avoid divergence after an informed path regeneration step. Line 1090 contains the patch. ............................................. 75

4.11 Distribution of minimum distances for the 939 targets not covered by the regression suites. ................................................................. 78

4.12 Example showing a bug found by KATCH, introduced in binutils revision 119e7b90. The bug is triggered on line 392. The highlighted lines are part of the patch. ................................................................. 83

4.13 Example showing a bug found by KATCH, introduced in binutils revision b895e9d. ................................................................. 83
5.1 Code snippet showcasing a bug missed by a test suite with 100% code coverage, e.g. x=50, x=100. 

5.2 Algorithm used by ZESTI to explore additional paths. 

5.3 Code snippet showcasing an execution generated by an input $x > 99$, annotated by ZESTI. 

5.4 Code snippet showcasing an execution generated by an input $99 < x \leq 199$, annotated by ZESTI. 

5.5 Probability to find a bug using a randomly picked fraction of an initial test suite of 100 test cases. 

5.6 Number of unique (by line of code) and total checks performed by ZESTI on Coreutils 6.10. 

5.7 ZESTI execution overhead compared to KLEE as an interpreter, when run with distance 0 on the Coreutils regression suite. 

5.8 Probability to find the Coreutils bugs at the minimum distance, relative to the size of a randomly chosen sub-test suite.
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of applications used in our empirical analysis, including number of executable lines of code and number of lines in test files in the last revision analysed.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Revisions used in our study. OK: code compiles and tests complete successfully, TF: some tests fail, TO: tests time out CF: compilation fails, Time: the number of months analysed, Age: the age of the project as of January 2014.</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Number of revisions where the test suite nondeterministically succeeds/-fails, and the maximum, median and average number of lines which are nondeterministically executed in a revision.</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Overall patch coverage bucketed by the size of the patch in ELOC. NP is the number of patches in the bucket and C is their overall coverage. Only patches which add or modify executable code are considered.</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Overall latent patch coverage: the fraction of the lines of code in all patches that are only executed by the regression suite in the next 1, 5 or 10 revisions. The overall patch coverage is listed for comparison.</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>Median coverage and the number of revisions achieving 100% coverage for the revisions containing bug fixes. The overall metrics are included for comparison.</td>
<td>50</td>
</tr>
</tbody>
</table>
4.1 Application suites used to evaluate KATCH along with the number of programs in each of them, the number of patches inspected and the timespan in months. ................................................................. 76

4.2 Number of targets covered by the manual test suite, and the manual test suite plus KATCH. ................................................................. 79

4.3 Number of targets covered by different combinations of heuristics. ....... 81

5.1 Bugs found by ZESTI along with the distance (from the concrete test path) and the depth (from the program start) at which the bug was found. New bugs are in bold. ................................................................. 98

5.2 A one byte corruption at offset 0x1073 in a libdwarf test file, which causes a division by zero. ................................................................. 101

5.3 Bug distance distribution for the bugs found by ZESTI in Coreutils and the number of tests available for each programs. ............................... 105
List of Publications

This thesis is based on work published in the following papers:


Chapter 1

Introduction

Testing currently accounts for a large fraction of the software development life cycle [87] and usually involves writing large numbers of manual tests that exercise various paths, often with the objective of maximising a certain coverage metric such as line or branch coverage. This is a tedious process that requires significant effort and a good understanding of the tested system.

As a result, we are witnessing a sustained research effort directed toward developing automatic techniques for generating high-coverage test suites and detecting software errors [8,15,16,24,27,33,45,90,97], with some of these techniques making their way into commercial and open-source tools such as Coverity, Fortify or KLEE. However, these techniques do not take advantage of the effort that developers expend on creating and updating the manual regression suites, which we believe could significantly enhance and speed up the testing process.

Our key observation is that well-written manual test suites exercise most program features, but often use a limited number of paths and input values, potentially missing corner case scenarios that the testers did not consider. In this thesis, we propose two complementary automatic testing techniques that amplify the effectiveness of a regression test suite. The first, KATCH, aims to create inputs that exercise previously unexecuted
Figure 1.1: Example showing at a high-level the added benefits of KATCH and ZESTI over regular regression testing (RT). KATCH synthesises inputs that execute additional parts of the code, while ZESTI thoroughly checks already executed code. In this example, ZESTI finds that a negative function argument triggers an invalid memory access.

code, thereby improving code coverage, while the second, ZESTI, checks the program paths explored by the regression suite against all possible input values, and also explores additional paths that slightly diverge from the original executions in order to thoroughly check potentially dangerous operations executed by the program. Figure 1.1 depicts these benefits using a simple example. Function \( f \) takes a single integer argument and uses it to index a fixed-sized array. The regression tests have already exercised the common scenario in which the function receives a small positive integer (Figure 1.1 (a)) but have not checked the function behaviour when the argument is outside the bounds of the array. KATCH aims to exercises code not executed previously (Figure 1.1 (b)) and therefore generates a large positive argument for the function, while ZESTI thoroughly checks the already executed code and finds that a negative argument causes an invalid memory access (Figure 1.1 (c)).

We envision both KATCH and ZESTI fully integrated in the software development cycle, similar in spirit to a continuous integration system, as shown in Figure 1.2. When developers create a new program version and add it to the repository, the systems automatically explore paths through the code using dynamic symbolic execution, augmented with several heuristics. The paths explored differ for the two systems: KATCH focuses exclusively on exploring new code, while ZESTI’s goal is to thoroughly test the code already exercised by the existing regression tests. Both approaches offer to the developers
Figure 1.2: KATCH and ZESTI are integrated in the software development life cycle and automatically find bugs and generate inputs that increase program coverage.

a set of test inputs which could be added to the regression suite and a report of any bugs introduced by the patch, accompanied by actual inputs that trigger them.

To be adopted by developers, such systems have to meet several requirements: (1) they have to be easy-to-use, ideally fully automatic; (2) they have to be fast, to encourage developers to run them after every single commit; and (3) they have to demonstrate “value” by finding bugs and generating inputs that cover more code through the patch than existing manual test suites. In this thesis, we provide some promising evidence that such systems could become a reality. In our experiments with KATCH we looked at all the patches written in a combined period of around six years for nineteen applications. KATCH was able to significantly increase the overall patch coverage and find fifteen distinct bugs, while spending only a relatively short amount of time per patch. ZESTI obtained promising results as well, finding two previously unknown bugs in Coreutils (despite these applications having been comprehensively checked before via symbolic execution [13, 16]), forty in libdwarf, and ten in readelf, in a manner completely transparent to developers.
The overall contribution of this thesis is to show that the effectiveness of standard regression tests can automatically be improved using symbolic execution-based techniques. We propose two such techniques and show that they can find bugs that are often out of reach for regression testing or standard symbolic execution alone. From a tester’s perspective, these techniques are automatic and do not require tuning a symbolic execution engine or deciding what symbolic inputs to provide to the program. More exactly:

1) We show through an empirical study the suitability of these techniques by building a system for automatic extraction of static and dynamic software evolution metrics from software repositories.

2) We present a first testing technique that combines symbolic execution with several novel heuristics based on program analysis that effectively exploit the program structure and existing program inputs to reach specific program points, and a system called KATCH, based on the state-of-the-art symbolic execution engine KLEE [16] that implements this technique for C programs. We thoroughly evaluate our technique by applying it on all the patches made to nineteen programs in the widely-used GNU diffutils, GNU binutils and GNU findutils application suites, during a combined period of six years.

3) We introduce a second technique, complementary to KATCH, for reasoning about all possible input values on the paths executed by the regression suite and for thoroughly exploring additional paths around sensitive instructions such as dangerous memory accesses, and demonstrate that this approach works well in practice by implementing it in a prototype named ZESTI (Zero-Effort Symbolic Test Improvement). We have applied ZESTI to several popular open-source applications, including the GNU Coreutils suite, the libdwarf library, and the readelf utility, where it found 52 previously unknown bugs. Furthermore, the inputs generated by ZESTI to reproduce the bugs discovered are almost well-formed, i.e. they differ
only slightly from the inputs included in the regression suite, making it easier for developers to analyse them.

4) We provide a theoretical and empirical analysis of ZESTI’s sensitivity to the quality of the test suite, and discuss how the probability of finding a bug varies with the number of test cases being considered.

The rest of the thesis is structured as follows. Chapter 2 discusses the essential background aspects related to our techniques, introduces symbolic execution, concolic testing and regression testing, and explains the program analysis terminology and notations used throughout the thesis.

Chapter 3 investigates through an empirical study the potential improvements that can be made to regression testing. We analyse the evolution of static and dynamic metrics in six widely used systems, focusing mainly on code coverage and its relationship to codebase evolution and to bugs. In this chapter, we answer seven research questions that aim to validate the assumptions that underlie KATCH and ZESTI. We analyse overall program coverage and its evolution, but also patch coverage and latent patch coverage, and finally correlate coverage with bugs extracted from the systems’ development history. Our main results show that (1) there is a significant amount of code added to our benchmark systems that is never executed by the regression tests and (2) many bugs lie in code that has high coverage. The following two chapters describe two techniques for automatic testing that are motivated by these observations.

Chapter 4 describes KATCH, a technique for automatically testing code patches. KATCH targets patch code which is not executed by the regression tests. To this purpose it uses a combination of symbolic execution and novel heuristics based on program analysis to quickly reach the code of the patch. In the evaluation, we have applied KATCH to all the patches written in a combined period of approximately six years for nineteen mature programs from the GNU diffutils, GNU binutils and GNU findutils utility suites, which are included in virtually all UNIX-based distributions. Our results show that KATCH can
automatically synthesise inputs that significantly increase the patch coverage achieved by the existing manual test suites, and find bugs at the moment they are introduced.

Chapter 5 presents ZESTI, a technique complementary to KATCH. While KATCH mainly aims to improve line coverage, ZESTI amplifies the effect of existing test suites using a lightweight symbolic execution mechanism, which thoroughly checks all sensitive operations (e.g. pointer dereferences) executed by the test suite for errors, and explores additional paths around sensitive operations; ZESTI looks for bugs in already executed code and, if appropriate, tries to exercise it on alternate paths. We have applied ZESTI to three open-source codebases—coreutils, libdwarf and readelf—where it found 52 previously unknown bugs, many of which are out of reach of standard symbolic execution. The technique works transparently to the tester, requiring no additional human effort or changes to source code or tests.

We conclude the thesis with related work, briefly reiterate the thesis achievements and discuss the applicability of the techniques presented in the broader context of the software development lifecycle, examining opportunities for integration into existing methodologies with minimal disruption.
Chapter 2

Background

This chapter introduces the general topics discussed in this thesis and provides definitions for concepts relevant to this work. We start by discussing symbolic execution (§2.1) and a variant that mixes concrete and symbolic execution, then present regression testing (§2.2) and finally introduce several relevant program analysis concepts (§2.3). Related work is discussed in detail in Chapter 6.

2.1 Symbolic Execution

Symbolic execution is a technique for program testing and verification that was first proposed several decades ago [52]. Driven by increased available computational power and advances in constraint solving [25, 31], the technique received much attention in recent years [17] due to its ability to systematically and exhaustively explore program paths and reason about the program’s behaviour along each of them. Tools such as KLEE [16], SAGE [36], JPF-SE [3], BitBlaze [82] and Pex [88] are just some of the dynamic symbolic execution engines currently used successfully in academia and in industry.
1 if (input < 100)
2   f(input+100);
3
4 if (input > 200)
5   f(input)
6
7 void f(int idx) {
8   char x[999]
9   if (idx < 999)
10      x[idx] = 0;
11 }

Figure 2.1: Code snippet used to illustrate symbolic execution.

Intuitively, symbolic execution (SE) works by systematically exploring all possible program executions and dynamically checking the safety of dangerous operations. SE replaces regular program inputs with *symbolic* variables that initially represent any possible value. Whenever the program executes a conditional branch instruction that depends on symbolic data, the possibility of following each branch is analysed and execution is forked for each *feasible branch*. To enable this analysis, symbolic execution maintains for each execution path a set of conditions which characterise the class of inputs that drive program execution along that path. At any time, the *path conditions* can be solved to provide a concrete input that exercises that path natively, making it easy to reproduce, report and analyse an execution of interest. SE also analyses all potentially dangerous operations as they are executed, verifying their safety for any input from the current input class. For example, reading from an array is safe on a certain execution path if and only if the array index used can be proven to be positive and smaller than the array size, given the current path conditions.

For example, during symbolic execution of the contrived code snippet in Figure 2.1, execution will be split into two paths at the branch on line 1: one following the *then* side of the branch, on which we add the constraint that *input* < 100, and one following the implicit *else* side of the branch, on which we add the constraint that *input* ≥ 100. When the path with the constraint *input* < 100 reaches line 4, only the *else* side is feasible, so no other path is spawned at this point. On the other hand, when the path with the
constraint \( \text{input} \geq 100 \) reaches line 4, both sides are feasible, so execution is again split into two paths, one on which we add the constraint that \( \text{input} > 200 \), and one on which we add the constraint that \( \text{input} \leq 200 \). Function \( f \) is hence called in two contexts: first from line 2 when \( \text{input} < 100 \), and second from line 5 when \( \text{input} > 200 \). After encountering the conditional at line 9, line 10 is reached with path constraints \( \text{input} < 100 \land \text{input}+100 < 999 \) and \( \text{input} > 200 \land \text{input} < 999 \) respectively. Finally, symbolic execution checks whether the array indexing at line 10 can access invalid memory. It does this by using a constraint solver to verify that the current path condition entails the in-bounds access condition, i.e. whether \( \text{input} < 100 \land \text{input}+100 < 999 \Rightarrow \text{input}+100 \geq 0 \land \text{input}+100 < 999 \) and \( \text{input} > 200 \land \text{input} < 999 \Rightarrow \text{input} \geq 0 \land \text{input} < 999 \) respectively. It can be seen that the implication is not valid in the former case, being false for an input such as -101. Symbolic execution has thus found a fault and moreover, can provide an actual input which triggers it.

Unfortunately, the number of execution paths in real programs often increases exponentially with the number of branches in the code, which may lead symbolic execution to miss important program paths. One solution is to employ sound program analysis techniques to reduce the complexity of the exploration [5, 11, 34, 54]. An orthogonal solution is to limit or prioritise the symbolic program exploration using different heuristics. For example, directed symbolic execution methods [7, 21] use techniques such as static analysis to find instructions or paths of interest which are then used to guide the symbolic exploration. Chapter 6 discusses these techniques in more details.

Concolic execution [35, 77] is a variant of symbolic execution, which involves instrumenting and running a program natively rather than building a full-fledged symbolic interpreter. Concolic executors start from the path executed by a concrete input, which they capture as an execution trace. Different program paths are then systematically explored by flipping the truth value of the branch conditions collected in the trace. These new sets of constraints are passed to a constraint solver which checks their feasibility and returns a satisfying assignment that acts as an input to the program in a new execution.
For the same example from Figure 2.1 and the initial input 1000, concolic execution starts by running the program using this input. This leads to the execution of lines 1, 4, 5 and 9. During execution, instrumentation embedded in the program gathers the safety conditions for all memory operations and their associated path condition, and the path conditions corresponding to branches not taken, i.e. \( \text{input} < 100 \) for line 1, \( \text{input} \geq 100 \land \text{input} \leq 200 \) for line 4, and \( \text{input} \geq 100 \land \text{input} > 200 \land \text{input} < 999 \) for line 9. Solving these constraints yields inputs which execute different paths through the program: 0, 101, 201 for example. The process then repeats for each of these inputs until no further unexplored feasible paths are found. When running the program on input 0, concolic execution finds the fault on line 10, by determining that the path constraints are not sufficient to guarantee the safety condition for the memory access, similarly to symbolic execution.

Our approaches follow the concolic paradigm, starting from the manually written regression tests. This leverages the knowledge incorporated by the developers in the test suite and inherently prioritises the exploration of paths from program features exercised by the regression tests, motivated by two observations: (a) well-written manual tests exercise the most important features for the correct functionality of the system and (b) testers and developers generally test for the common scenario but sometimes fail to consider corner cases.

### 2.2 Regression Testing

Practice has shown that as software evolves, emergence of new faults or reemergence of previously fixed faults is quite common. This may happen because of fragile code, unexpected interactions between different program features or simple human error. To mitigate this problem, good coding practice warrants recording a test case whenever a bug is found and fixed or when a new program feature is introduced. Together, all these
test cases create a safety net, which is the foundation of regression testing, one of the most popular testing techniques today.

At a minimum, regression testing involves maintaining the test suite consisting of program inputs and their expected outputs, and using it to check that no regressions have been introduced by a program change. Almost always a test driver such as automake\(^1\), dejagnu\(^2\) or LLVM lit\(^3\), automates this process by providing each of the inputs, in turn, to the subject program, monitoring its execution and checking its output against the corresponding expected output. A tester then only needs to invoke this driver using a command such as make check to run all the tests and obtain a report.

Both the advantages and disadvantages of regression testing stem from the manual effort involved in writing the tests. On the one hand, developers have a thorough knowledge of the system and are in the best position to provide inputs which exercise specific features, fragile or complex code. On the other hand, developers may miss corner case inputs, and do not usually test the code systematically since writing tests requires a significant amount of time.

### 2.3 Program Analysis

Both KATCH and ZESTI use several static and dynamic analysis techniques for prioritising the most promising paths during symbolic execution. This section contains background information and definitions useful to make their presentation self-contained. We defer the details of the actual analyses until later, in the context of the complete techniques.

A basic block is a maximal set of contiguous program instructions with only one entry point and only one exit point. In consequence, whenever the first instruction in a basic block is executed, the rest of the instructions are executed exactly once, in order.

\(^1\)http://www.gnu.org/software/automake/
\(^2\)http://www.gnu.org/software/dejagnu/
\(^3\)http://llvm.org/docs/CommandGuide/lit.html
A control flow graph (CFG) is a representation, using graph notation, of a program. The nodes of a CFG are the program’s basic blocks while directed edges represent jumps in the control flow. The control flow graph representation is convenient for many analyses such as reachability and domination.

A call graph is a directed graph which captures the calling relationship between procedures of a program. Each node corresponds to a procedure and each edge \((p_1, p_2)\) indicates that procedure \(p_1\) may call procedure \(p_2\).

An interprocedural control flow graph (iCFG) is a combination of a program’s call graph and the control flow graph of each procedure. Specifically, each call graph node is replaced with the control flow graph corresponding to its associated procedure, all edges pointing to it are redirected to the procedure’s entry basic block and all edges pointing out of it are assigned to the procedure’s basic blocks which contain the actual function calls. Figure 2.2 shows a simple code snippet and its associated iCFG. For simplicity of presentation, we consider all procedures to have a single exit block and we omit return edges which, as we will show, are not relevant to our analyses.

A Hoare triple is a way of describing how a computation changes the state of a program. It has the form \(\{P\}S\{Q\}\), where \(P\) and \(Q\) are predicates and \(S\) is a (possibly compound) statement. \(P\) and \(Q\) are called the precondition and the postcondition, respectively. The
meaning of a Hoare triple is that when the precondition is met, the statement establishes the postcondition.

Given a set of statements $S$, the \textit{weakest precondition} associated with $S$ is a function from postconditions to preconditions. The weakest precondition, denoted $wp(S, R)$ is (1) a precondition, meaning that if $wp(S, R)$ holds before executing $S$, then $S$ terminates and $R$ holds after executing it, and (2) the \textit{weakest} such precondition, meaning that for any other precondition $Q$, $Q \rightarrow wp(S, R)$. The special case $wp(S, true)$ is the weakest precondition for termination of the block $S$.

A variable \textit{definition} is a statement that assigns a value to the memory area associated with the variable. A \textit{reaching definition} relative to a program point $P$ and a variable $V$ is a definition of $V$ such that (1) there exists a path from the definition to $P$ and (2) $V$ is not redefined along this path. A \textit{reaching definitions analysis} is a program analysis technique which finds all reaching definitions for a given program point and variable.
Chapter 3

Empirical Analysis

3.1 Introduction

In this chapter we set out to explore the evolution of six popular software systems with a rich development history in order to assess the applicability of KATCH and ZESTI. In the process, we build a framework for software analytics capable of extracting and analysing static and dynamic metrics from a software repository.

Software repositories provide detailed information about the design and evolution of software systems. While there is a large body of work on mining software repositories—including a dedicated conference on the topic, the Working Conference on Mining Software Repositories (MSR)—past work has focused almost exclusively on static metrics, i.e. which can be collected without running the evolving software, e.g. number of lines of code, code complexity [73], or supplementary bug fixes [70].

We suspect that the main reason behind the scarcity of studies focusing on dynamic metrics lies in the difficulty of running multiple software versions,\(^1\) especially since doing so involves evolving dependencies and unstable (including non-compilable) versions. For example, prior work [101] cites the manual effort and the long time needed to run different

\(^1\)In this thesis, we use the terms version and revision interchangeably.
revisions as the reason for reporting dynamic measurements for only a small number of versions.

While static metrics can provide useful insights into the construction and evolution of software, there are many software engineering aspects which require information about software executions. For example, the research community has invested a lot of effort in designing techniques for improving the testing of software patches, ranging from test suite prioritisation and selection algorithms [40, 75, 84] to program analysis techniques for test case generation and bug finding [7, 10, 58, 71, 72, 86, 96] to methods for surviving errors introduced by patches at runtime [46].

Many of these techniques depend on the existence of a manual test suite, sometimes requiring the availability of a test exercising the patch [61, 89], sometimes making assumptions about the stability of program coverage or external behaviour over time [46, 74], other times using it as a starting point for exploration, as it is the case of KATCH (Chapter 4), ZESTI (Chapter 5) and others [36, 50, 95], and often times employing it as a baseline for comparison [16, 23, 28, 69].

However, despite the key role that test suites play in software testing, it is surprising how few empirical studies one can find in the research literature regarding the co-evolution of test suites and code and their impact on the execution of real systems.

In this chapter, we present COVRIG\textsuperscript{2}—an infrastructure for mining software repositories, which makes it easy to extract both static and dynamic metrics. COVRIG makes use of lightweight virtual machine technology to run each version of a software application in isolation, on a large number of local or cloud machines. We use COVRIG to conduct an empirical study on program evolution, in terms of code, tests and coverage. More precisely, we have analysed the evolution of three popular software systems with a rich development history over a combined period of twelve years, with the goal of answering the following research questions relevant to the applicability of KATCH and ZESTI:

\textsuperscript{2}The name emphasises one of the core aspects of the framework, its ability to measure coverage. COVRIG also means \emph{baget} in Romanian.
RQ1: Do executable and test code evolve in sync? Are coding and testing continuous, closely linked activities? Or are periods of intense development alternating with periods of testing? While KATCH and ZESTI can work with stale test suites, efficiently checking new program features benefits from the existence of adequate tests.

RQ2: Is code coverage deterministic? Does running the test suite multiple times cover the same lines of code? If not, how many lines are nondeterministically covered on average?

RQ3: How does the overall code coverage evolve? Is it stable over time? Does the overall coverage increase steadily over time, or does it remain constant? Are there revisions that significantly increase or decrease coverage? KATCH and ZESTI leverage existing regression tests, hence their results depend on the quality of these tests. A low coverage regression suite may not contain sufficient information to efficiently guide symbolic execution.

RQ4: What is the distribution of patch coverage across revisions? What fraction of a patch is covered by the regression test suite? Does patch coverage depend on the size of the patch? In particular, these questions seek to find whether KATCH can be beneficial in the testing process as a result of low patch coverage in the manual tests.

RQ5: What fraction of patch code is tested within a few revisions after it is added, i.e. what is the latent patch coverage? Are tests exercising recent patches added shortly after the patch was submitted? If so, how significant is this latent patch coverage?

RQ6: Are bug fixes better covered than other types of patches? Are most fixes thoroughly exercised by the regression suite? How many fixes are entirely executed?

RQ7: Is the coverage of buggy code less than average? Is code that contains bugs exercised less than other changes? Is coverage a reasonable indicator of code quality? These questions, along with RQ6, aims at providing more information about
bug fixes, which are a prime candidate for testing, and explores the opportunities that KATCH and ZESTI have to improve upon regression tests.

Our empirical results show that KATCH and ZESTI have the potential to improve the testing process. The systems analysed have an adequate number of tests for our techniques to leverage, have moderate to low patch coverage offering opportunities for improvement and, in roughly half of the cases, faulty code was fully executed by the existing tests without causing a failure, making symbolic execution a perfect fit for its ability to thoroughly check a program’s execution of regression tests.

In addition to this study, we present an infrastructure called COVRIG for dynamic (and static) software repository mining, which makes it easy to mine software repositories for dynamic metrics. At its core, COVRIG makes use of lightweight virtual machine technology to run each versions of a software application in isolation, on a large number of local or cloud machines. While the design of COVRIG was driven by our particular study, we believe it could be easily adapted to other tasks involving mining software repositories for dynamic information.

### 3.2 COVRIG Infrastructure

The overall architecture of the COVRIG infrastructure is depicted in Figure 3.1. It contains a generic driver which iterates through all the revisions in a given range and invokes routines specific to each system to compile, run, and collect statistics of interest.

**Lightweight software containers.** COVRIG employs software containers [81], an operating system-level virtualisation mechanism that provides the ability to run multiple isolated virtual Linux systems (“containers”) atop a single host OS. When launched,
COVRIG starts by loading the selected range of revisions from the project’s Git repository, and for each revision starts a new software container. The use of containers offers increased isolation and reproducibility guarantees by providing a consistent environment in which to run each software revision and ensuring that different revisions do not interfere with each other, e.g. by inadvertently leaving behind lock files or not properly freeing up resources.

The choice of lightweight OS-level virtualisation rather than more traditional virtual machines (e.g. KVM\(^3\) or Xen\(^4\)) reduces the performance penalty associated with spawning and tearing down VMs, operations performed for each revision analysed. To get a sense of this difference, we compared an LXC\(^5\) container, which required under a second for these operations, with a Xen VM, which needed over a minute.

In our implementation, we use Docker\(^6\) to create and manage the lower-level LXC

\(^{3}\)http://www.linux-kvm.org/
\(^{4}\)http://www.xenproject.org/
\(^{5}\)http://linuxcontainers.org/
\(^{6}\)https://www.docker.io/
containers, and deploy them on multiple local or cloud machines. Each container is
used to configure, compile and test one program revision, as well as collect the metrics of
interest, such as code size and coverage. The containers are remotely controlled through
SSH using the Fabric\textsuperscript{7} framework.

**Configuration file.** COVRIG has a modular architecture, which makes it possible to
analyse new systems with modest effort. A potential user of our infrastructure only needs
to provide a Python configuration file describing the system. A minimal file provides
the name of the system, its Git repository location, a method to compile the system,
e.g. install dependencies and run the appropriate `make` command, and a method to run
the regression tests, e.g. run the `make test` command. Finally, the configuration file can
also specify an *end revision* and a specific number of revisions to analyse. For accurate
test suite size measurements, the files or folders which make up the test suite can also
be indicated.

For each revision, COVRIG collects several static and dynamic metrics. The static metrics
are obtained either directly from the version control system (e.g. the number of lines of
test code) or after compiling each revision (e.g. the number of executable lines of code).
The dynamic metrics require running the regression tests (e.g. the overall line coverage
or the regression test success status). Further information and graphs—including the
ones presented in our empirical study—are automatically derived in the post-processing
stage from these primary metrics using a set of scripts.

**Bug data.** One possible application of COVRIG is finding useful data about software
bugs and correlating them with the static and dynamic metrics collected. For our study,
we mined bug data from both software repositories and, where available, bug tracking
systems. We automatically obtained a list of candidate bug-fixing revisions by iterating
through the list of commits and checking the commit message for words such as `fix`, `bug`
or `issue`, followed by a number representing the bug identifier. For example, a typical
Memcached bug fix commit message looks like “Issue 224 - check return of main event

\textsuperscript{7}http://fabfile.org/
The regular expression that we used to identify these commits is similar to the ones used in prior work [42]: `(?::bug|issue|fix|resolve|close)\s*#?\s?\d+`.

We confirmed that the bug identifier is valid by querying the associated bug tracking system and we further manually checked all reported revisions and confirmed that they included no false positives. While it is impossible to quantify the false negative rate without a knowledgeable developer manually checking all the revisions in a repository, we believe that the automatically obtained bug fixes create a representative subset of the fixes in the repository.

**Line mapping.** The ability to track how lines move and change across revisions is the cornerstone of many high-level software evolution analyses. A line mapping algorithm improves over the traditional `diff` algorithm by tracking the movement of individual lines rather than hunks. Conceptually, line mapping is a function which takes two revisions, $r_1$ and $r_2$, and a program location described by a pair `(file name 1, line number 1)` associated with $r_1$. The output is a pair `(file name 2, line number 2)` identifying the corresponding location in $r_2$.

COVRIG uses an external implementation of the line mapping algorithm, similar to the algorithms described in previous work [18, 51, 80, 91]. It makes use of the Levenshtein edit distance [57] to track line edits, and `tf-idf` [83] and cosine similarity [79] to track line movements. It also uses the Hungarian algorithm [53] to find the optimal matching of lines across versions.

In our study, we used line mapping to determine whether patches are tested within the next few revisions after they were created (§3.3.3).

**Cloud deployment.** To enable large-scale data collection and processing, we deployed COVRIG to our private cloud. We have built our system around a standard set of tools: Packer\(^8\) for building custom Docker-enabled machine images, Vagrant\(^9\) for controlling and

\(^8\)http://www.packer.io/
\(^9\)http://www.vagrantup.com/
provisioning the virtual machines based on these images, a Docker registry for serving COVRIG’s Docker containers and a \textit{fabfile} for orchestrating the entire cluster. The same set of tools and scripts can be used to deploy COVRIG to different private or public clouds.

### 3.3 Code, Test and Coverage Evolution of Six Popular Codebases

We used the COVRIG infrastructure to understand the evolution of six popular open-source applications written in C/C++, over a combined period of twelve years. Our empirical study has been successfully validated by the ISSTA 2014 artifact evaluation committee, and received the best artifact award. The six evaluated applications are:

1. **GNU Binutils\(^{10}\)** is a set of utilities for inspecting and modifying object files, libraries and binary programs. We selected for analysis the twelve utilities from the \texttt{binutils} folder (\texttt{addr2line, ar, cxxfilt, elfedit, nm, objcopy, objdump, ranlib, readelf, size, strings} and \texttt{strip}), which are standard user-level programs in many UNIX distributions.

2. **Git\(^{11}\)** is one the most popular distributed version control systems used by the open-source developer community.

3. **Lighttpd\(^{12}\)** is a lightweight web server used by several high-traffic websites such as Wikipedia and YouTube. We examined version 2, which is the latest development branch.

4. **Memcached\(^{13}\)** is a general-purpose distributed memory caching system used by several popular sites such as Craigslist, Digg and Twitter.

\(^{10}\)\url{http://www.gnu.org/software/binutils/}\n
\(^{11}\)\url{http://git-scm.com/}\n
\(^{12}\)\url{http://redmine.lighttpd.net/projects/lighttpd2}\n
\(^{13}\)\url{http://memcached.org/}\n
34
Table 3.1: Summary of applications used in our study. *ELOC* represents the number of executable lines of code and *TLOC* the number of lines in test files in the last revision analysed.

<table>
<thead>
<tr>
<th>App</th>
<th>Lang.</th>
<th>ELOC</th>
<th>Lang.</th>
<th>TLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>C</td>
<td>27,029</td>
<td>DejaGnu</td>
<td>5,186</td>
</tr>
<tr>
<td>Git</td>
<td>C</td>
<td>79,760</td>
<td>C/shell</td>
<td>108,464</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>C</td>
<td>23,884</td>
<td>Python</td>
<td>2,440</td>
</tr>
<tr>
<td>Memcached</td>
<td>C</td>
<td>4,426</td>
<td>C/Perl</td>
<td>4,605</td>
</tr>
<tr>
<td>Redis</td>
<td>C</td>
<td>18,203</td>
<td>Tcl</td>
<td>7,589</td>
</tr>
<tr>
<td>ØMQ</td>
<td>C++</td>
<td>7,276</td>
<td>C++</td>
<td>3,460</td>
</tr>
</tbody>
</table>

5. **Redis**\(^{14}\) is a popular key-value data store used by many well-known services such as GitHub and Flickr.

6. **ØMQ**\(^{15}\) is a high-performance asynchronous messaging middleware library used by a number of organisations such as Los Alamos Labs, NASA and CERN.

The six applications are representative for C/C++ open-source code: GNU Binutils are user-level utilities, Git is a version control system, Lighttpd, Memcached and Redis are server applications, while ØMQ is a library. All applications include a regression test suite.

**Basic characteristics.** Table 3.1 shows some basic characteristics of these systems: the language in which the code and tests are written, the number of executable lines of code (ELOC) and the number of lines of test code (TLOC) in the last revision analysed. To accurately measure the number of ELOC, we leveraged the information stored by the compiler in `gcov` graph files, while to measure the number of TLOC we did a simple line count of the test files (using `cloc`, or `wc -l` when `cloc` cannot detect the file types).

The code size for these applications varies from only 4,426 ELOC for Memcached to 79,760 ELOC for Git. The test code is written in a variety of languages and ranges from

\(^{14}\)http://redis.io/

\(^{15}\)http://zeromq.org/
2,440 lines of Python code for Lighttpd to 108,464 lines of C and shell code for Git. The test code is 36% larger than the application code in the case of Git, approximately as large as the application code for Memcached, around 40% of the application code for Redis and ØMQ, and only around 10% and 19% of the application code for Lighttpd and Binutils respectively. Running the test suite on the last version takes only a few seconds for Binutils, Lighttpd, and ØMQ, 110 seconds for Memcached, 315 seconds for Redis, and 30 minutes for Git, using a four-core Intel Xeon E3-1280 machine with 16 GB of RAM.

The version control system used by all these applications is Git. Four of these projects—Git, Memcached, Redis and ØMQ—are hosted on the GitHub online project site. The other two—Binutils and Lighttpd—use their own Git hosting.

**Selection of revisions.** Our goal was to select a comparable number of revisions across applications. The methodology was to start from the current version at the day of our experiments, and select an equal number of previous revisions for all systems. We only counted revisions which modify executable code, tests or both because this is what our analyses look at. We ended up selecting 250 such revisions from each system because some systems had non-trivial dependency issues further back than this, which prevented us from properly compiling or running them. We still had to install the correct dependencies where appropriate, e.g. downgrade `libev` for older versions of Lighttpd and `libevent` for Memcached.

Note that not all revisions compile, either due to development errors or portability issues (e.g. header files differing across OS distributions). Redis has the largest number of such transient compilation error—38. The prevailing reasons are missing `#include` directives, e.g. `unistd.h` for the `sleep` function, and compiler warnings subsequently treated as

---

16https://github.com/git/git.git  
17https://github.com/memcached/memcached.git  
18https://github.com/antirez/redis.git  
19https://github.com/zeromq/zeromq4-x.git  
20https://github.com/  
21git://sourceware.org/git/binutils.git  
22git://git.lighttpd.net/lighttpd/lighttpd2.git
Table 3.2: Revisions used in our study. OK: code compiles and tests complete successfully, TF: some tests fail, TO: tests time out CF: compilation fails, Time: the number of months analysed, Age: the age of the project as of January 2014.

<table>
<thead>
<tr>
<th>App</th>
<th>OK</th>
<th>TF</th>
<th>TO</th>
<th>CF</th>
<th>Time (mo)</th>
<th>Age (mo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>240</td>
<td>10</td>
<td>0</td>
<td>25</td>
<td>35</td>
<td>176</td>
</tr>
<tr>
<td>Git</td>
<td>249</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>105</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>145</td>
<td>105</td>
<td>0</td>
<td>13</td>
<td>36</td>
<td>66</td>
</tr>
<tr>
<td>Memcached</td>
<td>206</td>
<td>43</td>
<td>1</td>
<td>5</td>
<td>47</td>
<td>127</td>
</tr>
<tr>
<td>Redis</td>
<td>211</td>
<td>38</td>
<td>1</td>
<td>38</td>
<td>6</td>
<td>57</td>
</tr>
<tr>
<td>ØMQ</td>
<td>171</td>
<td>79</td>
<td>0</td>
<td>11</td>
<td>17</td>
<td>53</td>
</tr>
</tbody>
</table>

errors. The missing `#include` directives most likely slipped past the developers because on some systems other `libc` headers cause the missing headers to be indirectly included. The compiler warnings were generated because newer compiler versions, such as the one that we used, are more pedantic. Other reasons include forgotten files and even missing semicolons.

We decided to fix the errors which were not likely to be seen at the time a particular revision was created, for example by adding the compile flag `-Wno-error` in Binutils so that the warnings do not terminate the build process. In all situations when we could not compile a revision, we rolled over the changes to the next revisions until we found one where compilation was successful. Revisions which do not successfully compile are not counted towards the 250 limit.

Another important decision concerns the granularity of the revisions being considered. Modern decentralised software repositories based on version control systems such as Git do not have a linear structure and the development history is a directed acyclic graph rather than a simple chain. Different development styles generate different development histories; for example, Git, Redis and ØMQ exhibit a large amount of branching and merging while the other three systems have a linear history. Our decision was to focus on the main branch, and treat each merge into it as a single revision. In other words, we
considered each feature branch a single indivisible unit. Our motivation for this decision was twofold: first, development branches are often spawned by individual developers in order to work on a certain issue and are often “private” until they are merged into the main branch. As a result, sub-revisions in such branches are often unusable or even uncompileable, reflecting work-in-progress. Second, the main branch is generally the one tracked by most users, therefore analysing revisions at this level is a good match in terms of understanding what problems are seen in the field. This being said, there are certainly development styles and/or research questions that would require tracking additional branches; however, we believe that for our benchmarks and research questions this level of granularity provides meaningful answers.

On a secondary note, we remark that an additional complication with this approach is that version control systems do not associate a branch name to each revision, so some detective work might be required to follow the main development branch. However, since the projects exhibiting a branching structure are hosted on GitHub, an implicit central integrator exists (the project owner) and we considered their history to be the official one, essentially always following the first parent in a merge.

Table 3.2 summarises the revisions that we selected: they are grouped into those that compile and pass all the tests (OK), compile but fail some tests (TF), and compile but time out while running the test suite (TO). The time limit that we enforced was empirically selected for each system such that it is large enough to allow a correct revision to complete all tests. As shown in the table, timeouts were a rare occurrence, with at most one occurrence per application.

Table 3.2 also shows the development time span considered, which ranges from only 5-6 months for Git and Redis, which had a fast-paced development during this period, to almost 4 years for Memcached. The age of the projects at the first version that we analysed ranges from a little over 2 years for Lighttpd 2, to 11 years for Binutils.
Setup. All the programs analysed were compiled to record coverage information. In addition, we disabled compiler optimisations, which generally interact poorly with coverage measurements. For this we used existing build targets and configuration options if available, otherwise we configured the application with the flags `CFLAGS=-O0 -coverage` and `LDFLAGS=-coverage`. All code from the system headers, i.e. `/usr/include/` was excluded from the results.

Each revision was run in a virtualised environment based on Ubuntu 12.10 (12.04.3 for Git) running inside an LXC container. To take advantage of the inherent parallelism of this approach, the containers were spawned in one of 28 long-running Xen VMs, each with a 4 Ghz CPU, 6 GB of RAM, and 20 GB of storage, running a 64-bit version of Ubuntu 12.04.3.

The following subsections present the main findings of our analysis: each one starts with one or more research questions (RQ) that we attempt to answer in that section.

3.3.1 Code and Test Evolution

RQ1: Do executable and test code evolve in sync?

Figure 3.2 shows the evolution of each system in terms of ELOC. As discussed above, we measured the number of ELOC in each revision by using the information stored by the compiler in `gcov` graph files. This eliminates all lines which were not compiled, such as those targeting architectures different from our machine. One of the main reasons for which we have decided to measure ELOC rather than other similar metrics is that they can be easily connected to the dynamic metrics, such as patch coverage, presented in Sections 3.3.2 and 3.3.3.

As evident from this figure, all six systems grow over time, with periods of intense development that increase the ELOC significantly, alternating with periods of code tuning and testing, where the code size increases at a slower pace. It is interesting to note that there are also several revisions where the number of ELOC decreases (e.g.
in ØMQ): upon manual inspection, we noticed that they relate to refactorings such as using macros or removing duplicate code.

The total number of ELOC added or modified varies between 2,296 for Redis and 10,834 for Lighttpd, while the end-to-end difference in ELOC varies between 1,257 for Memcached and 4,466 for Lighttpd.

Figure 3.3 presents the evolution of the size of the test suite in each system, measured in textual lines of test code (TLOC). For each system, we manually identified the files responsible for regression testing and recorded the number of lines contained in them at each revision. It can be seen that test evolution is less dynamic than code evolution, developers adding less test code than regular code.

To better understand the co-evolution of executable and test code, we also merged the above data and plotted in Figure 3.4 only whether a revision changes the code (tests) or not; that is, the Code and Test values increase by one when a change is made.
to the code, respectively to the tests in a revision, and stay constant otherwise. For example, the Binutils plot shows that out of the 250 Binutils revisions analysed, over 200 modify code, but only about 70 modify tests. As it can be seen, while the Code line is smoothly increasing over time, the Test line frequently stays constant across revisions, indicating that testing is often a phased activity [101], that takes place only at certain times during the development cycle. One exception is Git, where code and tests evolve more synchronously, with a large number of revisions modifying both code and tests.

### 3.3.2 Overall Code Coverage

**RQ2: Is code coverage deterministic?**

As a large part of our study focuses on coverage metrics, we first investigate whether code coverage is deterministic, i.e. whether the regression test suite in a given revision
Figure 3.4: Co-evolution of executable and test code. Each increment represents a change.

achieves the same coverage every time it is executed. As we show, nondeterminism has implications in the reproducibility of test results—including the ones that we report—and the fault detection capability of the tests.

We measured the overall coverage achieved by the regression test suite using gcov. Interestingly, we found that all the programs from our experiments except binutils are nondeterministic, obtaining slightly different coverage in each run of the test suite. Therefore, we first quantified this nondeterminism by running the test suite five times for each revision and measuring how many revisions obtained mixed results, i.e. one run reports success while another reports failure. We were surprised to see a fair number of revisions displaying this behaviour, as Table 3.3 shows in the Nondet Result column.

We further counted for each pair of runs the number of lines whose coverage status differs. We used a 0/1 metric, i.e. we only considered a difference when one of the five
Table 3.3: Number of revisions where the test suite nondeterministically succeeds/fails, and the maximum, median and average number of lines which are nondeterministically executed in a revision.

<table>
<thead>
<tr>
<th>App.</th>
<th>Nondet. Result</th>
<th>Nondet. ELOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Median</td>
</tr>
<tr>
<td>Binutils</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Git</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>Memcached</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Redis</td>
<td>16</td>
<td>71</td>
</tr>
<tr>
<td>ØMQ</td>
<td>32</td>
<td>47</td>
</tr>
</tbody>
</table>

runs never executes a line and another one executes it. We only did this for revisions in which the test suite completes successfully to avoid spurious results that would occur if we compare a run which completed with one that was prematurely terminated due to a failure. As shown in Table 3.3, binutils seems to be completely deterministic with respect to its test suite, while Redis, for example, contains on average 30.98 lines that are nondeterministically executed.

We manually investigated the nondeterminism and pinpointed three sources: (1) multi-threaded code, (2) ordering of network events, and (3) nondeterminism in the test harness. As an example from the first category, the test from ØMQ `test_shutdown_stress` creates 100 threads to check the connection shutdown sequence. In a small percentage of runs, this test exposes a race condition. As an example in the third category, some Redis tests generate and store random integers, nondeterministically executing the code implementing the internal database data structures. The Memcached test `expirations.t` is representative of tests that make assumptions based on hardcoded wall-clock time values, which cause failures under certain circumstances. The test timings were previously adjusted in response to failures under Solaris’ `dtrace` and we believe that some of the failures that we encountered were influenced by the Docker environment.

\[\text{https://github.com/zeromq/zeromq4-x/commit/de239f3}\]
\[\text{https://github.com/memcached/memcached/commit/890e3cd}\]
Figure 3.5: Evolution of overall line and branch coverage.

The potential drawback of nondeterminism is the inability of coverage comparison across revisions, lack of reproducibility and consequent difficulty in debugging. Developers and researchers relying on test suite executions should take nondeterminism into account, by either quantifying its effects, or by using tools that enforce deterministic execution across versions [46], as appropriate. Tests with nondeterministic executions—such as the ones presented above—are fragile and should be rewritten. For example, tests relying on wall-clock time could be rewritten as event-based tests [47].

**RQ3: How does the overall code coverage evolve? Is it stable over time?**

When reporting the overall coverage numbers, we accumulated the coverage information across all five runs.\(^{25}\) Therefore, the results aim to count a line as covered if the test suite may execute it. The blue (upper) lines in Figure 3.5 plot the overall line coverage

\(^{25}\)With the exception of Git, where for convenience we considered a single run, as the number of lines affected by nondeterminism represent less than 0.3\% of the total codebase.
```c
#define zmq_assert(x) \
  do { \
    if (unlikely (!x)) { \
      fprintf(stderr, "Assertion failed: %s (%s:%d)\n", #x, \
       __FILE__, __LINE__); \
      zmq::zmq_abort (#x); \
    } \
  } while (false)
```

Listing 3.1: Example of an assertion macro used in the ØMQ codebase.

for all benchmarks. It can be seen that coverage level varies significantly, with Binutils at one end achieving only 17.39% coverage on average, and Git at the other achieving 80.74%, while in-between Lighttpd achieves 39.08%, Redis 59.97%, ØMQ 66.88%, and Memcached 72.98%.

One interesting question is whether coverage stays constant over time. As evident from Figure 3.5, for Binutils, Git, Memcached, and Redis, the overall coverage remains stable over time, with their coverage changing with less than 2 percentage points within the analysed period. On the other hand, the coverage of Lighttpd and ØMQ increase significantly during the time span considered, with Lighttpd increasing from only 2.02% to 49.37% (ignoring the last two versions for which the regression suite fails), and ØMQ increasing from 62.89% to 73.04%. An interesting observation is that coverage evolution is not strongly correlated to the co-evolution of executable and test code (RQ1). Even when testing is a phased activity, coverage remains constant because the already existing tests execute part of the newly added code.

One may notice that a few revisions from Lighttpd, Memcached and Redis cause a sudden decrease in coverage. This happens because either bugs in the program or in the test suite prevent the regression tests from successfully running to completion. In all cases, these bugs are fixed after just a few revisions.

Figure 3.5 also shows that the branch coverage closely follows the line coverage. The difference between line and branch coverage is relatively small, with the exception of Memcached and ØMQ. The larger difference is due to the frequent use of certain code
patterns which generate multiple branches on a single line, such as the one shown in Listing 3.1, which comes from the ØMQ codebase. The `zmq_assert` macro is expanded into a single line resulting in 100% line coverage, but only 50% branch coverage when executed in a typical run of the program (where assertions do not fail).

The fact that line and branch coverage closely follow one another suggests that in many situations only one of these two metrics might be needed. For this reason, in the remaining of the thesis, we focus only on line coverage.

Finally, we have looked at the impact on coverage of revisions that only add or modify tests. An interesting observation is that many of these revisions bring no improvements to coverage. For example, in Lighttpd only 26 out of 52 such revisions improve coverage. The other 26 either do not affect coverage (18) or decrease it (8). The revisions which do not affect coverage can be a sign of test driven development, i.e. the tests are added before the code which they are intended to exercise. The revisions which decrease coverage are either a symptom of nondeterminism—six of them, with small decreases in coverage—or expose bugs or bigger changes in the testing infrastructure (the other two). These two revisions exhibit a drop in coverage of several thousands lines of code. In one case, the tests cause Lighttpd to timeout which leads to a forceful termination and loss of coverage data. This problem is promptly fixed in the next revision. In the other case, the new tests require a specific (new) module to be built into the server, terminating the entire test suite prematurely otherwise.

### 3.3.3 Patch Coverage

**RQ4: What is the distribution of patch coverage across revisions?**

We define *patch coverage* as the ratio between the number of executed lines of code added or modified by a patch and the total number of executable lines in the patch, measured
Figure 3.6: Patch coverage distribution. Each colour bar represents a range of patch coverage values with its size proportional to the number of patches with coverage in that range.

in the revision that adds the patch. Low patch coverage is a strong sign that tools such as KATCH can improve the testing process.

Figure 3.6 shows the distribution of the patch coverage for each system. Each column corresponds to all patches which affect executable code in a system, normalised to 100%. The patches are further grouped into four categories depending on their coverage. As it can be observed, the patch coverage distribution is bi-modal across applications: the majority of the patches in Git, Memcached and ØMQ achieve over 75% coverage, while the majority of the patches in Binutils, Lighttpd and Redis achieve under 25%. One interesting aspect is that for all applications, there are relatively few patches with coverage in the middle ranges: most of them are either poorly (≤25%) or thoroughly (>75%) covered.
Table 3.4: Overall patch coverage bucketed by the size of the patch in ELOC. NP is the number of patches in the bucket and C is their overall coverage. Only patches which add or modify executable code are considered.

<table>
<thead>
<tr>
<th>App</th>
<th>≤10</th>
<th>11-100</th>
<th>&gt;100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>128</td>
<td>19.5%</td>
<td>63</td>
</tr>
<tr>
<td>Git</td>
<td>102</td>
<td>87.4%</td>
<td>65</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>120</td>
<td>41.9%</td>
<td>58</td>
</tr>
<tr>
<td>Memcached</td>
<td>122</td>
<td>73.7%</td>
<td>73</td>
</tr>
<tr>
<td>Redis</td>
<td>164</td>
<td>33.8%</td>
<td>51</td>
</tr>
<tr>
<td>ØMQ</td>
<td>119</td>
<td>65.5%</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 3.4 presents the same patch coverage statistics, but with the patches bucketed by their size into three categories: less than 10 ELOC, between 11 and 100 ELOC, and greater than 100 ELOC. For all benchmarks, patches are distributed similarly across buckets, with the majority of patches having ≤ 10 ELOC and only a few exceeding 100 ELOC. Across the board, the average coverage of patches with ≤10 ELOC is higher than for those with >100 ELOC, but the coverage of the middle-sized category varies.

Finally, the first column in Table 3.5 shows the overall patch coverage, i.e. the percentage of covered ELOC across all patches. For Binutils, Git and Memcached, it is within five percentage points from the overall program coverage, while for the other benchmarks it is substantially lower—for example, the average overall program coverage in Redis is 59.97%, while the overall patch coverage is only 30.4%. These results show that tools such as KATCH, which focus on executing previously uncovered code have the potential to improve the testing process, find more bugs and improve confidence in the code’s correctness. As we later show, KATCH can more than double the patch coverage of real systems.

**RQ5: What fraction of patch code is tested within a few revisions after it is added, i.e. what is the latent patch coverage?**

In some projects, tests exercising the patch are added only after the code has been submitted, or the patch is only enabled (e.g. by changing the value of a configuration
Table 3.5: Overall latent patch coverage: the fraction of the lines of code in all patches that are only executed by the regression suite in the next 1, 5 or 10 revisions. The overall patch coverage is listed for comparison.

<table>
<thead>
<tr>
<th>App</th>
<th>Overall</th>
<th>+1</th>
<th>+5</th>
<th>+10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>21.2%</td>
<td>0.1%</td>
<td>0.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Git</td>
<td>85.1%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>31.3%</td>
<td>0.9%</td>
<td>5.0%</td>
<td>6.1%</td>
</tr>
<tr>
<td>Memcached</td>
<td>68.9%</td>
<td>2.1%</td>
<td>3.4%</td>
<td>3.5%</td>
</tr>
<tr>
<td>Redis</td>
<td>30.4%</td>
<td>5.2%</td>
<td>5.5%</td>
<td>6.4%</td>
</tr>
<tr>
<td>ØMQ</td>
<td>56.9%</td>
<td>0.4%</td>
<td>3.5%</td>
<td>6.0%</td>
</tr>
</tbody>
</table>

parameter) after related patches or tests have been added. To account for this development style, we also recorded the number of ELOC in each patch which are only covered in the next few revisions (we considered up to ten subsequent revisions). We refer to the ratio between the number of such ELOC and the total patch ELOC as latent patch coverage.

We counted these lines by keeping a sliding window of uncovered patch lines from the past ten revisions and checking whether the current revision covers them. When a patch modifies a source file, all entries from the sliding window associated with lines from that file are remapped if needed, using the line mapping algorithm discussed in Section 3.2.

Table 3.5 shows the overall latent patch coverage i.e. the fraction of patch lines that are covered in the next few revisions after the patch is introduced. We report the results for three sliding window sizes: one, five and ten revisions. The latent patch coverage is significantly smaller compared to the overall patch coverage, accounting at most for 6.4% in Redis, where, as previously pointed, the developers almost never add code and tests in the same revision.

As conjectured, we found two main causes of latent patch coverage: tests being added only after the patch was written (this was the case in Lighttpd, where 12 revisions which only add tests cover an additional 74 ELOC) and patch code being enabled later on. In fact, the majority of latent patch coverage in Lighttpd—337 lines—is obtained by 6
Table 3.6: Median coverage and the number of revisions achieving 100% coverage for the revisions containing bug fixes. The overall metrics are included for comparison.

<table>
<thead>
<tr>
<th>App</th>
<th>Coverage (med)</th>
<th>Fully Covered</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overall</td>
<td>Fix</td>
<td>Overall</td>
<td>Fix</td>
</tr>
<tr>
<td>Memcached</td>
<td>89%</td>
<td>100%</td>
<td>45.4%</td>
<td>58.5%</td>
</tr>
<tr>
<td>Redis</td>
<td>0%</td>
<td>94.1%</td>
<td>25.5%</td>
<td>50.0%</td>
</tr>
<tr>
<td>ØMQ</td>
<td>76%</td>
<td>55.4%</td>
<td>33.3%</td>
<td>31.8%</td>
</tr>
</tbody>
</table>

revisions which change no test files. Upon manual inspection, we found that the code involved was initially unused, and only later revisions added calls to it.

3.3.4 Bug analysis

RQ6: Are bug fixes better covered than other types of patches?

RQ7: Is the coverage of buggy code less than average?

To answer these RQs, we collected bug data according to the methodology presented in Section 3.2 and we limited our analysis to the three systems which lend themselves to automatic identification of bug fixes based on commit messages: Memcached, Redis and ØMQ. The other three systems use non-specific commit messages for bug fixes, requiring an extensive manual analysis or more complex algorithms such as machine learning and natural language processing for understanding the contents of a specific revision [65]. We ignored revisions which do not affect executable files, such as fixes to the build infrastructure or to the documentation and then manually confirmed that the remaining revisions are indeed bug fixes [43] and further removed fixes which modify only non-executable lines (e.g. variable declaration). We thus obtained 41 fixes in Memcached, 22 in Redis and 22 in ØMQ.

We measured the patch coverage of these revisions and report the median values in Table 3.6, together with the corresponding overall metric, for comparison. For both
Memcached and Redis, the coverage for fixes is higher than that for other types of patches. For Redis, the median value jumps from 0% to 94.1%, while for Memcached the difference is less pronounced. On the other hand, the fixes in ØMQ are covered less than on average. The fraction of fixes which have 100% coverage follows the same trend.

To try to understand whether buggy code is less thoroughly tested than the rest of the code, we started from the observation that bug-fixing revisions are usually only addressing the bug, without touching unrelated code. Because of this, we can identify the code responsible for the bugs by looking at the code which is removed or modified by bug-fixing revisions and compute its coverage in the revision before the fix. The coverage for this code is 72.7% for Memcached—roughly the same as the overall patch coverage, 65.2% for Redis—much larger than the overall patch coverage, and 35.8% for ØMQ—significantly lower.

While these numbers cannot be used to infer the correlation between the level of coverage and the occurrence of bugs—the sample is too small, and the bugs collected are biased by the way they are reported—they clearly show the limitations of line coverage as a testing metric, with bugs even being introduced by patches which are fully covered by the regression test suite. Therefore, even for well-tested code, tools such as ZESTI, which thoroughly check each program statement for bugs using symbolic execution can be useful in practice.

### 3.3.5 Threats to Validity

The main threat to validity in our study regards the generalisation of our results. The patterns we have observed in our data may not generalise to other systems, or even to other development periods for the same systems. However, we regard the selected systems to be representative for open-source C/C++ code, and the analysis period was chosen in an unbiased way, starting with the current version at the time of our experiments.
Errors in the software underlying our framework could have interfered with our experiments. Both Docker and LXC were under development and not recommended for use in production systems at the time of our study. Furthermore, in case of some applications, we have observed test failures caused by the AuFS\textsuperscript{26} filesystem used by Docker. However, we have thoroughly investigated these failures and we believe they did not affect the results presented in our study.

Given the large quantity of data that we collected from a large number of software revisions, errors in our scripts cannot be excluded. However, we have thoroughly checked our results and scripts, and we are making our framework and data available for further validation.

### 3.4 Conclusion

Despite the important role that regression test suites play in software testing, there are surprisingly few empirical studies that report how they co-evolve with the application code, and the coverage level that they achieve. Our empirical study on six popular open-source applications, spanning a combined period of twelve years, aims to contribute to this knowledge base. To the best of our knowledge, the number of revisions executed in the context of this study—1,500—is significantly larger than in prior work, and this is also the first study that specifically examines patch coverage.

Our results indicate that both KATCH and ZESTI can potentially improve the effectiveness of the testing process in real systems. On the one hand, most of the systems analysed have reasonably good regression suites, which contain sufficient inputs to guide symbolic execution. On the other hand, there is significant room for improvement both in terms of overall coverage and patch coverage (RQ 3 and RQ 4), and in the thoroughness of the checks done by the existing tests (RQ6 and RQ7).

\textsuperscript{26}http://aufs.sourceforge.net/
Our experience has revealed two main types of challenges for conducting similar or larger studies that involve running a large number of program revisions. The first category relates to the inherent difficulty of running older revisions:

1. Decentralised repositories have non-linear histories, so even defining what a revision is can be difficult, and should be done with respect to the research questions being answered. In our case, we chose a granularity at the level of commits and merges to the main branch.

2. Older revisions have undocumented dependencies on specific compiler versions, libraries, and tools. We found it critical to run each revision in a separate virtual machine environment, to make it easy to install the right dependencies, or adjust build scripts.

3. Some older revisions do not compile. This may be due to errors introduced during development and fixed later, or due to incorrectly resolved dependencies. The execution infrastructure has to be flexible in tolerating such cases, and one needs a methodology for dealing with uncompileable revisions. In our case, we have skipped over the uncompileable revisions and incorporated their changes into the next compilable one.

4. The execution of the regression test suite is often nondeterministic—the test suite may nondeterministically fail and some lines may be nondeterministically executed. Studies monitoring the execution of the program need to take nondeterminism into account.

The second category of challenges relates to reproducibility and performance. Our COVRIG infrastructure ensures reproducibility through the use of software containers technology. Performance has two different aspects: at the level of an individual revision, we have found it essential to use a form of operating system-level virtualisation (in our case, Docker and LXC), in order to minimise the time and space overhead typically
associated with hardware virtualisation. Across revisions, we found it necessary to provide the ability of running our set of revisions on multiple local and cloud machines. For example, running the Git regression suite took in our case 26 machine days (250 revisions × 30 min/revision × 5 runs), which would have been too expensive if we used a single machine, especially since we also had to repeat some runs during our experimentation.

We believe this study provides useful empirical evidence regarding the evolution of code and tests in real software. We also hope it will encourage other similar studies, and to this end we are working on transforming COVRIG into a flexible extensible platform. We also make our experimental data available at http://srg.doc.ic.ac.uk/projects/covrig/.
Chapter 4

KATCH

4.1 Overview

While the code of popular software systems is frequently changing, these changes—or patches—are often poorly tested by developers, as we have shown in §3.3.3. In fact, as we further report in §4.4.1, developers often add or modify lines of code without adding a single test that executes them! To some extent, we have not found this result surprising, as we know from experience how difficult it can be to construct a test case that covers a particular line of code.

While the problem of generating inputs that cover specific parts of a program is generally undecidable, we believe that in many practical circumstances it is possible to automatically construct such inputs in a reasonable amount of time. Our system KATCH\(^1\) uses several insights to implement a robust solution. First, KATCH uses existing test cases from the program’s regression suite—which come “for free” and often already execute interesting parts of the code—as a starting point for synthesising new inputs. For each test case input, KATCH computes an estimated distance to the patch and then selects the closest input (§4.2.2) as the starting point for symbolic exploration. Second, symbolic

\(^1\)The name comes from K[LEE]+[P]ATCH. KLEE is an open-source symbolic execution engine on which KATCH is based.
Figure 4.1: The main stages of the KATCH patch testing infrastructure.

execution provides a framework for navigating intelligently through the intricate set of paths through a program, starting from the trace obtained by running the previously identified closest input. To reach the patch, KATCH employs three heuristics based on program analysis: greedy exploration, informed path regeneration (§4.2.3) and definition switching (§4.2.4).

Figure 4.1 presents the high-level architecture of KATCH. The framework takes as input a program, a set of existing program inputs and a patch description in the form of a diff file and automatically constructs new inputs that execute the patch code by following three steps.

*Patch preprocessing* is responsible for parsing the raw patch file and splitting it into lines of code. Lines of code that are part of the same basic block (and thus always executed together), are grouped to form a single *target*. Targets which are already executed by the program’s regression tests are identified by executing the tests, and dismissed at this step. For each remaining target, the following stages are executed to synthesise an input which exercises it.

*Input selection* leverages the fact that real applications already come with regression suites that contain a rich set of well-formed inputs created by the developers. Input selection takes as input the program, a target and an existing test suite. It then associates with each of the test inputs a *distance* estimating the effort required to modify it such that it executes the target. The *closest* input to the target is then used in the next stage.
The last step combines symbolic execution with three heuristics based on program analysis to derive a new input that exercises the target, starting from the input selected at the previous step. The role of symbolic execution is twofold. First, it provides a framework for inspecting the program branch decisions and their relation to program inputs, and gives the means to generate new inputs by changing the outcome of particular branches. Second, it thoroughly checks program operations such as memory accesses and assertions, in order to find errors. The heuristics based on program analysis complement symbolic execution by partly mitigating its scalability problems and steering it actively towards the target.

To scale this process to multiple systems and hundreds or thousands of patches, we have also built an infrastructure which executes automatically, as appropriate, each of the previous steps, requiring no changes to the systems under test nor to their regression suites (§4.3).

4.2 KATCH

This section describes in more detail the KATCH patch testing infrastructure: patch preprocessing (§4.2.1), input selection with weakest preconditions (§4.2.2), greedy exploration with informed path regeneration (§4.2.3) and definition switching (§4.2.4).

4.2.1 Patch Preprocessing

The first stage of our analysis is mainly responsible for retrieving each program version from the version control system, determining the differences from the previous version—i.e. the patch—and breaking this patch into lines which are then handled individually by the subsequent steps. In addition, the lines are filtered and consolidated when appropriate, as described next.
Figure 4.2 shows the preprocessing steps that each patch goes through. While each line in a patch is a potential target to KATCH, in practice, many lines do not need to be considered. First, source code contains many non-executable lines, such as declarations, comments, blank lines, or lines not compiled into the executable due to conditional compilation directives (Figure 4.2(b)). Second, we are not interested in lines already covered by the system’s regression test suite (Figure 4.2(c)). Finally, lines which are part of the same basic block are always going to be executed together, so we only need to keep one representative per basic block (Figure 4.2(d)).
The patch preprocessing stage is responsible for eliminating all these lines and works in two steps: a first step performs a simple static pass to eliminate non-executable code and all but one line in each basic block, and a second step runs the program’s regression suite to eliminate lines already covered by its test cases.

This results in a set of lines which are on the one hand executable and on the other hand are not executed by the program’s test suite—which we call targets. Each of them is processed individually in the following stages.

### 4.2.2 Seed Selection With Weakest Preconditions

Our input synthesis technique starts from an existing program input—called the seed—extracted from the program’s test suite, and iteratively changes it. The ideal seed executes code which is close to the target, in order to allow KATCH to quickly steer execution by switching only a few branch outcomes to reach the target.

To estimate the distance between the path executed by a seed and the target, we calculate the (static) minimum distance in the program’s interprocedural control flow graph (iCFG) between each basic block exercised by the seed and the target basic block. The minimum of these distances represents the distance from the seed to the target. Intuitively, the effort of symbolic execution lies in switching the outcome of branch statements, therefore we compute this distance in terms of the number of branch statements between the two basic blocks.

We also simplify the minimum distance computation by not requiring it to be context-sensitive. To do this, we note that pairs of matched function calls and returns should not contribute to the distance between two basic blocks. In practice, this means that we can “shortcut” function calls, i.e. each function call introduces an edge to the instruction immediately following the call, in addition to the edge to the target function. In turn, shortcutting function calls allows us to remove all return edges, simplifying the analysis.
However, the estimated distance outlined so far—which we call \textit{C-flow} distance, as it only takes the control flow into account—can select inputs which exercise paths close to a target, but cannot be easily changed to actually reach the target. In the interest of simplicity, we show a contrived example in Figure 4.3 to illustrate such a scenario. The code snippet takes a single integer as input and uses it to decide whether to call function \textit{f}, which contains the target. The only input which exercises the target is 999. The figure also shows the C-flow distance from each instruction to the target. For example, the C-flow distance for the instruction at line 5 is 2, because the shortest path to the target traverses two branches (on lines 5 and 9).

For simplicity, assume that we only want to assess whether input \textit{50} is better than input \textit{150}. From a pure control-flow perspective, \textit{50} appears to be a better choice because it exercises function \textit{f} and reaches the \textit{if} condition guarding the target (while \textit{150} does not call \textit{f} at all). Upon closer inspection however, it is clear that the target guard condition \texttt{x == 999} is always \texttt{false} on this path because function \textit{f} is called with argument \texttt{0} on line 2, and therefore the target cannot get executed through this call. This observation
led us to create a technique which automatically prunes control flow graph edges which provably do not lead to the target.

To find such edges we use an interprocedural data-flow analysis which computes for each target and basic block in the program a necessary condition to reach that target, a form of weakest precondition (§2.3). If by traversing an edge we obtain a false condition, we conclude that the target is unreachable through that edge. Considering the same example, the branch on line 9, which guards the target, creates the condition \( x = 999 \), while the edge from the function call at line 2 defines \( x \) to be 0. By simple syntactic substitution, we obtain the formula \( 0 = 999 \) which evaluates to false, and conclude that the function call on line 2 cannot help in reaching the target. Column WP of Figure 4.3 shows the minimum distance from each instruction to the target after removing the edge introduced by this function call. Lines 1 and 2 have their distances updated.

For the interested reader, we present the data-flow equations which compute the preconditions, relative to a target, at the beginning and at the end of each basic block, and give an intuition on their correctness.

\[
\begin{align*}
(1) \quad & \text{out}_b = \bigvee_{s \in \text{succ}} (\text{cond}_{b \to s} \land \text{in}_s) \\
(2) \quad & \text{in}_b = \wp(b, \text{out}_b)
\end{align*}
\]

With initial values:

\[
\begin{align*}
\text{in}_{\text{target}} &= \text{true}, \quad \text{out}_{\text{target}} = \text{false} \\
\text{in}_b &= \text{out}_b = \text{false}, \forall b \neq \text{target}
\end{align*}
\]

\( \text{cond}_{b \to s} \) represents the condition required to go from basic block \( b \) to \( s \). For unconditional branches the condition is always \text{true}. \( \wp(b, \text{out}_b) \) is the standard weakest precondition function (§2.3), applied to basic block \( b \) and postcondition \( \text{out}_b \), which is easily computed for a single basic block as we describe below.
The equations guarantee that any edge \( b \to s \) for which \( \text{cond}_{b \to s} \land \text{in}_s \) is \texttt{false} and any basic block \( b \) for which \( \text{in}_b \) is \texttt{false} cannot lead to the target.

The first equation intuitively says that at the end of a basic block \( b \), the condition to reach the target is the disjunction of the conditions for all possible paths from that basic block to the target. The second equation obtains the weakest precondition for a basic block from its corresponding postcondition. This is done by iterating through the instructions of the basic block in reverse order and substituting all variables from the postcondition with their definition, as appropriate. A variable not defined in the current basic block is left unchanged. When applied to the target basic block, the \( \wp \) function always yields \texttt{true}.

Solving the system is done using a standard fixed-point computation approach. Our implementation makes two conservative approximations to make the analysis tractable even on large programs. First, the \( \wp \) function only handles assignments which do not involve function calls. If the basic block applies other operations to the postcondition variables, the returned value is \texttt{true}. Second, a disjunction of syntactically non-identical formulae in the first equation is also treated as \texttt{true}, to prevent formulae from growing exponentially.

These two approximations capture two common practical cases. First, formulae which become \texttt{false} when applying the \( \wp \) function usually correspond to code patterns which use boolean flags or enumerated type variables in branch conditions; basic blocks which, for example, set a flag to \texttt{false} and make a certain branch infeasible are recognised accordingly. The example in Figure 4.3 is such a case.

Second, formulae may become \texttt{false} because the set of conjuncts accumulated through the first data-flow equation becomes inconsistent. This case corresponds to patterns where the same variables are used in branch conditions multiple times, possibly in different parts of the program and some of the conditions are mutually incompatible. A simple example can be observed in Figure 4.3: the weakest precondition algorithm
can prove that the branch between lines 1 → 2 does not lead to the target because the branch condition $\text{input} < 100$ is incompatible with the condition $\text{input} > 100$ which appears subsequently on the only path to the target.

After obtaining the branches and basic blocks that cannot lead to the target, and the distance from each basic block to the target, the distance from each available seed input to the target is computed as follows:

1. Compute the subgraph $G$ of the program’s iCFG by running the program on the seed input. $G$’s nodes are the basic blocks executed and its edges are the branches taken and function calls made during execution;

2. Remove all nodes and edges in $G$ which were proven to make the target unreachable;
3. Iteratively remove from $G$ all nodes orphaned by the previous step, i.e. while there are nodes with in-degree 0 (except the program entry point), remove them and all their outgoing edges;

4. Choose the minimum from the distances of the remaining nodes to the target.

Step 3 is a dynamic refinement of the previous data-flow analysis. Informally, it propagates the infeasible property to all basic blocks which the input exercises only through infeasible basic blocks or branches, thus obtaining a more accurate estimate for the length of the shortest feasible path to the target.

Figures 4.4 and 4.5 show graphically the execution of the four steps on the code previously presented in Figure 4.3. It can be seen that input 50 is farther from the target than input 150.

### 4.2.3 Greedy Exploration with Informed Path Regeneration

The last and most challenging stage of KATCH is responsible for transforming the previously selected seed input into a new input that executes the target. Our approach is based on symbolic execution (§2.1), a program analysis technique that can systematically explore paths through a program. To recapitulate, the key idea behind symbolic execution is to run the program on symbolic input, which is initially allowed to have any value. Then, whenever a branch depending directly or indirectly on the symbolic input is encountered, execution is conceptually forked to follow both sides if both are feasible, adding appropriate constraints on each side of the branch. Finally, whenever a path terminates or hits an error, the constraints gathered on that path are solved to produce a concrete input that exercises the path. For example, if we run the code in Figure 4.3 treating the input variable as symbolic, then at branch 1 execution will be split into two paths: one following the then side of the branch, on which we add the constraint that $\text{input} < 100$, and one following the implicit else side of the branch, on which we
add the constraint that \( \text{input} \geq 100 \). When the path with the constraint \( \text{input} < 100 \) reaches line 4, only the else side is feasible, so no other path is spawned at this point. On the other hand, when the path with the constraint \( \text{input} \geq 100 \) reaches line 4 both sides are feasible, so execution is again split into two paths, one on which we add the constraint that \( \text{input} > 100 \), and one on which we add the constraint that \( \text{input} \leq 100 \) (which together with the existing constraint that \( \text{input} \geq 100 \) gets simplified to \( \text{input} = 100 \)). The branches at lines 5 and 9 similarly spawn new execution paths. Finally, when a path terminates, a constraint solver is used to generate a solution to all the constraints gathered on that path, which represents an input that can be used to exercise the path. For example, the path with the constraints \( \text{input} \geq 100 \), \( \text{input} > 100 \) and \( \text{input} \leq 200 \) may return the solution \( \text{input} = 150 \) which exercises that path.

In our approach, we start symbolic execution from an existing input, the seed, similarly to the approach taken in concolic execution (§2.1) and our ZESTI system (Chapter 5). The seed is then iteratively modified by exploring paths which get closer to the target; symbolic execution provides the framework for the exploration and constraint solving is used to map program paths back to inputs. The novelty of our approach lies in the way paths are selected for exploration.

The selection is based on a metric which estimates the distance from a path to the target, similar to the distance used by the input selection stage (§4.2.2). In each iteration, we execute the program using the latest input, and remember all branch points, e.g. if conditions, along with information necessary to continue execution on the other side of the branch, should we later decide to.

We then select the branch point whose unexplored side \( S \) is closest to the target (according to the estimated distance) and attempt to explore this side. If \( S \) is feasible, i.e. the conjunction of the branch condition towards \( S \) and the current path condition is satisfiable, we eagerly explore it, in what we call a greedy exploration step. Otherwise, we examine two possibilities: (1) the branch condition is symbolic, i.e. it has a data
void log(char input) {
    int file = open("access.log", O_WRONLY|O_APPEND);
    if (input >= ' ' && input <= '~') {
        write(file, &input, 1);
    } else {
        char escinput = escape(input);
        write(file, &escinput, 1);
    }
    close(file);
}

Figure 4.6: Example based on lighttpd patch 2660 used to illustrate the greedy exploration step. Lines 3, 5–8 represent the patch.

dependence on program input on the current path and (2) the branch condition is concrete, i.e. it has a control dependence on program input. Informally, a branch condition is data dependent on program input if data propagates from the input to at least one of the variables involved in the branch condition via a sequence of assignments. A condition is control dependent on the input if at least one variable involved in the condition has more than one reaching definition. Note that some conditions can be both data and control dependent.

For data dependent conditions (including those which are also control dependent), we apply informed path regeneration, where we travel back to the branch point that made $S$ infeasible and take there the other side of the branch. For control dependent conditions, we attempt to find a different definition for the variables involved in the condition, such that the condition becomes true. In the following, we examine each of these cases in detail.

To illustrate our approach, we use the code snippet in Figure 4.6, which is based on a patch introduced in revision 2660 of the lighttpd web server. The log function takes a single character as input and writes it into a text file. The function was initially writing all characters unmodified, but was patched in order to escape sensitive characters that could corrupt the file structure. However, the program was tested only with printable
if (0 == strcmp(requestVerb, "GET")) { ... }

for (char* p = requestVerb; *p; p++) {
    log(*p);
}

Figure 4.7: Example based on lighttpd patch 2660 used to illustrate the informed path regeneration step. As in Figure 4.6, the patch is on lines 3, 5–8 of the log function.

class inputs and thus the else branch was never executed. After seeding the analysis with such an input containing only printable characters, our technique determines that the else side of the symbolic branch point at line 3 is the unexplored branch side closest to the patch (in fact, it is part of the patch), and goes on to explore it (in a greedy exploration step) by negating the condition on line 3.

To understand when informed path regeneration is necessary, consider the example in Figure 4.7, in which the log function of Figure 4.6 is called for each character of the requestVerb string. Assuming that the seed request contains the GET verb, the comparison at line 1 constrains this input to the value GET for the remainder of the execution. Changing any of the characters in the requestVerb is impossible after this point because it would create an inconsistent execution, and thus on this path we cannot follow the else side of the branch in the log function.

Instead, our informed path regeneration step travels back just before the execution of the symbolic branch point that introduced the constraint that makes the patch unreachable, and then explores the other side of that branch point. In our example, that symbolic branch point is the one at which requestVerb[2] was constrained to be ‘T’, and thus our technique takes here the other side of the branch, in which requestVerb[2] is constrained to be different from ‘T’. With this updated path condition, execution reaches again line 3 of the log function, where execution is allowed to take the else path and thus cover the patch.
217 enum DIFF_wh_sp ig_white_space = ignore_white_space;
...
230 switch (ig_white_space) {
232    case IGNORE_ALL_SPACE:
233       while ((c = *p++) != '\n')
234          if (! isspace (c))
235             h = HASH (h, ig_case ? tolower (c) : c);
236          break;
}

src/diff.c
291 while ((c = getopt_long (argc, argv,
292           shortopts, longopts, NULL)) != -1) {
293     switch (c) {
295          case 'b':
296             if (ignore_white_space < IGNORE_SPACE_CHANGE)
297                 ignore_white_space = IGNORE_SPACE_CHANGE;
298             break;
299          case 'Z':
300             if (ignore_white_space < IGNORE_SPACE_CHANGE)
301                ignore_white_space |= IGNORE_TRAILING_SPACE;
302          case 'E':
303             if (ignore_white_space < IGNORE_SPACE_CHANGE)
304                ignore_white_space |= IGNORE_TAB_EXPANSION;
305          case 'w':
306             ignore_white_space = IGNORE_ALL_SPACE;
307             break;
308          ...
Figure 4.8: Example from diffutils revision 8739d45f showcasing the need for definition switching. The patch is on line 235 and is guarded by a condition that is control dependent on the input.

4.2.4 Definition Switching

Informed path regeneration does not work if the branch condition has a concrete value, essentially because we cannot reason symbolically about concrete expressions. This
case occurs when the condition does not have a data dependence on the input on the currently explored path, but only a control dependence. Figure 4.8, containing code from diffutils revision 8739d45f, showcases such a scenario. The revision modifies line 235, which is our target.

To execute the patch, one needs to pass through the switch statement on line 230, requiring ig_white_space, and in turn ignore_white_space to be equal to the IGNORE_ALL_SPACE constant. This only happens when the program is given the -w command line argument (line 495). Assuming the current input does not include -w, the lack of a data dependence between the switch condition and the command line arguments renders informed path regeneration unusable. To solve this problem, we use a lightweight approach that finds the reaching definitions for the variables involved in the condition using static analysis and then attempts to find a path to an uncovered definition using the two techniques previously presented. To further improve the chances of getting the right definition early, the algorithm gives priority to definitions that can be statically shown to satisfy the target branch condition. Furthermore, the algorithm works recursively on all definitions which were already executed, but for which the right-hand side is not a constant. That is, the algorithm can be nested multiple times by using a stack of intermediary targets; when a definition needs to be switched, the active target is saved on the stack and the selected definition becomes the new active target. As soon as the definition is executed, the previous target is popped off the stack.

To show how definition switching works in practice, consider the same code snippet and the input -a -y -- a b provided by input selection, which compares two files a and b treating them as text (-a), and outputs the results side-by-side (-y). This input reaches the guarding switch statement on line 230 but evaluates to a different case. To reach the target, we need to modify the input such that the condition ig_white_space == IGNORE_ALL_SPACE is satisfied. Because the condition does not have a data dependence on the input, KATCH attempts to find another definition for the ig_white_space local variable and discovers one on line 217. However, it detects that this definition was
Figure 4.9: Configuration file used to test diffutils. The file specifies the repository address, the folders which may contain relevant changes, the programs to test and the libraries required to build the system.

already executed, so it recursively attempts to find definitions for the right-hand side of the assignment, the ignore_white_space global variable.

At this point, KATCH finds four definitions, each corresponding to a different command line argument and decides to use ignore_white_space = IGNORE_ALL_SPACE because it matches exactly the original condition which it attempts to satisfy. KATCH now pushes the original target (line 235) to the stack and changes the active target to line 495. It then uses an informed path regeneration step to replace the first command line argument with the required -w option. This reaches the intermediary target which causes the original target to be popped off the target stack and transformed back into the active target. Execution continues and this time the ignore_white_space and ig_white_space variables have the appropriate values to reach the patch. The synthesised input which reaches the patch is -w -y -- a b.

4.3 Implementation

KATCH consists of patch preprocessing scripts, the input selection subsystem, the augmented symbolic execution tool and a set of scripts which automatically iterate through all patches in a given set of program revisions. Most components operate at the level of LLVM bitcode, the intermediate language used by the popular LLVM compiler [56].

At a high level, a tester is only required to create a configuration file with details about the system to test, such as the repository address and the names of targeted
executable files. Figure 4.9 shows the actual file used for testing diffutils. Optionally, the tester can also provide scripts for compiling the system and running its regression suite. Otherwise, the default configure, make and make check commands are used, adapted for creating LLVM bitcode along with the native executables. Having this setup, the tester only needs to issue a command such as:

```sh
$ katch diffutils rev1 rev2
```

to test all diffutils revisions between rev1 and rev2. This script could be easily added to a continuous integration system to automatically test the last patch.

### 4.3.1 Patch Preprocessing

Patch preprocessing is implemented via two LLVM passes: the first one statically prunes non-executable lines by traversing the compiled program and using debug information to map LLVM instructions back to source code; a line is deemed non-executable if no instruction maps back to it. The second pass instruments the program to obtain test suite coverage information and determine which patch lines are executed by the test suite.

### 4.3.2 Input Selection

Input selection uses a combination of scripts and LLVM passes to instrument the program and analyse the execution of its test suite. In this phase, the original executables specified in the configuration file are replaced with wrapper scripts that invoke an instrumented copy of the corresponding binary. For each target, the instrumentation computes and outputs to a file the minimum distance from each test suite input, allowing the wrapper to determine which input gets closest to the target. This input is identified transparently by its sequence number, i.e. the number of times the program was executed by the test
suite so far. Subsequently, we run the test suite again and when reaching the target sequence number, we invoke KATCH instead of the regular executable.

The only assumption made by our approach is that the order of running the tests is deterministic, which holds in all cases we have looked at. While we could have used other solutions, we found that they are either not as general or they do not perform as well. For example, a different solution would be to record the program arguments used to get to the minimum distance instead of the sequence number and then run KATCH directly using these arguments. However, this approach fails when the test suite harness creates any non-trivial setup, not captured by the command line arguments, such as files, pipes or environment variables. Another approach is to directly run the symbolic execution component of KATCH on all test inputs. The downside is the larger overhead: symbolically interpreting the program is several orders of magnitude slower than native execution, while the instrumented programs have a comparable execution time to their native counterparts.

Instrumenting the program is performed through an LLVM pass which takes as input the original program and the current target. The pass uses a standard shortest path algorithm to statically compute the distance from each basic block to the target in the program’s interprocedural control flow graph and adds instrumentation to record which basic blocks are executed, to finally determine the executed basic block at the minimum distance from the target. It further uses the weakest precondition data-flow analysis described in §4.2.2 to refine this distance and inserts code in the executable to eliminate from the computation those branches which provably cannot lead to the target. To increase maintainability, most of the instrumentation is written in C++ as a set of helper functions which are then statically linked with the target program.
4.3.3 Symbolic Exploration

KATCH is implemented on top of the KLEE [16] open-source symbolic execution engine. KATCH starts by executing the program using the selected seed input to completion or until a predefined timeout expires. On this path it records all the branches that the program does not take. This includes branches whose associated branch condition is symbolic and feasible (i.e. has a data dependency on the program input and an input exists which executes the branch and the same path before it), symbolic and infeasible (has a data dependency on the program input but no input exists which executes the branch and the same path before it), and concrete (does not have a data dependency on the input). This provides more information for selecting the next path, as opposed to previous approaches which only considered the branches that depend on the symbolic input. The branches are then considered in order of increasing distance to the target as candidates for one of the techniques employed by KATCH: greedy exploration for feasible branches, and informed path regeneration or definition switching for infeasible branches. Once a suitable branch is found, the process repeats, executing a batch of instructions and re-evaluating the available paths.

We decided to use a batch of instructions, instead of a single one because this offers the advantage of generating more paths to choose from at the next iteration, with only a small time penalty, effectively providing a form of look-ahead. In certain scenarios, this compensates for the underestimation of the distance between two instructions, by permitting the execution of longer paths than dictated by the static estimation. Re-evaluating the available paths after each instruction is also possible but has an increased overhead and is more likely to get stuck in local optima. Our implementation currently uses batches of 10,000 LLVM instructions.

KATCH uses another optimisation to handle efficiently several common functions whose use is expensive in a symbolic execution context: the getopt family of functions, and strcmp. The getopt functions are helpers used by many programs to process command
line arguments. They work by allowing the programmer to write a simple specification of the arguments accepted by the program, thus moving the bulk of the command line parsing code inside the library functions. KATCH is aware of the getopt semantics and uses this information to speed up processing. More precisely, whenever the return value of getopt is a reaching definition, instead of recursively descending in the function code, it inspects the function argument corresponding to the specification of accepted command line arguments and directly determines the command line option needed to obtain the desired definition. The new argument is added to the command line and program execution restarts from the beginning.

The strcmp family of functions compares lexicographically two strings and returns -1, 0 or 1, depending on their ordering. These functions are often used to examine input and execute parts of the program logic if the input equals a certain predefined string. Virtually all strcmp implementations compare their arguments element-by-element, and return as soon as a mismatch is found. While this is desirable for efficiency reasons, the constraints thus generated only offer KATCH information on the first character that does not match. Should KATCH decide that the best path towards the target needs to satisfy the string equality, it would have to go through multiple iterations to make the strings equal, modifying a single character at a time. This has an adverse impact on performance, but even worse, some of the intermediate strings may not be valid inputs, which can cause execution to diverge from the original path in an informed path regeneration step. Our implementation solves this problem using a strcmp model whose return value is a conjunction of equalities, one for each position in its input strings, rather than just the first mismatch. Even though this model can only indicate equality or inequality, but not ordering, we found it sufficient for all the programs in our evaluation.

Figure 4.10 shows an example where KATCH uses the strcmp model. Reaching the patch requires that the name of one of the files passed on the command line is - (single dash), which is interpreted as standard input. The input selected from the test suite is in-4067 in2-4067, while an input which would exercise the patch is - in2-4067. KATCH correctly
detects that to reach the patch, the condition on line 1086 should be true, which without a `strcmp` model, means that the first character of the file name should be `-`. To make this change, it needs to apply an informed path regeneration step as we described in Section §4.2.3, thus going back just before the command line processing code inspected the argument. At that point, the input will be transformed into `-n-4067 in2-4067`. Note in particular, that the first argument is now invalid: `diff` interprets the leading dash followed by more characters as a program option. Because no such option exists, the program exits with an error messages instead of following the original path to the target. The `strcmp` model solves this problem by adding an additional constraint forcing the second character in the file name to equal the terminating `NUL`, thus directly creating the desired input `- in2-4067`.

### 4.3.4 Limitations

We discuss below the most significant limitations of our current prototype. Most importantly, we currently do not handle targets which are accessible only through function pointer calls that have not been exercised by the regression suite. Such indirect calls pose problems both during the static analysis when computing the closest input, and during dynamic exploration. The problems could be mitigated by including support for pointer analysis [1, Chapter 12] which KATCH currently does not offer.
Table 4.1: Application suites used to evaluate KATCH along with the number of programs in each of them, the number of patches inspected and the timespan in months.

<table>
<thead>
<tr>
<th>System</th>
<th>Programs</th>
<th>Size (ELOC)</th>
<th>Patches</th>
<th>Timespan (mo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>findutils</td>
<td>3</td>
<td>14,939</td>
<td>125</td>
<td>26</td>
</tr>
<tr>
<td>diffutils</td>
<td>4</td>
<td>42,930</td>
<td>175</td>
<td>30</td>
</tr>
<tr>
<td>binutils</td>
<td>12</td>
<td>68,830</td>
<td>181</td>
<td>16</td>
</tr>
</tbody>
</table>

Second, our current implementation of definition switching does not support aggregate data types such as structures and arrays and cannot be applied to branch conditions which include variables of these types. Finally, KLEE’s environment model is incomplete, e.g. it does not handle certain system calls.

### 4.4 Experimental Evaluation

For an objective evaluation of our technique, we have set ourselves the following two requirements. First, we have decided to do no cherry picking: once we have chosen a set of benchmark programs, rather than selecting the 10 (or 20, or 30) patches on which our technique works well, we included all the patches written over an extended period of time. Second, we have decided to allow a short timeout for our system, of no more than 15 minutes, which we believe is representative for the amount of time that can be dedicated in an automatic, possibly overnight, testing system.

We evaluated KATCH on nineteen programs from the GNU diffutils, GNU binutils and GNU findutils application suites, summarized in Table 4.1. These are all mature and widely used programs, installed on virtually all UNIX-based distributions.

**GNU findutils** is a collection of three programs, `find`, `xargs` and `locate`. They are smaller in size than the other two benchmarks, having a combined 14,939 executable lines of code (ELOC)\(^2\) in the tools themselves, and include additional portions of code from

\(^2\)We report the number of ELOC in the latest version tested, measured using `cloc` (http://cloc.sourceforge.net).
gnulib, which totals more than 280,000 ELOC at the latest revision that we inspected. We examined the 125 patches written in the two years and two months period between November 2010 and January 2013.

**GNU diffutils** comprises four programs, `diff`, `sdiff`, `diff3` and `cmp`. They are of medium size, with 42,930 ELOC in the tools themselves, and include additional portions of code from gnulib, similarly to findutils. We have analysed all the 175 patches written during the 2.5 years between November 2009 and May 2012.

**GNU binutils** includes a variety of programs out of which, due to time constraints, we selected the twelve assorted binary utilities from the binutils folder (`addr2line`, `ar`, `cxxfilt`, `elfedit`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings` and `strip`). They contain 68,830 ELOC, and use the statically linked libraries `libbfd`, `libopcodes` and `libiberty`, which total over 800,000 ELOC. Because of the more accelerated development pace in binutils, we examined a shorter 1 year 4 months period between April 2011 and August 2012, in which 181 patches were added to the binutils directory.

We set a short timeout of ten minutes per target for findutils and diffutils and a timeout of fifteen minutes for the larger binutils programs. We used a four-core Intel Xeon E3-1280 machine with 16 GB of RAM, running a 64-bit Fedora 16 system. As an extra safety check, we verified that all inputs generated by KATCH execute the corresponding patch code on the natively compiled programs, using gcov for coverage measurement.

Our tool and results\(^3\) have been successfully evaluated by the ESEC/FSE 2013 artifact evaluation committee and obtained a Distinguished Artifact award.

### 4.4.1 Coverage Improvement

As a first measure of KATCH’s effectiveness, we looked at its ability to improve patch coverage. Because KATCH operates at the basic block level, we define patch coverage as

\(^3\)http://srg.doc.ic.ac.uk/projects/katch/preview.html
the number of executed basic blocks which contain statements added or modified by a patch over the total number of basic blocks which contain such statements.

The patches analysed contain altogether 9,873 textual lines of code. After processing these lines to remove non-executable statements and group related executable lines, we obtained 1,362 potential targets which are part of 122 patches. Upon manual inspection, we found that the rest of the patches only keep the build system up-to-date with the program dependencies, or make changes to the documentation or test suite. A total of 423 targets were already covered by the system’s test suite, leaving 939 targets for KATCH to analyse.

4This includes only lines in .c and .h files.
Table 4.2: Number of targets covered by the manual test suite, and the manual test suite plus KATCH.

<table>
<thead>
<tr>
<th>Program Suite</th>
<th>Targets</th>
<th>Covered Test</th>
<th>Covered Test + KATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>findutils</td>
<td>344</td>
<td>215 (63%)</td>
<td>300 (87%)</td>
</tr>
<tr>
<td>diffutils</td>
<td>166</td>
<td>58 (35%)</td>
<td>121 (73%)</td>
</tr>
<tr>
<td>binutils</td>
<td>852</td>
<td>150 (18%)</td>
<td>285 (33%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,362</strong></td>
<td><strong>423 (31%)</strong></td>
<td><strong>706 (52%)</strong></td>
</tr>
</tbody>
</table>

The first step performed by KATCH is to compute the minimum distance from the regression test inputs to each target. Figure 4.11 presents the distribution of the minimum distances, which also provides a rough estimate of the work that KATCH needs to do for each target. More than half of the targets have regression tests which get relatively close to the target, at a distance smaller than five. Just a small fraction of the targets are at a distance over 20, which are all contained in completely untested binutils features. The figure does not include 389 binutils targets accessible only through indirect function calls not exercised by the test suite, which are outside the current capabilities of KATCH.

Table 4.2 summarises the results obtained after applying KATCH to these 939 targets. The Targets column lists the total number of targets for each benchmark and the Covered column lists the number of targets covered by the regression test suite, respectively the regression test suite and KATCH. It can be seen that KATCH has automatically increased the findutils patch coverage from 63% to 87%, it more than doubled the diffutils patch coverage, and made a more modest improvement for binutils, while still discovering fourteen bugs (§4.4.2). Overall, the patch coverage was increased from 31% to 52% (covering 283 out of the 939 targets).

We analyse below the cases in which KATCH fails to reach the target, in order to illustrate its limitations. More than half of the cases are targets accessible only through indirect function calls never exercised by the test suite, which our current prototype does not handle (see §4.3.4).
Another large number of cases relate to complex or multiple guard conditions. To satisfy them, KATCH would need to alter the input structure or to have access to a richer test suite, containing different seed inputs. For example, many `binutils` targets are only executed when the input file contains specific sections, with an individually defined structure. When none of the test suite files contains such a section type, the targets are usually not covered because KATCH cannot synthesise a complete section from scratch in the allotted time.

A more subtle scenario involves data stored using variable-length encoding, which is often used by `binutils`. In this case, KATCH can easily change input values only as long as they would be encoded using the same length. Changing to a value with a different encoding length would require inserting or removing one or more bytes in the middle of the input, significantly increasing complexity by possibly affecting other parts of the input such as header offsets.

Therefore, KATCH works best when the seed input does not need to have its structure altered. This is an inherent limitation of symbolic execution, which does not treat the input structure (e.g. its size) symbolically. This limitation is mitigated as the test suite quality improves and the chances of finding a good seed input increase.

The fact that our definition switching analysis does not support aggregate data types (§4.3.4) also affects several targets. A smaller number of targets cannot be reached due to the incomplete environment model implemented in KLEE, such as unsupported system calls.

Finally, we also noticed that several targets were not covered because they correspond to unreachable code on our test system—e.g. are reachable only on operating systems which differentiate between text and binary files.

In addition to the overall coverage improvement, we also wanted to measure exactly the contribution of each heuristic used by KATCH. We therefore re-executed the same experiments, selectively disabling all possible combinations of heuristics (note that all heuristics
Table 4.3: Number of targets covered by different combinations of heuristics: greedy (G), greedy and informed path regeneration (G+IPR), greedy and definition switching (G+DS) and all (KATCH).

<table>
<thead>
<tr>
<th>Program Suite</th>
<th>G</th>
<th>G + IPR</th>
<th>G + DS</th>
<th>KATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>findutils</td>
<td>74</td>
<td>85</td>
<td>78</td>
<td>85</td>
</tr>
<tr>
<td>diffutils</td>
<td>25</td>
<td>29</td>
<td>49</td>
<td>63</td>
</tr>
<tr>
<td>binutils</td>
<td>70</td>
<td>121</td>
<td>76</td>
<td>135</td>
</tr>
<tr>
<td>Total</td>
<td>169</td>
<td>235</td>
<td>203</td>
<td>283</td>
</tr>
</tbody>
</table>

depend on greedy). Table 4.3 shows the results. It can be seen that the improvement brought by each heuristic varies from system to system. At one end of the spectrum diffutils covers 152% more targets when using all heuristics compared to greedy alone, while at the other end findutils sees only a 15% improvement. Overall, informed path regeneration and definition switching combined brought a 67% improvement.

We have also run our experiments using KLEE instead of KATCH, to see how well a pure dynamic symbolic execution approach performs. We ran KLEE for 30 minutes on each revision, and we used an appropriate set of symbolic arguments. The results were very poor, with only two targets covered in the smaller findutils programs.

### 4.4.2 Bugs Found

KATCH was also able to identify a total of fifteen distinct crash bugs. We could verify that thirteen of these are also present in the latest version and we reported them to the developers, providing automatically-generated inputs which trigger them. Eleven of the bugs were discovered as a direct consequence of KATCH’s goal to reach the target: six bugs are in the actual targets and are discovered as they are introduced, while the other five are discovered because a patch is applied in their vicinity. Reaching this patch then leads to the discovery of the fault as KATCH continues executing the program to completion.
One bug was found in `findutils`, and the rest were found in `binutils`, the largest and most complex of all three application suites. A manual analysis of the bugs revealed that they relate to the handling of unexpected inputs. Interestingly, `binutils` generally does a good job handling such situations, but in several cases, the checks performed are incomplete. An example is bug 15206 in `objdump`, a buffer overflow caused by improperly checked buffer bounds. The bug appears in revision `119e7b90`, shown in part in Figure 4.12. Line 251 reads the buffer size from the buffer itself and lines 391 and 392 rely on this size to iterate through the entire buffer. The overflow occurs if the size read does not match the allocated buffer size.

Another example is the `readelf` bug 15191, shown in Figure 4.13. This bug was detected in revision `b895e9d`, when code was added to conditionally execute several existing lines. None of the code shown was executed by the regression tests. Line 12238 was newly added, therefore KATCH used it as a target and eventually executed it. It then attempted to run the program until the end and reached the next line (12240) where it discovered an overflow when reading through the `external` pointer. We have not debugged the exact root cause of the bug ourselves, but we sent an input triggering the crash to the developers, who fixed it shortly.

### 4.5 Discussion and Conclusion

We have presented KATCH, an automated technique for testing software patches. Our approach relies on symbolic execution, augmented by several synergistic heuristics based on static and dynamic program analysis. We have applied KATCH to all the patches written for nineteen programs over a combined period of approximately six years, and have shown that our technique can find bugs and significantly increase patch coverage with only a few minutes per target.

---

5 http://sourceware.org/bugzilla/show_bug.cgi?id=15206
6 http://sourceware.org/bugzilla/show_bug.cgi?id=15191
binutils/dwarf.c

243 process_ext_line_op (unsigned char *data, int is_stmt)
...
251 len = read_leb128 (data, &bytes_read, 0);
252 data += bytes_read;
...
380 unsigned int rlen = len - bytes_read - 1;
...
391 for (; rlen; rlen--)
392 printf ("%02x", *data++);

Figure 4.12: Example showing a bug found by KATCH, introduced in binutils revision 119e7b90. The bug is triggered on line 392. The highlighted lines are part of the patch.

binutils/readelf.c

12232 while (external < (Elf_External_Note *) ((char *) pnotes + length))
12233 {
...
12238 if (!is_ia64_vms ())
12239 {
12240 inote.type = BYTE_GET (external->type);
12241 inote.namesz = BYTE_GET (external->namesz);

Figure 4.13: Example showing a bug found by KATCH, introduced in binutils revision b895e9d. The bug is triggered on line 12240. The highlighted line is part of the patch.

We have learned several lessons from this research. First, it has reminded us that achieving high patch coverage is hard, and that as a result most patches remain untested—e.g. for our benchmarks the manual patch coverage was a modest 31% overall.

Second, it has reinforced our belief that automatic techniques are able to increase patch coverage and find bugs in the process. On average, KATCH was able to increase patch coverage from 31% to 52%, while on the best performing benchmark (diffutils), it more than doubled it, from 35% to 73%. In addition, we found fifteen crash bugs in widely-used mature programs.
Finally, it has shown us that the state of the art needs more advances to reach the goal of fully automated testing of real patches: despite the increase in coverage and the bugs found, KATCH was still unable to cover most of the targets in the binutils programs. We hope our current results will act as a challenge to other researchers working in this area.
Chapter 5

ZESTI

5.1 Overview

KATCH leverages the regression test suite to generate new inputs that execute different parts of the program under test. However, bugs that hide in covered code (§3.3.4), are missed by this approach. We have created ZESTI to complement KATCH by targeting these bugs through symbolic execution along the program paths exercised by the regression tests, allowing slight divergences around sensitive operations.

The main insight used by ZESTI is that regression test suites exercise interesting program paths. Such test suites are often created by the programmers who wrote the application and benefit from deep knowledge of the program logic, or by dedicated QA teams which systematically evaluate the main program features and possible corner cases. Furthermore, regression tests often cover program paths that previously triggered bugs, which are more likely to contain further errors [64, 103]. For instance, while the visible symptoms of the offending bugs are fixed, it can happen that the root cause of the bugs is not; alternatively, slightly different executions could still trigger the same bug.

A common way to measure the quality of a test suite is code coverage. Testing methodologies often require creating test suites that achieve a certain level of line or branch
coverage, and many projects contain relatively high-coverage test suites: for instance, most applications that we analysed in Chapter 3 have manual test suites reaching over 60% line coverage.

Unfortunately, despite the effort invested in creating these manual regression suites, bugs still remain undetected in the code covered by the test inputs. First of all, note that line coverage can be misleading for quantifying the confidence in a system for two important reasons. First, executing an operation may or may not cause a violation depending on its arguments. For example accessing the \( i \)-th element of a vector is safe when \( i \) is within vector bounds but causes an error otherwise. Line coverage, however, considers the operation tested as soon as it is executed once. Second, code behaviour depends on the path used to reach it; an instruction can operate correctly when reached along one path but cause a violation along a slightly different path. These caveats also apply to other coverage metrics, such as branch coverage.

In this chapter, we propose to augment regression suites by using symbolic execution to (1) analyse instruction safety against all inputs that could exercise the instruction along the same paths (§5.2.1) and (2) carefully choose and explore slightly divergent paths from those executed by the regression suite (§5.2.2). Compared to standard regression suites, our approach tests the program on all possible inputs on the paths exercised by the regression suite and on a large number of neighbouring paths, without any additional developer effort. Compared to standard symbolic execution, the approach takes advantage of the effort spent creating these regression suites, to quickly guide symbolic execution along paths that exercise interesting program behaviours.

### 5.2 Zero-Effort Symbolic Test Improvement

This section describes the two main techniques used by ZESTI: improving regression suites with additional symbolic checks (§5.2.1), and exploring additional paths around sensitive operations (§5.2.2).
5.2.1 Thoroughly Testing Sensitive Operations

A standard regression test suite consists of multiple tests, each being an (input, expected output) pair. The test harness iterates through the tests and runs for each of them the target program with the given input and collects its output. ZESTI hooks into this process by interposing between the testing script and the tested program, gaining complete control over the system’s execution.

Similarly to [50], ZESTI replaces the program input with symbolic values and at the same time remembers the concrete input, which is used to drive program execution whenever a branch is encountered. While executing the program, path conditions are gathered and used to verify potentially buggy operations. For example, whenever the program accesses a symbolic memory location, ZESTI checks that the operation is safe for all inputs that satisfy the current path condition.

Consider the snippet of code in Figure 5.1. Function $f$ contains a bug: it accesses an invalid memory location when passed a negative argument. A test suite might call this function with different arguments and verify its behaviour, attempting to maximise a certain metric, e.g., line coverage. It can be easily noticed that choosing one value greater than 99 and one smaller than or equal to 99 exercises all instructions, branches and paths without necessarily finding the bug. On the other hand, our approach finds the bug whenever the function argument is smaller than 100: for such values, symbolic execution gathers the path constraint $x \leq 99$ on line 3, and then on line 5 checks whether there are any values for $x$ than can overflow the buffer $v$. More exactly, ZESTI checks whether the formula $x \leq 99 \Rightarrow (x \geq 0 \land x \leq 99)$ is valid and immediately finds a counterexample in the form of a negative integer assignment to $x$.

In order to be accepted by software developers, we strongly believe that ZESTI needs to work transparently. We envision ZESTI being used in a similar way in which memory debuggers such as Valgrind [66] or Purify [41] are employed today in conjunction with test suites to provide stronger guarantees. For example, many open-source programs
```c
int v[100];

void f(int x) {
  if (x > 99)
    x = 99;
  v[x] = 0;
}
```

Figure 5.1: Code snippet showcasing a bug missed by a test suite with 100% code coverage, e.g. $x=50$, $x=100$.

provide a simple way to integrate Valgrind into their regression test frameworks, with the user simply having to type “`make test-valgrind`” to execute the regression suite under Valgrind. We hope ZESTI will be used in a similar way, by simply typing a command like “`make test-zesti`”.

In other words, running an existing regression test suite under ZESTI should happen without user intervention. To accommodate all testing frameworks, ZESTI treats both the tests and the testing script as black boxes. It functions by renaming the original program and replacing it with a script that invokes the ZESTI interpreter, passing as arguments the original program and any command line arguments. ZESTI automatically detects several input classes, namely command-line arguments and files opened for reading, and treats them as sources of symbolic data. We found these two sources sufficient for our benchmarks; however, adding additional input sources is relatively straightforward.

The main downside of this approach is execution overhead. In particular, there are two main sources of overhead: first, the overhead of interpreting LLVM code. Second, the constraint solver overhead: however, note that unlike in regular symbolic execution, the constraint solver is invoked in ZESTI only to check sensitive operations.

---

1Because ZESTI is an extension of the KLEE symbolic execution engine, which operates on LLVM bitcode [56], users need to compile their code to LLVM in order to use ZESTI. However, this is not a fundamental limitation of our approach, which could be integrated within a symbolic execution framework that works directly on binaries.
Inputs: $MaxDist$, the maximum distance to search, 
$S$, the set of sensitive instructions, 
$P$, the set of divergence points 
$f$, the distance estimation function

1: for $D = 1$ to $MaxDist$
2:   for sensitive instructions $I \in S$
3:     if $\exists$ divergence point $J \in P$
4:         at distance $D$ from $I$
5:             symbolically execute program starting
6:             from $J$, without restriction to a
7:             single path, with depth bound $f(D)$

Figure 5.2: Algorithm used by ZESTI to explore additional paths.

5.2.2 Exploring Additional Paths Around Sensitive Operations

The version of ZESTI described thus far has the disadvantage of being highly dependent on the thoroughness of the regression test suite. While a quality test suite is expected to test all program features, it is likely that not all corner cases are taken into account. Our analysis of Coreutils (§5.4), a mature set of applications with a high quality test suite, showed that only one out of the ten bugs previously found via symbolic execution could be detected by the version of ZESTI described so far. As a result, we extended ZESTI to explore paths that slightly diverge from those executed by the regression suite, according to the likelihood they could trigger a bug.

To mitigate the path explosion problem, ZESTI carefully chooses divergent paths via two mechanisms: (1) it only diverges close to sensitive instructions, i.e instructions that might contain a bug, and (2) it chooses the divergence points in order of increasing distance from the sensitive instruction. The key idea behind this approach is to exercise sensitive instructions on slightly different paths, with the goal of triggering a bug if the respective instructions contain one. Choosing a close divergence point ensures that only a small effort is needed to reach the same instruction again.

ZESTI identifies sensitive instructions dynamically. As it executes the concrete program path, it keeps track of all instructions that might cause an error on alternative executions. We consider two types of sensitive instructions: memory accesses and divisions. We
treat all pointer dereferences as sensitive, while for divisions we only consider those with a symbolic denominator as sensitive. At the LLVM level, ZESTI treats as sensitive all memory accesses to symbolic addresses, as well as those (concrete or symbolic) memory accesses preceded by a GetElementPtr instruction, and all division and modulo operations with symbolic denominators. While we currently track only sensitive memory accesses and divisions, we could also extend the technique to other types of sensitive operations, such as assertions.

To comprehensively exercise the sensitive instructions with different inputs, ZESTI tries to follow alternative execution paths that reach these instructions. To this end, it identifies all points along the concrete path where execution can diverge, i.e. the branches depending on symbolic input. ZESTI then prioritises the divergence points in increasing order of distance from sensitive instructions and uses them as starting points for symbolic execution. Figure 5.2 outlines the strategy used by ZESTI. Line 1 goes through distances from 1 to a user-specified maximum and line 2 iterates through all sensitive instructions. If any divergence point is found at the current distance from the current instruction, it is used to start a depth-bounded symbolic execution run, with bound \( f(D) \). The function \( f \) should be a function that closely overestimates the distance between the divergence point and the sensitive instruction on an alternative path. A function which underestimates this distance will give an SE depth bound too small to reach the sensitive instruction, while a function that largely overestimates it would needlessly increase ZESTI’s overhead. (However, note that not all additional paths explored by ZESTI are guaranteed to reach the sensitive instruction.) We empirically found that a linear function works well, and in our experiments we used \( f(D) = 2D \).

As an optimisation, line 2 of the algorithm considers sensitive instructions in decreasing order of distance from the program start. This favours the exploration of deeper states first, on the premises that (1) deeper states are more interesting because they exhibit the functionality exercised by the test suite as opposed to the shallow states that are often related to command-line parsing or input validation, and (2) standard symbolic
execution is less likely to be able to reach those states in reasonable time due to path explosion.

Intuitively, the metric used by ZESTI to measure the distance between two execution points needs to estimate the effort required by symbolic execution to reach one point from the other. To this end, ZESTI defines the distance between two instructions in a way similar to KATCH, as the number of branches between the instructions. Different from KATCH, ZESTI only counts branches where inputs could allow execution to follow either side of the branch. This metric captures the number of points where the program could have taken a different path (and which ZESTI could explore), and is inherently insensitive to large blocks of code that use only concrete data. Note that KATCH and ZESTI use the instruction distance for different purposes: guiding, respectively bounding symbolic execution.

In practice, the optimal maximum distance (MaxDist in Figure 5.2) for which to run ZESTI is hard to determine. Using a small value may miss bugs, while using a large value may be too time-consuming and leave no time to execute the rest of the tests within the allocated time budget. Our approach to solve this problem is to allocate a certain time budget to the entire testing process and use an iterative deepening approach: conceptually, all the tests are first executed without exploring any additional paths, then up to distance 1, 2, 3, etc., until the time budget expires.

To illustrate ZESTI’s exploration of additional paths, consider again the code in Figure 5.1. In the previous section we showed how ZESTI finds the bug starting from a test that calls function f with an argument smaller than 100. We now show how it can find the bug for any argument value. For values smaller than 100, the previous analysis applies and the bug is found without having to explore divergent paths. Therefore, we only discuss arguments greater than or equal to 100. Figure 5.3 shows the same code, annotated by ZESTI, when executed using such an argument. While running the function, ZESTI records all the sensitive instructions (S), and divergence points (D) being executed.
<table>
<thead>
<tr>
<th>Depth</th>
<th>Code</th>
<th>InstrType</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int  v[100]; void f(int x) { if (x &gt; 99) x = 99; }</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>v[x] = 0;</td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 5.3: Code snippet showcasing an execution generated by an input $x > 99$, annotated by ZESTI. The Depth column records the distance from the start of the execution, and the InstrType column keeps track of divergence points (D) and sensitive instructions (S).

<table>
<thead>
<tr>
<th>Depth</th>
<th>Code</th>
<th>InstrType</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int  v[100]; void f(int x) { if (x &gt; 99) { if (x &gt; 199) return; x = 99; } }</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>v[x] = 0;</td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 5.4: Code snippet showcasing an execution generated by an input $99 < x \leq 199$, annotated by ZESTI. The Depth and InstrType columns have the same meaning as in Figure 5.3.

(InstrType field), and computes their distance from the start of the execution (Depth field).

After running the entire function, ZESTI looks for instructions labelled as sensitive located at distance 1 after a divergence point (i.e., the difference between their Depth fields is 1), and finds instruction $v[x] = 0$ with corresponding divergence point if $(x > 99)$. These steps correspond to lines 2 and 3 of Figure 5.2. ZESTI then starts bounded symbolic execution from D (line 4 of Figure 5.2). The new path discovered corresponds to an input that makes the code take the (empty) else branch at D, i.e. a value smaller than 100. On this path x is no longer set to 99 but is used directly to index v. When executing the sensitive instruction $v[x] = 0$, ZESTI checks whether a violation can occur based on
the current path condition, and finds that a negative function argument causes a memory violation.

To further illustrate ZESTI’s algorithm, we consider the slightly more complicated code snippet in Figure 5.4. The code contains an additional if statement that creates a new divergence point $D_2$. Assuming a test input between 100 and 199, the sensitive instruction is at distance 1 from divergence point $D_2$ and at distance 2 from $D_1$. Therefore, ZESTI first considers $D_2$, and explores its then path, which does not trigger the bug. Going further, it finds $D_1$ which leads to the bug as in the previous example.

5.2.3 Improving Efficiency by Discarding Test Cases

An interesting question is how sensitive ZESTI is to the program test suite. The time in which ZESTI finds a bug depends on three main factors: the number of tests that are run, the percentage of them that expose the bug, and the minimum distance at which the bug is found.

As discussed above, because the distance at which a certain test case exposes the bug is difficult to predict, ZESTI first checks the concrete execution path and then uses an iterative deepening approach to check divergent paths. Under this strategy, the only other parameter that ZESTI can vary is the number of test cases that are run. In the rest of this section, we provide a theoretical estimate of the probability of finding a bug if ZESTI runs only a randomly chosen fraction of the test suite. Section 5.4.3 evaluates this probability in practice.

Creating a sub-test suite out of the initial test suite by randomly picking tests is an instance of the urn model without replacement [49], i.e. the marbles (tests) are not replaced in the urn (initial test suite) once picked. Assuming there exists a bug, and that the bug can be exposed at some minimum distance, consider that the urn model has the following parameters: $N$ – the total number of tests, $m$ – the number of tests
which expose the bug at the minimum distance, and \( k \) – the number of tests picked. The probability distribution which characterises the number of successes (i.e. tests which find the bug at the minimum distance) in the sequence of \( k \) picks is called the \textit{hypergeometric distribution} [49].

In terms of this model, we are interested in the probability of having at least one success, which is \( 1 - P(\text{failure}) \), the probability of having only failures:

\[
P(\text{success}) = 1 - P(\text{failure}) = 1 - \left( \frac{N - m}{k} \right) / \left( \begin{pmatrix} N \\ k \end{pmatrix} \right)
\]

where the fraction denominator represents the total number of possible test combinations and the numerator represents the number of combinations which contain zero successes.

Figure 5.5 plots the probability of finding a bug using a subset of a hypothetical initial test suite of 100 test cases for three fractions of tests exposing the bug: 6%, 10% and 30%, which are representative for the programs that we analysed with \textsc{zesti} (see §5.4). As this graph shows, it is possible to discard a large fraction of the test suite while still finding the bug with high probability. For example, for a test suite of size 100, in order to find the bug with at least 90% probability, it is enough to run only 7 (when \( m=30\% \)), 20 (when \( m=10\% \)), and 32 tests (when \( m=6\% \)). If the minimum distance at which the bug is found is relatively large, discarding a large number of tests can have a big positive impact on \textsc{zesti}'s performance, without significantly lowering the probability of finding the bug. In Section 5.4.3 we show that our analysis holds in practice by examining the test suite characteristics of real programs.

### 5.3 Implementation

Like \textsc{katch}, \textsc{zesti} is integrated in the \textsc{klee} symbolic execution engine [16]; a user can choose whether to run \textsc{zesti} via a command line switch. When enabled, \textsc{zesti} intercepts all calls that create symbolic data, (e.g., read from a file), and records the
Figure 5.5: Probability to find a bug using a randomly picked fraction of an initial test suite of 100 test cases. The three lines show the probability considering that 6%, 10% and respectively 30% of the initial tests find the bug.

can take the concrete value of the variables in a shadow data structure. ZESTI also intercepts all calls made to KLEE’s constraint solver, via a custom concretising module inserted in KLEE’s solver chain between the front-end and the query optimisers. When enabled, this module replaces all symbolic variables in a query with their concrete values and evaluates the resulting concrete expression, obtaining a value that is then returned directly back to KLEE. This implementation allows enabling and disabling symbolic execution by disabling and respectively enabling ZESTI’s concretizing module. The module is always disabled before executing a sensitive operation such as a memory access and re-enabled afterwards. This permits checking sensitive operations symbolically while executing the rest of the program concretely.

In order to explore paths around sensitive instructions, ZESTI associates with each program state that is not on the concrete path a time-to-live (TTL) value which keeps track of how long this state continues to be executed before it is suspended. This mechanism allows executing states in any order and guarantees execution for the exact
desired distance. The TTL uses the same metric used to measure distances between program states, i.e. symbolic branch count. It is initialised with the distance for which the state has to be executed, and decremented whenever the state is forked as a result of a symbolic branch condition.

ZESTI also implements its own state prioritization algorithm based on a breadth-first traversal of the state space, consistent with the distance metric used. The algorithm is implemented as a searcher, a pluggable abstraction used by KLEE to encapsulate the prioritization logic. This approach decouples the search algorithm from the symbolic execution functionality and allows updating or replacing the implementation with ease.

5.4 Experimental Evaluation

This section covers the results obtained with ZESTI, describing our benchmarks and methodology (§5.4.1), bugs found (§5.4.2), and quantifying the test improvements and overhead of using ZESTI (§5.4.3).

5.4.1 Benchmarks

To evaluate ZESTI, we used three different software suites:

1) **GNU Coreutils 6.10**, a suite of commonly-used UNIX utilities such as `ls`, `cat` and `cp`. Coreutils consists of a total of 89 individual programs and has a comprehensive regression test suite totalling 393 individual tests obtaining overall 67.7% line coverage. We used the older 6.10 version in order to facilitate the comparison against KLEE, which was previously used to comprehensively check this version of Coreutils [16]. The largest Coreutils program (`ls`) has 1429 effective lines of code (ELOC) but also uses part of a monolithic library shared by all the utilities, making it hard to compute an accurate line count. We therefore employed the same
approach used by KLEE’s authors, of computing the number of LLVM instructions after compiler optimisations are applied (especially the dead code elimination pass). This yields 20,700 instructions for ls.\textsuperscript{2}

2) \texttt{libdwarf 20110612}, a popular open-source library for inspecting DWARF debug information in object files. \texttt{libdwarf} has 13,585 ELOC as reported by gcov and 31,547 LLVM instructions, as reported by KLEE. Its test suite consists of two parts: 88 manually-created tests and a larger number of automatically-generated tests obtained by exhaustively mixing common command-line arguments and input files, achieving in total 68.6\% line coverage.

3) \texttt{readelf 2.21.53}, a component of GNU binutils for examining ELF object files, included in most Linux distributions. \texttt{readelf} has 9,938 ELOC and 30,070 LLVM instructions, and comes with a small test suite of only seven tests obtaining 24\% line coverage. One reason we included this benchmark was to see how ZESTI performs with a weaker regression suite. The other was that both \texttt{libdwarf} and \texttt{readelf} need large inputs (executable files), which would make a pure symbolic execution choke. For example, executing \texttt{libdwarf} using KLEE and a relatively small, 512 byte input file consumes all available memory on our test machine within a few tens of minutes.

To test these programs, we imposed a per-test time limit dependent on program complexity: we chose 15 minutes for the \texttt{Coreutils} programs and 30 minutes for \texttt{libdwarf} and \texttt{readelf}. For \texttt{libdwarf}, we used the 88 manual tests and 12 of the automatically-generated ones, picked to increase the diversity of the tests’ command line arguments. We ran all \texttt{libdwarf} experiments on a 64bit Fedora 16 Xeon E3-1280 machine with 16GB of RAM, while the rest were performed on a 64bit Ubuntu 10.04 i5-650 machine with 8GB of RAM.

\textsuperscript{2}Line count and coverage information was obtained using gcov 4.4.3 and LLVM 2.9. Numbers can vary slightly between different versions.
5.4.2 Bugs Found

ZESTI found a total of 58 bugs, out of which 52 were previously unknown. The new bugs were reported to the maintainers and most of them have already been fixed by the time of this writing. Table 5.1 shows a summary of the bugs found by ZESTI, along with the distance from the concrete path and the depth at which they were found. We compute the depth as the number of visited symbolic branches from the program start where both sides could be explored, as this is a rough estimation of the effort required by standard symbolic execution to find the bug. If the same bug is discovered by two or more test cases we report the minimum distance and for equal distances the minimum depth. Both the minimum distance and depth are influenced by program inputs; it may be possible to reach the bugs by traversing fewer symbolic branches when using other inputs.

Table 5.1: Bugs found by ZESTI along with the distance (from the concrete test path) and the depth (from the program start) at which the bug was found. New bugs are in bold.

<table>
<thead>
<tr>
<th>Bug no.</th>
<th>Location</th>
<th>Distance</th>
<th>Min Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cut.c:267</td>
<td>0</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>printf.c:188</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>seq.c:215</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>paste.c:107</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>mkdir.c:192</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>mknod.c:169</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>mkfifo.c:117</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>md5sum.c:213</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>9</td>
<td>dwarf_form.c:458</td>
<td>2</td>
<td>491</td>
</tr>
<tr>
<td>10</td>
<td>dwarf_form.c:503</td>
<td>0</td>
<td>1229</td>
</tr>
<tr>
<td>Bug no.</td>
<td>Location</td>
<td>Distance</td>
<td>Min Depth</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------</td>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>11</td>
<td>dwarf_form.c:525</td>
<td>0</td>
<td>490</td>
</tr>
<tr>
<td>12</td>
<td>dwarf_elf_access.c:663</td>
<td>0</td>
<td>382</td>
</tr>
<tr>
<td>13</td>
<td>dwarf_elf_access.c:664</td>
<td>0</td>
<td>383</td>
</tr>
<tr>
<td>14</td>
<td>dwarf_arange.c:160</td>
<td>0</td>
<td>319</td>
</tr>
<tr>
<td>15</td>
<td>dwarf_arange.c:179</td>
<td>0</td>
<td>321</td>
</tr>
<tr>
<td>16</td>
<td>dwarf_util.c:90</td>
<td>0</td>
<td>746</td>
</tr>
<tr>
<td>17</td>
<td>dwarf_util.c:396</td>
<td>0</td>
<td>923</td>
</tr>
<tr>
<td>18</td>
<td>dwarf_elf_access.c:640</td>
<td>0</td>
<td>495</td>
</tr>
<tr>
<td>19</td>
<td>dwarf_print_lines.c:385</td>
<td>0</td>
<td>514</td>
</tr>
<tr>
<td>20</td>
<td>dwarf_global.c:305</td>
<td>0</td>
<td>2057</td>
</tr>
<tr>
<td>21</td>
<td>dwarf_global.c:239</td>
<td>0</td>
<td>1508</td>
</tr>
<tr>
<td>22</td>
<td>dwarf_global.c:267</td>
<td>2</td>
<td>400</td>
</tr>
<tr>
<td>23</td>
<td>dwarf_leb.c:58</td>
<td>0</td>
<td>396</td>
</tr>
<tr>
<td>24</td>
<td>dwarf_leb.c:62</td>
<td>1</td>
<td>650</td>
</tr>
<tr>
<td>25</td>
<td>dwarf_leb.c:69</td>
<td>1</td>
<td>650</td>
</tr>
<tr>
<td>26</td>
<td>dwarf_leb.c:128</td>
<td>1</td>
<td>650</td>
</tr>
<tr>
<td>27</td>
<td>esb.c:117</td>
<td>0</td>
<td>1248</td>
</tr>
<tr>
<td>28</td>
<td>print_die.c:1523</td>
<td>0</td>
<td>1292</td>
</tr>
<tr>
<td>29</td>
<td>dwarf_util.c:116</td>
<td>0</td>
<td>488</td>
</tr>
<tr>
<td>30</td>
<td>dwarf_util.c:363</td>
<td>0</td>
<td>1248</td>
</tr>
<tr>
<td>31</td>
<td>dwarf_util.c:418</td>
<td>0</td>
<td>498</td>
</tr>
<tr>
<td>32</td>
<td>dwarf_query.c:325</td>
<td>0</td>
<td>648</td>
</tr>
<tr>
<td>33</td>
<td>dwarf_abbrev.c:119</td>
<td>0</td>
<td>543</td>
</tr>
<tr>
<td>34</td>
<td>dwarf_frame2.c:936</td>
<td>1</td>
<td>376</td>
</tr>
<tr>
<td>35</td>
<td>dwarf_frame2.c:948</td>
<td>0</td>
<td>389</td>
</tr>
<tr>
<td>36-48</td>
<td>dwarf_line.c:*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

*Bugs were found at 13 different locations in dwarf_line.c. For brevity we omit the details.*
We describe below three representative errors found by ZESTI, and then compare its bug-finding ability against standard symbolic execution.

**cut case study:** The bug found in the `cut` utility is a memory access violation. The test leading to its discovery uses the command line arguments `-c3-5,6- --output-d=:` `file.inp`. The `-c` argument specifies two ranges, from the 3rd to the 5th character and from the 6th character to the end of the line. Internally, `cut` allocates a buffer that is later indexed by the range endpoints. Its size is computed as the maximum of the right endpoints across all ranges. However, in this case, the ranges unbounded to the right are incorrectly not considered in the computation. Therefore the value 6 is used to index a (zero-based) vector of only 6 elements. However, because the `cut` implementation uses a bitvector, allocations are inherently done in chunks of 8 elements and the bug is not triggered by the test input (and thus a tool such as Valgrind could not find it).
Table 5.2: A one byte corruption at offset 0x1073 in a *libdwarf* test file, which causes a division by zero.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Original</th>
<th>Buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>7F 45 4C 46</td>
<td>7F 45 4C 46</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1070</td>
<td>00 00 00 04</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2024</td>
<td>69 74 00</td>
<td>69 74 00</td>
</tr>
</tbody>
</table>

However, ZESTI detects the problem by deriving a new input which triggers the bug, namely `-c3-5,8- --output-d=: file.inp`.

**libdwarf case study:** One of the bugs found in *libdwarf* is a division by zero, caused by improper handling of debug information data. Before reading the debug *ranges* section, *libdwarf* computes the size of each entry by looking at two fields in the executable file: the address size and the segment size. The entry size is computed using the formula:

\[
\text{entry\_size} = 2 \times \text{address\_size} + \text{segment\_size}.
\]

A check is then made to ensure that the section size is a multiple of the entry size via a modulo operation, which causes an exception when the entry size equals zero.

Table 5.2 shows the input generated by ZESTI by changing one byte in the original test file. The byte corresponds to the address size, which is changed from 4 to 0 (the segment size is already 0). The new file causes the division by zero when passed to *libdwarf*.

One advantage of ZESTI over standard symbolic execution is that it can generate well-formed inputs. While symbolic execution can only use the current path constraints to generate an input, leaving all unconstrained data to a default value, ZESTI creates an input that matches as close as possible the test data, while still reproducing the bug. The feedback to our bug reports indicates that this approach creates inputs that are easier to understand by programmers.

**printf case study:** ZESTI found a previously unknown bug in the *printf* program, a utility that prints formatted text in a similar fashion to the *printf* libc function.
The bug was found at distance 1, i.e., ZESTI had to flip the outcome of one branch in order to trigger it. The bug resides in a program feature that interprets a character as its integer ASCII code if preceded by a single or double quote. The implementation incorrectly assumes that all quotes are followed by at least one character; when a lone quote is provided as input, an off-by-one memory access is performed. ZESTI infers from the `printf %c x` test, the input `printf %d '`, which triggers the bug.

**Comparison with standard symbolic execution:** In terms of bug-finding capabilities, ZESTI and KLEE enjoy different advantages. On the one hand, ZESTI is able to avoid certain scalability problems that symbolic execution is facing, by using the paths executed by the regression suite to reach interesting program states. For example, ZESTI was able to find forty bugs in libdwarf and ten in readelf, while KLEE was not able to find any of them, because it ‘got lost’ in the large program state space, ending up consuming all available memory on our test machine. The large depth at which the libdwarf and readelf bugs are found in the symbolic state tree (Min Depth column in Table 5.1) shows that symbolic execution needs to search through a significantly larger number of states. For example, to find a bug at depth 100 requires searching through roughly $2^{90}$ times more states than it does for a bug at depth 10.

On the other hand, four of the bugs found by KLEE were not detected by ZESTI, showing ZESTI’s limitations. One of the bugs, found in the tac utility, is only triggered when providing more than one input file to the program. Because none of the tests do so, the buggy code is never executed in the inconsistent state. The two bugs found by KLEE in ptx are missed because the regression suite does not contain any tests for this program. Finally, the bug in the pr utility was not found due to the highly solver-intensive test inputs, which were consuming all the allocated time budget on the concrete path, not allowing ZESTI to explore beyond it in the allocated time budget. However, note that the input specifications used by KLEE could have been used to create seed inputs that could have allowed ZESTI to find these bugs.
5.4.3 Symbolic Bug Checks and Performance Overhead

Symbolic bug checks: One measure of ZESTI’s effectiveness is the number of symbolic checks (in our case memory access checks) made when running a regression suite. Figure 5.6 shows the number of total and unique checks performed for each program in the Coreutils suite when running ZESTI on the regression suite with distance 0 (i.e., with no additional paths explored) and a timeout of two minutes per program execution. Uniqueness was determined solely through the line of code that triggers the check.

Figure 5.6 shows 46 bars, one for each Coreutils application in which ZESTI performed symbolic checks while running the regression suite. The rest of the Coreutils programs do not provide any opportunities for such checks because they either are too simple (e.g., yes), do not take user input, or do not use it to access memory, (e.g., id, uname). This does not represent a limitation of ZESTI but instead shows that not all programs are susceptible to memory access bugs.
Overhead of \textit{ZESTI’s checks}: Under the same setup, we also measured the time taken by \textit{ZESTI} to run each test in the regression suite. To compute \textit{ZESTI}’s overhead, we use as baseline \textit{KLEE} as an interpreter only, i.e. without any symbolic data. Because no symbolic data is introduced, \textit{KLEE} uses its system call models, object management system and the same internal program representation as in symbolic execution mode but follows only one execution path and does not use the constraint solver.

To eliminate potential inconsistencies, we only consider tests that complete successfully, as reported by the regression suite. This eliminates 21 tests that result in \textit{ZESTI} timeouts and a small number of early program exits due to imperfections in \texttt{uClibc} or \textit{KLEE}’s models, which would otherwise add noise to our experiments.

The results are presented in Figure 5.7, which shows one pair of bars for each program execution: one for the time taken by the interpreter, and one for the time taken by \textit{ZESTI}. The times are sorted by interpreter time. The last two tests, not completely shown, take 250 seconds to terminate under the interpreter and have less than 1\% overhead under
Table 5.3: Bug distance distribution for the bugs found by ZESTI in Coreutils and the number of tests available for each programs (#T). The percentage of tests that find each bug at the minimum distance is in bold. The Not found value corresponds to not finding the bug in 15 minutes (60 minutes for md5sum).

<table>
<thead>
<tr>
<th>#T</th>
<th>cut</th>
<th>printf</th>
<th>md5sum</th>
<th>mkdir</th>
<th>mknod</th>
<th>mkfifo</th>
<th>paste</th>
<th>seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>163</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>17</td>
<td></td>
<td>44</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>33</td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Found</td>
<td>90.8%</td>
<td>53.0%</td>
<td>86.4%</td>
<td>79.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ZESTI. We see that for most tests, the execution times are virtually identical for KLEE and ZESTI. However, there are several executions for which ZESTI takes significantly more time, due to the constraint solver queries that it issues while making the symbolic checks. Finally, note that the interpreter time adds significant overhead on top of native execution (which for Coreutils usually takes only milliseconds per program execution), and one way to improve ZESTI’s performance is to speed up the interpreter (which in KLEE is not optimised, because in standard symbolic execution it is rarely a bottleneck).

**Effect of discarding test cases:** Table 5.3 shows the size of the test suite for each application from the Coreutils utilities in which ZESTI found a bug (#T), and the distribution of all available tests for each program, relative to the distance at which they exposed the bug, from zero to twelve (D0–D12). The Not found value corresponds to not finding the bug in 15 minutes (60 minutes for md5sum). For example, 9.2% of the 163 cut tests allow ZESTI to find the bug at distance 0, while the rest of 90.8% do not expose the bug in 15 minutes.
Based on the information in Table 5.3 and using the formula presented in Section 5.2.3, we plotted in Figure 5.8 the probability of finding the bug at the minimum distance for each of these applications, relative to the size of a randomly chosen sub-test suite. It can be noticed that the worst scenarios correspond to the `printf` and `md5sum` programs, where more than half of the tests are needed to have at least 90% confidence in finding the bug. For the rest of the programs, a confidence of at least 90% can be achieved by using roughly one third (or fewer) of the tests. This indicates that in practice, it might be possible to improve ZESTI’s efficiency—without significantly affecting the probability of finding a bug—by randomly discarding a large part of the test suite. `libdwarf`’s test suite corroborates these results, while `readelf` has a test suite too small to be considered for this analysis.
5.5 Conclusion

We have presented ZESTI, a lightweight symbolic execution-based tool that automatically improves regression test suites with the ability to reason about all possible input values on paths executed by the test suite, as well as explore additional paths around sensitive instructions. ZESTI approaches testing from two different angles: first, it significantly broadens the number of bug checks performed by a regression suite and therefore the number of bugs found. Second, by using the regression suites as a starting point, ZESTI provides an effective solution for guiding the exploration of the symbolic search space.

As a result of these features, we were able to successfully apply ZESTI to three popular software systems—GNU Coreutils, readelf, and libdwarf—where it found 52 previously unknown errors, including two in the Coreutils suite, which was previously checked thoroughly via symbolic execution.

We believe our technique can be effectively integrated with existing regression suites, and could help bridge the gap between standard regression testing and symbolic execution, by providing a lightweight, incremental way of combining the two techniques.

We are making our tool available as open-source at http://srg.doc.ic.ac.uk/projects/zesti.
Chapter 6

Related Work

The techniques behind KATCH and ZESTI can be analysed from two perspectives: they aim at scaling symbolic execution using regression tests while at the same time they improve the effectiveness of regression testing using symbolic execution. We structure this section by examining the two perspectives in turn, and finally discussing other related techniques.

6.1 Symbolic Execution

Both KATCH and ZESTI are based on symbolic execution: KATCH uses it to synthesise inputs which execute a given program location, while ZESTI uses it to perform thorough safety checks, guided by regression test inputs. We discuss each technique in turn.

KATCH

Synthesising inputs which cover a target is an essential problem in test generation and debugging and has been addressed through a variety of techniques, including symbolic execution, dependence analysis, iterative relaxation and search-based software testing, among others [7, 26, 38, 86, 94, 96, 102].
Borrowing ideas from the state of the art in these areas, katch treats the task as an optimisation problem for which it tries to compute an optimal solution. More exactly, it attempts to minimise the control-flow distance between the currently executing state and the target. For any reachable target, the minimum distance is zero, achieved when executing it. Guided by this distance and various heuristics based on program analysis, katch explores new paths using symbolic execution seeded with existing test inputs.

Our technique fits within the paradigms of longitudinal and differential program analysis [67,92], in which the testing effort is directed toward the parts of a program that have changed from one version to the next, i.e. software patches. In particular, differential symbolic execution [71] introduces a general framework for using symbolic execution to compute the behavioural characterisation of a program change, and discusses several applications, including regression test generation.

The work most closely related to katch is that on directed symbolic execution. Xu and Rothermel introduced directed test suite augmentation [96], in which existing test suites are combined with dynamic symbolic execution to exercise uncovered branches in a patch. The technique is similar to the greedy step in katch, without any of our additional analyses. Given an uncovered branch \( s_i \rightarrow d_i \) and a test case that reaches \( s_i \), the technique uses dynamic symbolic execution to try to generate a test case that executes the branch, and then repeats this process until no more branches can be covered. The technique depends on the availability of tests that reach the source node of an uncovered branch and do not constrain the input to take only the already covered branch, while our approach tries to actively steer execution toward the patch by combining the greedy exploration with informed path regeneration techniques and definition switching.

Researchers have proposed several improvements to this technique: eXpress [86] prunes CFG branches which provably do not lead to the patch; directed symbolic execution [58] introduces call-chain-backward symbolic execution as a guiding technique for symbolic execution and statically-directed test generation [7] uses the size of the target’s backward
slice reachable from the current state as an estimate for the likelihood of reaching it. Directed incremental symbolic execution [72] is a related technique which improves the efficiency of symbolic execution when having to analyse only the differences between two program versions. It can dynamically prune program paths which exercise the same behaviours in two program versions, and could be combined with KATCH if multiple behaviourally different inputs which cover the patch are desired.

While it is difficult to accurately compare these techniques with KATCH or among each other, we believe that KATCH improves upon previous work in several ways. First, by using the definition switching heuristic, KATCH takes into account more than the currently explored set of paths—and reasoning about unexecuted statements is critical for reaching certain targets. Second, informed path regeneration uses a “surgical” approach to reaching previously infeasible states by making changes to exactly those variables involved in infeasible branch conditions. Third, our evaluation is performed on significantly more patches than in prior work, which gives a better insight into the strengths and limitations of such a technique. Finally, we believe KATCH could be combined with some of these prior approaches, e.g. it could dynamically prune paths that are shown not to lead to the target.

ZESTI

ZESTI is designed to be a testing tool that integrates seamlessly in the software development life cycle, similar in spirit to Valgrind, but more effective through the use of symbolic reasoning. The idea of augmenting concrete executions with the ability to reason symbolically about potential violations was first proposed in [55], which introduces a technique that keeps track of lower and upper bounds of integer variables, and of the NUL character in strings. Based on this information, it can flag bugs such as buffer overflows and incorrect uses of libc string functions. The technique can only reason about limited types of constraints, and does not explore any additional paths.
Two related approaches, which lend themselves to regression test seeding are concolic testing [35, 77] and whitebox fuzz testing [36]. Concolic testing starts from the path executed by a concrete input and then systematically flips the truth value of the branch conditions collected on that path. Previous research has shown that the coverage and bug-finding abilities of concolic testing can be improved by combining it with random testing [59] or with well-formed inputs [36], and the effectiveness of fault-localization can be increased by aiming to maximize the similarity with the path constraints of faulty executions [4]. Combining concolic execution with manual test suites was first proposed in [50], where it was augmented by assertion hoisting in order to increase the number of bug checks, and then explored in [95], in which it was compared against a genetic algorithm test augmentation technique. ZESTI extends previous work by proposing techniques that explore paths around potentially dangerous instructions executed by the regression suite, by providing an analysis of the sensitivity of this approach to the quality of the test suite, and by presenting a thorough evaluation on real and complete regression suites of several popular applications.

By identifying potentially dangerous operations and performing depth-bounded symbolic execution around them, ZESTI limits the program state space that is symbolically explored, making our approach scalable. Other solutions for limiting or prioritising the symbolic program exploration use orthogonal approaches. For example, directed symbolic execution methods [7, 21, 58, 72, 86, 96] limit exploration by trying to target specific parts of a program, as discussed above. Redundant state detection [11, 14] aims at identifying and discarding states that can be proven to be redundant, e.g. they would execute again a previously explored path. Dynamic state merging [54] studies the opportunities and trade-offs of combining execution states that have different constraints and reach the same program location. Veritesting [5] combines dynamic and static symbolic execution, decreasing the number of paths explored at the expense of more complex constraints. Compositional dynamic test generation [34] uses function summaries to speed up symbolic execution by reusing them when possible instead of
actually executing the function code. The technique was later refined to compute the summaries on demand [2]. Selective symbolic execution [20] minimises the code which needs to be executed symbolically by separating the code of the system under test in a symbolic and concrete part, and seamlessly transitioning back and forth between symbolic and concrete execution.

6.2 Regression Testing

Research on improving regression testing generally falls under four main categories: (1) finding redundant tests [40], (2) ordering tests for finding defects earlier [84], (3) selectively running only relevant tests on new program versions [75] and (4) enhancing a system’s test suite as the system evolves [9, 39, 76, 78, 86, 100]. The first three categories address the problem of high resource usage in regression testing, which usually occurs in mature systems which have accumulated a large number of tests during their lifetime. They are orthogonal and can be combined with our techniques when targeting a particular part of the program.

The state-of-the-art for enhancing a system’s test suite combines control- and data-dependence chain analysis and partial symbolic execution to identify tests that are likely to exercise the effects of changes to a program [76]. This approach targets a more complex adequacy criterion than statement coverage by requiring the generated tests not only to execute the change, but also infect the program state and propagate to the output. Making this approach tractable requires a reasonably small set of differences between program versions and a depth-bounded analysis on dependence chains. Another technique [78] for achieving the same adequacy criterion uses a search-based approach, relying on evolutionary algorithms and a fitness function based on structural coverage, object distance and control-flow distance.
While our approach could be used for test augmentation, we see ZESTI and KATCH primarily as bug-finding techniques that can increase the effectiveness of regression suites by combining them with symbolic execution, following the manner in which dynamic execution tools such as Valgrind are often integrated with existing test suites.

6.3 Other Related Techniques

Search-based software testing (SBST) [29, 62, 94, 99] is an area of research which applies the principles of search-based software engineering to testing. Generally speaking, SBST formulates a testing task as an optimisation problem, and finds an approximate solution by exploring the program state space using a metaheuristic algorithm in conjunction with a fitness metric. For example, EXYST [37] uses genetic algorithms to automatically generate system-level tests and find bugs in GUI applications, by optimising a code coverage-based fitness criterion, without triggering false positives, and requiring only lightweight instrumentation.

KATCH and ZESTI share characteristics with SBST. First, our notion of estimated distance is similar to that of fitness in SBST. Second, the idea of reusing existing test cases has also been successfully employed in SBST [29, 99]. However, unlike SBST, we use a set of specialised heuristics which operate based on symbolic constraints and we search for an exact solution. Researchers have already started to develop techniques that combine SBST and dynamic symbolic execution [6, 28, 30, 60, 93] and similar ideas could be applied for the purpose of patch testing.

Fuzzing is a testing technique that involves providing random data to a program while monitoring its execution for abnormal behaviour. In its simplest form—black-box fuzzing—it consists of sending random bits to a program, either as command line arguments, input files, environment variables, network packets etc., but assuming no knowledge of either the input or the program. Miller et al. [63] were among the first to
apply this technique to actual programs. However, this basic approach is inadequate for programs whose valid inputs are scattered throughout the input space, e.g. compilers, or for programs whose inputs follow a very specific pattern, e.g. a checksum for a network packet or a *magic* field that identifies the input type. In these cases, a completely random input will almost always be invalid and thus only exercise the program’s input validation code. This problem has been addressed by leveraging information regarding the input or the program structure. Solutions which leverage only the input structure fall in the gray-box fuzzing category: mutating valid inputs [68, 85], using grammar-based fuzzing [98] and a combination of the two [44]. Such a combination is similar to KATCH, in that it generates new program inputs starting from regression tests. However, while KATCH uses the program structure and symbolic execution to reach a specific target, fuzzing randomly creates new inputs using a grammar which has to be provided by the user. Finally, solutions which take advantage of the program structure [32, 36] systematically explore different paths and find boundary conditions, similarly in principle to concolic execution. This approach, dubbed white-box fuzzing, can, similarly to KATCH and ZESTI, reason more thoroughly about the program’s behaviour and generally finds deeper bugs at the cost of increased complexity and overhead introduced by constraint solving and the use of program instrumentation. Given the same time budget, black-box fuzzing can generate significantly more, but potentially less diverse inputs, while gray-box fuzzing is a middle-ground between the two.

Research on **automatic generation of filters** based on vulnerability signatures [12, 22] addresses the problem of executing a specific target from a different angle. Given an existing input which exploits a program vulnerability, the goal is to infer the entire class of inputs that lead to that vulnerability. Similarly, generating inputs with the same effect as a crashing input but which do not leak sensitive data, is used in bug reporting to preserve user privacy [19]. In the context of automated debugging, execution synthesis [102] and BugRedux [48] attempt to solve a similar problem: generating an input or a path starting from a set of ‘waypoints’ through which execution has to pass.
Chapter 7

Conclusion

7.1 Summary of Thesis Achievements

We briefly reiterate the main contributions of this thesis:

1) We have shown that the effectiveness of standard regression tests can be automatically improved using symbolic execution-based techniques.

2) We have shown through an empirical study the suitability of these techniques to real applications and have built a system for automatic extraction of static and dynamic software evolution metrics from software repositories.

3) We have presented and evaluated KATCH, a testing technique that combines symbolic execution with several novel heuristics based on program analysis that effectively exploit the program structure and existing program inputs to reach specific program points.

4) We have introduced and demonstrated ZESTI, a technique complementary to KATCH, for reasoning about all possible input values on the paths executed by the regression suite and for thoroughly exploring additional paths around sensitive instructions such as dangerous memory accesses.
7.2 Applications

The ultimate goal of our project is to make KATCH and ZESTI accessible to testers through a simple interface. Continuous integration systems such as BuildBot\(^1\), Jenkins\(^2\) or Travis CI\(^3\) are widely adopted in industry for automating tasks such as building and regression testing. We envision KATCH and ZESTI as “plug-ins” to such a system, working behind the scenes, requiring little configuration and minimally disrupting the existing development workflows.

Using KATCH requires providing only a simple configuration file, a build script and a regression test suite, which already exist in most systems. When invoked, KATCH automatically detects the latest source code changes, runs the existing regression tests, checks their patch coverage, and finally creates new inputs based on the existing tests to improve the coverage and detect more errors.

ZESTI on the other hand, plugs directly into this regression test suite. The regular tests are usually executed from the command line via a `make check` or `make test` command. Some systems also allow running the regression suite through a memory debugger such as Valgrind, using a simple command such as `make test-valgrind`, in order to catch invalid memory operations which do not result in observable errors. We envision exposing ZESTI through a similar command, e.g. `make test-zesti`, which would enable all the additional checks it makes.

The approach implemented by KATCH and ZESTI has several advantages: (1) it does not require changes to the program source code or to the regression tests, as both systems interpose transparently between the test harness and the actual program; (2) it takes advantage of the effort put in the original test cases, as they are reused to drive symbolic execution; and (3) for each bug found, an input that reproduces the bug is generated;

\(^1\)http://buildbot.net
\(^2\)http://jenkins-ci.org
\(^3\)https://travis-ci.org/
furthermore, to help developers understand the bug, this input is kept as similar as possible to the initial test input used to drive exploration.

Unlike standalone symbolic execution, KATCH and ZESTI eliminate the guesswork involved in setting up symbolic data. In particular, choosing the appropriate number and size of symbolic inputs is non-trivial: on the one hand, small inputs may miss bugs, while on the other hand large inputs can significantly increase execution overhead by generating very expensive constraint solving queries, or by causing symbolic execution to spend most of its time in uninteresting parts of code. For example, while analysing the two Coreutils bugs detected by ZESTI but missed by KLEE, we found that carefully tuning the symbolic input size allows standard symbolic execution to find them. Surprisingly, one of the bugs can be found only with larger inputs, while the other only with smaller ones. The cut bug can be found only when using two long arguments—but the original KLEE tests were using a single long argument—and the printf bug can only be found with an argument of size one—but the original KLEE tests used a larger size. Good regression test suites invoke applications with representative arguments, in number, size and value, which KATCH and ZESTI successfully exploit.

Another problem of symbolic execution is that it can get stuck in uninteresting parts of the code, such as input parsing code, and therefore miss interesting “deep paths.” KATCH and ZESTI solve this problem by first executing the entire program along the paths induced by the regression tests, and then exploring adjacent program parts symbolically driven by a control flow-based metric.

The main disadvantage of this approach is that it can take significantly more time than natively executing the regression tests. However, our empirical analysis showed that a good regression suite allows finding bugs close to the concrete execution path, thus minimising the time spent symbolically executing the program. Furthermore, KATCH and ZESTI can be tuned to specific time budgets through various configurable settings which limit the exploration via timeouts (per-instruction, per-solver query, per-branch...
from the concrete path) or by disallowing execution beyond a certain distance. Finally, if necessary, developers can run only a part of the test suite under Zesti, often without significantly lowering the probability of finding bugs.

One problem that we observed in the pr utility from the Coreutils suite is a very expensive—in terms of symbolic checks—concrete path. This prevents Zesti from exploring paths which diverge from the test suite in the given time budget. In the future, we plan to incorporate in both KATCH and ZESTI heuristics for adaptively skipping memory access and branch feasibility checks. For example, the tools could consider an instruction correct after it has been checked for a predetermined number of times on different paths. Similarly, they would limit the number of branches spawned by the same static instruction. We intend to further study the trade-offs that we can make in the symbolic state-space exploration.
Bibliography


24th International Symposium on Software Reliability Engineering (ISSRE’13), November 2013.


