Filtering overfitted automatically-generated patches by using automated test generation

by

Alexey Zhikhartsev

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

Statement of contributions

This thesis is based on the following joint work:

Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

The contributions of the author of this thesis include:

- Designing and conducting the experiments for filtering overfitted automatically-generated patches by using fuzz-tests and the crash oracle, including implementing the software necessary for the experiments
- Conducting a case study to investigate *how* automatically-generated tests filter overfitted automatically-generated patches
- Performing the relevance analysis to discover *weakly relevant* test cases among the generated fuzz-tests; forming definitions of weak relevance for benchmark defects
- Exploring directions of future work and conducting a preliminary study to explore the feasibility of using weakly relevant test cases to further improve automated program repair
- Writing of this thesis.

Since this work would not have been possible without the contributions of all the coauthors of Yang et al. (2017), the author uses an editorial "we", rather than "I", even when describing work performed by the author of this thesis individually.

Abstract

"Generate-and-Validate" (G&V) approaches to automatic program repair first *generate* candidate patches and then *validate* the patches against a test suite. Current G&V tools accept the first patch that passes all the test cases and are at risk of making a mistake: they can choose an ultimately buggy patch while leaving a correct patch unfound. These mistakes are often due to developer test suites being insufficient to correctly validate a patch.

In our approach, we aim to improve existing test suites with automatic test generation. To circumvent the oracle problem, we compare the behavior of the buggy program with the behavior of the newly patched program, and if the patched program fails more, then the patch is considered to be overfitted and it is filtered. We evaluate our approach on 441 patches (both overfitted and correct) from three automatic repair systems and show that 67% (279/417) of overfitted patches are filtered. In addition, by further exploring the search space of patches for one of the defects, our approach filters overfitted patches until the correct patch is found and accepted.

We also conduct a post-mortem relevance analysis of automatically-generated test cases to evaluate (1) how many of the test cases should aid the developer in manual debugging, (2) how many of the test cases should filter out more overfitted patches if better oracles are used, and (3) how many of the test cases are relevant to filtering overfitted patches (i.e., how many of them execute lines of code patched by SPR—one of the automatic repair systems in our study). The analysis shows that up to 40% of the automatically-generated test cases per bug can help developers conduct manual debugging and can filter out more overfitted patches if automatic repair systems are empowered with better oracles.

Acknowledgements

I would like to thank my advisor, professor Lin Tan, for her exceptional expertise and professionalism, and all the life opportunities that she gave me.

Dedication

This is dedicated to the ones I love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent research efforts in the area of automatic program repair aim at reducing developer workload and minimizing the cost of fixing a bug. Software developers receive a myriad of bug reports. They rarely have sufficient time to address every one and are forced to release programs with known bugs [15]. The time that developers spend to analyze and fix unimportant bugs would be better spent on more vital tasks. Ideally, the process of bug fixing is automated, where a repair tool transforms buggy programs into working ones. In practice, the outlook is not so bright, and problems arise from different areas:

- How can a defect be localized in the program?
- How can a defect be fixed in an acceptable amount of time?
- How can a fix be verified?

Researchers tackle these problems in various ways. We focus on the general approach known as "Generate-and-Validate" (G&V, or search-based repair): the repair tool first *generates* a candidate patch and then *validates* the patch against the program's test suite. The search is over either when a candidate patch passes the test suite or when the search space is exhausted.

GenProg [14], SPR [18], Prophet [20] and Angelix [24] are among the most prominent G&V tools. As an example of their efficacy, Prophet was able to fix 15 out of 69 defects in the acceptable period of time of 12 hours per defect [19]. Despite the promising results, modern automatic repair tools suffer from producing patches that pass the entire validation test suite but nonetheless are incorrect. Long et al. [19] showed that for SPR's and

Prophet's search spaces (i.e., the sets of all the patches that these tools are capable of generating) such patches prevail over the correct ones by orders of magnitude. G&V patch generation systems are designed to stop the search after encountering the first patch that passes the validation test suite. Thus, if the first patch happened to be incorrect and there is a correct one in the search space for this particular defect, then the correct patch will never be found. Search spaces might contain hundreds of patches that pass the test suite, which rules out the option of manual evaluation of each patch. Our goal is to find a way of automatically discarding incorrect patches until we find a correct one.

We call these plausible but incorrect patches *overfitted* patches; they are analogous to overfitted classification models in machine learning in that they overfit existing developer test cases. In machine learning, one can tackle the problem of overfitting by adding more training data; in G&V approaches, one can similarly tackle the problem of overfitting by performing additional tests. These new tests can potentially prune incorrect patches and clear the way to previously blocked correct patches.

There are several ways of generating test cases automatically: random fuzzing [23], static symbolic execution (e.g., King's approach [12] of static test case generation, generating tests *without* ever executing the program), or concolic testing [7, 32, 21]. No matter the approach, there is a problem of how to employ newly-generated tests to detect overfitted patches. The naive way is to run an automatically-patched program with all the new tests and if any of the tests fail, consider the patch overfitted. However, both previous work [29] and this thesis show that this naive approach is likely to filter correct patches since even correctly-patched programs might contain hidden bugs (unrelated to the bug that the repair is aiming to fix).

Our methodology for pruning overfitted patches is based on the following assumption: the correctly patched version should not behave worse than the buggy version. To decide whether an automatically generated patch is overfitted, we run the buggy and the patched versions of the program with test cases previously generated by means of fuzzing on the buggy program. If the number of errors found for the patched version is greater than the corresponding number for the buggy version, we consider the patch to be overfitted. This approach is evaluated on patches generated by Genprog, AE, Kali, and SPR on the GenProg ManyBugs 2012 benchmark [14]; it has been shown that a significant portion of overfitted patches is filtered.

Not all newly-generated test cases contribute to improving software repair; one of the reasons for this lies in weakness of the oracle used: simple *crash* oracle used in this study is not always sufficient. To see how many automatically-generated test cases have the potential of filtering overfitted patches, an analysis of so-called *weakly relevant* test cases is conducted. This study shows that up to 40% of tests generated are weakly relevant to the target bug.

The contributions of this thesis are as follows:

- Filtering 279 out of 417 overfitted patches generated by Genprog, AE, Kali, and SPR
- Fixing one additional defect, after filtering out overfitted patches
- A manual analysis of the automatically-generated test cases to gauge the number of test cases that have the potential of pruning overfitted patches if equipped with better oracles; up to 40% of tests have such a potential.
- Results of debugging benchmark defects to show *how* the approach filters overfitted patches.
- A feasibility study to investigate usage of weakly relevant test cases to further improve automated program repair.

Chapter 2

Background on Automated Program Repair

In this chapter, we provide the reader with the necessary background information on G&V automatic software repair and the problem of overfitted patches.

Figure 2.1 presents a high-level workflow of an automatic repair tool. As input, it accepts a buggy program and a validation test suite. The suite contains two types of test cases: *failing* test cases that expose the target bug (i.e., the bug it aims to fix) and *passing* test cases that verify the existing functionality. First, the repairer runs a fault localization algorithm that produces a ranked list of program's statements that are suspected to be buggy. Then, the tool uses this information to devise a fix and patch the program to attempt to fix the target bug. The patched program is verified by running the program's validation test suite; if no tests fail, then the patch is considered to be correct and is presented to the developer; otherwise, the automatic repair tool applies a different patch and repeats the process. The term *search space* corresponds to all patches that might be devised for a particular bug.

It is important to have passing test cases along with failing test cases. To illustrate this importance, we present a hypothetical bug that (1) is exposed by a failing test case but (2) is otherwise untested, e.g., by any passing test case. A generated patch can pass this validation test suite just by removing the buggy functionality altogether. Yet, most likely, such a patch would be considered incorrect by the developer and would not be accepted. Qi et al. [28] showed that the majority of patches produced by GenProg, AE and RSRepair are semantically equivalent to a functionality deletion.

Such patches that merely make the test cases pass but do not fix the target bug are called

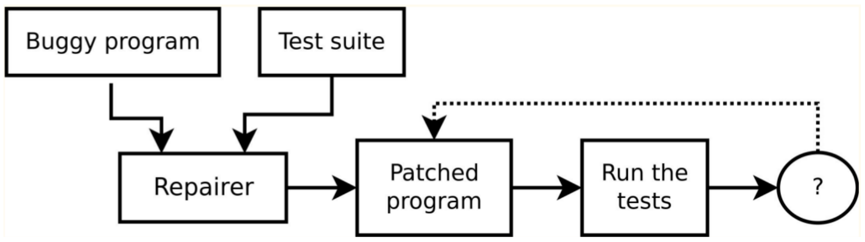


Figure 2.1: The workflow of a G&V automatic repair tool

overfitted patches. Existence of overfitted patches undermine the process of automatic software repair. Usually, only the first patch that passes all the tests is presented to the developer; thus, overfitted patches *block* correct patches and prevent the defect to be fixed correctly.

The workflow of a G&V automatic repair tool is defined by three main components: fault localization, patch generation and patch validation. Fault localization techniques do not differ much between the tools and usually use statement rankings based on the results of running the buggy program on passing and failing test cases. The same is true for the stage of patch validation: automatic repair tools usually do not go farther than simply running developer test cases to verify a patch. However, for patch generation, there is far more variety. For example, GenProg [14] uses existing program statements as fix ingredients and employs genetic programming to devise the best fix; RSRepair [27] substitutes genetic programming with random search; the only types of fixes considered by Kali [28] is functionality deletion. SPR [18] uses predefined patch templates in conjunction with optimizations for searching for a specific value in the template.

Validation and *verification* are well-defined terms in requirements engineering. According to the PMBOK Guide [9], "Validation is the assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers." and "Verification is the evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process." In other words, validation (of a system) answers the question "Have we implemented the right system?", and verification (of a system) answers the question "Have we implemented the system right?" Strictly speaking, in automated program repair, patches are not validated but rather verified, according to the PMBOK definition. However, in this work we use the term "validation" due to (1) the terminology being already well-established in the domain of program repair and (2) "software verification" has a separate meaning in computer science.

Chapter 3

Filtering overfitted patches

In this chapter, we describe our approach for filtering out overfitted automatically-generated patches, present its evaluation, and list threats to validity.

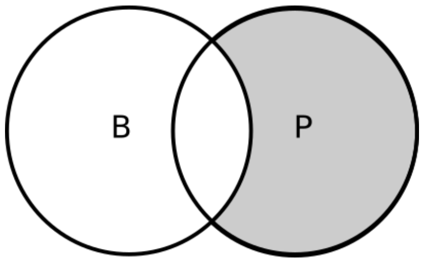
3.1 Approach

There are many established ways to generate more tests for a program. However, automatic generation of oracles is still a challenging research problem. To be able to use automatically-generated tests when deciding whether a patch is overfitted in the absence of an oracle, we use the following intuition: a correctly-patched program should not behave worse than the buggy program. A patch is considered to be overfitted if it fails on test cases, which the buggy version passes. Overfitted patches are identified by using the *O-measure* metric [36]:

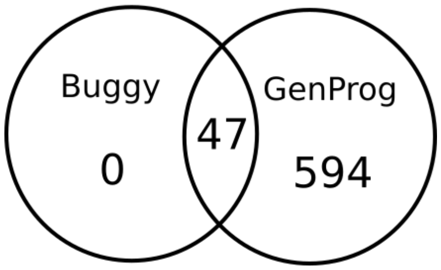
Definition. "Given a test suite T, B: the set of test cases that make the buggy version fail ($B \subset T$), $\overline{B}$: the set of test cases that make the buggy version pass ($\overline{B} \subset T$), P: the set of test cases that make the patched version fail ($P \subset T$). O-measure is defined as the size of $\overline{B} \cap P$."

Our approach accepts as input the buggy version of a program, and an automatically-generated patch (that already passes all the developer tests); the approach is comprised of the following steps:

1. Using developer tests as seeds, we perform fuzz-testing of the buggy version of the program and generate new test cases. This step is performed only once for each bug.



(a) Shaded area represents tests that fail on the patched version (P) but pass on the buggy version (B).



(b) The numbers of failing test cases for the buggy version and the version patched by GenProg patch #5; the bug gzip-3fe0c-39a36.

Figure 3.1: Sets of failing test cases for the buggy and the patched versions.

2. The buggy version is tested on these new test cases; for each test case, we record whether the program has failed.
3. The patch is applied and the same is performed on the patched version.
4. Then, the approach compares the set of failing test cases for the buggy and the patched version. If a patched version fails on a test case that the buggy version passes, the patch is deemed to be overfitted, and the approach filters it out.

In Figure 3.1a, the relationship between the failed test cases on the buggy version and the patched version is represented in a venn diagram. The left circle represents the failed test cases on the buggy version, while the right one represents the patched version. If $|\overline{B} \cap P|$ (cardinality of the shaded area, i.e., the O-measure) is greater than zero, then the patch is considered to be overfitted. With an effective test suite to define the bug, different behaviors are expected between the buggy version and the patched one. Thus, the two circles in Figure 3.1a should not completely overlap. For example, for the bug `gzip-3fe0c-39a36`, we execute the buggy version with the automatically-generated test cases and discover 47 test cases that make it fail (Figure 3.1b). Then, we run the same test cases on the version patched by one of the GenProg patches and record the failing test cases for this version as well ($47 + 594 = 641$ failing test cases). Finally, we compare the sets of failing test cases between the buggy and the patched versions; we observe 594 test cases that fail only for the patched version, suggesting that this GenProg patch is overfitted (and this is indeed the case).

The definition of O-measure that we use is not the only possible one. The filtering approach could have used a simpler metric: if a patched version fails on *any* newly-generated test cases, then consider the patch to be overfitted. However, previous work [29] shows that this metric is likely to filter even correct patches, since even correctly-patched programs usually contain hidden bugs (i.e., bugs that are unrelated to the target bug). Yang et al. [36] also present the theoretical basis for the chosen definition of O-measure by comparing it with the *ideal* definition of O-measure.

Yang et al.[36] propose a tool, *Opad*, for filtering overfitted patches. Opad employs two types of oracles that define the failure of an automatically-generated test. The first one is a simple crash oracle: consider a test to be failed, if the program under test crashes with a segmentation violation error, assertion, etc. The second oracle involves more sophisticated memory-safety checking: to compare program behaviors between the buggy version and the patched version, Opad checks whether the patch introduces new memory errors (e.g., buffer overflows) or whether the patch makes the program leak more memory. The author of this thesis was mostly involved with designing and conducting the experiments with the crash oracle; thus, we explain it in more detail. Here, we mention the results of using the memory-safety oracle only briefly. More details on the part of the study that involves memory-safety oracle can be found in Yang et al. [36].

Test case generation. We use fuzzing as means to generate new test cases. Fuzz-testing is a well-established technique that consists of bombarding the program under test with random (or partially random) input with the purpose of finding defects. In our study, we use American Fuzzy Lop (AFL) [1]: a coverage-guided fuzzer that has previously uncovered many bugs in various open-source projects. AFL accepts valid inputs as seeds and mutates them in order to uncover more paths in the program. These mutants are mutated further until all the paths in the program are explored or the time limit is reached. In this study, we terminate test case generation once AFL cannot discover any new paths during two hours.

3.2 Evaluation

In this section, we present the results of using our approach to filter overfitted patches produced by automatic software repair systems.

Table 3.1: Benchmarks used in the study.

App.	LOC	GenProg/AE	Kali	SPR
gzip	491k	10/10	1/1	2/2
libtiff	77k	23/23	5/5	255/258
lighttpd	62k	35/35	5/5	5/5
php	1046k	31/35	6/8	7/17
python	407k	5/9	1/2	6/6
wireshark	2814k	12/12	4/4	4/4
fbc	97k	1/1	1/1	1/1
gmp	145k	3/3	1/1	1/2

3.2.1 Benchmarks

For our evaluation, we chose GenProg ManyBugs 2012 benchmark [14] since all the repair systems in our study were previously evaluated on that benchmark. It consists of 105 defects from various systems of different scale written in C. Table 3.1 shows an overview of the benchmark. The column "LOC" shows the number of code lines per each project; columns "GenProg/AE", "Kali", and "SPR" show the number of patches generated for defects in each project (number of overfitted patches/total number). Each defect has a set of passing and failing test cases; the passing test cases test existing functionality and ensure lack of regressions, whereas failing test cases expose a defect. A correct patch is known for each defect; correctness is defined with respect to the developer fix: a developer patch is always assumed to be correct. Thus, if an automatically-generated patch is semantically equivalent to the developer patch, then it is assumed to be correct. We evaluate our approach on every defect for which at least one of the automatic repair systems (GenProg/AE, Kali, and SPR) produces a patch. The subjects `gmp` and `fbc` were excluded from the study due to how tests are implemented for them: they use unit tests that cannot be used as seeds for fuzz-testing. Note, that this is not a limitation of the approach but rather an implementation issue that could be avoided by using a different test-generation technique. Also note, that for two `libtiff` bugs, we continue exploring SPR search space and encounter many overfitted patches; thus, a large number of patches.

3.2.2 Results of filtering overfitted patches

Filtering overfitted patches is important since it allows a repair system to continue exploring the search space and potentially find the correct patch. Due to large numbers of patches

Table 3.2: Numbers of test cases that fail on the patched version but pass on the buggy version of the program.

Bug ID	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	AE	Kali	SPR
gzip-3fe0c-39a36	0	0	4	0	594	0	-	0	5	-	0	1	0
libtiff-08603-1ba75	-	23	-	23	23	-	23	-	-	-	23	23	23
libtiff-5b021-3dfb3	4	5	5	5	4	5	-	5	5	-	5	5	4
libtiff-d13be-ccadf	-	1	1	0	142	-	-	-	1	-	0	1	1✓

in search spaces of G&V automatic repair tools [19], it is not reasonable to present to the developer the whole search space for a particular bug. Thus, automatic repair systems usually stop at the first patch that passes all the tests. Note, that even if filtering overfitted patches does not lead to finding a correct patch, it still has benefits due to reducing developers' workload associated with evaluating patches produced by automatic repair systems.

Our approach filters 67% (279/417) of overfitted patches (by only employing the crash oracle). We evaluated our approach on 441 patches from GenProg/AE, Kali, and SPR (both overfitted and correct). Table 3.2 shows the number of test cases that fail on the patched version but pass on the buggy version (the bugs, for which our approach is unable to filter any patches, are omitted). For instance, the GenProg patch #5 for `gzip-3fe0c-39a36` fails on 594 automatically-generated test cases which the buggy version passes. In accordance with our approach, a non-zero entry indicates that a patch is filtered. The columns "G1–G10" denote GenProg patches generated by using different random seeds. Note, that for three defects, `libtiff-08603-1ba75`, `libtiff-5b021-3dfb3` and `libtiff-d13be-ccadf`, we continue exploring the search space of SPR; the table shows the number of test cases only for the first patch evaluated. The case in which a correct patch is found after filtering overfitted patches is denoted by "✓".

Our approach is able to filter out many overfitted patches for two of the projects in the ManyBugs 2012 benchmark. This should significantly decrease the costs associated with manual patch evaluation in the real-world scenario of using automated program repair systems in practice. Although the approach does find a correct patch for `libtiff-d13be-ccadf`, it also mistakenly filters a correct patch produced by SPR for `libtiff-5b021-3dfb3`. Note, that the patch that was filtered by mistake is inside the search space for the defect; i.e., it is not the first patch produced by SPR and presented to the developer for evaluation. The reason behind the mistake lies in the presence of hidden bugs in `libtiff`; the patch changes the control flow of the program and exposes a previously hidden bug (unrelated to the bug that SPR aims to fix). Thus, the patched version

Table 3.3: Numbers of patches filtered by using both the crash oracle and the memory-safety oracle

Bug ID	GP/AE	Kali	SPR	Bug ID	GP/AE	Kali	SPR
gzip-3fe0c-39a36	3	1	0	php-307562-307561	-	-	0
gzip-a1d3d-f17cb	0	-	0	php-307846-307853	-	-	0
libtiff-08603-1ba75	5	0	1	php-307914-307915	-	-	0
libtiff-5b021-3dfb3	9	1	238	php-307931-307934	1	0	0
libtiff-90d13-4c666	0	0	0	php-308262-308315	-	-	2
libtiff-d13be-ccadf	3	1	13	php-308323-308327	-	-	1
libtiff-ee2ce-b5691	0	0	0	php-308525-308529	0	0	1
lighttpd-1794-1795	0	-	0	php-308734-308761	-	-	0
lighttpd-1806-1807	0	-	0	php-309111-309159	0	-	0
lighttpd-1913-1914	0	-	0	php-309516-309535	-	-	0
lighttpd-1948-1949	-	-	0	php-309579-309580	-	-	1
lighttpd-2330-2331	2	-	0	php-309688-309716	-	-	0
lighttpd-2661-2662	0	-	0	php-309892-309910	0	0	0
python-69223-69224	0	-	0	php-309986-310009	5	0	1
python-69368-69372	-	-	1	php-310011-310050	8	1	5
python-69709-69710	-	-	0	php-310370-310389	-	1	0
python-69783-69784	0	0	0	php-310673-310681	0	0	0
python-70019-70023	-	-	0	php-310991-310999	-	-	0
python-70098-70101	1	1	0	php-311323-311300	-	-	0
wireshark-37112-37111	10	1	1	php-311346-311348	-	0	0
wireshark-37172-37171	1	0	0	gmp-13420-13421	-	-	0
wireshark-37172-37173	1	0	1	gmp-14166-14167	-	0	0
wireshark-37284-37285	-	0	0				

fails on a test case which the buggy version passes (the details are discussed in Section 3.3). Although we recognize this as a threat to our approach, we argue that detecting and fixing hidden bugs is still beneficial towards improving overall software quality.

In Table 3.3, we also show the results of filtering automatically-generated patches by using two types of oracles, as proposed by Opad [36]: crash and memory-safety. The columns "GP/AE", "Kali", and "SPR" show the total number of filtered patches generated by GenProg/AE, Kali, and SPR respectively. The case in which an automatic repair tool does not produce any patch is denoted by "-". Adding an additional memory-safety oracle helps with filtering overfitted patches for a wider range of projects (i.e., not only `gzip` and `libtiff`, but also `php`, `python`, `wireshark`, and `lighttpd`).

```
- if (nstrips > 1        // buggy
+ if (nstrips > 2        // developer
+ if (nstrips > 1 && 0   // overfitted
    && compression == COMPRESSION_NONE
    && stripbytecount[0] != stripbytecount[1])
  {
    TIFFWarning("Wrong␣field,␣ignoring␣and␣calculating␣from␣imagelength");
    if (estimate(tif, ...) < 0)
      goto bad;
  }
```

Figure 3.2: Patches for libtiff-d13be-ccadf

3.3 Case studies

In this section, we perform a closer look as to *how* automatically-generated test cases improve automated program repair systems used in our study. We manually debug two cases: `libtiff-d13be-ccadf`, for which our approach uncovers a correct patch, and `libtiff-5b021-3dfb3`, for which a correct patch is filtered.

Figure 3.2 presents the code for `libtiff-d13be-ccadf`, and three versions of the program: the buggy one, the one patched by the developer (i.e., the correct version), and the one patched by an overfitted patch produced by SPR. The `libtiff` library is an image-manipulation library and the code in question resides in an image-reading routine. The first if-condition is supposed to identify ill-formed images and call the function `estimate()` to attempt to fix the internal representation of an image; this representation is partially stored in an array. The buggy version calls `estimate()` for some well-formed images that should not be fixed, and the developer version correctly fixes the bug by tweaking the if-condition. The overfitted version, however, merely removes the call to `estimate()` so it is never called for any types of images, neither well- nor ill-formed. One of the automatically-generated test cases, produced for `libtiff-d13be-ccadf`, contains garbage values in the image array. When this test case is executed with the buggy version, the program fixes the values in the array and successfully finishes execution. The version with the overfitted patch, on the other hand, skips the call to `estimate()` and later uses garbage values from the array to index another array; this leads to a segmentation violation and a crash. All overfitted patches that precede the correct patch in the search space of SPR have the same flaw and, therefore, are filtered out. SPR's correct patch fixes the internal representation of the image and is consequently accepted by our approach.

Figure 3.3 shows the code for a hidden bug in `libtiff-5b021-3dfb3`: the patched version fails on the assertion at line 2, whereas the buggy version exits before reaching this

```
 1 int TIFFWriteDirectoryTagCheckedRational(double value, ...) {
 2   assert(value >= 0.0);  // failed assertion
 3   // the earliest version
 4 - if (value == (uint32)value) {
 5 -     ...
 6 - } else if (value < 1.0) {
 7 -     ...
 8 - }
 9   // the current version
10 + if (value <= 0.0) {
11 +     ...
12 + }
13   ...
14   }
```

Figure 3.3: A hidden bug exposed in the patched version of libtiff-5b021-3dfb3

piece of code (due to the target bug). We argue that the assertion at line 2 is redundant and should be removed. The earliest version of the `TIFFWriteDirectoryTagCheckedRational` function did not contain an if-clause that handles `value` less or equal to zero so the assertion was relevant. However, later the code had been changed and the function started to handle non-positive values explicitly, which rendered the assertion obsolete. After we reported this bug to the developers of `libtiff`, they confirmed our understanding of the bug[1].

3.4 Threats to validity

3.4.1 Non-determinism

In some cases, benchmark programs in our study show non-deterministic behavior; i.e., a program might either fail or pass an automatically-generated test due to random chance. To mitigate this, we run each version of the program on each test case 10 times; if at least one of the runs leads to a failure, then we deem such a test case failed. Admittedly, this solution does increase overall overhead of our approach. Alternatively, Address Space Layout Randomization (ASLR) could have been disabled to reduce non-determinism, as our preliminary experiments showed that ASLR is the main culprit of non-deterministic behavior. Disabling ASLR would indeed lead to smaller overhead; however, our experiments also showed that ASLR helps to uncover more crashes. Thus, disabling it might inhibit our approach and lead to fewer overfitted patches being filtered.

[1] http://bugzilla.maptools.org/show_bug.cgi?id=2535

3.4.2 Hidden bugs

Our approach relies on comparing the behavior of the buggy program with the behavior of the patched program; if the patched program fails more, then the approach considers a patch to be overfitted. However, the patched program might fail on test cases which the buggy program passes not due to defects that the patch brings, but rather due to hidden bugs being exposed after the patch changes the control flow of a program. For example, in the case of `libtiff-5b021-3dfb3`, patches produced by SPR uncover a redundant assertion that leads to program termination (after we had reported this bug, the developers fixed it). Although this threat might lead to correct patches being filtered, fixing hidden bugs detected by automatically-generated tests should lead to an increase in overall quality of a software project.

Chapter 4

Test case relevance analysis

In this chapter, we analyze automatically-generated test cases to study how many of them should help developers in manual debugging and filter more overfitted patches if automated program repair systems are empowered with better oracles.

4.1 Types of test case relevance

Even the state of the art program repair techniques can only fix a small portion of bugs. Still, labor-intensive manual debugging is largely required to fix as many bugs as possible. Automatically-generated test cases can help manual debugging in addition to improving G&V techniques. To evaluate whether our automatically-generated test cases can indeed help developers fix the bugs, we conduct relevance analysis as a post-mortem assessment of the automatically-generated test cases. Test cases are **relevant** if they can help developers in diagnosing the bugs. For example, for an integer overflow bug, relevant test cases may contain big integers to trigger the integer overflow (i.e., causing a failure); then, this test case no longer fails when this bug is fixed. Relevance can be determined based on whether different behaviors can be observed between a buggy version and the patched version. Different behaviors can be either from program output or program state. Therefore, in order to mimic manual debugging process and better assess the relevance of automatically-generated test cases, we define two types of relevance—*strongly relevant* and *weakly relevant*—to represent differences on program output and program state respectively. For automatically-generated test cases, differences on program output are explicit as crash or not; however, differences on program states are implicit, which requires manual analysis which mimics developers' debugging process.

The definitions of relevance are inspired by the concept of strongly and weakly killing a mutant in mutation testing [8, 16]. Mutation testing is a technique to evaluate the quality of test suites. It works by modifying a program, running the test suite on the original program and the modified program, and observing differences in the behavior between two versions of the program. If such differences are captured, then the mutant is "killed". In order to differentiate whether different program behaviors are from program output or from program state, there are two definitions of killing a mutant: (1) *strongly* killing a mutant and (2) *weakly* killing a mutant [2]. The mutant is killed strongly, if the output between the original program and the mutant differs. The mutant is killed weakly, if the program states differ between the versions.

In summary, we propose to categorize relevant test cases into three categories (Table 4.1). One test case can be strongly/weakly relevant to the target bug or all bugs. Note that Type 1 is a subset of Type 2; we separate Type 1 out of Type 2 because Type 1 is more helpful when diagnosing one particular bug. Since weak relevance concerns program states which vary from bug to bug, weakly relevant test cases to all bugs is not an applicable category (✗ in Table 4.1). We describe details of each category, and criteria we use to determine which category one test case belongs to as follows.

Table 4.1: Categories of relevant test cases.

	Strongly Relevant	Weakly Relevant
Target Bug	Type 1	Type 3
All Bugs	Type 2	✗

- *Type 1: A test case is strongly relevant to the target bug*, if it makes the buggy version fail while the developer version passes. We denote such test cases as $B \cap \overline{P}$ (B is the set of failed test cases for the buggy version, P—for the patched version). In Figure 4.1, the left circle represents the set of the failed test cases on the buggy version (B), and the right circle is the set of failed test cases on the developer version. We denote *Type 1*—test cases strongly relevant to the target bug as $B \cap \overline{P}$. Therefore, in Figure 4.1, the number of *Type 1* test cases—$|B \cap \overline{P}|$ is 47. This definition of *Type 1* covers the test cases that expose a bug (i.e., failure) in the buggy version, but the developer patch makes the test cases pass. Therefore such test cases should be relevant to the target bug. Note that here we are able to use the developer patches because this is a post-mortem analysis to evaluate whether the test cases can improve manual debugging. This is not to show whether the test cases can improve automated program repair.

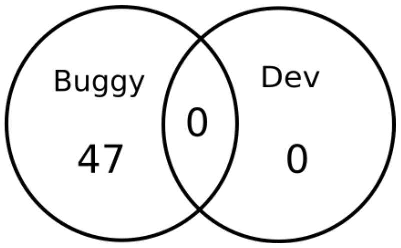


Figure 4.1: The sets of test cases on which the buggy (left circle) and developer (right circle) versions fail for the bug gzip-3fe0c-39a36.

- *Type 2: A test case is strongly relevant to all bugs*, if it makes either the buggy or the developer version fail. *Type 2* is denoted as $B \cup P$. In Figure 4.1, the size of $B \cup P$ equals $47 + 0 + 0 = 47$). Such test cases should be helpful to developers to resolve the target bug (as they make the buggy version fail) or to resolve other bugs in the software (as they make the patched version fail too). Facilitation in solving these issues is a side effect of our approach that helps to improve overall software reliability.
- *Type 3: A test case is weakly relevant*, if it exposes the target bug inexplicitly by showing differences in program states. Compared to a strongly relevant test case, a weakly relevant test case usually does not expose a bug explicitly. Instead, the program states differ between the buggy version and the developer version when executing test cases of *Type 3*.

To illustrate the concept of weak relevance, we use `libtiff-08603-1ba75` that has a buggy check for an integer overflow: many inputs that do not actually contain an integer overflow are being erroneously rejected (see Figure 4.2). The condition at line 6 is incorrect due to an arithmetic error; the correct condition is at line 7. For this bug, a test case is relevant, if it is rejected by the buggy version and accepted by the developer version. To detect a weakly relevant input in this particular case, we instrument the buggy program to count how many times the "true" branch is taken (i.e., how many times the function rejects the input); this number is denoted as M. Then, we apply the developer patch and execute the program on the same input again: the number of times the program rejects the input is denoted as N. If $M > N$ for any particular test case, then we state that the test case is weakly relevant to the bug `libtiff-08603-1ba75`. We list definitions of weak relevance for all the defects in our study in Appendix A.

```
void TIFFFetchData(TIFF tif) {
  ...
  tsize_t cc = dir.tdir_count * w;

  /* Check for overflow. */
- if (dir.tdir_count / w != cc)     /* BUG */
+ if (cc / dir.tdir_count != w)     /* correct check */
    goto bad;

  memcpy(cp, tif.tif_base + dirdir_offset, cc);

bad:
  return 0;
}
```

Figure 4.2: An incorrect check for an integer overflow in libtiff-08603-1ba75

We argue that the benefits of weakly relevant test cases are twofold: on one hand, they should be helpful in the process of manual debugging, on the other hand, they should benefit automated program repair systems that have better oracles. With respect to manual debugging, weakly relevant test cases explore program paths that are relevant to the target bug; additionally, fuzz-testing changes program input in somewhat random fashion, which might lead to variables changing their values at the point of a bug in unexpected, to the developer, fashion (e.g., corner cases). Handling those corner cases, which the original test suite did not contain, should be helpful to more fully fix the bug. Note that developers do not need do create a definition of weakly relevance for the bug they are investigating but rather naturally use the automatically-generated test cases that they find relevant to the bug; definitions of weakly relevance are created by the author of this work solely to discover weakly relevant test cases among all the automatically-generated test cases.

The oracle that we employ in our approach for filtering overfitted patches is a simple crash oracle, which is limiting factor in the effectiveness of the approach. However, automatic oracle generation is an open problem and a hard challenge [2]. If a test case explores relevant paths in the program with relevant data but does not cause a crash, then it does not mean that such a test case is useless in filtering overfitted patches. Rather, it shows lack of better oracles. We argue that weakly relevant test cases might help to filter out more overfitted patches if automated repair systems are empowered with better oracles (either automatically-generated or manually-written).

Table 4.2: Results of relevance analysis of automatically-generated test cases.

Bug ID	Strongly relevant		Weakly	Prun.	Total
	Type 1	Type 2	Type 3		
gzip-3fe0c-39a36	47	47	47	893	2,052
gzip-a1d3d-f17cb	0	0	0	1,443	1,443
libtiff-08603-1ba75	0	29	1,312	1,303	1,312
libtiff-5b021-3dfb3	0	281	745	906	2,863
libtiff-90d13-4c666	9	155	982	979	2,693
libtiff-d13be-ccadf	0	24	249	1,331	1,361
libtiff-ee2ce-b5691	0	611	2,310	2,363	5,967
lighttpd-1794-1795	0	0	0	0	11,372
lighttpd-1806-1807	0	0	0	0	11,372
lighttpd-1913-1914	0	0	0	0	11,372
lighttpd-1948-1949	0	0	0	0	11,372
lighttpd-2330-2331	0	0	51	7,629	11,372
lighttpd-2661-2662	0	0	0	0	11,372
python-69223-69224	0	0	0	0	356
python-69368-69372	0	0	0	0	230
python-69709-69710	0	0	0	0	284
python-69783-69784	0	0	0	0	552
python-70019-70023	0	0	0	0	529
python-70098-70101	0	0	0	0	338
wireshark-37112-37111	0	14	0	0	858
wireshark-37172-37171	0	2	0	0	452
wireshark-37172-37173	0	12	0	0	889
wireshark-37284-37285	0	2	0	0	483
php-307562-307561	0	400	9	9	26,556
php-307846-307853	0	528	0	8	31,904
php-307914-307915	0	16	0	0	6,791
php-307931-307934	0	11	0	0	5,945
php-308262-308315	0	2	0	0	10,695
php-308323-308327	0	2	0	0	6,192
php-308525-308529	0	11	0	0	4,981
php-308734-308761	0	10	0	0	12,817
php-309111-309159	0	1	0	0	4,205
php-309516-309535	0	3	0	0	12,588
php-309579-309580	10	10	10	17	5,590
php-309688-309716	0	0	0	0	8,245
php-309892-309910	0	5	0	0	5,033
php-309986-310009	0	0	0	0	3,567
php-310011-310050	0	4	0	2,095	4,642
php-310370-310389	0	0	0	1,762	2,143
php-310673-310681	0	0	0	0	4,329
php-310991-310999	0	0	0	0	7,275
php-311323-311300	0	5	0	0	6,805
php-311346-311348	0	4	0	0	5,456

4.2 Results

In this section, we present the results of the relevance analysis on automatically-generated test cases. The relevance analysis is to evaluate the usefulness of automatically-generated test cases in helping manual debugging. As stated in Section 4.1, we propose to categorize relevant test cases into three categories. To determine strongly relevant test cases, we conduct the analysis by running the subject binaries with the automatically-generated test cases to detect failures. For the weakly relevant ones, we manually inspect each bug and create a definition specific for that particular bug. Then, we instrument the program according to the definition and run the subject binary with our test cases.

Table 4.2 presents the results. The columns "Strongly relevant" and "Weakly" present the number of relevant test cases according to the definitions of strong and weak relevance (discussed in Section 4.1). These test cases are relevant to debugging; i.e., they should help the developer to improve the software quality. The column "Prun." shows the number of test cases that execute the portion of the code patched by SPR. These test cases are relevant to pruning overfitted patches. The column "Total" shows the total number of automatically-generated test cases generated by means of fuzzing for a particular bug. For example, in the case of the bug `libtiff-5b021-3dfb3`, there are zero test cases that fail for the buggy version and at the same time pass for the developer version, and 281 cases that make either the buggy or the developer version crash. After formulating a definition of weakly relevance for this bug, we discovered 745 relevant test cases out of 2,863. In addition, there are 906 test cases that execute the patch generated by SPR.

Based on the results in Table 4.2, for 26 bugs, we generated test cases that make the software crash. Finding the root cause and fixing the bug that causes the crash should lead to an improvement to overall quality of the software. Moreover, in many cases we can generate weakly relevant test cases that expose the target bug. These test cases make the program follow different paths (in conformance with the fuzzer's workflow); this should give the developer a better understanding of the target bug.

As a side effect of our technique, we can generate test cases that can help the developer both to pinpoint the target bug in the process of manual debugging and to improve the overall software quality.

Chapter 5

Related work

Automated program repair systems. SPR [18] is a state-of-the-art search-based tool for automated program repair. It is novel for the following approaches: (1) a notion of a "schema" (i.e., a patch template, where different conditions are represented by different templates) that leads to a large search space with many correct patches; (2) a "target value search" that optimizes the search for a specific value inside a schema instantiation; and (3) a "condition synthesis" that produces a patch. SPR was evaluated on the benchmarks compiled by Le Goues et al. [14] and it was able to generate successful repairs for 11 out of 69 defects. In many cases, there is a correct patch in the search space but it is blocked by an overfitted patch; our work aims at filtering out these overfitted patches. Prophet [20] further improves SPR by learning correct fixes from a large corpus of developer fixes; then, this information is used to better prioritize candidate patches inside search spaces. Admittedly, Prophet is able to find a correct patch for `libtiff-d13be-ccadf`, the bug for which our approach finds a correct patch after filtering out SPR's overfitted patches. However, Prophet still produces many overfitted patches for other defects and our approach should be able to mitigate that. We leave exploring the effectiveness of using our approach in conjunction with Prophet as future work. GenProg [14] uses fix ingredients from existing program statements and leverages evolutionary algorithms to create the best patch according to the fitness function. AE [34] is an improvement upon GenProg that aims to reduce its search space by identifying equivalent patches; thus it improves GenProg's efficiency by running the developer test suite for fewer patches. Kali [28] is an automatic repair system that focuses only on patches that remove functionality; Qi et al. [28] showed that Kali not only performs as good as GenProg in terms of the number of correct patches produced but also eclipses GenProg in terms of performance. Angelix [24] is a tool that implements program repair based on semantics of the code that it analyzes. The tool is scalable through means of controlled symbolic execution and is relatively light-weight in comparison to other tools that implement repair through the search-based method. Angelix conducts program repair by taking into advantage semantic relations between error prone areas within its subject. Our approach can potentially be used to improve Angelix in filtering out the overfitted patches that it may generate.

Benchmarks for automated software repair. Le Goues et al. [14] introduced a set of benchmarks as a part of their work on the GenProg automatic repair tool. Although the authors had a goal of creating a large representative set of bugs, these benchmarks have a number of limitations: e.g., the bugs are only those of C programs, the bugs are mostly simple, some of the bugs are not even bugs but deliberate functionality changes. Defects4J [10] is an alternative set of benchmarks written in Java. It consists of 357 real-world (rather than artificial) bugs from large open-source projects. Reproducing our study on the Defects4J benchmark is an interesting direction of future work, especially considering that two out of four automatic repair tools in our study have been previously reimplemented to fix Java defects (jGenProg and jKali [22]). However, to the best of our knowledge, there is no current reimplementation of SPR for Java and creating one requires significant engineering efforts. We leave the direction of reproducing our study on the Defects4J benchmark as future work.

Automatic test generation. There are various ways to generate new tests and expose faults in a program. Fuzzing, or fuzz-testing, was invented by B.P. Miller in the late 1980s [25] after noticing that random noise to the input can cause some UNIX utilities to fail. Despite the surprising efficacy of this approach, random input often finds only “shallow” bugs; i.e., the bugs in the code that performs initial input checks. To mitigate this, researchers came up with the fuzzers that either mutate a well-formed input in a random manner (mutation-based fuzzers) or generate test cases based on the formal specifications (generation-based fuzzers) [30]. Though these approaches are able to detect bugs and vulnerabilities hidden deeper in code’s logic, both of them share a common disadvantage: they do not leverage knowledge of the tested program about how the input is processed. Mutation- and generation-based fuzzers view a tested program as a black box and are doomed to produce test cases blindly; this takes a great amount of time and resources, and bugs often remain undetected. White-box fuzzing and symbolic execution tools leverage such knowledge and produce test cases in a smarter way; these test cases would theoretically be able to detect even “deeper” bugs. Examples of such include KLEE [4], S^2E [5], Zesti [21] and SAGE [7]. In this work, we use directed random fuzzing as implemented by American Fuzzy Lop [1] due to its scalability and effectiveness in finding real defects.

Requirement defects, i.e., defects that arise due to misunderstandings of software requirements, are plentiful among many projects [13, 26]; in addition, they are harder to detect and, as a consequence, harder to fix. To the best of our knowledge, there is no automated program repair effort solely focused on fixing requirement bugs. Creating such a tool might be a promising research direction.

Anti-patterns. Tan et al. [31] improve the effectiveness and efficiency of automatic repair by filtering out patches based on *anti-patterns*—code constructs that characterize overfitted patches. For example, a patch that simply removes functionality or always executes an if-branch is likely to be overfitted. This approach is built on top of existing G&V automatic repair tools, and rejects (or accepts) a patch after the developer test cases pass. Our approach is similar in this sense; however, we tackle the problem of overfitted patches from a different angle: rather than filtering out a patch based on a set of predefined metrics, we compare the buggy version with the patched version and expect the behavior of the patched version to not worsen.

Impact of test suites' quality on automatic repair. Long et al. [19] empirically measure the correlation between the "richness" of a search space (i.e., the number of patches in a search space) and the effectiveness of this search space (i.e., the number of correct patches produced). The study shows that the more rich a search space is, the less the chance to find a correct patch due to great prevalence of overfitted patches in rich search spaces. The study also shows that the ratio of correct patches to overfitted patches is greater for subjects with stronger test suites; i.e., the stronger a test suite is, the fewer overfitted patches there are in the search space. Martinez et al. [10] perform an empirical study on the Defects4J benchmark; three automated program repair systems are evaluated: jGenProg (the authors reimplemented GenProg to target Java bugs, rather than C bugs), jKali (the same for Kali) and Nopol—the only semantic-based system in the study. The results show unsatisfactory performance for all three tools: only 11 out of 84 generated patches are actually correct (beyond just passing the test suite). Consequently, our approach aims to improve existing developer test suites with automatically-generated tests and filter overfitted patches from search spaces.

Using automatic test generation to improve automated program repair. Xin et al. [35] propose a tool called DiffTGen that generates new test cases to discover semantic differences between the buggy and the patched program with the purpose of identifying overfitted patches. DiffTGen is a promising tool; however, at this stage, it leverages the correctly patched version of a program as an oracle. Our approach, on the other hand, does not need the perfect oracle, as it relies on differences in execution between the buggy version and the automatically-patched version of the program. Yu et al. [37] propose two techniques for filtering overfitted automatically-generated patches by the means of test generation. The first one targets search-based repair systems and works by evaluating each test-suite-adequate patch in the search space on automatically-generated tests. Then, the patch with the fewest number of failed test cases is picked as the correct one. Our approach, on the contrary, does not need to evaluate each patch in the search space as it stops once the approach encounters a patch that does not make the program behave worse compared to the buggy version. The second technique targets semantic-based automated program repair tools and aims to generate more test cases that would be relevant to existing program functionality (rather than bug-exposing tests). These new test cases improve repair constraints and help semantics-based automated program systems to generate more correct patches. Liu et al. [17] propose an approach based on the following intuition: after applying a patch, program's behavior should not change drastically on *passing* test cases. Their approach compares stack trace similarities between the buggy and the patched version and makes a decision of whether the stack trace of the patched version is different enough for the patch to be considered overfitted. Automatically-generated tests are employed as well to enhance existing developer test suites. Our approach employs a different idea of using crash as an oracle to detect overfitted patches.

Chapter 6

Future work

6.1 Overview

The analysis of weakly relevant test cases presented in Section 4.2 shows that many automatically-generated test cases have the potential of filtering overfitted patches. However, weakly relevant test cases are often unable to do so due to insufficiencies of the oracles employed. In other words, although our study has shown that using crash as oracle can help to filter many overfitted patches, it is too simple an oracle. Thus, a question arises: what are the different ways to leverage automatically-generated test cases to filter overfitted patches while taking into consideration the findings of the relevance analysis?

As described in Chapter 2, test suites used by automated program repair systems consist of two types of test cases: failing test cases that expose the bug, and passing test cases that ensure lack of regressions. A correct patch is expected to introduce changes that amend program's behavior *only* on failing test cases; changes in program's behavior on passing test cases would imply regressions that must be avoided. Due to potential ambiguities of the terms "passing" and "failing" test cases, we introduce the following:

Definition. A test case is called *bad* if the behavior of the buggy program should be altered on such a test case to fix the bug; a test case is called *good* if the behavior of the buggy program should be preserved so no regressions are introduced.

To illustrate the definitions with an example, we use `libtiff-08603-1ba75`—a bug in the `libtiff` image manipulation library (see Figure 6.1): the developer incorrectly implements a check for an integer overflow at line 6, the correct check should be `cc / tdir_count != w` which is true when the variable `cc` is overflowed. The incorrect check leads to many *benign* inputs (i.e., the ones that do not contain an overflow) being rejected. Assume that the test suite contains 2 test cases (the tuples denote the values of the variables [`cc, tdir_count, w`]):

1. The values are [6, 3, 2] and the program execution should not go to the error handling branch (since there is no integer overflow). However, due to the bug, the execution goes to error handling ($3/2 \neq 6$) and the test fails.

2. The values [22, 22, 1] and the execution should not go to `bad`; this test passes ($22/1 == 22$).

The failing tests cases, such as the test case 1, we call bad test cases; the passing test cases, such as the test cases 2, we call good tests cases. To fix the bug, the automatic repair tool should change the program behavior in a way that causes the first test to pass but does not cause the second test to fail.

Similarly, we distinguish good program behavior from bad behavior. Based on the above, we formulate the core intuition of our approach: a correct patch must change bad program behavior, and must not change good program behavior. If either of the statements does not hold true, then a patch is considered to be overfitted.

One of the crucial parts of the approach is a classifier that distinguishes good tests from bad ones. We have studied the feasibility of two possible classification approaches; below, we present our preliminary findings.

6.2 Using machine learning to classify test cases as bad or good

One way to distinguish good tests from bad ones is to apply machine-learning classification algorithms using values of relevant variables as input. The basic idea is to identify variables relevant to the target bug by means of fault localization, and use their values collected from developer test cases as training data. Then, to decide whether an automatically-generated test case is good or bad, collect values of the variables by running the test case on the buggy version and feed the values to the classification algorithm, which would produce a binary output.

We illustrate the approach with `libtiff-08603-1ba75`—a defective integer-overflow check. The first stage of G&V automatic software repair is fault localization; let us assume

```
 1 void TIFFFetchData(TIFF tif) {
 2   ...
 3   tsize_t cc = dir.tdir_count * w;
 4 
 5   /* Check for overflow. */
 6 - if (dir.tdir_count / w != cc)     /* BUG */
 7 + if (cc / dir.tdir_count != w)     /* correct check */
 8 + if (0)                            /* overfitted check */
 9     goto bad;
10 
11   memcpy(cp, tif.tif_base + dirdir_offset, cc);
12 
13 bad:
14   return 0;
15 }
```

Figure 6.1: Checks for an integer overflow in libtiff-08603-1ba75

that the fault localization algorithm correctly picks line 6 as the faulty line. Next, the tool applies a patch to the faulty line; the simplest patch to be applied is to remove the faulty functionality altogether (see line 8). Indeed, this patch would cause all the tests to pass ([6, 3, 2] does not cause the program to go to `bad` anymore, and [22, 22, 1] still passes). However, this patch is obviously incorrect and would not be accepted by the developer. The reason behind the tool accepting an incorrect fix lies in imperfections of the test suite: if the test suite contained a test case that overflows `cc`, the incorrect patch would not be accepted. It is possible to produce such a test case with automatic test generation (e.g., fuzz-testing), and use it to augment existing tests; we denote the test case that overflows `cc` as a tuple of [small-number, big-number, big-number] (since multiplication of two large integers leads to an overflow and produces a small integer value).

One of the crucial parts of our approach is a classifier that accepts values of faulty variables and decides whether those values belong to a bad or a good test. The classifier is trained on developer test cases and used to predict whether fuzz test cases are bad or good. Logistic regression is used as a classification algorithm. To collect values of faulty variables from developer test cases, we instrument the buggy program at the point of the failure (detected by fault localization), run the buggy program with developer test cases and collect values of the variables. Each execution of the faulty line of code produces a tuple of variable values. We label *all* the tuples of variable values from *passing* test cases as good and *the last* tuple from *failing* test cases as bad (the rest of values are ignored). After the classifier is trained on developer test cases, it is subsequently used to predict whether an *automatically* generated test cases is bad or good in the later stages of the approach.

Table 6.1: Results of classifying automatically-generated test cases.

Bug ID	Train.	Test.	Pr.	Rec.
gzip-a1d3d-f17cb	1/1	1/0	1.00	1.00
libtiff-08603-1ba75	5/4	11/5	0.46	0.62
libtiff-5b021-3dfb3	1/3	1/1	0.25	0.50
libtiff-90d13-4c666	1/0	-	-	-
libtiff-d13be-ccadf	1/8	1/145	0.99	0.01
libtiff-ee2ce-b5691	1/0	-	-	-

In the example, a tuple of specific values [`cc, tdir_count, w`] is a single classification instance; e.g., [6, 3, 2] or [22, 22, 1]. One run of a passing test case can produce several instances (since e.g. `TIFFFetchData` might be called multiple times); however, one run of a failing test case produces only one instance (we consider only the last tuple by assuming that only the last tuple leads to the failure). The classifier labels the new test case [small-number, big-number, big-number] as good: on this test case, the program behavior should be preserved; i.e., the program should still go to `bad`. However, given an oracle that detects incorrect behavior, the overfitted patch is filtered since the behavior was incorrectly changed.

We conducted a preliminary evaluation of the classifier on a number of defects from the GenProg benchmark [14]. As test data, we used values collected from fuzz-tests generated by American Fuzzy Lop [1] on the buggy version of the program. Note that the training data contains noise due to imperfections in developer oracles; therefore, some training instances were labeled as both bad and good. To mitigate this, we assume that the developer's oracle in never wrong; however, the developer might fail to incorporate the oracle into all test cases. Thus, there is a possibility of a situation in which a test case passes, although it should have failed. Based on the above, we say that if at least one of the developer tests says that an instance is bad, then we trust that it is indeed bad. Table 6.1 presents the results. The column "Bug ID" shows bug identifiers, the columns "Train." and "Pass." show the number of unique instances in the training set and in the test set respectively (number of bad instances/number of good instances), the columns "Pr." and "Rec." show the precision and recall of the classifier. Note that precision and recall are *average* precision and recall between results in classifying good and bad instances (i.e., if classifier's precision in recognizing bad instances is 0.5, and its precision in classifying good instances is 0.0, then the average precision is 0.25). To collect the ground truth for test instances, we manually created definitions of good and bad test cases based on the nature of the bug (similar to definitions of weakly relevant test cases from Chapter 4). Results

Table 6.2: True functions for classifying test cases as bad or good.

Bug ID	Definitions of good/bad test cases
gzip-a1d3d-f17cb	`var1 == 0`
libtiff-08603-1ba75	`var1 == 1 || var1 == 0 || var2 == 0`
libtiff-5b021-3dfb3	`(var1 == 2 || var1 == 3) && (var2 # {1, 2, 4, 16})`
libtiff-d13be-ccadf	`var1 == 2`
python-69223-69224	`var1 < 0`
python-70098-70101	`var1 == 0 && var2 > 0`

show that number of instances in the training set is often minuscule, ranging from 1 to 9. In some cases (`libtiff-90d13-4c666` and `libtiff-ee2ce-b5691`), there are no good instances to learn from which renders supervised classification impossible.

6.3 Reducing the hypothesis space

As a logistic-regression classifier showed unsatisfactory results in classifying test cases, there is a need for an alternative approach. Given our knowledge about what good and bad test cases are, it is possible to stop treating the classifier as a black box and apply this knowledge to form better hypotheses. In machine learning, the term "hypothesis space" refers to the set of all possible hypotheses that a classifier can form [3]. For example, if the classifier aims to divide a two-dimensional plane into clusters, classifier's hypothesis space might be a set of all possible rectangles, or a set of all possible lines to divide the plane. Thus, with our domain knowledge, we can *reduce* the hypothesis space to improve the effectiveness of classification.

We used the domain knowledge about the bugs to create *true functions*, or the best hypotheses. The hypotheses that our classifier would form are hoped to be as close to the true function as possible. Again, we intend to use values of the variables at the point of interest as input data to the classifier. Table 6.2 presents true functions for a number of defects to give the reader a sense of what types of hypotheses the classifier should aim for. The first column lists bug identifiers, the second column shows definitions of good or bad test cases based on the values of the variables at the point of interest (i.e., the buggy line). We do not state which definitions are for bad tests and which ones are for good tests since it is of low importance to reducing the hypothesis space. For example, for the bug `python-70098-70101`, the buggy line of code contains two variables, values of which can distinguish good tests from bad ones. A test case is bad if the value of the first variable equals to zero and the value of the second variable is greater than zero. The preliminary results show that the majority of true functions are fairly simple, so there might be a possibility of learning these functions based on data collected from developer test cases. The only exception is `libtiff-5b021-3dfb3`, which has a more complex true function; the hash symbol `#` stands for "one of the bits is set": e.g., `var # {0, 1}` denotes a number that has either bit 0 or bit 1 set to true (010_2, or 011_2, but not 100_2).

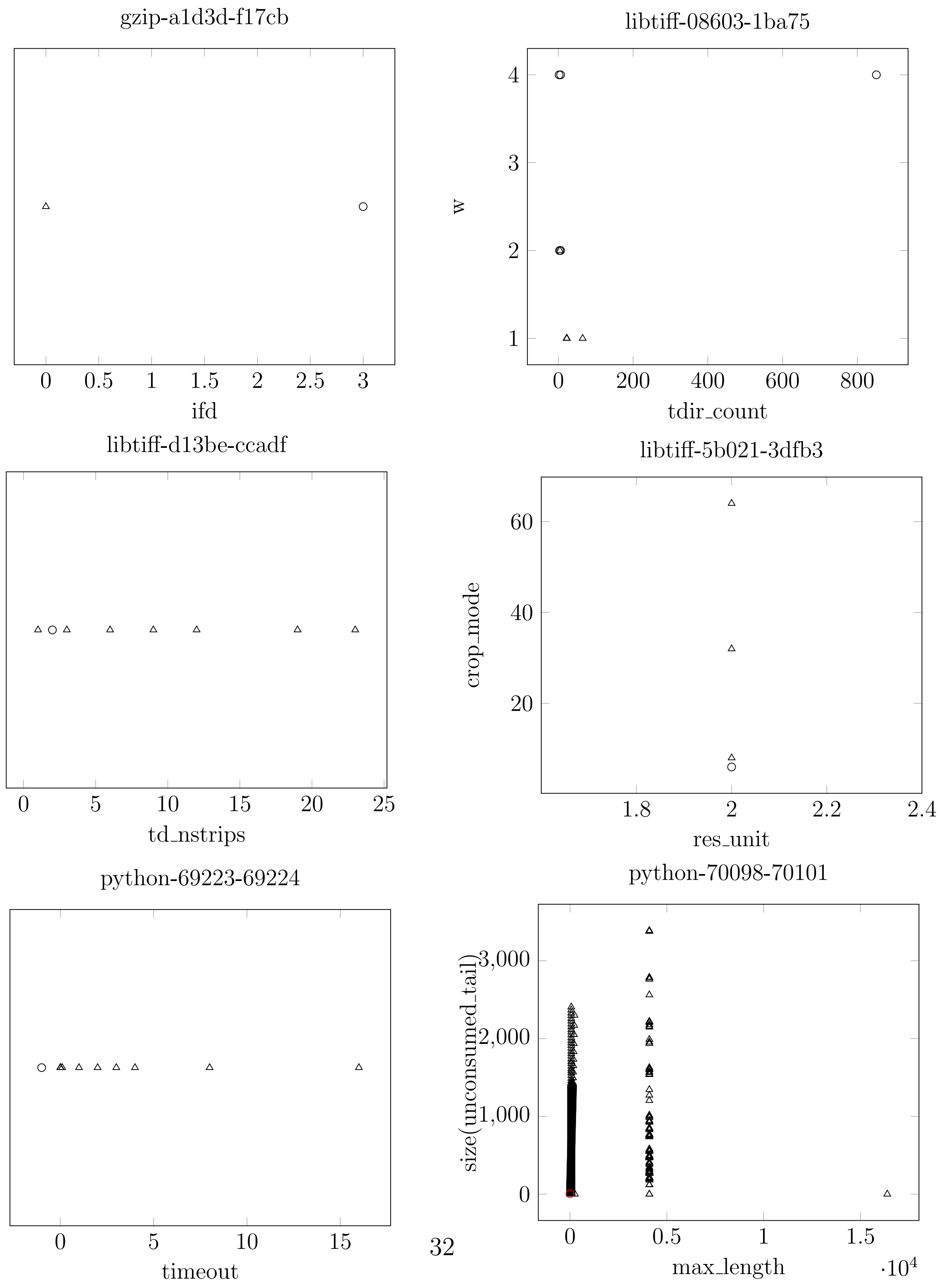
Figure 6.2 shows the training instances collected from the developer test cases. Some of the instances are one-dimensional (e.g., the ones for `gzip-a1d3d-f17cb`, which has only one variable that determines whether a test is bad or good), the other ones are two-dimensional (e.g., `libtiff-08603-1ba75`). Good instances are denoted by "$\triangle$", bad instances are denoted by "$\circ$". For example, the bug `libtiff-d13be-ccadf` has one bad instance, in which `var1` equals to two, and several good instances: 1, 3, 6, 9, 12, 19, and 23.

Figure 6.3 shows the instances collected from the automatically-generated fuzzing test cases for four defects. AFL, the fuzz-tester employed, was unable to generate any new test cases that would exercise the code in question. For three out of four defects, with `gzip-a1d3d-f17cb` as an exception, fuzzing produced new instances that can potentially be used in our approach for filtering overfitted patches.

One of the possible candidates to be used for test classification is DAIKON [6], a widely known dynamic invariant-detection tool. DAIKON instruments the program at the point of interest and executes it with given input. Then, based on the input, the hypotheses about potential program invariants are made (e.g., "variable `a` is less than zero"); these hypothesis are refined or rejected based on the following input. DAIKON's output consists of a set of invariants that are true for given variables under given input. Types of invariants that can be deduced by DAIKON are predefined and include a variable being a constant, a variable being in a range of values, linear relationships over two variables, etc. For example, in case of `libtiff-d13be-ccadf`, DAIKON can be used to infer invariants of the variable `td_nstrips`. First, DAIKON performs required instrumentation to start inferring invariants about `td_nstrips`. Then, to infer invariants that represent bad behavior, the program is executed with the bug-exposing test cases; in case of `libtiff-d13be-ccadf`, there is only one value that triggers the bug: `td_nstrips == 2`. As mentioned previously, DAIKON supports this kind of template which renders it possible to correctly infer that that value of `td_nstrips` represents bad behavior (and all other values correspond to good behavior). At the later stages of our approach, when evaluating automatically-generated patches, this information can be used to check that good behavior is preserved while bad behavior is altered.

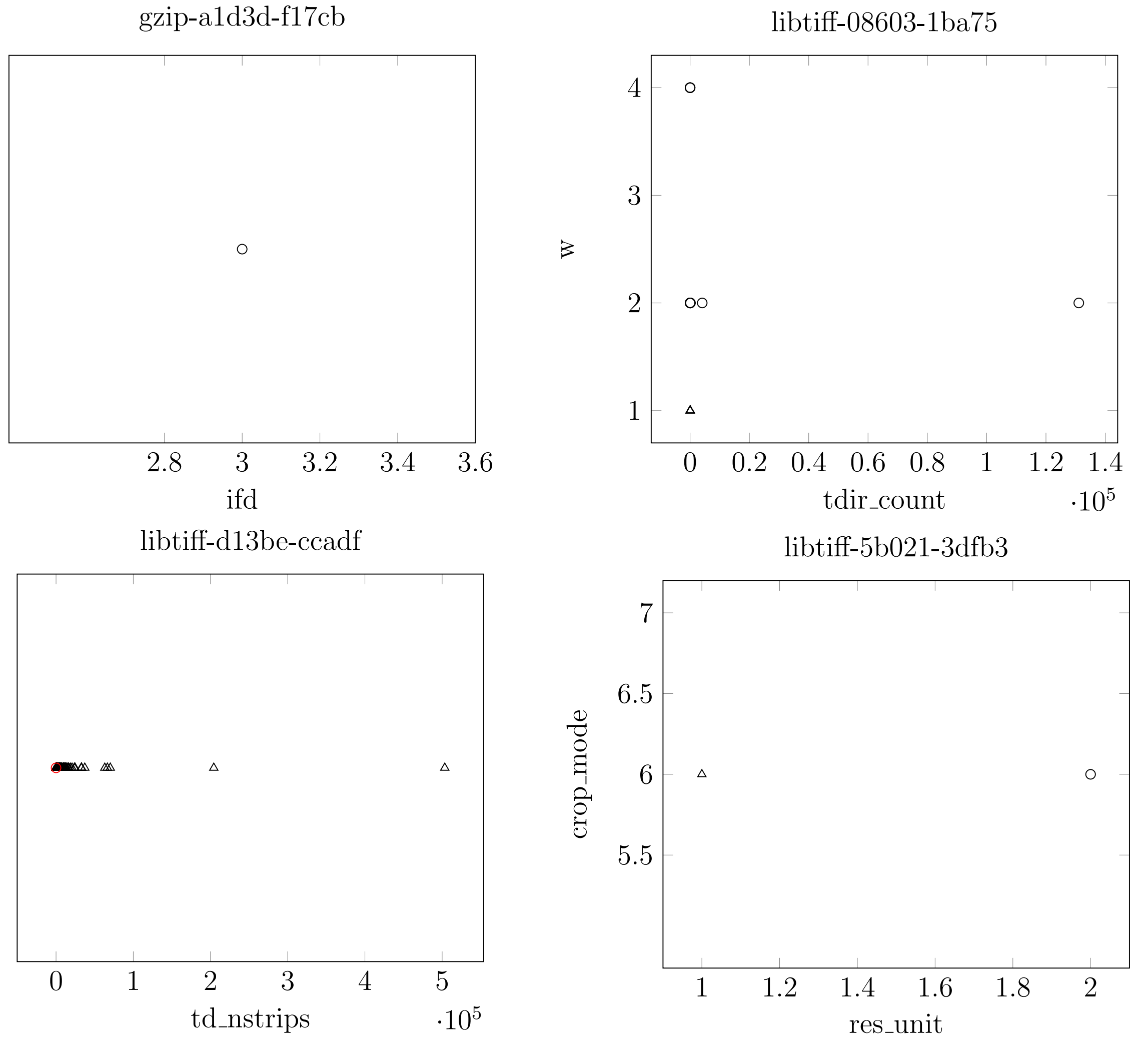
6.4 Related work

Weimer et al. [33] released a short technical report that lists ideas similar to the ones proposed in this chapter. For instance, Weimer et al. propose using supervised classification algorithms to detect whether the patched program has changed its behavior when executed on the regression test suite (compared to the buggy program). The work by Deborah S. Katz [11] provides more details in regard to the classification approach; the approach employs low-level information about the program execution, such as: “the number of unique instructions executed, the mean of all addresses read, the address of the most frequent stack write”, etc. In our approach, we employ values of variables at the point of interest as input to the classifier.



"△" good instances, "○" bad instances

Figure 6.2: Plots representing good and bad developer instances.



“△” good instances, “○” bad instances

Figure 6.3: Plots representing good and bad instances from the fuzz tests.

Chapter 7

Conclusions

Automated software repair offers a promising and enticing approach to reducing developers' bug-fixing workload. However, state-of-the-art automated software repair systems suffer from producing many incorrect patches due to insufficiencies of the test suites used for patch validation. By leveraging automatically-generated test cases and comparing the behavior between the buggy program and the patched program to detect worsening in behavior, our approach is able to filter 67% (279/417) of overfitted patches. For one of the defects, filtering overfitted patches lead to discovering a correct patch that was previously blocked.

In addition, we conducted a relevance analysis to study how many of the automatically-generated test cases can help developers in manual debugging and also potentially filter more overfitted patches: up to 40% of test cases have such a potential.

Finally, we outline the directions for future work to incorporate the findings of the relevance analysis into a new approach for filtering overfitted patches. We propose using a tailor-made classification algorithm to classify automatically-generated test cases as bad or good with the assumption that the patched program should change its behavior only on bad test cases. A feasibility study is conducted to show the promises of the approach.

References

[1] American fuzzy lop, 2017. http://lcamtuf.coredump.cx/afl/.

[2] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2008.

[3] Hendrik Blockeel. *Hypothesis Space*, pages 511–513. Springer US, Boston, MA, 2010.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number DSLAB-CONF-2009-002, 2009.

[6] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[7] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[8] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, (4):279–290, 1977.

[9] Project Management Institute. A guide to the project management body of knowledge. In *Project Management Institute*, volume 3, 2004.

[10] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014*

International Symposium on Software Testing and Analysis, pages 437–440. ACM, 2014.

[11] Deborah S Katz. Understanding intended behavior using models of low-level signals. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 424–427. ACM, 2017.

[12] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[13] Sakthi Kumaresh and R Baskaran. Defect analysis and prevention for software process quality improvement. *International Journal of Computer Applications*, 8(7), 2010.

[14] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[15] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[16] Richard J Lipton, Richard A DeMillo, and FG Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.

[17] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. *CoRR*, abs/1706.09120, 2017.

[18] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[19] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. ACM, 2016.

[20] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312. ACM, 2016.

[21] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 716–726. IEEE Press, 2012.

[22] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, pages 1–29, 2016.

[23] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: The state of the art. Technical report, DTIC Document, 2012.

[24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[25] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[26] TR Nair and V Suma. A paradigm for metric based inspection process for enhancing defect management. *ACM SIGSOFT Software Engineering Notes*, 35(3):1, 2010.

[27] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 254–265, 2014.

[28] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[29] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. ACM.

[30] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[31] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[32] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. Springer, 2008.

[33] Westley Weimer, Stephanie Forrest, Miryung Kim, Claire Le Goues, and Patrick Hurley. Trusted software repair for system resiliency. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 238–241. IEEE, 2016.

[34] Westley Weimer, Zachary P Fry, and Stephen Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[35] Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2017, 2017.

[36] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.

[37] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *ArXiv e-prints*, March 2017.

APPENDICES

Appendix A

Definitions of weak relevance

In addition to improving the developer test suites to prune overfitted patches, the test cases generated by random fuzzing can be used to facilitate the process of manual debugging; they also have the potential of filtering out more overfitted patches if automated program repairs systems are empowered with better test oracles. These test cases we call relevant. We propose the definition of a weakly relevant test case analogous to a weakly killed mutant in the context of mutation testing: a test case is weakly relevant, if it exposes a defect in any way. Usually, to distinguish weakly relevant test cases from the irrelevant ones, we compare the memory state of a program between the buggy and the developer versions when they are executed with a particular test case.

We present weakly definitions of relevant test cases using the following format:

Bug ID

- *Bug*: a description of the target bug
- *Definition*: the definition of a weakly relevant test case for this specific bug
- *Methodology* describing how to distinguish relevant test cases from irrelevant ones

Bug ID

- *Bug*:
- *Definition*:

- *Methodology*:

Note, that for the PHP and lighttpd defects, we conducted preliminary filtering to decrease the number of defects to be manually examined. For these defects, we instrumented the buggy version at the point where the developer patch should be; then we executed the binary on all the test cases we generated for this bug. If we generated no test cases that would execute the patched portion of the code, we do not create a definition of weakly relevance for this defect as no test cases can be relevant in this case. Below, such defects are omitted.

gzip-bug-3fe0c-39a36

- *Bug*: a malformed input file can cause `gzip` to crash with a segmentation violation or hang in an endless loop.
- *Definition*: the same as for the strong relevance—a test case is relevant if it makes the buggy version crash and the correct version pass.
- *Methodology*: follows from the definition

gzip-bug-a1d3d-f17cb

- The *bug* is in the command line arguments handling. Definition of weakly relevance is not applicable since no input can expose the bug (only the right combination of command line arguments can).

libtiff-bug-0860361d-1ba75257

- *Bug*: many benign inputs are erroneously rejected.
- *Definition*: test case is relevant if it is erroneously rejected.
- *Methodology*: if a test case is rejected by the buggy version and is accepted by the correct version, then it is relevant.

libtiff-bug-5b02179-3dfb33b

- *Bug*: `res_unit` erroneously defaults to ‘INCH‘ in the function `loadImage`

- *Definition*: every test case in which `res_unit` erroneously defaults to 'INCH' is a relevant test case.
- *Methodology*: if `res_unit` gets defaulted, the error branch is subsequently executed in the buggy version and the error branch is not executed in the developer version then the test case is relevant.

libtiff-bug-90d136e4-4c66680f

- *Bug*: if the program executes neither `goto fail` nor `goto success`, then the default return code is `EXIT_FAILURE`. However, according to a developer's comment[1], the default exit code of the program should be `EXIT_SUCCESS`.
- *Definition*: test case is relevant if the buggy version returns `EXIT_FAILURE` and the correct version returns `EXIT_SUCCESS`.
- *Methodology*: follows from the definition.

libtiff-bug-d13be72c-ccadf48a

- *Bug*: the method `EstimateStripByteCounts` is executed when it should not be.
- *Definition*: if the buggy version executes the method and the correct version does not, then the test case is relevant.
- *Methodology*: follows from the definition.

libtiff-bug-ee2ce5b7-b5691a5a

- Equivalent to `libtiff-90d136e4-4c66680f`, except it is not a regression case

lighttpd-bug-2330-2331

- *Functionality change*: "Add possibility to disable methods in `mod_compress` (#1773)"[2]
- *Definition*: test case is relevant if it executes the portion of the code that is to-be-patched and it contains `Accept-Encoding` in the request.

[1] http://git.ghostscript.com/?p=user/chrisl/libtiff.git;a=commit;h=b5691a5a

[2] http://redmine.lighttpd.net/projects/lighttpd/repository/1/diff?utf8=%E2%9C%93&rev=2331&rev_to=2330

- *Methodology*: filter out the patches that do not execute the patched portion of the code; out of the rest, pick the ones with `Accept-Encoding`.

php-bug-307562-307561

- *Bug*: `DOMDocument->saveHTML()` does not produce anything if it has an argument (a tag to save).
- *Definition*: test case is relevant if it contains a call to `saveHTML()` with an argument that represents an HTML tag.
- *Methodology*: manually examine the test cases that execute the patched portion and pick the ones that confirm to the definition.

php-bug-307846-307853

- *Bug*: "Bug #52290 ('setDate', 'setISODate', 'setTime' works wrong when 'DateTime' created from timestamp)"[3]
- *Definition*: test case is relevant if it creates an instance of `DateTime` from a time stamp.
- *Methodology*: manually examine the test cases that execute the patched portion and pick the ones that confirm to the definition.

php-bug-308262-308315

- *Bug*: an error message should not be emitted if `type == BP_VAR_IS`.
- *Definition*: if the buggy version emits the message and the developer version does not, then the test case is relevant.
- *Methodology*: follows from the definition, performed only on those test cases that execute the patched portion of the code.

php-bug-309579-309580

- *Bug*: calling the constructor of the `DatePeriod` class with the `NULL` argument crashes the interpreter.

[3]http://svn.php.net/viewvc?view=revision&revision=307853

- *Definition*: a test case is relevant if the constructor `DatePeriod(NULL)` is called
- *Methodology*: follows from the definition.

php-bug-311323-311300

- *Functionality change*: "Increase the overly conservative `pcre` backtrack limit from 100,000 to 1,000,000"[4]
- *Definition*: the buggy version does not go to error handling, however the developer test case indicates that it should. An automatically generated test case is relevant if it makes the developer version go to the error branch but not the buggy version.
- *Methodology*: B is number of times the buggy version goes to the error branch; P is the same for the developer version. A test case is relevant if $B < P$.

python-bug-69223-69224

- *Bug*: when the `select` function is called with negative timeout, it raises a `SelectError` exception instead of `ValueError`.
- *Definition*: a test case is relevant if the function `select_select` from the file `Module/selectmodule.c` is executed with a negative timeout.
- *Methodology*: follows from the definition.

python-bug-69368-69372

- *Bug*: in some cases code optimization must be skipped; the bug is that it is not.
- *Definition*: a test case is relevant if `python` optimizes the code when it should not.
- *Methodology*: the developer patch puts an if-condition into the code that handles the case in which the code should not be optimized. The test case is relevant if it makes the program go into that if-condition.

python-bug-69709-69710

[4] http://svn.php.net/viewvc?view=revision&revision=311323

- A bug or a functionality change? The issue #11223 describes two separate problems. One of them is a defect: "interruption of locks by signals not guaranteed when locks are implemented using POSIX condition variables". The other is a functionality change: "replace `threading._info()` by `sys.thread_info`". The SPR paper categorizes `python-bug-69709-69710` as a defect. However, the changeset 69709-69710 does not contain changes relevant to the defect, those changes are relevant only to the functionality change. In addition, the bug itself is not in the python source code but rather in a python's test case: one of the test cases should be skipped if the target platform is FreeBSD 6. This defect is fixed here. Thus, we categorize python-bug-69709-69710 as a functionality change and create a definition according to the category.

- *Functionality change*: "Replace threading._info() by sys.thread_info"[5].

- *Definition*: Every test case that accesses the fields of `sys.thread_info` is relevant.

- *Methodology*: no need in a methodology since all the newly generated test cases are derived from `test_os.py` that does not access any fields from `sys.thread_info`. To be certain, we grepped through the generated files to find any mentions of `sys.thread_info` and found nothing relevant.

python-bug-69783-69784

- *Functionality change*: two-digit year is no longer acceptable when supplying a date. Python used to perform the conversion of years as "01" → "2001", now it is "01" → "0001".

- *Definition*: a test case is relevant if it has a year value less than 1000 (expected to be converted).

- *Methodology*: if in the function `gettmarg` from the file `timemodule.c` we encounter year less than 1000, then such a test case is relevant.

python-bug-70019-70023

- *Bug*: python crashes when encoding highly-nested JSON documents.

[5]https://hg.python.org/cpython/rev/2b21fcf3d9a9

- *Definition*: test case is relevant if it contains a highly-nested JSON document.
- *Methodology*: the developer patch puts if-conditions into the code that handle the case of highly-nested JSON documents. A test case is relevant if it makes the program go into at least one of those if-conditions.

python-bug-70098-70101

- *Bug*: `zlib.decompressobj().decompress()` does not clear the `unconsumed_tail` attribute when called without the `max_length` argument.
- *Definition*: a test case is relevant if the attribute should have been cleared.
- *Methodology*: the developer patch puts an if-condition into the code that handles the case in which the attribute should be cleared. The test case is relevant if it makes the program go into that if-condition.

wireshark-bug-37112-37111

- *Bug*: a regression case—putting call to free inside the `functionfree_all_reassembled_fragmen` leads to a double-free error.
- *Definition*: test case is relevant if the buggy version crashes with double free error and the fixed version does not.
- *Methodology*: a test case is relevant if (1) for the buggy version, Valgrind shows double free and (2) for the fixed version Valgrind does not show double free.
- Note: After a double-free, the program should crash. However, the buggy and the patched version crashes on the very same test cases. Thus, there is no need to run them with Valgrind, we can immediately state that there are no relevant test cases.

wireshark-37172-37171, wireshark-37172-37173, wireshark-37284-37285

- *Bug*: depending on a CLI argument, the `tshark` binary might receive unexpected information from the `dumpcap` binary.
- *Definition* of weak relevance is not applicable since no input can expose the bug (only the right command line arguments can).