Better Test Cases for Better Automated Program Repair

Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, Lin Tan
j223yang,azhikhar,yuefei.liu,lintan@uwaterloo.ca
Electrical and Computer Engineering, University of Waterloo
Waterloo, ON, Canada

ABSTRACT

Automated generate-and-validate program repair techniques (G&V techniques) suffer from generating many overfitted patches due to in-capabilities of test cases. Such overfitted patches are incorrect patches, which only make all given test cases pass, but fail to fix the bugs. In this work, we propose an overfitted patch detection framework named *Opad* (*O*verfitted *PA*tch *D*etection). *Opad* helps improve G&V techniques by enhancing existing test cases to filter out overfitted patches. To enhance test cases, *Opad* uses fuzz testing to generate new test cases, and employs two test oracles (crash and memory-safety) to enhance validity checking of automatically-generated patches. *Opad* also uses a novel metric (named *O-measure*) for deciding whether automatically-generated patches overfit.

Evaluated on 45 bugs from 7 large systems (the same benchmark used by GenProg and SPR), *Opad* filters out 75.2% (321/427) overfitted patches generated by GenProg/AE, Kali, and SPR. In addition, *Opad* guides SPR to generate correct patches for one more bug (the original SPR generates correct patches for 11 bugs). Our analysis also shows that up to 40% of such automatically-generated test cases may further improve G&V techniques if empowered with better test oracles (in addition to crash and memory-safety oracles employed by *Opad*).

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

OverfiTting in automated program repair, Patch validation, Testing

ACM Reference format:
Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106274

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106274

1 INTRODUCTION

Automated generate-and-validate program repair techniques [14, 16, 18, 28, 29, 34] (G&V techniques) show promising results to reduce manual quality assurance efforts and to improve software reliability. G&V techniques automatically generate patches to repair buggy programs with the guidance of test cases and validate the correctness of the generated patches using the same set of test cases.

Despite the great potential, G&V techniques suffer from generating incorrect patches due to in-capabilities of test suites [18, 29, 32]. Qi et al. [29] pointed out that 98% of the patches that are generated by GenProg [16] are incorrect. A large portion of such incorrect patches are equivalent to deletion of buggy functionalities. These incorrect patches make test cases pass after the entire buggy code is removed, simply because the test cases do not cover the expected correct behaviors of the buggy code. For example, for the bug libtiff-08603-1ba75 (an arithmetic bug), GenProg generates incorrect patches that remove an integer overflow check to make these given test cases pass because they do not expose the integer overflows. Following previous work [32], we call such incorrect patches **overfitted patches**, since they are overfitted to pass only the given tests, but fail to fix the bugs.

Overfitted patches prevent G&V techniques from generating correct patches. The terminating condition of G&V techniques is to make all given test cases pass; thus, once an overfitted patch is generated, G&V techniques often stop exploring other patch candidates. This happens when the failing test case cannot well define the bug and/or if the original passing test cases fail to define all correct behaviors of the software. Thus, we need to improve test cases to precisely decide whether the generated patches overfit and make G&V techniques continue to generate correct patches.

There are limited prior efforts to enhance test cases for large and complex systems to further improve G&V techniques. Previous studies [32, 34] focus on illustrating the impacts of low-quality test suites on the quality of automatically-generated patches. Recent work [37, 39] on this direction demonstrates the challenges of using automated test generation to improve G&V techniques in a real-world setting. Xin et al. [37] design a new test generation technique to cover the generated patches by G&V techniques. However, it requires correctly-patched programs to get oracles (e.g., expected outputs), which is difficult to obtain in practice.

In this work, we propose an *O*verfitted *PA*tch *D*etection framework, *Opad*, that combines automated test generation, two oracles (crash and memory-safety), and a novel overfitness metric to detect overfitted patches. First, *Opad* improves existing test suites to better define bugs and preserve the desired functionalities from two angles: (1) generating new test cases automatically, and (2) leveraging additional oracles (i.e., memory-safety oracles) to improve validity checking of automatically-generated patches. For (1), *Opad* applies

fuzz strategies on input from existing test cases to automatically generates new test cases [1]. For (2), in addition to crash oracles for automatically-generated tests and manual oracles for existing test cases (e.g., expected output from developers), *Opad* obtains memory-safety oracles using Valgrind [3], a well-known memory bug detection tool, to ensure program memory safety.

This approach is analogous to the treatment of sickness: to determine if a patient has recovered (analogous to whether a bug has been fixed by a patch), in addition to checking if symptoms have been improved, doctors often (1) order laboratory tests such as blood tests (analogous to generating new tests), and (2) check if medical metrics such as white blood cell counts have been improved compared to those when a patient is sick (analogous to the improved validity checking).

Second, *Opad* leverages a novel metric, *the overfitness measure, O-measure* in short, to assist the improved test suite in detecting overfitted patches. The proposed *O-measure* is shown to be an effective approximation of the ideal metric that can best distinguish a correct patch from an overfitted patch. A prior study [32] shows that deciding whether a patch is overfitted using whether the patched version fails on any of the additional tests is imprecise in distinguishing overfitted from correct patches. Different from this prior study, our O-measure is built based on the assumption that a correctly patched program should not behave worse than the corresponding buggy program (e.g., fail on more test cases).

Third, we investigate how many of *Opad*'s automatically-generated test cases have the potential to filter out more overfitted patches if empowered with better test oracles (in addition to crash and memory-safety) through a post-mortem manual analysis. The usefulness of automatically-generated test cases is limited by weak oracles. Thus, we believe that there exist some automatically-generated test cases that, although they currently do not contribute to identifying overfitted patches due to the limitation of oracles, have the potential to filter out more overfitted patches. We call such test cases *weakly relevant* to the target bug.

We apply *Opad* to improve four G&V techniques, GenProg [16], AE [35], Kali [29], and SPR [18], in generating patches for 45 bugs. *Opad* automatically generates between 452 to 31,904 new test cases per bug, which include both passing and failing test cases. Our evaluation shows that:

- *Opad* filters out a significant portion (75.2%, 321/427) of overfitted patches generated by the four G&V techniques. With *Opad*, GenProg/AE, Kali, and SPR generate correct patches for 2, 3, and 12 bugs respectively.
- By filtering out overfitted patches, *Opad* helps SPR [18] generate a correct patch for one additional bug (libtiff-d13be-ccadf) compared to the original SPR (vanilla SPR generates correct patches for 11 bugs). In other words, without our approach, SPR fails to generate this correct patch. Although many overfitted patches are filtered out, *Opad* does not *always* lead to the generation of more correct patches, since, (1) to generate the correct patch, *all* overfitted patches that precede the correct one in the search space must be filtered, and (2) the search space must contain the correct patch.
- Our relevance analysis shows that, for each bug, up to 40% (2,310/5,967) of automatically-generated test cases are *relevant* to the target bug. The result indicates that a large portion of the automatically-generated test cases may filter out more overfitted patches if empowered with better test oracles.

In summary, this paper makes the following contributions:

- We enhance test suites for improving G&V techniques.
- We formulate the ideal and theoretical metric for determining if a generated patch is overfitted, and propose a novel practical metric for it.
- We explore and identify a scalable and practical approach to enhance existing test suites by generating new test cases and leveraging two oracles (crash and memory-safety).
- We evaluate the proposed approach by applying it to improve four G&V techniques to repair large and complex systems.
- We conduct a relevance analysis on automatically-generated test cases: we identify promising test cases that can filter out more overfitted patches if empowered with better test oracles (in addition to crash and memory-safety).

Availability. We make the data from this work available at http://asset.uwaterloo.ca/tests4repair.

2 BACKGROUND ON AUTOMATED G&V PROGRAM REPAIR

We briefly describe how G&V techniques (GenProg [16], Kali [29], AE [35], and SPR [18]) automatically generate patches given a buggy version, and given both failing and passing test cases. Figure 1 shows the typical structure of G&V techniques (at the top) and the structure of our patch validation framework (at the bottom). G&V techniques start with a buggy program (i.e., code that contains a target bug) and a set of failing and passing test cases to define the target bug. Then, G&V techniques utilize spectrum-based fault localization techniques to narrow down the scope of the faulty source code. After that, G&V techniques use specific approaches to construct a search space of patch candidates: 1) GenProg/AE, RSRepair [28], and Kali leverage different template-based operators to generate patch candidates (i.e., fixes); and 2) SPR uses parameterized templates. After constructing the search space, G&V techniques iterate the patches in the search space until they find a patch that can pass the patch validation (i.e., whether the patch can make the *same set of test cases* pass).

Despite their differences, G&V techniques share the same techniques of fault localization and patch validation. Imperfect patch validation (e.g., using the *same test cases* for both patch generation and validation) may lead to overfitted patches [32]. Overfitting hinders the effectiveness of G&V techniques: when there is a correct patch in the search space, G&V techniques risk failing to present it to the developer due to the correct patch being preceded by overfitted patches. In this paper, we propose an approach that filters out overfitted patches that precede the correct patch and helps G&V techniques generate more correct patches.

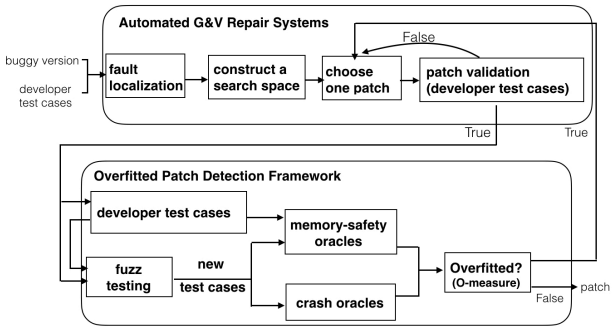

Figure 1: Overview of the Proposed Overfitted Patch Detection Framework (*Opad*) and How *Opad* is Integrated with G&V Techniques.

3 APPROACH

Overview. Figure 1 shows an overview of the proposed *O*verfitted *PA*tch *D*etection framework (*Opad*) for validating the correctness of automatically-generated patches. We also show how *Opad* can be used to improve G&V techniques. *Opad* employs automatic test generation, two test oracles (crash and memory safety), and a metric (Overfitness-measure: ***O-measure***) to assess the correctness of automatically-generated patches. First, to generate new test cases, *Opad* leverages fuzz testing and uses existing test suites as fuzzing seeds. Second, for all test cases (including automatically-generated test cases and developers' original test cases), *Opad* employs two additional oracles, a crash and a memory-safety oracle (e.g., buffer overflows, uninitialized variables, and memory leaks), to improve validity checking of automatically-generated patches. Third, based on the validity results, for each automatically-generated patch, *Opad* uses *O-measure* to decide whether a patch is overfitted. Figure 1 shows how *Opad* complements G&V techniques by deciding whether a generated patch is overfitted. *Opad* guides G&V techniques to continue choosing the next patch candidate in the search space if a patch is identified as overfitted.

Challenges. There are two main challenges in designing an overfitted patch detection framework based on automatically-generated new test cases for large and complex systems. The first challenge is how to leverage the generated tests and bug detection tools to determine if a patch is overfitted. A naive approach is that if a patch causes any automatically-generated test to fail the improved validity checking (e.g., the patched version contains a memory bug as reported by a bug detection tool), then we consider the patch overfitted. However, this approach is likely to filter out correct patches, because there are other irrelevant bugs (i.e., bugs are not related to the target bug) in the program. A correct patch may correctly fix the target bug, but fail to fix other *irrelevant* bugs in the program, i.e., bugs that are not targeted by the G&V tool (current G&V approaches are designed to fix only the target bug as defined by developer failing test cases). To address such irrelevant bugs, *Opad* uses O-measure (Section 3.3) that only considers a patch to be overfitted if the patched program performs worse than the buggy program under the same set of tests. Our assumption is that a correctly patched version should not behave worse than the buggy version, e.g., the patched version should not fail on the test cases on which the buggy version passes. Section 3.3 presents how and why we define O-measure.

The second challenge is the lack of test oracles: developer-written tests usually contain manually defined test oracles (e.g., `assert` statements that compare the expected output of a program with the actual output); however, it is an open challenge to automatically generate such test oracles [4]. To address this challenge, we leverage two oracles (crash and memory-safety) to help ensure the correctness of the patches. These two oracles are correct because programs *should not* crash under any circumstances (i.e., a crash is a definite indication of a bug in the program) and *should not* violate memory safety (e.g., memory leaks). By adding new test cases, the memory-safety oracles can guarantee memory safety of more code execution paths in the program by patched G&V techniques.

3.1 Generating New Test Cases Using Fuzz Testing

In order to generate new test cases, we use fuzz testing [25]—a well-established bug-finding technique that feeds the program under test with randomly-generated input. We choose fuzz testing due to the following constraints when improving G&V techniques on large and complex systems. First, fuzz testing is scalable to large and complex systems (i.e., programs of millions of lines of code). Currently, many other advanced automatic test generation techniques do not work for programs of such scale. Second, fuzz testing can be applied to a wide spectrum of software (from image manipulation programs to interpreters). Finally, our benchmark consists of C programs, for which there are limited tools available; unlike other languages, there are well-established tools (e.g., Randoop [26] and EvoSuite [8] for Java). Primitive fuzzing techniques rarely find errors deep within programs' control flow because the randomly-generated input is usually rejected at early stages of error checking. To mitigate this issue, mutation-based fuzzing was proposed [11, 33]. Mutation-based fuzzers perform random mutations on well-formed input which allows mutated input to pass initial sanity checks and trigger the bugs that lie deeper in the program. In this work, we use American Fuzz Lop (AFL) [1], a coverage-guided fuzz-testing tool, to generate new test cases for the bugs in the evaluated benchmark. AFL is a mature mutation-based fuzz-testing tool that detects significant vulnerabilities in mature C projects [1]. AFL works by applying mutation rules on input, by selecting the new input that explores new paths (to achieve higher coverage), and by continually mutating the newly created input until all inputs are explored or AFL is terminated manually.

3.2 Generating Memory-Safety Oracles

Opad employs memory-safety oracles on both newly automatically-generated test cases and developer test cases to improve validity checking of automatically-generated patches. *Weak oracles* (e.g., checking only whether a program crashes) are not sufficient to guarantee program correctness. This is true for both developer test cases and automatically-generated test cases. To mitigate this, *Opad* enhances validity checking of patches by inspecting the quality of memory management and ensuring memory safety.

To validate large and complex systems, we need a practical and scalable memory-safety checker. We chose dynamic analysis over

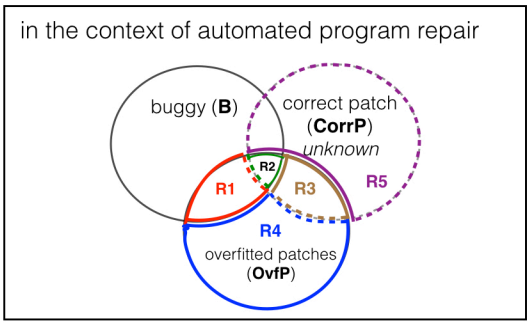

Figure 2: Sets of failing test cases on the buggy version (B), the versions with overfitted patches (OvfP), and the version with correct patch (CorrP).

static analysis, since static analysis tools may generate too many false positives. This makes static analysis unsuitable for our purpose since false positives might erroneously prune overfitted patches (not due to the defect in the patch); in addition, false positives are likely to prune correct patches as well.

Opad leverages Valgrind [3] (i.e., Memcheck) for memory-safety oracles. Specifically, *Opad* applies Valgrind with each test case (i.e., either from a developer or automatically-generated) and records the detection results from Valgrind (i.e., memory errors and leaked memory bytes). Valgrind inspects memory safety by instrumenting the program under test, keeping track of validity of all unallocated/allocated memory, and reporting errors once memory safety is violated. Valgrind can detect various memory-related problems, such as using undefined values, accessing already-freed memory, and memory leaks.

3.3 Measuring the Overfitness of a Patch Using an Overfitness Metric (O-measure)

In this subsection, we present our definition of *O-measure*. *Opad* uses O-measure to determine whether automatically-generated patches overfit. Then, we provide a justification about why our proposed definition of *O-measure* works best under both theoretical and practical constraints for G&V techniques.

3.3.1 Defining O-measure. We propose a metric, *O-measure*, to identify overfitted patches. The proposed *O-measure* is calculated based on the results of executing test cases (both developer and automatically-generated) against two oracles (crash and memory-safety).

We present the definition of O-measure and how to use O-measure to decide overfitness of patches below.

Definition 3.1. 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 $\overline{B} \cap P$.

Definition 3.2. A patch is overfitted if it has a non-zero O-measure, and not overfitted otherwise.

3.3.2 Calculating O-measure. *Opad* executes each test case on both versions (the buggy and the patched versions) and records the oracle-related execution results (i.e., whether the program crashes and memory-safety detection results). Based on the results, *Opad* calculates O-measure to determine the overfitness of patches. If O-measure is non-zero for a patch, *Opad* determines the patch to be overfitted, and not overfitted otherwise.

It is straightforward to calculate O-measure for test cases with crash oracles. For memory-safety oracles, *Opad* decides whether a test case contributes to O-measure ($\overline{B} \cap P$) by checking whether the patched version exposes more memory issues than the buggy version. Different from crash oracles, for which the result is a binary value (whether the program crashes), memory-safety oracles produce comprehensive memory detection results. Thus, simply using whether memory safety is violated for deciding failure is not sufficient. Instead, we calculate O-measure by checking whether the patched version exposes more memory issues than the buggy version. For example, Valgrind reports memory errors (e.g., "use of uninitialized values") and the number of bytes leaked (definitely/indirectly/possibly lost). If for a test case, the patched version contains extra memory errors or extra leaked bytes of the three types above-mentioned, the value of O-measure of this patch is incremented by one.

3.3.3 Reasons Behind Our Choice of O-measure. The proposed definition of O-measure (Definition 3.1) is merely one possible way to define overfitness of patches. We illustrate why we propose this O-measure definition from both theoretical and practical aspects.
The Ideal Overfitness Measure (O-measure). We define the *ideal O-measure* as the O-measure that can perfectly distinguish overfitted patches from correct patches. Figure 2 demonstrates the relationship among the sets of failing test cases on the buggy version (annotated as B), on the correctly-patched version ($CorrP$), and on the overfittedly-patched version ($OvfP$). We use $\overline{B}$, $\overline{CorrP}$, and $\overline{OvfP}$ to annotate the sets of *passing* test cases on the buggy version, the correctly-patched version, and the overfittedly-patched version. In Figure 2, the five regions are highlighted: $R1$ is $B \cap OvfP \cap \overline{CorrP}$; $R2$ is $B \cap OvfP \cap CorrP$; $R3$ is $\overline{B} \cap OvfP \cap CorrP$; $R4$ is $\overline{B} \cap OvfP \cap \overline{CorrP}$; and $R5$ is $\overline{OvfP} \cap CorrP$.

The *ideal O-measure* should be able to differentiate between correct and overfitted patches. This means that there exists at least one test case that shows different behaviors (i.e., fail or pass on the oracle) on the two versions (i.e., the one with the correct patch, and the one with an overfitted patch). So, the *ideal O-measure* for deciding overfitness is the size of the set $(OvfP \cap \overline{CorrP}) \cup (\overline{OvfP} \cap CorrP)$ ($R1 \cup R4 \cup R5$ in Figure 2). If the *ideal O-measure* of a patch is non-zero, this patch is overfitted as it has different behaviors from the correct patch on at least one test case.
From the Ideal O-measure to Our Definition of O-measure (Definition 3.1). The *ideal O-measure* is annotated as $R1 \cup R4 \cup R5$ (Figure 2). In the context of automated program repair, the correct patch is not available. This means that $R5$ (which is a subset of $CorrP$) is hard to approximate in practice. Thus, we take the *first-step* approximation of the *ideal O-measure*: using $R1 \cup R4$ (a subset of $OvfP$). However, $R1 \cup R4$ (a subset of $OvfP$) still cannot be directly computed due to the unavailability of $CorrP$: $R2$ and $R3$ cannot

be excluded precisely. We take the *second-step* approximation: using $R3 \cup R4$ in Figure 2 (our O-measure definition, Definition 3.1) to approximate $R1 \cup R4$ by excluding $R1$ and including $R3$. $R1$ is $B \cap OvfP \cap \overline{CorrP}$, and $R3$ is $\overline{B} \cap OvfP \cap CorrP$. The inclusion of $R3$ in inevitable to approximate $R4$ due to the unavailability of correct patch. Below, we illustrate why the exclusion of $R1$ is a reasonable choice in the context of using *Opad* to improve G&V techniques.

First, we prove that for a particular type of bugs and their corresponding overfitted patches, $R1$ is empty in theory. If $R1$ is empty, $R1 \cup R4$ (the first-step approximation described above) equals to $R4$. Thus, for these cases, using $R3 \cup R4$ (our O-measure, a superset of $R4$) will identify all overfitted patches that can be identified by the first-step approximation of the *ideal O-measure*. We describe the proof in "*Proving the emptiness of* $B \cap OvfP \cap \overline{CorrP}$ *for specific cases*" below. We manually investigate how many bugs and their corresponding overfitted patches (GenProg 2012 benchmark that we use for evaluation) fall into this particular pattern that we prove. $R1$ is empty for 19% of the bugs (7/36, 36 bugs for which there is at least one overfitted patch from the four G&V techniques), and their corresponding 34 overfitted patches.

Second, $R1$ has to be approximated using $R1 \cup R2$ ($B \cap OvfP$). Such approximation introduces the inclusion of $R2$. Since $R2$ is part of $CorrP$ and is not part of the *ideal O-measure*, the inclusion of $R2$ causes two risks: 1) ineffectiveness in filtering out overfitted patches, especially if $R2 == B \cap OvfP$; and 2) incorrectly filtering out correct patches. Empirically, we find that both of the two risks are true in the evaluation: the bugs in the GenProg 2012 benchmark, the patches from the four G&V techniques (GenProg/AE, Kali, and SPR), and automatically-generated test cases by *Opad*. Particularly, for 92% of the overfitted patches in the evaluation, $R2 == B \cap OvfP$. This shows that using $B \cap OvfP$ to approximate $R1$ is ineffective to filter out overfitted patches since for most cases, $R1$ is empty. In addition, $B \cap CorrP$ (i.e., $R1 \cup R2$ when an automatically-generated patch is correct instead of overfitted) is not empty for correct patches of 53% of the bugs. This shows that using $R1 \cup R2$ as O-measure or part of O-measure would incorrectly filter out correct patches for 53% of the evaluated bugs. This echoes with previous work [32], which shows that using $OvfP$ as O-measure is ineffective.

In summary, we choose the definition of O-measure (Definition 3.1) due to both theoretical and practical concerns. The intuition behind our O-measure is that the patched program should not behave worse than the buggy program.

Proving the emptiness of $B \cap OvfP \cap \overline{CorrP}$ ***for specific cases***. This proof is to show that for a particular type of bugs and their corresponding overfitted patches, the proposed O-measure is the most reasonable metric to distinguish between correct and overfitted patches. Note that the proposed O-measure is not tied to this particular type of bugs, and it also applies to other bugs (as shown in the evaluation).

We first describe the particular type of bugs and its corresponding overfitted patches, and then show that for these bugs and patches, there do not exist test cases in $R1$ ($B \cap OvfP \cap \overline{CorrP}$) in Figure 2. First, the code structure of this particular type of bugs is:

if (cond) S1; **else** S2;

where S1 and S2 are code statements. Second, this particular type of bugs and their corresponding overfitted patches satisfy the following conditions (which constitute 19% of the studied benchmark):

(A_1) I represents the entire input space; $I = I1 \cup I2$ and $I1 \cap I2 = \varnothing$.

(A_2) On a buggy program (B), for every input i in I, S1 is always executed.

We use "$B(I \rightsquigarrow \text{S1})$" to represent that on a buggy program, S1 is executed for every input in I. In the context of G&V techniques, there must exist at least one test case (i.e., a pair of an input i and an oracle) so that $B(i \rightsquigarrow \text{S1})$ leads to a failure as the oracle is not satisfied. This proof should cover all possible test cases in theory. It is unnecessary and unrealistic to obtain the result of executing every possible test case because the proof is generalizable for both cases: $B(I \rightsquigarrow \text{S1})$ leads either a failure or a pass.

(A_3) Overfitted patches modify conditions to redirect every input in I to execute S2.

(A_4) Correct patch makes the failing test case pass by redirecting every input in $I1$ to execute S2, while keeping every input in $I2$ to execute S1.

(A_5) Both overfitted and correct patches change program executions by only modifying cond. Thus, such patches have no side effects on other parts of the program other than that the execution flow is changed, e.g., from executing S1 to S2. This means, for example, for the same input i, the results of executing $B(\text{i} \rightsquigarrow \text{S1})$, $CorrP(\text{i} \rightsquigarrow \text{S1})$, and $OvfP(\text{i} \rightsquigarrow \text{S1})$ are the same as long as they all execute S1.

Proof of emptiness of $B \cap OvfP \cap \overline{CorrP}$. We start by inferring the following facts from the conditions:

(F_1) $CorrP(I1 \rightsquigarrow \text{S2})$.
From A_4.

(F_2) $OvfP(I1 \rightsquigarrow \text{S2})$.
From A_1 and A_3.

(F_3) $CorrP(I1 \rightsquigarrow \text{S2}) == OvfP(I1 \rightsquigarrow \text{S2})$.
From F_1, F_2 and A_5. This means that $CorrP(I1 \rightsquigarrow \text{S2})$ and $OvfP(I1 \rightsquigarrow \text{S2})$ have the same result, i.e., either both failure or both pass.

(F_4) $CorrP(I2 \rightsquigarrow \text{S1})$.
From A_4.

(F_5) $B(I2 \rightsquigarrow \text{S1})$.
From A_1 and A_2.

(F_6) $CorrP(I2 \rightsquigarrow \text{S1}) == B(I2 \rightsquigarrow \text{S1})$.
From F_4, F_5, and A_5. Similar to F_3.

We prove by contradiction. If $B \cap OvfP \cap \overline{CorrP}$ is not empty, there exists at least one test case that satisfies all three conditions: fails on B (denoted as $Condition_1$), fails on $OvfP$ ($Condition_2$), and passes on $CorrP$ ($Condition_3$).

From A_1, the input of such test case must be either $I1$ or $I2$: 1) if the input is $I1$, based on F_3, $CorrP(I1 \rightsquigarrow \text{S2})$ and $OvfP(I1 \rightsquigarrow \text{S2})$ should have the same result, either both failures or both passes. This means that $Condition_2$ and $Condition_3$ cannot be satisfied at the same time; and 2) if the input is $I2$, based on F_6, $CorrP(I2 \rightsquigarrow \text{S1})$ and $B(I2 \rightsquigarrow \text{S1})$ should have the same result, thus $Condition_1$ and $Condition_3$ cannot be satisfied at the same time.

Thus, such test case that satisfies all the three conditions does not exist, which means $B \cap OvfP \cap \overline{CorrP}$ is empty.

□

The proof above shows that the proposed O-measure is the most reasonable one for this particular type of bugs. In addition, the O-measure also works well for other bugs (as shown in the evaluation).

3.4 An Optimized Setting of *Opad*

Opad calculates O-measure based on running test cases against test oracles. Since *Opad* uses O-measure by only asserting whether it is zero or not, *Opad* can be optimized by deciding a patch is overfitted as soon as O-measure becomes non-zero. For example, for a patch from G&V techniques, once a test case (new or developer test case) against test oracles (i.e., crash or memory-safety) fails on the patched version but not on the buggy version, *Opad* decides this patch is overfitted. Furthermore, when examining the next patch from the search space of G&V techniques, *Opad* can prioritize running the test cases with oracles that have contributed to filtering out overfitted patches before. In our evaluation, we evaluated *Opad* without this optimization to get a full understanding of the effectiveness of O-measure unless specified. However, we find that, by using this optimization, we can significantly speed up *Opad* (e.g., from over 100 to less than 10 minutes for *Opad* to guide SPR to generate a correct patch for libtiff-d13be-ccadf, a loose condition bug).

4 EVALUATION

In this section, we present the experimental setup and the three research questions we answer.

Experimental Setup. We evaluate *Opad* on the same set of bugs evaluated by previous work (GenProg, AE, Kali, and SPR). Particularly, we select all bugs for which at least one of the four repair tools have generated at least one patch. In total, we apply our approach on 45 bugs from 7 systems, and 449 corresponding patches (both overfitted and correct ones) that are generated by G&V techniques. To generate new test cases, we feed AFL (Section 3.1) with input from non-crashing developer test cases (i.e., test cases that do not make the program crash). Such non-crashing test cases include all passing test cases and some failing test cases if the failures are observed by non-crash oracles (e.g., defined expected output). The reason is that AFL, by its design, does not mutate crashing test cases in order to avoid focusing on the exact same crash. We terminate AFL when no new paths are explored within two hours, since AFL may keep running without manual interruption. AFL leverages coverage to guide the mutation for better performance, and the coverage is obtained by running executables from the program under test. For some evaluated systems that contain more than one executable, we only apply AFL on the executables that are identified to expose the target bug by developer test cases. We run each automatically-generated test case against crash oracles ten times to mitigate possible non-determinism. This number was chosen as an acceptable trade-off between efficiency of running test cases and efficacy of mitigating non-determinism. The experiment is primarily conducted in the virtual machine image released by Le Goues et al. [16], except for SPR's patches that are obtained from the SPR virtual machine [18]. We host the virtual machines on computers with 16G RAM and 3.10 GHz Intel i5 CPU.

RQ1: How many overfitted patches does *Opad* filter out?

Motivation. Identifying overfitted patches is crucial for G&V techniques since it allows them to continue exploring the search space to eventually find the correct patch. Note that it is not realistic for G&V techniques to iterate the entire search space to find all patches that make the test cases pass. As stated in a recent study [19], there can be up to thousands of overfitted patches per search space. So, stopping at the first patch that makes all the test cases pass is a reasonable design choice for G&V techniques. Even if one generates all patches that make the test cases pass, filtering out overfitted patches could still save developers' time in selecting the correct one, as often a few correct patches are hidden among many overfitted patches [19].

Approach. We evaluate *Opad* on four automated G&V techniques (GenProg, AE, Kali, and SPR) to study whether our approach can correctly filter out overfitted patches while preserving correct patches. Specifically for GenProg, AE and Kali, the generated patches are publicly available. So we apply our approach on the released patches and report how many overfitted patches are successfully pruned by *Opad*. We obtained the ground truth for correct and overfitted patches from Qi et al. [29]. A patch is correct if it fixes the bug. Conversely, a patch is overfitted if it merely causes the test cases to pass and does not fix the bug. For the bugs that we evaluate, the correct patches from G&V techniques are semantically equivalent to developer patches.

Results. In total, *Opad* filters out 75.2% (321/427) overfitted patches from the four automated G&V techniques. Table 1 shows the overall result of filtering out overfitted patches. Table 1 contains all the bugs (45 in total) from the GenProg 2012 benchmark, for which at least one of the four G&V techniques can generate patches (i.e., overfitted or correct). Since AE is an adaptive version of GenProg based on a different search algorithm, we merge GenProg and AE into one column. To show the improvement from different components of *Opad*, we show the number of pruned patches in four settings: 1) using crash oracles on new test cases from fuzz testing (Column "Crash + Fuzz"); 2) using memory-safety oracles on developer test cases (Column "Mem. + Dev."); and 3) using memory-safety oracles on new test cases from fuzz testing (Column "Mem. + Fuzz"); 4) using the combination of all the above (*Opad*, Column "All").

GenProg/AE often generate several patches for a bug, so we show the total number of patches per bug in Column "GenProg/AE"/"Total". Kali generates one patch per bug, which is a total of 17 overfitted patches from Kali (we omit the column that shows the total number of patches for Kali). For some bugs that SPR has correct patches in the search space, we set SPR to continue exploring the search space until a patch is accepted by *Opad*. Therefore, the number of patches from SPR that are filtered out by *Opad* may be more than one. Column "SPR"/"Total" shows the total number of overfitted patches from SPR that are evaluated by *Opad*.

The three components of *Opad* mostly complement each other: 1) using crash oracles on fuzz test cases filters out 276 overfitted

Table 1: The results of using *Opad* to filter out overfitted patches from GenProg/AE, Kali, and SPR. 'Total' is the number of overfitted patches evaluated by *Opad*; for Kali, this number is always one (unless a Kali's patch is correct). Check symbol (✓) means that GenProg/AE, Kali, or SPR find the correct patch, and these correct patches are not incorrectly pruned by our approaches. Double check symbol (✓✓) means that *Opad* guides SPR to generate a correct patch (original SPR does not generate this correct patch).

Bug	GenProg/AE					Kali				SPR				
	Total	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Total	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All
gzip-3fe0-39a3	9	3	0	0	3	1	0	0	1	1	0	0	0	0
gzip-a1d3-f17c	1	0	0	0	0	-	-	-	-	1	0	0	0	0
libtiff-08603-1ba75	6	5	0	0	5	0	0	0	0	1	1	1	1	1
libtiff-5b021-3dfb3	9	9	1	9	9	1	0	1	1	238	238	0	0	238
libtiff-90d13-4c666	1	0	0	0	0	0	0	0	0	1	0	0	0	0
libtiff-d13be-ccadf	6	3	0	0	3	1	0	0	1	13	13 ✓✓	0	0	13 ✓✓
libtiff-ee2ce-b5691	1	0	0	0	0	0	0	0	0	0	0 ✓	0 ✓	0 ✓	0 ✓
lighttpd-1794-1795	10	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1806-1807	6	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1913-1914	1	0	0	0	0	-	-	-	-	1	0	0	0	0
lighttpd-1948-1949	-	-	-	-	-	-	-	-	-	1	0	0	0	0
lighttpd-2330-2331	9	0	2	0	2	-	-	-	-	1	0	0	0	0
lighttpd-2661-2662	9	0	0	0	0	-	-	-	-	1	0	0	0	0
python-69223-69224	1	0	0	0	0	-	-	-	-	1	0	0	0	0
python-69368-69372	-	-	-	-	-	-	-	-	-	1	0	0	1	1
python-69709-69710	-	-	-	-	-	-	-	-	-	1	0	0	0	0
python-69783-69784	3	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	1	0	0	0	0
python-70019-70023	-	-	-	-	-	-	-	-	-	1	0	0	0	0
python-70098-70101	1	-	1	0	1	0	1	0	1	1	0	0	0	0
wireshark-37112-37111	10	0	10	0	10	0	1	1	1	1	0	0	1	1
wireshark-37172-37171	1	0	1	0	1	0	0	0	0	1	0	0	0	0
wireshark-37172-37173	1	0	1	0	1	0	0	0	0	1	0	1	0	1
wireshark-37284-37285	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-307562-307561	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0	0
php-307846-307853	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0	0
php-307914-307915	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-307931-307934	9	0	0	1	1	0	0	0	0	1	0	0	0	0
php-308262-308315	-	-	-	-	-	-	-	-	-	3	0	2	0	2
php-308323-308327	-	-	-	-	-	-	-	-	-	1	0	0	1	1
php-308525-308529	1	0	0	0	0	0	0	0	0	1	0	1	1	1
php-308734-308761	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309111-309159	1	0	0	0	0	-	-	-	-	1	0	0	0	0
php-309516-309535	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309579-309580	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309688-309716	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-309892-309910	0	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0 ✓	0	0 ✓	0 ✓	0 ✓	0 ✓
php-309986-310009	10	0	5	0	5	0	0	0	0	1	0	1	0	1
php-310011-310050	9	0	6	5	8	0	1	0	1	6	0	5	0	5
php-310370-310389	-	-	-	-	-	0	1	0	1	1	0	0	0	0
php-310673-310681	2	0	0	0	0	0	0	0	0	1	0	0	0	0
php-310991-310999	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
php-311323-311300	-	-	-	-	-	-	-	-	-	1	0	0	0	0
php-311346-311348	-	-	-	-	-	0 ✓	0 ✓	0 ✓	0 ✓	0	0 ✓	0 ✓	0 ✓	0 ✓
gmp-13420-13421	-	-	-	-	-	-	-	-	-	0	0 ✓	0 ✓	0 ✓	0 ✓
gmp-14166-14167	3	0	0	0	0	0	0	0	0	1	0	0	0	0
Sum	120	20	27	15	49	3	4	2	7	290	252	11	5	265

patches ("Crash+Fuzz"); 2) using memory-safety oracles on developer test cases filters out 42 overfitted patches ("Mem+Dev."); and 3) using memory-safety oracles on fuzz test cases filters out 24 overfitted patches ("Mem+Fuzz").

Opad does not filter out correct patches incorrectly for ten bugs. We use '✓' or '✓✓' in Table 1 to annotate the bugs that *Opad* preserves the correct patches for them. *Opad* filters out three correct patches (libtiff-5b021-3dfb3, php-307562-307561, and php-307846-307853) because the correctly-patched programs behave worse than

Table 2: Results of using *Opad* to improve SPR (SPR+*Opad*) on the 19 bugs from the GenProg 2012 benchmark. Each cell contains two symbols. The first symbol shows whether SPR+*Opad* generates a correct patch (Y or N); and the second symbol shows how *Opad* contributes in the patch generation process—✓: filtering out overfitted patches, –: not filtering out patches (neither overfitted nor correct), *B*: filtering out both overfitted and correct patches, and ×: filtering out correct patches only.

Bug ID	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All	Bug ID	Crash + Fuzz	Mem. + Dev.	Mem. + Fuzz	All
gzip-a1d3d-f17cb	N –	N –	N –	N –	php-309111	N –	N –	N –	N –
libtiff-5b021-3dfb3	N B	N –	N –	N B	php-309516	Y –	Y –	Y –	Y –
libtiff-d13be-ccadf	Y ✓	N –	N –	Y ✓	php-309579	Y –	Y –	Y –	Y –
libtiff-ee2ce-b5691	Y –	Y –	Y –	Y –	php-309688	N –	N –	N –	N –
python-69783-69784	N –	N –	N –	N –	php-309892	Y –	Y –	Y –	Y –
php-307562	Y –	Y –	N ×	N ×	php-310011	N –	N ✓	N –	N ✓
php-307846	Y –	Y –	N ×	N ×	php-310991	Y –	Y –	Y –	Y –
php-307914	Y –	Y –	Y –	Y –	php-311346	Y –	Y –	Y –	Y –
php-308262	N –	N ✓	N –	N ✓	gmp-13420	Y –	Y –	Y –	Y –
php-308734	Y –	Y –	Y –	Y –					

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

Figure 3: A bug hidden in the buggy version of libtiff-5b021-3dfb3.

the buggy programs based on O-measure (i.e., either crash or fail the memory-safety oracles on some test cases, while the buggy program does not). *This happens because there are some hidden bugs in the buggy version, and such hidden bugs are exposed after the patches are applied.* Such hidden bugs are exposed in the patched version once the patch changes the control flow of the program (some of these hidden bugs are later fixed by the developers).

We show an example of a hidden bug. In libtiff-5b021-3dfb3, a hidden bug that is caused by a failed assertion is newly exposed by a correct patch (line 2 in Figure 3). We reported the bug to libtiff developers and the bug has been fixed[1]. This failed assertion should have been removed after the functionality had been changed. The earlier version of this function contains the assertion to abort the program if `value` is invalid. However, this function was later modified to capture an invalid `value` in the `if`-branch (line 10) and the assertion became obsolete. The buggy version exits before reaching the assertion (due to the nature of the target bug), but the correctly patched version continues the execution until the assertion fails; this results in a non-zero O-measure for the correct patch. Nonetheless, the generated test cases and our approach should help developers fix the new bugs in the patched version and, ultimately, improve the quality of the software.

RQ2: Can *Opad* guide SPR to generate correct patches for more bugs?

Motivation. We want to evaluate whether *Opad* can improve automated G&V techniques in terms of generating correct patches for more bugs. In this evaluation, we focus on SPR because SPR is shown to have great potential: there are many correct patches in the SPR search space that are not discovered because they are blocked by overfitted patches. A prior study [19] shows that there are no more correct patches in the search space of GenProg/AE and Kali to be discovered for the bugs in this evaluation. Therefore, although our approach filters out many overfitted patches generated by GenProg/AE, continuing running GenProg/Kali will not generate more correct patches. In contrast, SPR has correct patches for eight more bugs in its search space (but fails to generate correct patches due to having too many overfitted patches).

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

```
1| - if (nstrips > 1   // buggy
2| + if (nstrips > 2   // developer
3| + if (nstrips > 1 && 0  // overfitted
4|   && compression == COMPRESSION_NONE
5|   && stripbytecount[0] != stripbytecount[1]) {
6|     TIFFWarning("Wrong␣field,␣ignoring␣and"
7|       "calculating␣from␣imagelength");
8|   if (estimate(tif, ...) < 0)
9|     goto bad;
10| }
```

Figure 4: Patches for libtiff-d13be-ccadf (a loose condition bug).

Approach. We integrate *Opad* with SPR to see if *Opad* can guide SPR to generate correct patches for more bugs (see the integration in Figure 1). Particularly, whenever SPR generates one patch that makes existing test cases pass, that patch will be validated by *Opad* (by calculating O-measure based on new test cases and two oracles). If this patch is determined as overfitted by *Opad* (O-measure is non-zero), SPR will continue exploring the search space until it finds the next patch that can pass both the original validation (developer test cases) and *Opad*.

Results. Our approach guides SPR to generate a correct patch for one additional bug (SPR previously fixed 11 bugs). Table 2 shows the results of applying SPR+*Opad* on the 19 bugs (from the GenProg 2012 benchmark) for which there are correct patches in SPR's search space. We use '**Y**✓' to annotate the case that, *Opad* helps SPR generate the correct patch. For libtiff-d13be-ccadf (a loose condition bug), SPR cannot generate a correct patch without *Opad*: our approach prunes 13 overfitted patches that block the correct one.

Finding the correct patch. We describe how *Opad* exposes the flaws in overfitted patches for libtiff-d13be-ccadf (a loose condition bug), filters them out and finds the correct patch. The bug is in the image-reading routine (simplified code is presented in Figure 4). The function `estimate()` at line 8 should only be called when input images are ill-formed. However, since the condition at line 1 is incorrect, `estimate()` is called for some well-formed images as well. The correct patch fixes the condition so that `estimate()` is only called when it should be. However, an overfitted patch removes the entire branch (line 3) by adding `&&0` to the condition. Thus, `estimate()` is never called, even for ill-formed images. When some automatically-generated test cases exercise the overfitted patch with ill-formed images, one garbage item, which should otherwise be cleaned up in the `estimate()` routine, is used as an array index and this leads to a segmentation violation. All the overfitted patches that precede the correct one has the described flaw and, therefore, have a non-zero O-measure and are filtered by *Opad*. Thus, with the help of automatically-generated test cases, *Opad* successfully guides SPR to generate the correct patch.

RQ3: How many of *Opad*'s automatically-generated test cases may filter out more overfitted patches if empowered with better oracles?

Motivation. The usefulness of test cases in filtering out overfitted patches is limited by oracles, i.e., oracles might fail to precisely define correct behaviors. Generating effective oracles is an open challenge in software testing [4]. Even if an automatically-generated test case explores relevant program paths, which is required to identify an overfitted patch, the test case cannot filter out overfitted patches without sufficient oracles. This does not mean such a test case is useless in identifying overfitted patches. If with better oracles (e.g., manually-defined oracles, automatically-generated regression oracles, or `assert` statements to distinguish different program states), such test cases may filter out more overfitted patches. We call such test cases *weakly relevant* to the target bug that we want to repair.

Approach. We first define *weakly relevant* test cases, then we describe how we perform relevance analysis manually.

A test case is *weakly relevant* if it exposes the target bug *inexplicitly* by showing differences in program states. A weakly relevant test case usually does not expose the target bug explicitly due to the limitations of its oracle. Instead, running weakly relevant test cases shows the differences in the program states between the buggy version and the correctly-patched version. This means that, if empowered with better oracles, *weakly relevant* test cases can *explicitly* tell the differences between the buggy version and the correctly-patched version. For example, libtiff-08603-1ba75 (an arithmetic bug) is a bug in a check for an integer overflow; due to a developer's mistake, many benign inputs (that do not contain an integer overflow) are rejected. For this specific bug, we say that the test case (essentially, the input of the test case) is weakly relevant if it is rejected by the buggy version and accepted by the developer version. Another example is php-307562-307561: a bug is in the `saveHTML()` routine of the `DOMDocument` class. Invoking `saveHTML()` with an optional parameter generates empty results (this behavior is incorrect). We define a test case to be weakly relevant if it contains a call to the `saveHTML()` with a parameter.

For each bug, we manually investigate the root cause of the target bug and create methodologies that can identify weakly relevant from all the automatically-generated test cases. Then, we instrument the buggy version and the correctly-patched version at the points of interest, run each test case on both versions, and use the collected data to determine the relevance of the test case. For example, for the bug libtiff-08603-1ba75 (an arithmetic bug) described above, the root cause of the bug is that many benign inputs are erroneously rejected. To find weakly relevant test cases, we instrument the buggy version and the correctly-patched version using GCOV coverage instrumentation [2] and observe the number of times the input was rejected by each version. If the buggy version rejects the input from a test case M times, the correctly-patched version rejects the same input N times, and $M > N$, then we say that this particular test case is weakly relevant. The detailed methodology of weakly relevance of each bug is on our project website.

Results. Our relevance analysis shows that up to 40% (2,310/5,967) of the automatically-generated test cases are weakly relevant to the target bug; and they may filter out more overfitted patches if empowered with better oracles. Table 3 shows the results of our manual relevance analysis on the automatically-generated test cases. The *weakly relevant* test cases that are identified by relevance analysis could filter out more overfitted patches if empowered with better oracles. The column "Weakly Relevant" presents the number of *w*eakly relevant test cases from all the automatically-generated test cases in *Opad*. The column "Total" shows the total number of automatically-generated test case by fuzz testing for a particular bug. For example, in the case of the bug libtiff-5b021-3dfb3, we discovered 745 weakly relevant test cases out of 2,863. This shows that these 745 test cases can indeed identify the differences between the buggy version and the correctly-patched version from program states (i.e., behaviors); however, these test cases currently cannot do so due to limitations of the oracles.

Table 3: Weakly relevant test cases from *Opad* through manual analysis.

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

In summary, for 9/45 bugs, there exist automatically-generated test cases that can potentially identify more overfitted patches if with better oracles (for each bug, up to 40% of test cases have such a potential). These test cases can observe different behaviors between the buggy version and the correctly-patched version. Those differences in behaviors can be encoded in oracles (e.g., `assert`, or manually-defined expected output) to allow *weakly relevant* test cases to identify more overfitted patches.

5 THREATS TO VALIDITY

Non-determinism. Some studied programs show non-deterministic behaviors during the execution of the automatically-generated test cases. For example, a program may only crash one out of 10 times given the same input (e.g., due to address space layout randomization). To mitigate this issue, we execute each automatically-generated test case 10 times, which reduces the risk of getting spurious results and erroneously filtering out a patch.

Hidden Bugs. In some cases, a correct patch can have a non-zero O-measure because of hidden bugs. Such correct patches change the control flow of a program and reveal the bugs that were hidden in the buggy version; this leads to more crashes in the patched

version compared to the buggy version and results in a non-zero O-measure (as described in Section 4 for libtiff-5b021-3dfb3). Although the correct patch would not be accepted by *Opad* in this scenario, the test cases that we generated would help developers fix such hidden bug manually, which remains as future work.

Limitations of Fuzz Testing. Fuzz testing has a limitation of targeting *initial* levels of input-parsing in programs under test [22]. If a particular patch correctly fixes a bug in deeper levels of programs, but the program contains other bugs in initial levels, then the fuzz test cases would crash the program without even reaching the patched code. Our definition of O-measure eliminates this issue by a comparison with the buggy version: both versions will fail on such a test case and the correct patch will not be filtered out.

6 RELATED WORK

Automated Program Repair. Researchers have been working on various G&V techniques. GenProg [16] is the pioneer work in this area, followed by Par [14], RSRepair [28], Kali [29], SPR [18], relifix [34], and Nopol [38]. The above-mentioned techniques differ from GenProg in terms of either search space and/or search algorithm. Par [14] uses hard-coded patch templates to construct search space. RSRepair [28] employs a random search algorithm instead of genetic programming (which is used by GenProg). Kali [29] uses a restricted search space—emphasizing on deleting operations and an exhaustive search strategy. SPR [18], which outperforms the previous work, constructs search space based on predefined transformation schemas and leverages a targeted search algorithm. The constructed search space contains more useful patches and provides a larger set of fix templates than that of Par [14]. As an alternative to G&V techniques, semantic-based automatic repair tools are proposed (e.g., DirectFix [23], Angelix [24]). Semantic-based automatic repair uses symbolic execution and constraint solvers to synthesize a patch that by design passes all the developer test cases. Our current approach focuses solely on G&V systems.

Recently, innovative approaches are proposed on top of existing automated program repair (e.g., GenProg/AE and SPR). Fan et al. [20] apply probability model to improve the ranking of patches in search space so that the correct patches can be selected first. Le et al. [15] propose to prioritizefix candidates based on frequent fix patterns that are mined from software fix history. Tan et al. [31] use anti-patterns to identify overfitted patches, e.g., one anti-pattern is to eliminate patches that only contain deletions. Different from prior work, we focus on filtering out overfitted patches and generating correct ones from a new angle—improving test cases.

Using Testing to Improve G&V Techniques. Xin et al. [37] propose techniques to guide test generation techniques to cover patches by G&V techniques with an assumption that perfect oracles are already available. Our work shows that basic oracles can improve G&V techniques. Yu et al. [39] aim to leverage test generation to guide G&V techniques to generate patches that are less overfitted. Differently, we use automatic test generation to improve G&V techniques by filtering out overfitted patches, and then continuing G&V techniques to generate correct patches. Liu et al. [17] propose a novel technique which leverages the similarity of execution traces to heuristically determine the correctness of the generated patches by G&V techniques. Differently, we use new test inputs and oracles directly to detect overfitted patches.

Empirical Study on Program Repair. Barr et al. [5] study what percentage of fixes can be constructed from historical fixes. Martinez et al. [21] study the possibility of constructing fixes using the code from the same version (i.e., buggy version). These papers focus on the theoretical correctness of the fundamental assumption of search-based automated program repair—whether fixes can be constructed given a search space. Differently, our work empirically studies the impacts of the quality of test cases on G&V techniques on 45 bugs from 7 systems. Smith et al. [32] compare the quality of automatically-generated patches with developers' patches. In terms of quality of patches, they use the passing ratio of an independent set of test cases from white-box test generation. This work focuses on empirically studying whether patches of G&V techniques have lower quality or not. Differently, our work focuses on ways to distinguish overfitted patches from correct ones.

Automated Test Generation. There exist different types of automated test generation. Random test generation techniques [7, 27] and fuzz testing tools scale to large systems, but lack of direction. Dynamic symbolic execution [6] and concolic testing [30] tools aim to generate test cases which achieve high coverage. Search-based test generation techniques [9, 10] integrate search algorithms to guide unit test generation to achieve high coverage. All the above techniques use crashes as oracles. Alternatively, regression oracles are automatically generated [36] by recording variable values during running black-box test cases written by developers. In this work, we use crash, which is a widely-accepted oracle, and memory-safety oracles to improve validity checking of patches.

Empirical Study on Test Effectiveness. Previous work on test suite effectiveness shows that coverage [12] is not strongly correlated with test suite effectiveness, and mutation detection is strongly correlated with real fault detection [13]. Zhang et al. [40] show assertions are strongly correlated with test suite effectiveness. In this work, we study the effectiveness of test cases in the context of automated program repair.

7 CONCLUSIONS

In conclusion, we experiment with ways to improve existing test cases in order to improve G&V techniques. We propose an approach to filter out incorrect patches by augmenting existing test cases. *Opad* improves existing test cases from two angles—better validity checking by employing memory-safety oracles and new test cases from fuzz testing. We propose O-measure, to filter out overfitted patches based on the new test cases and oracles. Our evaluation on 45 bugs from 7 systems shows that *Opad* filters out 75.2% of the overfitted patches. More importantly, *Opad* helps SPR generate the correct patch for one additional bug (original SPR can only generate correct patches for 11 bugs). In addition, we identify how many of the automatically-generated test cases can filter out more overfitted patches if used with better test oracles. Our findings highlight promising research directions on improving G&V techniques.

ACKNOWLEDGEMENTS

We thank the reviewers for their invaluable comments. We thank Fan Long for clarifying SPR and the GenProg 2012 benchmark used in the SPR paper. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 2016. American Fuzzy Lop. (2016). http://lcamtuf.coredump.cx/afl/.

[2] 2016. gcov—a Test Coverage Program. (2016). https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[3] 2016. Valgrind. (2016). http://valgrind.org/.

[4] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, New York, NY, USA.

[5] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 306–317.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*. USENIX Association, Berkeley, CA, USA, 209–224.

[7] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34, 11 (Sept. 2004), 1025–1050.

[8] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (FSE 2011)*. ACM, 416–419.

[9] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)*. ACM, New York, NY, USA, 416–419.

[10] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2014. Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution (Tool paper). In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, ACM, New York, NY, USA, 421–424.

[11] S Hocevar. 2011. zzuf—multi-purpose fuzzer. (2011).

[12] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 435–445.

[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665.

[14] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013)*. IEEE Press, Piscataway, NJ, USA, 802–811.

[15] X. B. D. Le, D. Lo, and C. L. Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, Vol. 1. 213–224.

[16] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 3–13.

[17] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. 2017. Identifying Patch Correctness in Test-Based Automatic Program Repair. *CoRR* abs/1706.09120 (2017).

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

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

[20] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 298–312.

[21] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 492–495.

[22] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. 2012. *Fuzzing: the state of the art.* Technical Report. DTIC Document.

[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*, Vol. 1. IEEE, 448–458.

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

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

[26] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.

[27] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, Washington, DC, USA, 75–84.

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

[29] Zichao Qi, Fan Long, Sara Anchor, 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 (ISSTA 2015)*. 24–36.

[30] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 263–272.

[31] Tan Shin Weiand, H Yoshida, Prasad M, and A. Roychoudhury. Anti-patterns in Search-based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 727–738.

[32] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. 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)*. ACM, New York, NY, USA, 532–543.

[33] Michael Sutton. 2012. FileFuzz. (2012).

[34] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *Proceedings of the 2015 Internaltional Conference on Software Engineering (ICSE 2015)*. 471–482.

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

[36] Tao Xie. 2006. Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*. Springer-Verlag, Berlin, Heidelberg, 380–403.

[37] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2017)*.

[38] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* PP, 99 (2016), 1–1.

[39] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. 2017. Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. *ArXiv e-prints* (March 2017). arXiv:cs.SE/1703.00198

[40] Yucheng Zhang and Ali Mesbah. 2015. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 214–224.