An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard
MIT EECS & CSAIL
{zichaoqi, fanl, sarachour, rinard}@csail.mit.edu

ABSTRACT

We analyze reported patches for three existing generate-and-validate patch generation systems (GenProg, RSRepair, and AE). The basic principle behind generate-and-validate systems is to accept only *plausible patches* that produce correct outputs for all inputs in the test suite used to validate the patches.

Because of errors in the patch evaluation infrastructure, the majority of the reported patches are not plausible — they do not produce correct outputs even for the inputs in the validation test suite. The overwhelming majority of the reported patches are not correct and are equivalent to a single modification that simply deletes functionality. Observed negative effects include the introduction of security vulnerabilities and the elimination of desirable standard functionality.

We also present Kali, a generate-and-validate patch generation system that only deletes functionality. Working with a simpler and more effectively focused search space, Kali generates at least as many correct patches as prior GenProg, RSRepair, and AE systems. Kali also generates at least as many patches that produce correct outputs for the inputs in the validation test suite as the three prior systems.

We also discuss the patches produced by ClearView, a generate-and-validate binary hot patching system that leverages learned invariants to produce patches that enable systems to survive otherwise fatal defects and security attacks. Our analysis indicates that ClearView successfully patches 9 of the 10 security vulnerabilities used to evaluate the system. At least 4 of these patches are correct.

1. INTRODUCTION

Automatic patch generation holds out the promise of correcting defects in production software systems without the time and expense required for human developers to understand, triage, and correct these defects. The prominent *generate-and-validate* approach starts with a test suite of inputs, at least one of which exposes a defect in the software. The patch generation system applies program modifications to generate a space of candidate patches, then searches the generated patch space to find *plausible patches* — i.e., patches that produce correct outputs for all inputs in the test suite. In this paper we start by considering the GenProg [26], RSRepair [48], and AE [59] systems.

The reported results for these systems are impressive: GenProg is reported to fix 55 of 105 considered bugs [26], RSRepair is reported to fix all 24 of 24 considered bugs (these bugs are a subset of the 55 bugs that GenProg is reported to fix) [48], and AE is reported to fix 54 of the same 105 considered bugs [59].[1] If these results are accurate, these systems represent a significant step forward in our ability to automatically eliminate defects in large software systems.

1.1 Plausibility Analysis and Weak Proxies

Motivated to better understand the capabilities and potential of these systems, we performed an analysis of the patches that these systems produce. Enabled by the availability of the generated patches and the relevant patch generation and validation infrastructure [7, 5, 3, 8, 1], our analysis was driven by the following research questions:

RQ1: Do the reported GenProg, RSRepair, and AE patches produce correct outputs for the inputs in the test suite used to validate the patch?

The basic principle behind generate-and-validate systems is to only accept *plausible* patches that produce correct outputs for all inputs in the test suite used to validate the patches. Despite this principle, our analysis shows that because of errors in the patch validation infrastructure, many of the reported patches are, in fact, not plausible – they do not produce correct outputs even for the inputs in the test suite used to validate the patch.

For 37 of the 55 defects that GenProg is reported to repair [26], none of the reported patches produce correct outputs for the inputs in the test suite. For 14 of the 24 defects that RSRepair is reported to repair [48], none of the reported patches that we analyze produce correct outputs for the inputs in the test suite. And for 27 of the 54 defects reported in the AE result tar file [1], none of the reported patches produces correct outputs.

Further investigation indicates that *weak proxies* are the source of the error. A weak proxy is an acceptance test that does not check that the patched program produces correct output. It instead checks for a weaker property that may (or may not) indicate correct execution. Specifically, some of the acceptance tests check only if the patched program produces an exit code of 0. If so, they accept the patch (whether the output is correct or not).

Because of weak proxies, all of these systems violate the underlying basic principle of generate-and-validate patch generation. The result is that the majority of the patches ac-

[1]Our analysis of the commit logs and applications indicates that 36 of these bugs correspond to deliberate functionality changes, not actual bugs. That is, for 36 of these bugs, there is no actual bug to fix. To simplify the presentation, however, we refer to all of the 105 bugs as defects.

cepted by these systems do not generate correct outputs even for the inputs in the validation test suite. See Section 3.

1.2 Correctness Analysis

Despite multiple publications that analyze the reported patches and the methods used to generate them [26, 48, 59, 41, 23, 32, 49, 21], we were able to find no systematic patch correctness analysis. We therefore analyzed the remaining plausible patches to determine if they eliminated the defect or not.

RQ2: Are any of the reported GenProg, RSRepair, and AE patches correct?

The overwhelming majority of the patches are not correct. Specifically, GenProg produced a correct patch for only 2 of the 105 considered defects.[2] Similarly, RSRepair produced a correct patch for only 2 of the 24 considered defects. AE produced a correct patch for only 3 of the 105 considered defects. For each of the incorrect patches, we have a test case that exposes the defect in the patch [50]. Because of weak proxies, many of these test cases were already present in the existing test suites. For the remaining plausible but incorrect patches, we developed new test cases that exposed the defects. See Section 4.

1.3 Stronger Test Suites

One hypothesis is that stronger test suites with additional inputs that provide better coverage would enable these systems to generate more correct patches:

RQ3: Do stronger test suites enable GenProg to produce more correct patches?

To investigate this question, we reran all of the GenProg runs that generated incorrect patches. We used corrected test scripts and enhanced test suites that contained defect-exposing test cases for all of these patches. These reexecutions produced no patches at all. We next discuss two potential explanations for this result.

Search Space: A necessary prerequisite for the success of any search-based patch generation algorithm is a search space that contains successful patches. One potential explanation is that the GenProg, RSRepair, and AE search spaces do not contain correct patches for these defects. This explanation is consistent with recent results from the staged program repair (SPR) project, whose search space contains correct patches for 20 of the 105 defects in the GenProg benchmark set [34, 35]. Only 3 of these correct patches are within the GenProg search space.

Random Genetic Search: Another potential explanation is that these systems do not use a search algorithm that can explore the search space efficiently enough. GenProg's genetic search algorithm uses the number of passed test cases as the fitness function. For most of the defects in the benchmark set, there is only one negative test case (so even the unpatched program passes all but one of the test cases). With this fitness function, the difference between the fitness of the unpatched code and the fitness of a plausible patch that passes all test cases is only one. There is therefore no smooth gradient for the genetic search to traverse to find a solution. In this situation, genetic search can easily devolve into random search. Indeed, RSRepair (which uses random search) is reported to find patches more quickly and with less trials than GenProg [48]. See Section 5.

[2]We note that the paper discusses only two patches: one of the correct patches for one of these two defects and a patch that is obtained with the aid of user annotations [26].

1.4 Functionality Deletion

As we analyzed patch correctness, it became clear that (despite some surface syntactic complexity), the overwhelming majority of the plausible patches were semantically quite simple. Specifically, they were equivalent to a single functionality deletion modification, either the deletion of a single line or block of code or the insertion of a single return or exit statement.

RQ4: How many of the plausible reported GenProg, RSRepair, and AE patches are equivalent to a single functionality deletion modification?

Our analysis indicates that 104 of the 110 plausible GenProg patches, 37 of the 44 plausible RSRepair patches, and 22 of the 27 plausible AE patches are equivalent to a single modification that deletes functionality.

Our analysis also indicates that (in contrast to previously reported results [31]) the plausible patches had significant negative effects, including the introduction of new integer and buffer overflow security vulnerabilities and the elimination of standard desriable functionality. These negative effects highlight some of the risks associated with the combination of functionality deletion and generate-and-validate patch generation. See Section 6.

Despite their potential for negative effects, functionality deletion patches can be useful in helping developers locate and better understand defects. For obtaining functionality deletion patches for this purpose, we advocate using a system that focuses solely on functionality deletion (as opposed to a system that aspires to create correct patches). Such an approach has at least two advantages. First, it is substantially simpler than approaches that attempt to generate more complex repairs. The search space can therefore be smaller, simpler, and searched more efficiently. Second, focusing solely on functionality deletion can produce simpler, more transparent patches for developers who want to use the patches to help them locate and better understand defects.

1.5 Original GenProg

Our analysis of the original GenProg system [60, 22] yields similar results. Out of 11 defects evaluated in the two papers, the generated patches for 9 defects are incorrect (in some cases because of the use of weak proxies, in other cases because of weak test suites). The patches for 9 of the 11 defects simply delete functionality (removing statements, adding return statements, or adding exit statements). The only two defects for which the original GenProg system generates correct patches are small motivating examples (less than 30 lines of code). See Section 7.

1.6 Revisiting Hypotheses and Explanations

At this point there is a substantial number of papers that present hypotheses and explanations related to phenomena associated with the GenProg, RSRepair, and AE automated patch generation systems. We next revisit some of these hypotheses and explanations in light of the results presented in this paper.

Simple vs. Complex Patches: Previous papers have considered (but not satisfactorily answered) the following question: "why is GenProg typically able to produce simple patches for bugs when humans used complex patches?" [59, 26]. The results in this paper provide additional insight into this question: the simple GenProg patches are not correct — they either fail to produce correct outputs even for the inputs in the validation test suite, or they simply remove lightly tested functionality. Humans used complex patches because complex patches are required to eliminate the defect. This fact highlights how test suites that may be suitable for human developers may not be suitable for automated patch generation systems that operate with less context.

Targeted Defect Classes: In an essay in ICSE 2014, Monperrus focuses on the importance of "target defect classes," i.e., the set of defects for which the technique is designed to work [41]. He notes that the GenProg research does not explicitly address the question, but observes "hints that GenProg works best for manipulating defensive code against memory errors (in particular segmentation faults and buffer overruns)" [41]. Our results indicate that the defect class for which GenProg works best is defects that can be repaired with a single modification that deletes functionality. And while some of the patches do manipulate defensive code, observed effects include the deletion of critical checks and the introduction of new segmentation faults and buffer overruns.

Human Patch Acceptability: A paper investigating the developer maintainability of a subset of the GenProg patches found "statistically significant evidence" that the GenProg patches "can be maintained with equal accuracy and less effort than the code produced by human-written patches" [23]. In retrospect, this potentially surprising result may become more plausible when one considers that the majority of the GenProg patches are equivalent to a single functionality deletion modification.

The evaluation focused on asking human developers a variety of questions designed to be relevant to maintainability tasks [23]. There is no indication if any of the developers thought that the patches were incorrect. The motivation for the paper, referencing GenProg patches (among others), states "while these patches may be functionally correct, little effort has been taken to date to evaluate the understandability of the resulting code". We note that the referenced patches are not functionally correct, and question the relevance of a human evaluation of patch understandability that does not expose the obvious incorrectness of the patches.

Time and Effort: The GenProg, RSRepair, and AE research projects devoted significant time and effort to evaluating variants of the basic GenProg patch generation mechanisms [26, 21, 32, 48, 59]. The results presented in this paper show that (at least for the considered benchmark defects) all of the time and effort invested in developing, evaluating, analyzing, and evolving these mechanisms only produced complex systems whose patches are no better than those generated by the much simpler Kali patch generation mechanism (which simply deletes functionality).

Community Perception: It is our understanding that the broader software engineering community may understand (incorrectly) that the GenProg patches actually fix the defects. Example quotes that reflect this understanding include "We selected GenProg for the reason that it is almost the only state-of-the-art automated repair tool having the ability of fixing real-world, large-scale C faulty programs" [49] and "in an evaluation of GenProg, a state-of-the-art repair approach guided by genetic programming, this approach repaired 55 out of 105 defects" [38]. We believe that this understanding should be revisited in light of the results presented in this paper.

1.7 Realistic Patch Generation Expectations

Given this backdrop, what can one realistically expect from automatic patch generation systems moving forward? Currently available evidence indicates that improvements will require both 1) the use of richer search spaces with more correct patches and 2) the use of more effective search algorithms that can search the space more efficiently.

Perhaps most importantly, our results highlight important differences between machine-generated and human-generated patches. Even the plausible GenProg, AE, and RSRepair patches are overwhelming incorrect and simply remove functionality. The human-generated patches for the same defects, in contrast, are typically correct and usually modify or introduce new program logic. This result indicates that information other than simply passing a validation test suite is (at least with current test suites) important for producing correct patches.

Automatic Code Transfer: One way to obtain correct code is to obtain it from another application. Working with an input that exposes a potential security vulnerability, CodePhage searches an application database to automatically locate and transfer code that eliminates the vulnerability [56, 55]. CodePhage successfully repaired 10 defects in 7 recipient applications via code transfer from 5 donor applications.

Learning From Successful Patches: Another way to obtain additional information is to learn from successful human patches. Prophet [33] analyzes a large database of revision changes extracted from open source project repositories to automatically learn features of successful patches. It then uses these features to recognize and prioritize correct patches within a larger space of candidate patches. On the GenProg benchmark set, Prophet generates correct patches for 14 defects (12 more than GenProg and 11 more than AE).

Learned Invariants: Successful executions are yet another source of useful information. Learning data structure consistency specifications from successful executions can enable successful data structure repair [14]. ClearView [47] observes successful executions to dynamically learn and enforce invariants that characterize successful executions. ClearView automatically generates successful patches that eliminate security vulnerabilities in 9 of 10 evaluated defects [47].

Targeted Patch Generation: Another source of information is to identify a specific set of defects and apply techniques that target that set of defects. Researchers have successfully targeted out of bounds accesses [53, 10, 42], null pointer dereferences [36, 18], divide by zero errors [36], memory leaks [44, 25], infinite loops [11, 29, 37], and integer and buffer overflows [57]. For the defects in scope, a targeted technique tends to generate patches with better quality than a search-based technique. For example, RCV [36], a recovery tool for divide-by-zero and null-dereference defects, successfully enables applications to recover from the majority of the systematically collected 18 CVE defects so that they exhibit identical behavior as the developer-patched application.

Specifications: Specifications, when available, can enable patch generation systems to produce patches that are guaranteed to be correct. AutoFix-E produces semantically sound candidate bug patches with the aid of a design-by-contract programming language (Eiffel) [58]. CodeHint uses partial specifications to automatically synthesize code fragments with the specified behavior [24]. Data structure repair techniques detect the inconsistency between a data structure state and a set of specified model constraints and enforce the violated constraints to repair the data structure state [15].
Realistic Expectations: By combining more productive search spaces and search algorithms with the exploitation of additional information other than generating correct outputs on a validation test suite, we expect future systems to be able to generate successful patches for defects that can be fixed with small changes (via search-based techniques) and defects that follow specific patterns (via targeted techniques).

1.8 Kali

Inspired by the observation that the patches for the vast majority of the defects that GenProg, RSRepair, and AE were able to address consisted (semantically) of a single functionality deletion modification, we implemented a new system, the *Kali* automatic patch generation system, that focuses only on removing functionality. Kali generates patches that either 1) delete a single line or block of code, 2) replace an if condition with true or false (forcing the then or else branch to always execute), or 3) insert a single return statement into a function body. Kali accepts a patch if it generates correct outputs on all inputs in the validation test suite. Our hypothesis was that by focusing directly on functionality removal, we would be able to obtain a simpler system that was at least as effective in practice.

RQ5: How effective is Kali in comparison with existing generate-and-validate patch generation systems?

Our results show that Kali is more effective than GenProg, RSRepair, and AE. Specifically, Kali finds correct patches for at least as many defects (3 for Kali vs. 3 for AE and 2 for GenProg and RSRepair) and plausible patches for at least as many defects (27 for Kali vs. 18 for GenProg, 10 for RSRepair, and 27 for AE). And Kali works with a simpler and more focused search space.
Efficiency and Automatic Operation: An efficient search space is important for automatic operation. An examination of the GenProg patch generation infrastructure indicates that GenProg (and presumably RSRepair and AE) require the developer to specify the source code file to attempt to patch [6]. This requirement significantly reduces the size of the patch search space, but also prevents these prior systems from operating automatically without developer involvement. Kali, in contrast, is efficient enough to operate fully automatically without requiring the developer to specify a target source code file to attempt to patch.

RQ6: Can Kali provide useful information about software defects?

Although Kali is more effective than GenProg, RSRepair, and AE at finding correct patches, it is not so much more effective that we would advocate using it for this purpose (any more than we would advocate using any of these previous systems to find correct patches). But Kali's plausible patches can still be useful. The Kali patches often precisely pinpoint the exact line or lines of code to change. And they almost always provide insight into the defective functionality, the cause of the defect, and how to correct the defect.

1.9 Research Result Availability

We were able to perform the research in this paper because the reported GenProg, RSRepair, and AE patches (along with the GenProg patch generation and evaluation infrastructure) were available online [7, 3, 5, 1, 8]. This paper therefore supports a move to greater transparency and availability of reported experimental results and systems. The Kali and ClearView patches, new test cases, and the plausibility, correctness, and functionality deletion analyses are all available [50]. Kali is available on request for the purpose of checking the results presented in this paper.

The PAR system is another prominent generate-and-validate automatic patch generation system [28]. We do not include PAR in this study because (despite repeated requests to the authors of the PAR paper), we were unable to obtain the PAR patches.

1.10 Contributions

This paper makes the following contributions:

- **Plausibility Analysis:** It shows that the majority of the reported GenProg, RSRepair, and AE patches, contrary to the basic principle of generate-and-validate patch generation, do not produce correct outputs even for the inputs in the test suite used to validate the patches.

- **Weak Proxies:** It identifies *weak proxies*, acceptance tests that do not check that the patched application produces the correct output, as the cause of the reported implausible patches.

- **Correctness Analysis:** It shows that the overwhelming majority of the reported patches are not correct and that these incorrect patches can have significant negative effects including the introduction of new security vulnerabilities and the elimination of desirable standard functionality.

- **Stronger Test Suites Don't Help:** It shows that stronger test suites do not help GenProg produce correct patches — they simply eliminate the ability of GenProg to produce any patches at all.

- **Semantic Patch Analysis:** It reports, for the first time, that the overwhelming majority of the plausible GenProg, RSRepair, and AE patches are semantically equivalent to a single modification that simply deletes functionality from the application.

- **Kali:** It presents a novel automatic patch generation system, Kali, that works only with simple patches that delete functionality.

- **Kali Results:** It presents results that show that Kali outperforms GenProg, RSRepair, and AE. With a simpler search space and no identification of a target source code file to patch, Kali generates at least as many correct patches and at least as many plausible patches as these prior systems. Even though the majority of these patches are not correct, they successfully target the defective functionality, can help pinpoint the defective code, and often provide insight into important defect characteristics.

2. OVERVIEW OF ANALYZED SYSTEMS

GenProg: GenProg combines three basic modifications, specifically delete, insert, and replace, into larger patches, then uses genetic programming to search the resulting patch space. We work with the GenProg system used to perform a "systematic study of automated program repair" that "includes two orders of magnitude more" source code, test cases, and defects than previous studies [26]. As of the submission date of this paper, the relevant GenProg paper is referenced on the GenProg web site as the recommended starting point for researchers interested in learning more about GenProg [4]. The GenProg patch evaluation infrastructure works with the following kinds of components [7, 5, 3]:

- **Test Cases:** Individual tests that exercise functionality in the patched application. Examples include php scripts (which are then evaluated by a patched version of php), bash scripts that invoke patched versions of the libtiff tools on specific images, and perl scripts that generate HTTP requests (which are then processed by a patched version of lighttpd).
- **Test Scripts:** Scripts that run the application on a set of test cases and report either success (if the application passes all of the test cases) or failure (if the application does not pass at least one test case).
- **Test Harnesses:** Scripts or programs that evaluate candidate patches by running the relevant test script or scripts on the patched application, then reporting the results (success or failure) back to GenProg.

It is our understanding that the test cases and test scripts were adopted from the existing software development efforts for each of the benchmark GenProg applications and implemented by the developers of these projects for the purpose of testing code written by human developers working on these applications. The test harnesses were implemented by the GenProg developers as part of the GenProg project.

A downloadable virtual machine [7], all of the patches reported in the relevant GenProg paper (these include patches from 10 GenProg executions for each defect) [5], source code for each application, test cases, test scripts, and the GenProg test harness for each application [3] are all publicly available. Together, these components make it possible to apply each patch and run the test scripts or even the patched version of each application on the provided test cases. It is also possible to run GenProg itself.

RSRepair: The goal of the RSRepair project was to compare the effectiveness of genetic programming with random search [48]. To this end, the RSRepair system built on the GenProg system, using the same testing and patch evaluation infrastructure but changing the search algorithm from genetic search to random search. RSRepair was evaluated on 24 of the 55 defects that GenProg was reported to repair [48, 26]. The reported patches are publicly available [8]. For each defect, the RSRepair paper reports patches from 100 runs. We analyze the first 5 patches from these 100 runs.

AE: AE is an extension to GenProg that uses a patch equivalence analysis to avoid repeated testing of patches that are syntactically different but equivalent (according to an approximate patch equivalence test) [59]. AE focuses on patches that only perform one edit and exhaustively enumerates all such patches. The AE experiments were "designed for direct comparison with previous GenProg results" [59, 26] and evaluate AE on the same set of 105 defects. The paper reports one patch per repaired defect, with the patches publicly available [1]. AE is based on GenProg and we were able to leverage the developer test scripts available in the GenProg distribution to compile and execute the reported AE patches.

3. PLAUSIBILITY ANALYSIS

The basic principle behind the GenProg, RSRepair, and AE systems is to generate patches that produce correct results for all of the inputs in the test suite used to validate the patches. We investigate the following research question:

RQ1: Do the reported GenProg, RSRepair, and AE patches produce correct results for the inputs in the test suite used to validate the patch?

To investigate this question, we downloaded the reported patches and validation test suites [7, 5, 3, 8, 1]. We then applied the patches, recompiled the patched applications, ran the patched applications on the inputs in the validation test suites, and compared the outputs with the correct outputs. Our results show that the answer to RQ1 is that the majority of the reported GenProg, RSRepair, and AE patches do not produce correct outputs for the inputs in the validation test suite:

- **GenProg:** Of the reported 414 GenProg patches, only 110 are plausible — the remaining 304 generate incorrect results for at least one input in the test suite used to validate the patch. This leaves 18 defects with at least one plausible patch.
- **RSRepair:** Of the analyzed 120 AE patches, only 44 are plausible — the remaining 76 generate incorrect results for at least one input in the test suite used to validate the patch. This leaves 10 defects with at least one plausible patch.
- **AE:** Of the reported 54 AE patches, only 27 are plausible — the remaining 27 generate incorrect results for at least one input in the test suite. This leaves 27 defects with at least one plausible patch.

Test Harness Issues: The GenProg 2012 paper reports that GenProg found successful patches for 28 of 44 defects in php [26]. The results tarball contains a total of 196 patches for these 28 defects. Only 29 of these patches (for 5 of the 44 defects, specifically defects php-bug-307931-307934, php-bug-309892-309910, php-bug-309986-310009, php-bug-310011-310050, and php-bug-310673-310681) are plausible. GenProg accepts the remaining 167 patches because of integration issues between the GenProg test harness and the developer test script.

For php, the developer test script is also written in php. The GenProg test harness executes this developer test script using the version of php with the current GenProg patch under evaluation applied, not the standard version of php. The current patch under evaluation can therefore influence the behavior of the developer test script (and not just the behavior of the test cases).

The GenProg test harness does not check that the php patches cause the developer test scripts to produce the correct result. It instead checks only that the higher order 8 bits of the exit code from the developer test script are 0. This can happen if 1) the test script itself crashes with a segmentation fault (because of an error in the patched version of php that the test case exercises), 2) the current patch under evaluation causes the test script (which is written in php) to exit with exit code 0 even though one of the test cases fails, or 3) all of the test cases pass. Of the 167 accepted patches, 138 are implausible — only 29 pass all of the test cases.

We next present relevant test infrastructure code. The GenProg test harness is written in C. The following lines determine if the test harness accepts a patch. Line 8564 runs the test case and shifts off the lower 8 bits of the exit code. Line 8566 accepts the patch if the remaining upper 8 bits of the exit code are zero.

```
php-run-test.c:8564 int res = system(buffer) >> 8
php-run-test.c:8565
php-run-test.c:8566 if (res == 0) { /* accept patch */
```

Here `buffer` contains the following shell command:

```
./sapi/cli/php ../php-helper.php -p
  ./sapi/cli/php -q <php test file>
```

where `./sapi/cli/php` is the patched version of the php interpreter. This patched version is used both to run the php test file for the test case and the `php-helper.php` script that runs the test case.

Test Script Issues: The GenProg libtiff test scripts do not check that the test cases produce the correct output. They instead use a *weak proxy* that checks only that the exercised libtiff tools return exit code 0 (it is our understanding that the libtiff developers, not the GenProg developers, developed these test scripts [9]). The test scripts may therefore accept patches that do not produce the correct output. There is a libtiff test script for each test case; 73 of the 78 libtiff test scripts check only the exit code. This issue causes GenProg to accept 137 implausible libtiff patches (out of a total of 155 libtiff patches). libtiff and php together account for 322 of the total 414 patches that the GenProg paper reports [26].

One of the gmp test scripts does not check that all of the output components are correct (despite this issue, both gmp patches are plausible).

AE: The reported AE patches exhibit plausibility problems that are consistent with the use of weak proxies in the GenProg testing infrastructure. Specifically, only 5 of the 17 reported libtiff patches and 7 of the reported 22 php patches are plausible.

RSRepair: RSRepair uses the same testing infrastructure as GenProg [48]. Presumably because of weak proxy problems inherited from the GenProg testing infrastructure, the reported RSRepair patches exhibit similar plausibility problems. Only 5 of the 75 RSRepair libtiff patches are plausible. All of these 5 patches repair the same defect, specifically libtiff-bug-d13be72c-ccadf48a. The RSRepair paper reports patches for only 1 php defect, specifically php-bug-309892-309910, the 1 php defect for which all three systems are able to generate a correct patch.

ManyBugs: It is our understanding that the developers of the GenProg benchmark suite are aware of the test infrastructure errors and are working to correct them. As of the submission date of this paper, the php test harness uses the patched version of php to run the test cases (the segmentation fault error described above has been corrected) and the libtiff test scripts check only for exit code 0, not for correct output [30]. The result is accepted patches that produce incorrect outputs for inputs in the validation test suite.

4. CORRECTNESS ANALYSIS

We analyze each plausible patch in context to determine if it correctly repairs the defect.

RQ2: Are any of the reported GenProg, RSRepair, and AE patches correct?

Patch Correctness Results: Our analysis indicates that only 5 of the 414 GenProg patches (3 for python-bug-69783-69784 and 2 for php-bug-309892-309910) are correct. This leaves GenProg with correct patches for 2 out of 105 defects. Only 4 of the 120 RSRepair patches (2 for python-bug-69783-69784 and 2 for php-bug-309892-309910) are correct. This leaves RSRepair with correct patches for 2 out of 24 defects. Only 3 of the 54 AE patches (1 for php-bug-309111-309159, 1 for php-bug-309892-309910, and 1 for python-bug-69783-69784) are correct. This leaves AE with correct patches for 3 out of 54 defects.

For each plausible but incorrect patch that GenProg or AE generate, and each plausible but incorrect RSRepair patch that we analyze, we developed a new test case that exposes the defect in the incorrect patch [50].

Patch Correctness Clarity: We acknowledge that, in general, determining whether a specific patch corrects a specific defect can be difficult (or in some cases not even well defined). We emphasize that this is not the case for the patches and defects that we consider here. The correct behavior for all of the defects is clear, as is patch correctness and incorrectness.

Developer Patch Comparison: For each defect, the GenProg benchmark suite identifies a corrected version of the application that does not have the defect. In most cases the corrected version is a later version produced by a developer writing an explicit developer patch to repair the error. In other cases the corrected version simply applies a deliberate functionality change — there was no defect in the original version of the application. In yet other cases the identified correct version is an earlier version of the application. In these cases, it is possible to derive an implicit developer patch that reverts the application back to the earlier version.

Our analysis indicates that the developer patches are, in general, consistent with our correctness analysis. Specifically, 1) if our analysis indicates that the reported GenProg, RSRepair, or AE patch is correct, then the patch has the same semantics as the developer patch, 2) if our analysis indicates that the reported GenProg, RSRepair, or AE patch is not correct, then the patch has different semantics than the developer patch, and 3) if we developed a new test case to invalidate generated plausible but incorrect patches for a defect, the corresponding developer patched version of the application produces correct output for the new input.

python-bug-69783-69784: Figure 2 (see Appendix A) presents the GenProg patch for python-bug-69783-69784. Figure 3 presents the developer patch. Both of the patches remove an if statement (lines 1-25 in Figure 3, lines 1-52 in Figure 2). Because GenProg generates preprocessed code, the GenProg patch is larger than but semantically equivalent to the developer patch. AE and RSRepair also generate correct patches that are semantically equivalent to this GenProg patch. Note that python-bug-69783-69784 is in fact not a bug. It instead corresponds to a deliberate functionality change. The relevant code (correctly) implemented python support for two-year dates. This functionality was deliberately removed by the developer in revision 69784.

php-bug-309892-309910: Figure 4 (see Appendix A) presents the GenProg patch for php-bug-309892-309910. Figure 5 presents the developer patch. Both of the patches remove an obsolete check implemented by the deleted if statement (lines 14-18 in Figure 4 and lines 7-9 in Figure 5). AE and RSRepair generate semantically equivalent patches.

php-bug-309111-309159: Figure 6 (see Appendix A) presents the AE patch for php-bug-309111-309159. Figure 7 presents the developer patch. php-309111-309159 is an url parsing defect — the PHP function parse_url() may incorrectly parse urls that contain question marks. The AE patch (with __genprog_mutant equal to 25) copies the if statement (lines 23-29 of Figure 6) to the location after the assignment p = pp. Therefore p is equal to pp when the copied block executes. In this context, the copied block is semantically equivalent to the block that the developer patch adds before the assignment statement. In the AE patch, the code involving __genprog_mutant works with the AE test infrastructure to compile multiple generated patches into the same file for later dynamic selection by the AE test infrastructure.

5. GENPROG REEXECUTIONS

We next consider the following research question:

RQ3: Do stronger test suites enable GenProg to produce more correct patches?

To determine whether GenProg [26] is able to generate correct patches if we correct the issues in the patch evaluation infrastructure and provide GenProg with stronger test suites, we perform the following GenProg reexecutions:

Corrected Patch Evaluation Infrastructure: We first corrected the GenProg patch evaluation infrastructure issues (see Section 3). Specifically, we modified the php test harness to ensure that the harness correctly runs the test script and correctly reports the results back to GenProg. We strengthened the 73 libtiff test scripts to, as appropriate, compare various metadata components and/or the generated image output with the correct output. We modified the gmp test scripts to check all output components.

Augmented Test Suites: We augmented the GenProg test suites to include the new test cases (see Section 4) that expose the defects in the plausible but incorrect GenProg patches.

GenProg Reexecution: For each combination of defect and random seed for which GenProg generated an incorrect patch, we reexecuted GenProg with that same combination. These reexecutions used the corrected patch evaluation infrastructure and the augmented test suites.

Results: These reexecutions produced 13 new patches (for defects libtiff-bug-5b02179-3dfb33b and lighttpd-bug-2661-2662). Our analysis indicates that the new patches that GenProg generated for these two defects are plausible but incorrect. We therefore developed two new test cases that exposed the defects in these new incorrect patches. We included these new test cases in the test suites and reexecuted GenProg again. With these test suites, the GenProg reexecutions produced no patches at all. The new test cases are available [50].

6. SEMANTIC PATCH ANALYSIS

For each plausible patch, we manually analyzed the patch in context to determine if it is semantically equivalent to either 1) the deletion of a single statement or block of code, or 2) the insertion of a single return or exit statement. This analysis enables us to answer the following research question:

RQ4: Are the reported GenProg, RSRepair, and AE patches equivalent to a single modification that simply deletes functionality?

Our analysis indicates that the overwhelming majority of the reported plausible patches are equivalent to a single functionality deletion modification. Specifically, 104 of the 110 plausible GenProg patches, 37 of the plausible 44 RSRepair patches, and 22 of the plausible 27 AE patches are equivalent to a single deletion or return insertion modification. Note that even though AE contains analyses that attempt to determine if two patches are equivalent, the analyses are based on relatively shallow criteria (syntactic equality, dead code elimination, and equivalent sequences of independent instructions) that do not necessarily recognize the functionality deletion equivalence of syntactically complex sequences of instructions. Indeed, the AE paper, despite its focus on semantic patch equivalence, provides no indication that the overwhelming majority of the reported patches are semantically equivalent to a single functionality deletion modification [59].

6.1 Weak Test Suites

During our analysis, we obtained a deeper understanding of why so many plausible patches simply delete functionality. A common scenario is that one of the test cases exercises a defect in functionality that is otherwise unexercised. The patch simply deletes functionality that the test case exercises. This deletion then impairs or even completely removes the functionality.

These results highlight the fact that *weak test suites* — i.e., test suites that provide relatively limited coverage — may be appropriate for human developers (who operate with a broader understanding of the application and are motivated to produce correct patches) but (in the absence of additional techniques designed to enhance their ability to produce correct patches) not for automatic patch generation systems that aspire to produce correct patches.

6.2 Impact of Functionality Deletion Patches

Our analysis of the patches also indicated that (in contrast to previously reported results [31]) the combination of test suites with limited coverage and support for functionality deletion can promote the generation of patches with negative effects such as the introduction of security vulnerabilities and the elimination of standard functionality.

Check Elimination: Several defects are caused by incorrectly coded checks. The test suite contains a test case that causes the check to fire incorrectly, but there is no test case that relies on the check to fire correctly. The generated patches simply remove the check. The consequences vary depending on the nature of the check. For example:

- **Integer Overflow:** libtiff-bug-0860361d-1ba75257 incorrectly implements an integer overflow check. The generated patches remove the check, in effect reintroducing a security vulnerability from a previous version of libtiff (CVE-2006-2025) that a remote attacker can exploit to execute arbitrary injected code [2].
- **Buffer Overflow:** Defect fbc-bug-5458-5459 corresponds to an overly conservative check that prevents a buffer overflow. The generated patches remove the check, enabling the buffer overflow.

Standard Feature Elimination: Defects php-bug-307931-307934, gzip-bug-3fe0ca-39a362, lighttpd-bug-1913-1914, lighttpd-bug-2330-2331 correspond to incorrectly handled cases in standard functionality. The test suite contains a test case that exposes the incorrectly handled case, but no test case that exercises the standard functionality. The patches impair or remove the functionality, leaving the program unable to process standard use cases (such as decompressing non-zero files or initializing associative array elements to integer values).
Undefined Accesses: Patches often remove initialization code. While the resulting undefined accesses may happen to return values that enable the patch to pass the test cases, the patches can be fragile — different environments can produce values that cause the patch to fail (e.g., the AE patch for fbc-bug-5458-5459).
Deallocation Elimination: The patches for wireshark-bug-37112-37111 and php-bug-310011-310050 eliminate memory management errors by removing relevant memory deallocations. While this typically introduces a memory leak, it can also enhance survival by postponing the failure until the program runs out of memory (which may never happen). We note that human developers often work around difficult memory management defects by similarly removing deallocations.
Survival Enhancement: One potential benefit of even incorrect patches is that they may enhance the survival of the application even if they do not produce completely correct execution. This was the goal of several previous systems (which often produce correct execution even though that was not the goal) [53, 10, 42, 11, 29, 36, 44, 18, 47]. Defect lighttpd-bug-1794-1795 terminates the program if it encounters an unknown configuration file setting. The generated patches enhance survival by removing the check and enabling lighttpd to boot even with such configuration files. We note that removing the check is similar to the standard practice of disabling assertions in production use.
Relatively Minor Defects: We note that some of the defects can be viewed as relatively minor. For example, python-bug-69223-69224 causes the unpatched version of python to produce a SelectError message instead of a ValueError message — i.e., the correct behavior is to produce an error message, the defect is that python produces the wrong error message. Three of the wireshark defects (wireshark-bug-37172-37171, wireshark-bug-37172-37173, wireshark-bug-37284-37285) were caused by a developer checking in a version of wireshark with a debug macro flag set. The relevant defect is that these versions generate debugging information to the screen and to a log file. The correct behavior omits this debugging information.

7. ORIGINAL GENPROG PATCHES

We also analyzed the reported patches from the original GenProg system [60, 22]. Out of the 11 defects evaluated in the two papers, the corresponding patches for 9 defects are plausible but incorrect. 9 of the 11 patches simply eliminate functionality (by removing statements or adding a return or exit statements). We next discuss the reported patch for each application in turn.

- **uniq:** The patch is semantically equivalent to removing the statement *buf++ = c at uniq.c:74. The effect is that the patched application will ignore the user input file and operate as if the file were empty. Because of the use of a weak proxy, this patch is not plausible. The test scripts check the exit code of the program, not whether the output is correct.

- **look-u:** The patch is semantically equivalent to removing the condition argv[1] == "-" from the while loop at look.c:63. The effect is that look will treat the first command line argument (-d, -f, -t) as the name of the input file. Unless a file with such a name exists, look will then immediately exit without processing the intended input file.

- **look-s:** The patch is semantically equivalent to replacing the statement mid = (top+bot)/2 at look.c:87 with exit(0). The effect is that look will always exit immediately without printing any output (if the input file exists, if not, look will print an error message before it exits).

  The patch is plausible because the use of a weak test suite — the correct output for the positive and negative test cases is always no output. If the correct output for any of the test cases had been non-empty, this patch would have been implausible.

- **units:** The units program asks the user to input a sequence of pairs of units (for example, the pair meter and feet) and prints out the conversion factor between the two units in the pair. The patch is semantically equivalent to adding init() after units.c:279. The unpatched version of the program does not check for an overflow of the user input buffer. A long user input will therefore overflow the buffer.

  The GenProg patch does not eliminate the buffer overflow. It instead clears the unit table whenever the program reads an unrecognized unit (whether the unit overflows the user input buffer or not). Any subsequent attempt to look up the conversion for any pair of units will fail. It is also possible for the buffer overflow to crash the patched program.

- **deroff:** When deroff reads a backslash construct (for example, \L), it should read the next character (for example, ") as a delimiter. It should then skip any text until it reaches another occurrence of the delimiter or the end of the line.

  The patch is semantically equivalent to removing the statement bdelim=c (automatically generated as part of a macro expansion) at deroff.c:524. The effect is that the patched program does not process the delimiter correctly — when it encounters a delimiter, it skips all of the remaining text on the line, including any text after the next occurrence of the delimiter.

- **nullhttpd:** The patch is semantically equivalent to removing the call to strcmp(..., "POST") httpd_comb.c:4092-4099. The effect is that all POST requests generate an HTML bad request error reply.

- **indent:** The patch is semantically equivalent to adding a return after indent.c:926. The GenProg 2009 paper states that "Our repair removes handling of C comments that are not C++ comments." Our experiments with the patched version indicate that the patched version correctly handles at least some C comments that are not C++ comments (we never observed a C comment that it handled incorrectly). In many cases, however, the patched version simply exits after reading a { character, truncating the input file after the {.

- **flex:** The patch is semantically equivalent to removing the call to strcpy() at flex_comb.c:13784. This call transfers the token (stored in yytext) into the variable nmdef. Removing the call to strcpy() causes flex to incorrectly operate with an uninitialied nmdef. This variable holds one of the parsed tokens to process. The effect is that flex fails to parse the input file and incorrectly produces error messages that indicate that flex encountered an unrecognized rule.

- **atris:** The commments in the atris source code indicate that it is graphical tetris game. atris has the ability to load in user options stored in the .atrisrc in the user's home directory. A call to sprintf() at atrix_comb.c:5879 initializes the string buffer that specifies the filename of this .atrisrc file. If the home directory is longer than 2048 characters this sprintf() call will overflow the buffer.

  The patch is semantically equivalent to removing the call to sprintf() at atrix_comb.c:5879. The result is that the program passes an uninitialized filename to the procedure that reads the .atrisrc file.

The remaining 2 programs, for which GenProg generates correct patches, are used as motivating examples in the ICSE 2009 and GECCO 2009 papers [60, 22]. These two programs contain less than 30 lines of code.

We note that many of the test scripts use weak proxies. Specifically, all uniq, look-u, and look-s test cases do not compare the output of the patched program to the correct output. They instead check only that the patched program produces the correct exit code. Similarly, the deroff and indent negative test case test scripts only check the exit code.

8. KALI

The basic idea behind Kali is to search a simple patch space that consists solely of patches that remove functionality. There are two potential goals: 1) if the correct patch simply removes functionality, find the patch, 2) if the correct patch does not simply remove functionality, generate a patch that modifies the functionality containing the defect. For an existing statement, Kali deploys the following kinds of patches:

- **Redirect Branch:** If the existing statement is a branch statement, set the condition to true or false. The effect is that the then or else branch always executes.
- **Insert Return:** Insert a return before the existing statement. If the function returns a pointer, the inserted return statement returns NULL. If the function returns an integer, Kali generates two patches: one that returns 0 and another that returns -1.
- **Remove Statement:** Remove the existing statement. If the statement is a compound statement, Kali will remove all substatements inside it as well.

Statement Ordering: Each Kali patch targets a statement. Kali uses instrumented executions to collect information and order the executed statements as follows. Given a statement s and a test case i, $r(s, i)$ is the recorded execution counter that identifies the last execution of the statement s when the application runs with test case i. In particular, if the statement s is not executed at all when the application runs with the test case i, then $r(s, i) = 0$. Neg is the set of negative test cases (for which the unpatched application produces incorrect output) and Pos is the set of positive test cases (for which the unpatched application produces correct output). Kali computes three scores $a(s)$, $b(s)$, $c(s)$ for each statement s:

$$a(s) = |\{i \mid r(s,i) \neq 0,\, i \in \text{Neg}\}|$$
$$b(s) = |\{i \mid r(s,i) = 0,\, i \in \text{Pos}\}|$$
$$c(s) = \Sigma_{i \in \text{Neg}} r(s,i)$$

A statement s_1 has higher priority than a statement s_2 if $\text{prior}(s_1, s_2) = 1$, where prior is defined as:

$$prior(s_1,s_2) = \begin{cases} 1 & a(s_1) > a(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) > b(s_2) \\ 1 & \begin{array}{l} a(s_1) = a(s_2), b(s_1) = b(s_2), \\ c(s_1) > c(s_2) \end{array} \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, Kali prioritizes statements 1) that are executed with more negative test cases, 2) that are executed with less positive test cases, and 3) that are executed later during the executions with negative test cases. The Kali search space includes the top 500 ranked statements regardless of the file in which they appear.

Search: Kali deterministically searches the patch space in tiers: first all patches that change an if condition, then all patches that insert a return, then all patches that remove a statement. Within each tier, Kali applies the patch to the statements in the priority order identified above. It accepts a patch if the patch produces correct outputs for all of the inputs in the validation test suite.

8.1 Kali Evaluation Methodology

We evaluate Kali on all of the 105 defects in the GenProg set of benchmark defects [3]. We also use the validation test suites from this benchmark set. Our patch evaluation infrastructure is derived from the GenProg patch evaluation infrastructure [3]. For each defect, Kali runs its automatic patch generation and search algorithm to generate a sequence of candidate patches. For each candidate patch, Kali applies the patch to the application, recompiles the application, and uses the patch evaluation infrastructure to run the patched application on the inputs in the patch validation test suite. To check if the patch corrects the known

Defect	**GenProg**	**RSRepair**	**AE**	**Kali**			
				Result	**Search Space**	**Search Time**	**Type**
fbc-5458-5459	Plausible	-	Plausible‡	Plausible	737	2.4m	SL†
gmp-14166-14167	Plausible	Plausible‡	Plausible	Plausible	1169	19.5m	DP
gzip-3fe0ca-39a362	Plausible	Plausible	Plausible‡	Plausible	1241	28.2m	SF (119)*
gzip-a1d3d4-f17cbd	No Patch	-	Plausible	No Patch			
libtiff-0860361d-1ba75257	Plausible‡	Implausible	Plausible‡	Plausible	1525	16.7m	SL*
libtiff-5b02179-3dfb33b	Plausible‡	Implausible	Plausible‡	Plausible	1476	4.1m	DP
libtiff-90d136e4-4c66680f	Implausible	Implausible	Plausible‡	Plausible	1591	45.0m	SL†
libtiff-d13be72c-ccadf48a	Plausible‡	Plausible‡	Plausible‡	Plausible	1699	42.9m	SL*
libtiff-ee2ce5b7-b5691a5a	Implausible	Implausible	Plausible‡	Plausible	1590	45.1m	SF(10)*
lighttpd-1794-1795	Plausible‡	-	Plausible‡	Plausible	1569	5.9m	
lighttpd-1806-1807	Plausible‡	Plausible‡	Plausible‡	Plausible	1530	55.5m	SF(21)†
lighttpd-1913-1914	Plausible‡	Plausible‡	No Patch	Plausible	1579	158.7m	SL*
lighttpd-2330-2331	Plausible‡	Plausible‡	Plausible‡	Plausible	1640	36.8m	SF(19)†
lighttpd-2661-2662	Plausible‡	Plausible‡	Plausible‡	Plausible	1692	59.7m	DP
php-307931-307934	Plausible‡	-	Plausible‡	Plausible	880	9.2m	DP
php-308525-308529	No Patch	-	Plausible‡	Plausible	1152	234.0m	SL†
php-309111-309159	No Patch	-	Correct	No Patch			
php-309892-309910	Correct‡	Correct‡	Correct‡	Correct	1498	20.2m	C
php-309986-310009	Plausible‡	-	Plausible‡	Plausible	1125	10.4m	SF(27)*
php-310011-310050	Plausible	-	Plausible‡	Plausible	971	12.9m	SL*
php-310370-310389	No Patch	-	No Patch	Plausible	1096	12.0m	DP
php-310673-310681	Plausible‡	-	Plausible‡	Plausible	1295	89.00m	SL*
php-311346-311348	No Patch	-	No Patch	Correct	941	14.7m	C
python-69223-69224	No Patch	-	Plausible	No Patch			
python-69783-69784	Correct‡	Correct‡	Correct‡	Correct	1435	16.1m	C
python-70098-70101	No Patch	-	Plausible	Plausible	1233	6.8m	SL*
wireshark-37112-37111	Plausible‡	Plausible	Plausible‡	Plausible	1412	19.6m	SL†
wireshark-37172-37171	No Patch	-	Plausible‡	Plausible	1459	10.9m	SL†
wireshark-37172-37173	No Patch	-	Plausible‡	Plausible	1459	10.9m	SL†
wireshark-37284-37285	No Patch	-	Plausible‡	Plausible	1482	11.5m	SL†

Figure 1: Experimental Results

incorrect behavior from the test suite, Kali first runs the negative test cases. To check if the patch preserves known correct behavior from the test suite, Kali next runs the positive test cases. If all of the test cases produce the correct output, Kali accepts the patch. Otherwise it stops the evaluation of the candidate patch at the first incorrect test case and moves on to evaluate the next patch.

Kali evaluates the php patches using the modified php test harness described in Section 3. It evaluates the gmp patches using a modified gmp test script that checks that all output components are correct. It evaluates the libtiff patches with augmented test scripts that compare various elements of the libtiff output image files from the patched executions with the corresponding elements from the correct image files. Other components of the image files change nondeterministically without affecting the correctness. The libtiff test scripts therefore do not fully check for correct outputs. After Kali obtains patches that pass the modified libtiff test scripts, we manually evaluate the outputs to filter all Kali patches that do not produce correct outputs for all of the inputs in the validation test suite. This manual evaluation rejects 7 libtiff patches, leaving only 5 plausible patches. Effective image comparison software would enable Kali to fully automate the libtiff patch evaluation.

We perform all of our Kali experiments (except for the fbc defects) on Amazon EC2 Intel Xeon 2.6GHz Machines running Ubuntu-64bit 14.04. The fbc application only runs in 32-bit environment, so we use a virtual machine with Intel Core 2.7GHz running Ubuntu-32bit 14.04 for fbc.

8.2 Experimental Results

Figure 1 presents the experimental results from our analysis of these patches. The figure contains a row for each defect for which at least one system (GenProg, RSRepair, AE, or Kali) generates a plausible patch. The second to fifth columns present the results of GenProg, RSRepair, AE, and Kali on each defect. "Correct" indicates that the system generates at least one correct patch for the defect. "Plausible" indicates that the system generates at least one plausible patch but no correct patches for the defect. "Implausible" indicates that all patches generated by the system for the defect are not plausible. "No Patch" indicates that the system does not generate any patch for the defect. "-" indicates that the RSRepair researchers chose not to include the defect in their study [48]. "‡" indicates that at least one of analyzed patches is not equivalent to a single functionality elimination modification.

Our results show that for the defects in the GenProg benchmark set, Kali generates correct patches for at least as many defects (3 for Kali vs. 3 for AE and 2 for GenProg and RSRepair) and plausible patches for at least as many defects (27 for Kali vs. 18 for GenProg, 10 for RSRepair, and 27 for AE).

Search Space and Time Results: The sixth column of Figure 1 presents the size of the search space for each defect (which is always less than 1700 patches). The seventh column presents the search times. Kali typically finds the patches in tens of minutes. If the search space does not contain a plausible patch, Kali typically searches the entire space in several hours and always less than seven hours.

It is not possible to directly compare the reported performance numbers for GenProg, RSRepair, and AE [26, 48, 59] with the numbers in Figure 1. First, the reported aggregate results for these prior systems include large numbers of implausible patches. The reported results for individual defects ([48], Table 2) report too few test case executions to validate plausible patches for the validation test suite (specifically, the reported number of test case executions is less than the number of test cases in the test suite). Second, these prior systems reduce the search space by requiring the developer to identify a target source code file to attempt to patch (Kali, of course, works with the entire application). Nevertheless, the search space sizes for these prior systems appear to be in the tens of thousands ([59], Table I) as opposed to hundreds for Kali. These numbers are consistent with the simpler Kali search space induced by the simpler set of Kali functionality deletion modifications.

Patch Classification: The last column of Figure 1 presents our classification of the Kali patches. "C" indicates that the Kali patch is correct. There are three defects for which Kali generates a correct patch. For two of the defects (php-bug-309111-309159, python-bug-69783-69784) both the Kali and developer patch simply delete an if statement. For php-bug-311346-311348, the Kali patch is a then redirect patch. The developer patch changes the else branch, but when the condition is true, the then branch and modified else branch have the same semantics.

"SL" indicates that the Kali and corresponding developer patches modify the same line of code. "*" indicates that the developer patch modified only the function that the Kali patch modified. "†" indicates that the developer patch modified other code outside the function. In many cases the Kali patch cleanly identifies the exact functionality and location that the developer patch modifies. Examples include changing the same if condition (fbc-bug-5458-5459, libtiff-bug-d13be72c-ccadf48a), changing the condition of an if statement when the developer patch modifies the then and/or else clause of that same if statement (python-bug-70098-70101, libtiff-bug-0860361d-1ba75257, wireshark-bug-37112-37111), deleting code that the developer patch encloses in an if statement (lighttpd-bug-1913-1914, php-bug-310673-310681, and deleting the same code (php-bug-308525-308529, libtiff-bug-0860361d-1ba75257, libtiff-bug-90d136e4-4c66680f, wireshark-bug-37172-37171, wireshark-bug-37172-37173, wireshark-bug-37284-37285) as the developer patch. Many of the patches correspond quite closely to the developer patch and move the application in the same direction.

"SF" indicates that the Kali and corresponding developer patches modify the same function. The number in parentheses is the distance in lines of code between the Kali patch and developer modifications. The Kali and developer patches typically modify common functionality and variables. Examples include reference counting (php-bug-309986-310009), caching (lighttpd-bug-1806-1807), and file encoding mechanism functionality (lighttpd-bug-2330-2331).

"DP" indicates that the Kali and developer patches modify different functions, but there is some dependence that connects the Kali and developer patches. Examples include changing the return value of a function invoked by code that the developer patch modifies (gmp-bug-14166-14167), deleting a call to a function that the developer patch modifies (php-bug-307931-307934), modifying memory management code for the same data structure (php-bug-310370-310389), and accessing the same value, with either the Kali or the developer patch changing the value (lighttpd-bug-2661-2662, libtiff-bug-5b02179-3dfb33b).

The Kali patch for lighttpd-bug-1794-1795 (like the GenProg and AE patches) is an outlier — it deletes error handling code automatically generated by the yacc parser generator. The developer patch changes the yacc code to handle new configuration parameters. We do not see the any of the automatically generated patches as providing useful information about the defect.

python-bug-69783-69784: Figure 8 (see Appendix B) presents the Kali patch for python-bug-69783-69784. Like the GenProg, RSRepair, and AE patches, the patch for this defect deletes the if statement that implements two-digit years. Note that unlike these previous systems, which generate preprocessed code, Kali operates directly on and therefore preserves the structure of the original source code. To implement the deletion, Kali conjoins false (i.e., !1) to the condition of the if statement.

php-bug-309892-309910: Figure 9 (see Appendix B) presents the Kali patch for php-bug-309892-309910. Like the GenProg, RSRepair, and AE patches, this patch deletes the if statement that implements the obsolete check.

php-bug-311346-311348: Figure 10 (see Appendix B) presents the Kali patch for php-bug-311346-311348. This code concatenates two strings, ctx->buf.c and output. The original code incorrectly set the result handled_output to NULL when the first string is empty. The Kali patch, in effect, deletes the else branch of the if statement on line 1 so that handled_output is correctly set when ctx->buf.c is empty and output is not empty. Figure 11 presents the developer patch. The developer patches the else branch to correctly set handled_output when ctx->buf.c is empty. The two patches have the same semantics.

8.3 Discussion

While many of the plausible but incorrect Kali patches precisely pinpoint the defective code, that is far from the only useful aspect of the patch. The fact that the patch changes the behavior of the program to produce the correct output for the negative input provides insight into what functionality is defective and how the defect affects that functionality. Even when the Kali patch is not correct, it often moves the program in the same direction as the developer patch, for example by deleting code that the developer patch causes to execute only conditionally.

We note that, by directly implementing functionality elimination patches (as opposed to using a broader set of modifications to generate more complex patches that, in the end, are equivalent to functionality elimination), the Kali patches can be more transparent and easier to understand. Many GenProg patches, for example, contain multiple modifications that can obscure the semantic simplicity of the patch. Unlike GenProg, RSRepair, and AE, Kali operates directly on the original source code. The prior systems, in contrast, operate on preprocessed code, which in our experience significantly impairs the transparency and comprehensibility of the patches.

9. CLEARVIEW

Of course GenProg, RSRepair, and AE (and now Kali) are not the only generate-and-validate patch generation systems. ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [47]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches that take a repair action when the invariant is violated. Subsequent executions enable ClearView to determine if 1) the patch eliminates the defect while 2) preserving desired benign behavior. ClearView, GenProg, RSRepair, AE, and Kali all use the basic generate-and-validate approach of generating multiple patches, then evaluating the patches based on their impact on subsequent executions of the patched application. But ClearView leverages additional information not immediately available in the test suite, specifically invariants learned in previous executions on benign inputs. This additional information enables ClearView to produce more targeted patches. The experimental results indicate that this additional information enables ClearView to produce more successful patches in less time (ClearView produces patches in five minutes on average for the evaluated defects [47]).

ClearView differs from GenProg, RSRepair, and AE in two important ways. First, ClearView's goal is to enable the program to survive defect-triggering inputs and security attacks and to continue on to process subsequent inputs successfully. Therefore the correct outputs for the inputs that trigger the defects are not required and not available. In some cases it is not even clear what the correct output should be. More generally, how one would obtain correct outputs for defect-triggering inputs is an open question — in many cases we anticipate that the easiest way of obtaining such outputs may be to manually fix the defect, then use the new version of the program to produce the correct output. For such cases, the utility of automatic patch generation systems that require correct outputs for defect-triggering inputs is not clear. The difficulty is especially acute for fully automatic systems that must respond to new defect-triggering inputs with no human intervention and no good way to obtain correct outputs for these inputs.

A second difference is that ClearView generates binary patches and applies these binary patches to running programs without otherwise interrupting the execution of the program. It is, of course, possible to automatically generate source-level patches, but binary patching gives ClearView much more flexibility in that it can patch programs without source and without terminating the application, recompiling, and restarting.

ClearView Patch Evaluation: ClearView was evaluated by a hostile Red Team attempting to exploit security vulnerabilities in Firefox [47]. The Red Team developed attack web pages that targeted 10 Firefox vulnerabilities. These attack web pages were used during a Red Team exercise to evaluate the ability of ClearView to automatically generate patches that eliminated the vulnerability. During the Red Team exercise, ClearView automatically generated patches for 7 of the 10 defects. Additional experiments performed after the Red Team exercise showed that a ClearView configuration change enables ClearView to automatically patch one of the remaining vulnerabilities and that an enhanced set of learning inputs enables ClearView to automatically patch another of the remaining vulnerabilities, for a total of 9 ClearView patches for 10 vulnerabilities. An examination of the patches after the Red Team exercise indicates that each patch successfully eliminated the corresponding security vulnerability. A manual translation of these ClearView patches into source-level patches is available [51].

To evaluate the quality of the continued execution after the presentation of the attack, the Red Team also developed 57 previously unseen benign web pages that exercised a range of Firefox functionality. ClearView learned invariants not from these 57 web pages but from other benign web pages developed independently of the Red Team. All of the 9 generated patches enabled Firefox to survive the security attacks and continue successful execution to produce correct behavior on these 57 benign web pages.

Correctness and Functionality Elimination: Unlike (apparently) the GenProg, RSRepair, and AE patches, the ClearView patches were subjected to intensive human investigation and evaluation during the Red Team evaluation. The ability of the generated patches to eliminate security vulnerabilities is therefore well understood. Although generating correct patches was not a goal, our evaluation of the patches indicates that at least 4 of the patches are correct (for defects 285595, 290162, 295854, and 296134). 3 of the patches implement conditional functionality elimination — they insert an if statement that checks a condition and returns if the condition is true. 5 of the patches implement a conditional assignment — they insert an if statement that checks a condition and, if the condition is true, sets a variable to a value that enforces a learned invariant.

10. THREATS TO VALIDITY

The data set considered in this paper was selected not by us, but by the GenProg developers in an attempt to obtain a large, unbiased, and realistic benchmark set [26]. The authors represent the study based on this data set as a "Systematic Study of Automated Program Repair" and identify one of the three main contributions of the paper as a "systematic evaluation" that "includes two orders of magnitude more" source code, test cases, and defects than previous studies [26]. Moreover, the benchmark set was specifically constructed to "help address generalizability concerns" [26]. Nevertheless, one potential threat to validity is that our results may not generalize to other applications, defects, and test suites. In particular, if the test suites provide more coverage of the functionality that contains the defect, we would not expect functionality deletion modifications to be as effective in enabling applications to produce plausible patches.

For each defect, we analyze only the first five patches that RSRepair generates. It is possible that the remaining patches may have other characteristics (although our initial examination of these other patches revealed no new characteristics).

11. RELATED WORK

SPR: SPR is a generate-and-validate patch generation system [34, 35] that uses *staged condition synthesis*. SPR first selects parameterized patch schemas, some of which contain abstract values to represent branch conditions. For each schema, SPR determines whether there is any instantiation of the schema that can pass all test cases. If so, SPR then instantiates the abstract value in the schema to obtain and validate candidate patches. For schemas with abstract conditions, SPR finds a set of desired branch directions for that abstract condition and synthesizes an expression that realizes these branch directions. This technique significantly reduces the number of candidate patches that SPR attempts to validate. For the same GenProg benchmark set [26] (excluding the 36 deliberate functionality changes), the SPR search space contains correct patches for 19 defects, 11 of which SPR finds as the first patch that passes the test suite. SPR also finds the correct patch for python-bug-69783-69784.

Prophet: Prophet is a patch generation system that learns a probabilistic model over candidate patches from a large code database that contains many past successful human patches [33]. It defines the probabilistic model as the combination of a distribution over program points based on an error localization algorithm and a parameterized log-linear distribution over program modification operations. It then learns the model parameters via maximum log-likelihood, which identifies important characteristics of successful human patches. Prophet uses the learned model to identify likely correct patches within the Prophet search space. One goal is to overcome the limitations of weak test suites by learning characteristics of correct patches. For the same GenProg benchmark set [26] (excluding the 36 deliberate functionality changes), Prophet finds 14 correct patches as the first patch that passes the test suite.

Multi-Application Code Transfer: CodePhage [56, 55] automatically locates and transfers correct code from donor applications to eliminate defects in recipient applications. CodePhage successfully repaired 10 defects in 7 recipient applications via code transfer from 5 donor applications.

Failure-Oblivous Computing: Failure-oblivious computing [53] checks for out of bounds reads and writes. It discards out of bounds writes and manufactures values for out of bounds reads. This eliminates data corruption from out of bounds writes, eliminates crashes from out of bounds accesses, and enables the program to continue execution along its normal execution path.

Failure-oblivious computing was evaluated on 5 errors in 5 server applications. For all 5 errors, this technique enabled the servers to survive otherwise fatal errors and continue on to successfully process subsequent inputs. For 2 of the 5 errors, it completely eliminates the error and, on all inputs, deliver the same output as the developer patch that corrects the error (we believe these patches are correct).

Boundless Memory Blocks: Boundless memory blocks store out of bounds writes in a hash table to return as the result of corresponding out of bounds reads [52]. The technique was evaluated on the same set of applications as failure-oblivious computing and delivered the same results.

Bolt and Jolt: Bolt [29] attaches to a running application, determines if the application is in an infinite loop, and, if so, exits the loop. A user can also use Bolt to exit a long-running loop. In both cases the goal is to enable the application to continue useful execution. Bolt was evaluated on 13 infinite and 2 long-running loops in 12 applications. For 14 of the 15 loops Bolt delivered a result that was the same or better than terminating the application. For 7 of the 15 loops, Bolt completely eliminates the error and, on all inputs, delivers the same output as the developer patch that corrects the error (we believe these patches are correct). Jolt applies a similar approach but uses the compiler to insert the instrumentation [11].

RCV: RCV [36] enables applications to survive null dereference and divide by zero errors. It discards writes via null references, returns zero for reads via null references, and returns zero as the result of divides by zero. Execution then continues along the normal execution path.

RCV was evaluated on 18 errors in 7 applications. For 17 of these 18 errors, RCV enables the application to survive the error and continue on to successfully process the remaining input. For 11 of the 18 errors, RCV completely eliminates the error and, on all inputs, delivers either identical (9 of 11 errors) or equivalent (2 of 11 errors) outputs as the developer patch that corrects the error (we believe these patches are correct).

Memory Leaks: Cyclic memory allocation eliminates memory leaks by statically bounding the amount of memory that can be allocated at any allocation site [44]. LeakFix [25] proposes to fix memory leaks in C programs by inserting deallocations automatically. LeakFix guarantees that the inserted fix is safe, i.e., the inserted fix will not cause free-before-allocation, double-free, or use-after-free errors.

Integer and Buffer Overflows: TAP automatically discovers and patches integer and buffer overflow errors [57]. TAP uses a template-based approach to generate source-level patches that test for integer or buffer overflows. If an overflow is detected, the patches exit the program before the overflow can occur.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [15, 17, 16, 14]. Data structure repair enforces a data structure consistency specification. This specification can be provided by a human developer or automatically inferred from correct program executions [14]. Assertion-based data structure repair [20] starts from an erroneous data structure state that triggers an assertion violation and uses symbolic execution to explore possible structure mutations that can repair the state.

APPEND: APPEND [18] proposes to eliminate null pointer exceptions in Java by applying recovery techniques such as replacing the null pointer with a pointer to an initialized instance of the appropriate class. The presented examples illustrate how this technique can effectively eliminate null pointer exceptions and enhance program survival.

Principled PHP Repair: PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair php programs that generate HTML [54]. By formulating the problem as a string constraint problem, PHPRepair obtains sound, complete, and minimal repairs to ensure the patched php program passes a validation test suite. PHPRepair therefore illustrates how the structure in the problem enables a principled solution that provides benefits that other program repair systems typically cannot provide.

Solver-Based Approaches: Several patch generation systems use SMT solvers to search the patch space. SemFix uses SMT solvers to find expressions that enable the patched program to generate correct outputs for all inputs in a validation test suite [43]. DirectFix extends SemFix to limit the search space to small patches [40]. NOPOL uses a solver to repair incorrect branch conditions [13]. The goal is to find a new branch condition that flips the taken branch direction for negative test cases and preserves the taken branch direction for positive test cases [13]. Infinitel uses a solver to find new loop exit conditions that eliminate infinite loops [37].

Debroy and Wong: Debroy and Wong present a generate-and-validate approach with two modifications: replacement of an operator with another from the same class and condition negation [12]. The results, on the Siemens suite (which contains seeded errors) and the Java Ant program, indicate that this approach can effectively fix a reasonable percentage of the studied errors. Our research differs in the scale of the benchmark programs (large production programs rather than, in the case of the Seimens suite, small benchmark programs), the nature of the faults (naturally occurring rather than seeded), and the identification of deletion as an effective way of obtaining plausible patches.

Mutation-Based Fault Localization: The Metallaxis-FL system operates on the principle that mutants that exhibit similar test suite behavior (i.e., fail and pass the same test cases) as the original program containing the defect are likely to modify statements near the defect [45, 46]. This principle is opposite to the Kali approach (Kali searches for modifications that pass all test cases even though the original program fails at least one). The results show that Metallaxis-FL works best with test suites with good coverage of the code with the defect. The production test suites with which we evaluate Kali, in contrast, have poor coverage of the code with the defect (which we would expect to typically be the case in practice). Metallaxis-FL was evaluated on the Siemens suite (with hundreds of lines of code per program). Kali, in contrast, was evaluated on large production applications.

Researchers have used automated program repair to measure the effectiveness of fault localization techniques (using the different techniques to drive patch generation locations) [49]. Kali, in contrast, uses automated program repair to, in part, generate patches that provide useful information about defects.

Human Patch Acceptability: A study comparing the acceptability of GenProg, PAR, and human-generated patches for Java programs found that the PAR patches were more acceptable to human developers than the GenProg patches [28] (GenProg works on C programs. It is not clear how the authors of the PAR paper managed to get GenProg to generate patches for Java programs). The study only addresses human acceptability, with apparently no investigation of patch correctness. The study also found that PAR generated plausible patches for more defects than GenProg. The study noted that GenProg produced plausible patches for only 13% (16 out of 119) of the defects in the study as opposed to the 52% (55 out of 105) reported in the GenProg paper [26]. One potential explanation for this discrepancy is the use of weak proxies in the GenProg paper, which substantially increases the number of reported patches.

Patch Characteristics: Researchers have investigated the relative frequencies of different patch generation modifications, both for automated patch generation [32] and for patches generated by human developers (but with applications to automatic patch generation) [39]. Both of these papers characterize the syntactic modifications (add, substitute, delete) used to create the patch. Our semantic patch analysis, in contrast, works with the patch semantics to recognize the semantic equivalence of different syntactic modifications — the vast majority of the patches that we analyze are semantically equivalent to a single functionality deletion modification (even though the patch itself may be implemented with a variety of different syntactic modifications).

Fix Ingredients: Martinez et. al. studied more than 7,000 human commits in six open source programs to measure the *fix ingredient availability*, i.e., the percentage of commits that could be synthesized solely from existing lines of code. The results show that only 3-17% of the commits can be synthesized from existing lines of code. The study provides another potential explanation for the inability of GenProg, RSRepair, and AE to generate correct patches — the GenProg, RSRepair and AE search space only contains patches that can be synthesized from existing lines of code (specifically by copying and/or removing existing statements without variable replacement or any other expression-level modification).

Correctness Evaluation: A very recent paper [19] evaluates patches generated by GenProg, Kali, and NOPOL [13] for 224 Java program defects in the Defects4J dataset [27]. The results are, in general, consistent with ours — out of 42 manually analyzed plausible patches, the analysis indicates that only 8 patches are undoubtedly correct.

Discussion: A common pattern that emerges is that more structure enhances the ability of the system to produce effective repairs (but also limits the scope of the defects that it can handle). The availability of specifications can enable systems to provide repairs with guaranteed correctness properties, but impose the (in many cases unrealistic) burden of obtaining specifications. PHPRepair exploits the structure present in the domain to provide principled patches for a specific class of defects. Kali, GenProg, RSRepair, and AE aspire to address a different and less structured class of defects, but without the guarantees that PHPRepair can deliver. One of the benefits of exploiting the structure is a reduced risk of producing patches with negative effects such as the introduction of security vulnerabilities.

A primary goal of many of the targeted systems is to enable applications to survive otherwise fatal inputs [53, 10, 42, 11, 29, 36, 44, 15, 47]. The rationale is that the applications (conceptually) process multiple inputs; enabling applications to survive otherwise fatal inputs enables the applications to successfully process subsequent inputs. The techniques therefore focus on eliminating fatal errors or security vulnerabilities in potentially adversarial use cases. Generating correct output for such adversarial inputs is often not a goal; in some cases it is not even clear what the correct output should be. Nevertheless, the techniques often succeed in delivering patches with identical functionality as the subsequent developer patches. They are also quite simple — each repair typically consists of a check (such as an out of bounds access) followed by a simple action (such as discarding the write). Although GenProg, RSRepair, and AE may, in principle, be able to generate larger and more complex repairs, in practice the overwhelming majority of the repairs that they do generate are semantically very simple.

12. CONCLUSION

The results presented in this paper highlight several important design considerations for generate-and-validate patch generation systems. First, such systems should not use weak proxies (such as the exit code of the program) — they should instead actually check that the patched program produces acceptable output. Second, the search space and search algorithm are critical — a successful system should use 1) a search space that contains successful patches and 2) a search algorithm that can search the space efficiently enough to find successful patches in an acceptable amount of time. Third, simply producing correct results on a validation test suite is (at least with current test suites) far from enough to ensure acceptable patches. Especially when the test suite does not contain test cases that protect desired functionality, unsuccessful patches can easily generate correct outputs.

Incorporating additional sources of information may be a particularly effective way of improving patch quality. Promising sources include correct code transferred from other applications [56, 55], learning characteristics of successful human-generated patches [33], learning characteristics of previous successful executions [14, 47], exploiting properties of targeted defect classes [53, 10, 42, 36, 18, 36, 44, 25, 11, 29, 37, 57], and specifications that identify correct program behavior [58, 15, 24].

Acknowledgments

We would like to thank Westley Weimer for the very useful and enlightening feedback he provided on various parts of an earlier draft of this paper. We also thank the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8650-11-C-7192).

13. REFERENCES

[1] AE results. `http://dijkstra.cs.virginia.edu/genprog/resources/genprog-ase2013-results.zip`.

[2] CVE-2006-2025. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2025`.

[3] GenProg benchmarks. `http://dijkstra.cs.virginia.edu/genprog/resources/genprog-icse2012-benchmarks/`.

[4] GenProg: Evolutionary Program Repair. `http://dijkstra.cs.virginia.edu/genprog/`.

[5] GenProg results. `http://dijkstra.cs.virginia.edu/genprog/resources/genprog-icse2012-results.zip`.

[6] GenProg source code. `http://dijkstra.cs.virginia.edu/genprog/resources/genprog-source-v3.0.zip`.

[7] GenProg virtual machine. `http://dijkstra.cs.virginia.edu/genprog/resources/genprog_images`.

[8] RSRepair results. `http://sourceforge.net/projects/rsrepair/files/`.

[9] Claire Le Goues, personal communication, May 2015.

[10] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM, 2006.

[11] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP 2011–Object-Oriented Programming*, pages 609–633. Springer, 2011.

[12] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[13] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[14] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[15] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03', pages 78–95, New York, NY, USA, 2003. ACM.

[16] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05', pages 176–185, New York, NY, USA, 2005. ACM.

[17] B. Demsky and M. C. Rinard. Static specification analysis for termination of specification-based data structure repair. In *14th International Symposium on Software Reliability Engineering (ISSRE) 2003), 17-20 November 2003, Denver, CO, USA*, pages 71–84, 2003.

[18] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 47–56, 2008.

[19] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *arXiv*, abs/1505.07002, 2015.

[20] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07', pages 64–73, 2007.

[21] E. Fast, C. L. Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 965–972, 2010.

[22] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09', pages 947–954, New York, NY, USA, 2009. ACM.

[23] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[24] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 653–663, 2014.

[25] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[26] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13, 2012.

[27] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440, 2014.

[28] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[29] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *ACM SIGPLAN Notices*, volume 47, pages 431–450. ACM, 2012.

[30] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (to appear)*.

[31] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[32] C. Le Goues, W. Weimer, and S. Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 959–966. ACM, 2012.

[33] F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful human patches. Technical Report MIT-CSAIL-TR-2015-019, 2015.

[34] F. Long and M. Rinard. Staged program repair in SPR. Technical Report MIT-CSAIL-TR-2015-008, 2015.

[35] F. Long and M. Rinard. Staged program repair in SPR. In *Proceedings of ESEC/FSE 2015 (to appear)*, 2015.

[36] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 26. ACM, 2014.

[37] S. L. Marcote and M. Monperrus. Automatic Repair of Infinite Loops. Technical Report 1504.05078, Arxiv, 2015.

[38] M. Martinez. Extraction and analysis of knowledge for automatic software repair. *Software Engineering. Universite Lille*, (tel-01078911), 2014.

[39] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, pages 1–30, 2013.

[40] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[41] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 234–242, 2014.

[42] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *Proceedings of the 2009 international symposium on Memory management*, pages 49–58. ACM, 2009.

[43] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[44] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 15–30, New York, NY, USA, 2007. ACM.

[45] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 2013.

[46] M. Papadakis and Y. Le Traon. Effective fault localization via mutation analysis: A selective mutation approach. In *ACM Symposium On Applied Computing (SAC'14)*, 2014.

[47] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[48] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265, 2014.

[49] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, 2013.

[50] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems (Supplementary Material). `http://hdl.handle.net/1721.1/97051`.

[51] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems (Supplementary Material). `http://hdl.handle.net/1721.1/93255`.

[52] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, pages 82–90, 2004.

[53] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, volume 4, pages 21–21, 2004.

[54] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[55] S. Sidiroglou, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2014-024, Aug. 2014.

[56] S. Sidiroglou, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[57] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard. Automatic discovery and patching of buffer and integer overflow errors. Technical Report MIT-CSAIL-TR-2015-018, 2015.

[58] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[59] W. Weimer, Z. P. Fry, and S. 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.

[60] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

APPENDIX

A. GENPROG AND AE CORRECT PATCHES

```
- if (y < 1000) {
-    tmp___0 = PyDict_GetItemString(moddict, "accept2dyear");
-    accept = tmp___0;
-    if ((unsigned int )accept != (unsigned int )((void *)0)) {
-      tmp___1 = PyObject_IsTrue(accept);
-      acceptval = tmp___1;
-      if (acceptval == -1) {
-        return (0);
-      } else {

-      }

-      if (acceptval) {
-        if (0 <= y) {
-          if (y < 69) {
-            y += 2000;
-          } else {
-            goto _L;
-          }
-        } else {
-          _L: /* CIL Label */
-          if (69 <= y) {
-            if (y < 100) {
-              y += 1900;
-            } else {
-              PyErr_SetString(PyExc_ValueError,
                               "year out of range");
-              return (0);
-            }
-          } else {
-            PyErr_SetString(PyExc_ValueError,
                             "year out of range");
-            return (0);
-          }
-        }

-        tmp___2 = PyErr_WarnEx(PyExc_DeprecationWarning,
-                   "Century info guessed for a 2-digit year.", 1);
-        if (tmp___2 != 0) {
-          return (0);
-        } else {

-        }
-      } else {

-      }
-    } else {
-      return (0);
-    }
-  } else {

-  }
   p->tm_year = y - 1900;
   (p->tm_mon) --;
   p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 2: GenProg patch for python-bug-69783-69784

```
-if (y < 1000) {
-    PyObject *accept = PyDict_GetItemString(moddict,
-                                            "accept2dyear");
-    if (accept != NULL) {
-        int acceptval =  PyObject_IsTrue(accept);
-        if (acceptval == -1)
-            return 0;
-        if (acceptval) {
-            if (0 <= y && y < 69)
-                y += 2000;
-            else if (69 <= y && y < 100)
-                y += 1900;
-            else {
-                PyErr_SetString(PyExc_ValueError,
-                                "year out of range");
-                return 0;
-            }
-            if (PyErr_WarnEx(PyExc_DeprecationWarning,
-                "Century info guessed for a 2-digit year.", 1) != 0)
-                return 0;
-        }
-    }
-    else
-        return 0;
-}
 p->tm_year = y - 1900;
 p->tm_mon--;
 p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 3: Developer patch for python-bug-69783-69784

```
if (offset >= (long )s1_len) {
    php_error_docref0((char const   *)((void *)0), 1 << 1L,
    "The start position cannot exceed initial string length");
    while (1) {
      __z___1 = return_value;
      __z___1->value.lval = 0L;
      __z___1->type = (unsigned char)3;
      break;
    }
    return;
  } else {

  }
- if (len > (long )s1_len - offset) {
-   len = (long )s1_len - offset;
- } else {

- }
  if (len) {
    tmp___1 = len;
  } else {
    if ((long )s2_len > (long )s1_len - offset) {
      tmp___0 = (long )s2_len;
    } else {
      tmp___0 = (long )s1_len - offset;
    }
    tmp___1 = tmp___0;
  }
  cmp_len = (unsigned int )tmp___1;
```

Figure 4: GenProg patch for php-bug-309892-309910

```
if (offset >= s1_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
        "The start position cannot exceed initial string length");
        RETURN_FALSE;
}

-if (len > s1_len - offset) {
-        len = s1_len - offset;
-}

cmp_len = (uint) (len ? len : MAX(s2_len, (s1_len - offset)));
```

Figure 5: Developer patch for php-bug-309892-309910

```
if (pp) {
   if ((unsigned int )pp < (unsigned int )p) {
+    ...
     p = pp;
+    ...
+    if (__genprog_mutant == 25) {
+      if (p - s) {
+        tmp___24 = _estrndup(s, (unsigned int )(p - s));
+        ret->path = (char *)tmp___24;
+        php_replace_controlchars_ex(ret->path, p - s);
+      } else {

+      }
     }
+    ...
    goto label_parse;
   } else {

   }
  } else {

  }
  if (p - s) {
   tmp___21 = _estrndup(s, (unsigned int )(p - s));
   ret->path = (char *)tmp___21;
   php_replace_controlchars_ex(ret->path, p - s);
  } else {

  }
```

Figure 6: AE patch for php-bug-309111-309159

```
if (pp && pp < p) {
+ if (pp - s) {
+     ret->path = estrndup(s, (pp-s));
+     php_replace_controlchars_ex(ret->path, (pp - s));
+ }
  p = pp;
  goto label_parse;
}

if (p - s) {
  ret->path = estrndup(s, (p-s));
  php_replace_controlchars_ex(ret->path, (p - s));
}

```

Figure 7: Developer patch for php-bug-309111-309159

B. KALI CORRECT PATCHES

```
-if (y < 1000) {
+if (y < 1000 && !1) {
     PyObject *accept = PyDict_GetItemString(moddict,
                                             "accept2dyear");
     if (accept != NULL) {
         int acceptval =  PyObject_IsTrue(accept);
         if (acceptval == -1)
             return 0;
         if (acceptval) {
             if (0 <= y && y < 69)
                 y += 2000;
             else if (69 <= y && y < 100)
                 y += 1900;
             else {
                 PyErr_SetString(PyExc_ValueError,
                                 "year out of range");
                 return 0;
             }
             if (PyErr_WarnEx(PyExc_DeprecationWarning,
                 "Century info guessed for a 2-digit year.", 1) != 0)
                 return 0;
         }
     }
     else
         return 0;
 }
 p->tm_year = y - 1900;
 p->tm_mon--;
 p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 8: Kali patch for python-bug-69783-69784

```
if (offset >= s1_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,
        "The start position cannot exceed initial string length");
        RETURN_FALSE;
}

-if (len > s1_len - offset) {
+if (len > s1_len - offset && !1) {
         len = s1_len - offset;
 }

cmp_len = (uint) (len ? len : MAX(s2_len, (s1_len - offset)));
```

Figure 9: Kali patch for php-bug-309892-309910

```
-if (ctx->buf.len) {
+if ((ctx->buf.len) || 1) {
     smart_str_appendl(&ctx->result, ctx->buf.c, ctx->buf.len);
     smart_str_appendl(&ctx->result, output, output_len);

     *handled_output = ctx->result.c;
     *handled_output_len = ctx->buf.len + output_len;

     ctx->result.c = NULL;
     ctx->result.len = 0;
     smart_str_free(&ctx->buf);
 } else {
     *handled_output = NULL;
 }
```

Figure 10: Kali patch for php-bug-311346-311348

```
 if (ctx->buf.len) {
     smart_str_appendl(&ctx->result, ctx->buf.c, ctx->buf.len);
     smart_str_appendl(&ctx->result, output, output_len);

     *handled_output = ctx->result.c;
     *handled_output_len = ctx->buf.len + output_len;

     ctx->result.c = NULL;
     ctx->result.len = 0;
     smart_str_free(&ctx->buf);
 } else {
-    *handled_output = NULL;
+    *handled_output = estrndup(output, *handled_output_len = output_len);
 }
```

Figure 11: Developer patch for php-bug-311346-311348