MIT-CSAIL-TR-2015-003 February 10, 2015

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

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard

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 prior generate-and-validate patch generation systems (GenProg, RSRepair, and AE). Because of experimental error, the majority of the reported patches violate the basic principle behind the design of these systems — they do not produce correct outputs *even for the inputs in the test suite used to validate the patches.* We also show that the overwhelming majority of the accepted patches are *not correct* and are *equivalent to a single modification that simply deletes functionality.*

We also present Kali, a generate-and-validate patch generation system that simply 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 plausible 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.

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 then 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 [15], RSRepair [32], and AE [39] systems.

GenProg in particular is visible, widely cited, and used in subsequent research [25, 9, 33]. Example quotes that reflect how GenProg is widely viewed as a system that can successfully fix bugs in large C programs 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" [33] and "successful, well-known automatic repair tools such as GenProg" [9].

The reported results for these systems are impressive: GenProg is reported to fix 55 of 105 considered bugs [15], 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) [32], and AE is reported to fix 54 of the same 105 considered bugs [39]. 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 Patch Evaluation Research Questions

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 public availability of the relevant patch generation and validation infrastructure and the generated patches [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 *experimental error* in the patch validation, 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 55 defects that GenProg is reported to repair, *none* of the reported patches produce correct outputs for the inputs in the test suite [15]. For 14 of the 24 defects that RSRepair is reported to repair, *none* of the reported patches that we analyze produce correct outputs for the inputs in the test suite [32]. And for 27 of the 54 defects reported in the AE result tar file [1], *none* of the reported patches produces correct outputs. See Section 3.

Weak Proxies: Further investigation indicates that *weak proxies* are the source of the experimental 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, for some of the benchmark programs, the patch generation systems do not check if the output is correct. They instead check 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 accepted by these systems *do not generate correct outputs even for the inputs in the validation test suite.* See Section 3.

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

Despite multiple publications that analyze the reported patches and the methods used to generate them [15, 32, 39, 27, 13, 21, 33, 12], we were able to find *no* systematic correctness analysis. We therefore analyzed the plausible patches to determine if they eliminated the defect or not. Despite what one might reasonably expect from reading the relevant papers [15, 32, 39], the overwhelming majority of the patches *are not correct*. Specifically, GenProg produced a correct patch for only 2 of the 105 considered defects.[1] 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. See Section 4.

> **RQ3:** How many of the plausible reported GenProg, RSRepair, and AE patches are equivalent to a single modification that simply deletes functionality?

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 statement.* 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. See Section 5.

1.2 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.

> **RQ4:** How effective is Kali in comparison with existing generate-and-validate patch generation systems?

Our results show that, *on their own sets of benchmark defects*, 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.

> **RQ5:** 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:

- **Enhanced Defect Reports:** Previous research shows that developers are more likely to fix defects if they come with sample patches that show how to eliminate the defect for sample defect-exposing inputs [38]. Kali provides an efficient way to obtain such patches without requiring model-checking and policy specification information as in prior research [38].
- **Defect Location and Characteristics:** An examination of the Kali patches provides insight into why including patches can promote more productive defect reports. 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. See Section 7.

1.3 Implications

Our analysis substantially changed our understanding of the capabilities of the analyzed automatic patch generation systems. It is now clear that GenProg, RSRepair, and AE are overwhelmingly less capable of generating meaningful patches than we initially understood from reading the relevant papers. Our results highlight several aspects:

Room for Improvement: The results highlight that there is significant room for improvement in automatic patch generation systems. They highlight, for example, the prominent role that simply deleting functionality can play in the generation of plausible but not correct patches, especially when the defect occurs in lightly tested functionality. We anticipate that future systems will need to navigate this pitfall if they are to produce correct patches (and avoid patches with significant negative effects such as the introduction of security vulnerabilities, see Section 5).

The success of more targeted patch generation approaches (see Sections 8 and 10.1) highlights the potential that automatic patch generation systems can offer. From this perspective, the challenge is generalizing these more targeted approaches while preserving their benefits and avoiding the demonstrated negative effects of less structured approaches.

Machine vs. Human-Generated Patches: Our results highlight important differences between machine-generated and human-generated patches. For the same defects, the machine-generated patches are overwhelming incorrect comparing to the human-generated patches. This result illustrates that information other than simply passing a validation test suite is (at least with current production test

[1]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 [15].

suites) important for producing correct patches. Future research directions include understanding other potential techniques for boosting the ability of automatic patch generation systems to produce correct patches (and not just plausible patches) with realistic test suites that may not provide extensive coverage of the defective functionality.

The ClearView hot patching system, for example, leverages invariants learned during program executions on benign inputs to produce candidate patches [30]. The success of ClearView in eliminating security vulnerabilities highlights the potential benefits that exploiting additional sources of information (in this case, the learned invariants) can provide [30].

Transparency and Reproducibility: 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. This paper therefore supports a move to greater transparency and availability of reported experimental results and systems.

The PAR system (winner of a best paper award at ICSE 2013) is another prominent generate-and-validate automatic patch generation system [18]. 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.

1.4 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 standard feature elimination, the introduction of integer and buffer overflow vulnerabilities, and new memory leaks.

- **Semantic Patch Analysis:** It reports, for the first time, that the overwhelming majority of the reported GenProg, RSRepair, and AE patches, including all of the correct 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 *on their own benchmark sets*, 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.

An accurate publication record is a baseline requirement for scientific progress. The uncorrected presence of incomplete or inaccurate information in the scientific literature hampers progress, especially when the results are visible, widely cited, and the foundation for additional research [40, 15, 32, 39, 25, 9, 33, 12, 31]. By providing a more complete understanding of the capabilities of prior automatic patch generation systems, this paper can help researchers better identify productive research directions. The presented Kali system is an example of a novel system that was enabled by this more complete understanding of prior research.

2. OVERVIEW OF ANALYZED SYSTEMS

GenProg: GenProg combines three basic modifications, specifically delete, insert, and replace, into larger patches, and 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 [15]. 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 [32]. 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 [32, 15]. 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) [39]. AE focuses on patches that only perform one edit and exhaustively enumerate all such patches. The AE experiments were "designed for direct comparison with previous GenProg results" [39, 15] 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 produce 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 as available at [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 violate the basic principle of generate-and-validate patch generation and 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 considered 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 [15]. 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 uses a *weak proxy* that checks only that the higher order 8 bits of the exit code from the developer test script are 0. This can happen 1) if 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) if 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) if 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.

Correction and Reexecution: To determine whether GenProg would be able to produce plausible patches if these issues were addressed, we modified the php test harness to ensure that the harness correctly runs the test script and correctly reports the results back to GenProg.

Using the modified test scripts and test harness, we reexecuted GenProg on all php defects potentially affected by the test script or test harness modifications. Recall that, by chance, the previous GenProg executions happened to generate some plausible patches. The reexecutions rediscovered these previously discovered patches, but discovered no new plausible patches.

Test Script Issues: The GenProg libtiff test scripts do not check that the test cases produce the correct output. They instead check that the exercised libtiff tools return exit code 0. 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 [15].

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 [32]. 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.

4. CORRECTNESS ANALYSIS

We next discuss our analysis of the following research question:

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

We analyze the correctness of each patch by examining the patch and determining if it correctly repairs the defect.

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.

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 some cases, however, 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 and 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.

5. SEMANTIC PATCH ANALYSIS

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

> **RQ3:** 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 [39].

5.1 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 a feature that is otherwise unexercised. The patch simply deletes functionality that the test case exercises. This deletion then impairs or even completely removes the feature.

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.

5.2 Impact of Functionality Deletion Patches

Our analysis of the patches also indicated that (in contrast to previously reported results [20]) 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 crippling of standard features:

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 features. 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) [35, 19, 22, 11, 30]. 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.

6. REVISITING PREVIOUS 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?" [39, 15]. 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 are operating 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 [27]. He notes that the GenProg research (as well as the PAR 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)" [27]. 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" [13]. In retrospect, this potentially surprising result may become more plausible when one considers that the machine-generated 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 [13]. 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 investigation of patch understandability that does not expose the obvious incorrectness of the patches.
Time and Effort: The AE, RSRepair, and GenProg research projects devoted significant time and effort to evaluating variants of the basic GenProg patch generation mechanisms [15, 12, 21, 32, 39]. 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 under-

stand (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" [33] 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" [23]. We believe that this understanding should be revisited in light of the results presented in this paper.

7. 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 type, the inserted return statement returns NULL. If the function returns an integer type, Kali will generate two patches returning zero or -1.
- **Remove Statement:** Remove the existing statement. If the statement is a compound statement, Kali will remove all sub-statements 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:

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

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:

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

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.

7.1 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 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.

7.2 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 [32]. A "‡" indicates that all analyzed patches are equivalent to a single functionality elimination modification

Defect	GenProg	RSPepair	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

(specifically removing a single statement or block of statements or inserting a single return statement). We omit the "‡" for Kali patches — they are all functionality elimination patches.

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). All of the Kali patches are available at [34].

Search Space and Time Results: The sixth column of Figure 1 presents the size of the search space. The numbers show that the Kali search space is in the hundreds to small thousands of patches. The seventh column presents the search times. The numbers show that 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. 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 the fbc experiments.

It is not possible to directly compare the reported performance numbers for GenProg, RSRepair, and AE [15, 32, 39] 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 ([32], 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 ([39], 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. A "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.

An "SL" indicates that the Kali and corresponding developer patches modify the same line of code. A "*" indicates that the developer patch modified only the function that the Kali patch modified. A "†" 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.

An "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 (lighttpd-bug-2330-2331) functionality.

A "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.

7.3 Discussion

Examining the Kali patches, it is clear why providing a plausible patch along with a defect report can help developers produce a correct patch [38]. 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. In this way the Kali patches can provide more useful information than standard fault localization techniques, which use statistics to rank order likely defective statements (but provide no insight into the nature of the defect or why the statements may be defective).

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 can operate 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.

8. 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 [30]. 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 [30]).

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 [30]. 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 [34].

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. 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.

9. 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 [15]. 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 [15]. Moreover, the benchmark set was specifically constructed to "help address generalizability concerns" [15]. 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. More coverage might also enable the other patch generation systems to produce more meaningful 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).

10. RELATED WORK

Debroy and Wong: Debroy and Wong propose a generate-and-validate approach using two kinds of modifications: replacement of an operator with another from the same class and condition negation [10]. 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.

Principled PHP Repair: PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair php programs that generate HTML [36]. 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.

Specification-Based Systems: Specifications, when available, can enable patch generation systems to produce patches that are guaranteed to be correct. Researchers have developed patch generation systems that leverage available specifications 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) [37].

CodeHint: In response to partial specifications of desired software behavior, CodeHint automatically synthesizes code fragments with that desired behavior [14]. CodeHint uses a generate-and-validate approaches and works with a probabilistic model that guides the search towards likely expressions that work in the context at hand.

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 [28, 29]. 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) [33]. Kali, in contrast, uses automated program repair to, in part, generate patches that provide useful information about defects.

Patches With Bug Reports: There is experimental evidence that bug reports that come with plausible (but not necessarily correct) patches are more likely to be fixed, presumably because the patches help developers better understand and fix the error [38]. The proposed method uses model-checking and policy specification information to construct an example patch. Working only with a test suite (and without model-checking information and policy specifications), Kali can deliver similarly useful patches.

Human Patch Acceptability: A study comparing the acceptability of GenProg, PAR, and human-generated patches for Java programs found that the patches that PAR generated patches were more acceptable to human developers than GenProg patches [18] (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 [15]. 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 [21] and for patches generated by human developers (but with applications to automatic patch generation) [24]. 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).

10.1 Targeted Repair Systems

Researchers have developed a variety of repair systems that are targeted at specific classes of errors.

Failure-Oblivous Computing: Failure-oblivious computing [35] 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 five errors in five server applications. The goal was to enable servers to survive inputs that trigger the errors and continue on to successfully process other inputs. For all five systems, the implemented system realized this goal. For two of the five errors, failure-oblivious computing completely eliminates the error and, on all inputs, delivers the same output as the official developer patch that corrects the error (we believe these patches are correct).

Bolt: Bolt [19] 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 official developer patch that corrects the error (we believe these patches are correct).

RCV: RCV [22] 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 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 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 official developer patch that corrects the error (we believe these patches are correct).

APPEND: APPEND [11] 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.

10.2 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. The rationale is that the target 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 official patches. They are also quite simple — each repair typically consists of a check for a condition (such as an out of bounds access) followed by a simple action (such as discarding the write or returning zero). Although the GenProg, RSRepair, and AE can, in principle, generate much larger and more complex repairs, in practice the overwhelming majority of the repairs that they do generate are semantically very simple.

In comparison with Kali, GenProg, RSRepair, and AE, the more structured repair modifications that these targeted approaches (as well as ClearView) deploy run substantially less risk of introducing security vulnerabilities or removing critical functionality. Other examples of more structured systems (that should also have less risk of introducing serious latent defects) include PAR [18], which uses a set of human-learned patterns to generate a more structured search space; Afix, which deploys a set of specialized transformations to eliminate single-variable atomicity defects [16]; and QuickStep, which uses a variety of accuracy-enhancing transformations such as synchronization introduction (which eliminates atomicity violations) and privatization (which gives each thread its own copy of shared variables) [26, 17].

11. CONCLUSION

An accurate publication record is essential to scientific progress. By providing a more complete picture of the capabilities of prior automatic patch generation systems, this paper can help researchers better understand the implications of these systems and more easily identify productive research directions. The presented Kali system provides an example of research inspired by this more complete understanding. Kali shows how a simple automatic patch generation mechanism based only on functionality deletion can, at least on the analyzed set of benchmark defects, outperform prior more complex patch generation systems.

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.

12. 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] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, China, November 2014.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. *ACM SIGPLAN Notices*, 46(6):389–400, 2011.

[17] D. Kim, S. Misailovic, and M. Rinard. Automatic parallelization with statistical accuracy bounds. 2010.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *36th International Conference on Software Engineering, ICSE '14, Companion*

Proceedings, Hyderabad, India, May 31 - June 07, 2014, pages 492–495, 2014.

[26] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):88, 2013.

[27] 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.

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

[29] 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.

[30] 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.

[31] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *ICSM*, pages 180–189. IEEE, 2013.

[32] 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.

[33] 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.

[34] 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`.

[35] 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.

[36] 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.

[37] 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.

[38] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, pages 181–190, 2006.

[39] 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.

[40] 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.