A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each

Claire Le Goues Michael Dewey-Vogt
Computer Science Department
University of Virginia
Charlottesville, VA
Email: legoues,mkd5m@cs.virginia.edu

Stephanie Forrest
Computer Science Department
University of New Mexico
Albuquerque, NM
Email: forrest@cs.unm.edu

Westley Weimer
Computer Science Department
University of Virginia
Charlottesville, VA
Email: weimer@cs.virginia.edu

***Abstract*—There are more bugs in real-world programs than human programmers can realistically address. This paper evaluates two research questions: "What fraction of bugs can be repaired automatically?" and "How much does it cost to repair a bug automatically?" In previous work, we presented *GenProg*, which uses genetic programming to repair defects in off-the-shelf C programs. To answer these questions, we: (1) propose novel algorithmic improvements to GenProg that allow it to scale to large programs and find repairs 68% more often, (2) exploit GenProg's inherent parallelism using cloud computing resources to provide grounded, human-competitive cost measurements, and (3) generate a large, indicative benchmark set to use for systematic evaluations. We evaluate GenProg on 105 defects from 8 open-source programs totaling 5.1 million lines of code and involving 10,193 test cases. GenProg automatically repairs 55 of those 105 defects. To our knowledge, this evaluation is the largest available of its kind, and is often two orders of magnitude larger than previous work in terms of code or test suite size or defect count. Public cloud computing prices allow our 105 runs to be reproduced for $403; a successful repair completes in 96 minutes and costs $7.32, on average.**

***Keywords*-genetic programming; automated program repair; cloud computing**

I. Introduction

Program evolution and repair are major components of software maintenance, which consumes a daunting fraction of the total cost of software production [1]. Automated techniques to reduce their costs are therefore especially beneficial. Developers for large software projects must confirm, triage, and localize defects before fixing them and validating the fixes. Although there are a number of tools available to help with triage (e.g., [2]), localization (e.g., [3], [4]), validation (e.g., [5]) and even confirmation (e.g., [6]), generating repairs remains a predominantly manual, and thus expensive, process. At the same time, *cloud computing*, in which virtualized processing power is purchased cheaply and on-demand, is becoming commonplace [7].

Research in *automated program repair* has focused on reducing defect repair costs by producing candidate patches for validation and deployment. Recent repair projects include ClearView [8], which dynamically enforces invariants to patch overflow and illegal control-flow transfer vulnerabilities; AutoFix-E [9], which can repair programs annotated with design-by-contract pre- and post-conditions; and AFix [10], which can repair single-variable atomicity violations. In previous work, we introduced *GenProg* [11], [12], [13], [14], a general method that uses genetic programming (GP) to repair a wide range of defect types in legacy software (e.g., infinite loops, buffer overruns, segfaults, integer overflows, incorrect output, format string attacks) without requiring *a priori* knowledge, specialization, or specifications. GenProg searches for a repair that retains required functionality by constructing variant programs through computational analogs of biological processes.

The goal of this paper is to evaluate dual research questions: "What fraction of bugs can GenProg repair?" and "How much does it cost to repair a bug with GenProg?" We combine three important insights to answer these questions. Our key algorithmic insight is to represent candidate repairs as patches [15], rather than as abstract syntax trees. These changes were critical to GenProg's scalability to millions of lines of code, an essential component of our evaluation. We introduce new search operators that dovetail with this representation to reduce the number of ill-formed variants and improve performance. Our key performance insight is to use off-the-shelf cloud computing as a framework for exploiting search-space parallelism as well as a source of grounded cost measurements. Our key experimental insight is to search version control histories exhaustively, focusing on open-source C programs, to identify revisions that correspond to human bug fixes as defined by the program's most current test suite.

We combine these insights and present a novel, scalable approach to automated program repair based on GP, and then evaluate it on 105 real-world defects taken from open-source projects totaling 5.1 MLOC and including 10,193 test cases.

The main contributions of this paper are:

- GenProg, a scalable approach to automated program repair based on GP. New GP representation, mutation, and crossover operators allow GenProg to scale to large programs and take advantage of cloud computing

parallelism. We evaluate directly against our previous approach on its own benchmarks [11] and find that the improved algorithm finds repairs 68% more often.

- A systematic evaluation of GenProg on 105 defects from 5.1 MLOC of open-source projects equipped with 10,193 test cases. We generate a benchmark set by exhaustively searching inclusive version ranges to help address generalizability concerns: All reproducible bugs in a time window are considered, and all defects considered were important enough for developers to test for and fix manually. This evaluation includes two orders of magnitude more source code than AutoFix-E [9] or our previous work [11], two orders of magnitude more test cases than ClearView [8], and two orders of magnitude more defects than AFix [10] (in addition to being strictly larger than each of those four previous projects on each of the those metrics separately).
- Results showing that GenProg repairs 55 of those 105 defects. Because our experiments were conducted using cloud computing and virtualization, any organization could pay the same rates we did and reproduce our results for $403 — or $7.32 per successful run on this dataset. A successful repair takes 96 minutes of wall-clock time, on average, while an unsuccessful run takes 11.2 hours, including cloud instance start up times.

II. Motivation

This section motivates automated program repair and identifies monetary cost, success rate and turnaround time as important evaluation metrics.

The rate at which software bugs are reported has kept pace with the rapid rate of modern software development. In 2006, one Mozilla developer noted, "everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle" [2, p. 363]; bugzilla.mozilla.org gives similar bug report numbers for 2011. Since there are not enough developer resources to fix all defects, programs ship with both known and unknown bugs [6].

In light of this problem, many companies have begun offering *bug bounties* to outside developers, paying for candidate repairs. Well-known companies such as Mozilla[1] and Google[2] offer significant rewards for security fixes, with bounties raising to thousands of dollars in "bidding wars."[3]

Although security bugs command the highest prices, more wide-ranging bounties are available. Consider Tarsnap.com,[4] an online backup provider. Over a four-month period, Tarsnap paid $1,625 for fixes for issues ranging from cosmetic errors (e.g., typos in source code comments), to general software engineering mistakes (e.g., data corruption), to security vulnerabilities. Of the approximately 200 candidate patches submitted to claim various bounties, about 125 addressed spelling mistakes or style concerns, while about 75 addressed more serious issues, classified as "harmless" (63) or "minor" (11). One issue was classified as "major." Developers at Tarsnap confirmed corrections by manually evaluating all submitted patches. If we treat the 75 non-trivial repairs as true positives (38%) and the 125 trivial reports as overhead, Tarsnap paid an average of $21 for each non-trivial repair and received one about every 40 hours. Despite the facts that the bounty pays a small amount even for reports that do not result in a usable patch and that about 84% of all non-trivial submissions fixed "harmless" bugs, the final analysis was: "Worth the money? Every penny."[5]

Bug bounties suggest that the need for repairs is so pressing that companies are willing to pay for outsourced candidate patches even though repairs must be manually reviewed, most are rejected, and most accepted repairs are for low-priority bugs. These examples also suggest that relevant success metrics for a repair scheme include the fraction of queries that produce code patches, monetary cost, and wall-clock time cost. We now present an automated approach to program repair with a use case similar to that of the outsourced "bug bounty hunters." The method is powerful enough to fix over half of the defects it tackles, and we evaluate it using these and other metrics.

Input: Full fitness predicate $\mathsf{FullFitness} : Patch \to \mathbb{B}$
Input: Sampled fitness $\mathsf{SampleFit} : Patch \to \mathbb{R}$
Input: Mutation operator $\mathsf{mutate} : Patch \to Patch$
Input: Crossover operator $\mathsf{crossover} : Patch^2 \to Patch^2$
Input: Parameter $PopSize$
Output: Patch that passes FullFitness

1: **let** $Pop \leftarrow$ map mutate over $PopSize$ copies of $\langle\ \rangle$
2: **repeat**
3: **let** $parents \leftarrow \mathsf{tournSelect}(Pop, Popsize, \mathsf{SampleFit})$
4: **let** $offspr \leftarrow$ map crossover over $parents$, pairwise
5: $Pop \leftarrow$ map mutate over $parents \cup offspr$
6: **until** $\exists\ candidate \in Pop.\ \mathsf{FullFitness}(candidate)$
7: **return** $candidate$

Figure 1. High-level pseudocode for the main loop of our technique.

III. Automated Repair Method

In this section we describe *GenProg*, an automated program repair method that searches for repairs to off-the-shelf programs. We highlight the important algorithmic and representational changes since our preliminary work [11] that enable scalability to millions of lines of code, improve performance, and facilitate implementation on a cloud computing service.

A. Genetic Programming

GenProg uses *genetic programming (GP)* [16], an iterated stochastic search technique, to search for program

[1] http://www.mozilla.org/security/bug-bounty.html $3,000/bug

[2] http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html $500/bug

[3] http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws

[4] http://www.tarsnap.com/bugbounty.html

[5] http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html

repairs. The search space of possible repairs is infinitely large, and GenProg employs five strategies to render the search tractable: (1) coarse-grained, statement-level patches to reduce search space size; (2) fault localization to focus edit locations; (3) existing code to provide the seed of new repairs; (4) fitness approximation to reduce required test suite evaluations; and (5) parallelism to obtain results faster.

High-level pseudocode for GenProg's main GP loop is shown in Figure 1; it closely resembles previous work [11]. Fitness is measured as a weighted average of the positive (i.e., initially passing, encoding required functionality) and negative (i.e., initially failing, encoding a defect) test cases. The goal is to produce a candidate patch that causes the original program to pass all test cases. In this paper, each individual, or variant, is represented as a repair patch [15], stored as a sequence of AST edit operations parameterized by node numbers (e.g.,$\mathsf{Replace}(81, 44)\rangle$; see Section III-B).

Given a program and a test suite (i.e., positive and negative test cases), we localize the fault (Section III-D) and compute context-sensitive information to guide the search for repairs (Section III-E) based on program structure and test case coverage. The functions SampleFit and FullFitness evaluate variant fitness (Section III-C) by applying candidate patches to the original program to produce a modified program that is evaluated on test cases. The operators mutate and crossover are defined in Section III-F and Section III-G. Both generate new patches to be tested.

The search begins by constructing and evaluating a population of random patches. Line 1 of Figure 1 initializes the population by independently mutating copies of the empty patch. Lines 2–6 correspond to one iteration or *generation* of the algorithm. On Line 3, *tournament selection* [17] selects from the incoming population, with replacement, parent individuals based on fitness. By analogy with genetic "crossover" events, parents are taken pairwise at random to exchange pieces of their representation; two parents produce two offspring (Section III-G). Each parent and each offspring is mutated once (Section III-F) and the result forms the incoming population for the next iteration. The GP loop terminates if a variant passes all test cases, or when resources are exhausted (i.e., too much time or too many generations elapse). We refer to one execution of the algorithm described in Figure 1 as a *trial*. Multiple trials are run in parallel, each initialized with a distinct random seed.

The rest of this section describes additional algorithmic details, with emphasis on the important improvements on our preliminary work, including: (1) a new patch-based representation (2) large-scale use of a sampling fitness function at the individual variant level, (3) fix localization to augment fault localization, (4) and novel mutation and crossover operators to dovetail with the patch representation.

B. Patch Representation

An important GenProg enhancement involves the choice of representation. Each variant is a *patch*, represented as sequence of edit operations (cf. [15]). In the original algorithm, each individual was represented by its entire abstract syntax tree (AST) combined with a weighted execution path [11], which does not scale to large programs in the cloud computing setting. For example, for at least 36 of the 105 defects considered in this paper, a population of 40–80 ASTs did not fit in the 1.7 GB of main memory allocated to each cloud node. In our dataset, half of all human-produced patches were 25 lines or less. Thus, two unrelated variants might differ by only 2×25 lines, with all other AST nodes in common. Representing individuals as patches avoids storing redundant copies of untouched lines. This formulation influences the mutation and crossover operators, discussed below.

C. Fitness evaluation

To evaluate the fitness of a large space of candidate patches efficiently, we exploit the fact that GP performs well with noisy fitness functions [13]. The function SampleFit applies a candidate patch to the original program and evaluates the result on a random sample of the positive tests as well as all of the negative test cases. SampleFit chooses a different test suite sample each time it is called. FullFitness evaluates to true if the candidate patch, when applied to the original program, passes all of the test cases. For efficiency, only variants that maximize SampleFit are fully tested on the entire test suite. The final fitness of a variant is the weighted sum of the number of tests that are passed, where negative tests are weighted twice as heavily as the positive tests.

D. Fault Localization

GenProg focuses repair efforts on statements that are visited by the negative test cases, biased heavily towards those that are not also visited by positive test cases [3]. For a given program, defect, set of tests T, test evaluation function $Pass : T \to \mathbb{B}$, and set of statements visited when evaluating a test $Visited : T \to \mathcal{P}(Stmt)$, we define the fault localization function $faultloc : Stmt \to \mathbb{R}$ to be:

$$faultloc(s) = \begin{cases} 0 & \forall t \in T.\ s \notin Visited(t) \\ 1.0 & \forall t \in T.\ s \in Visited(t) \implies \neg Pass(t) \\ 0.1 & \text{otherwise} \end{cases}$$

That is, a statement never visited by any test case has zero weight, a statement visited only on a bug-inducing test case has high (1.0) weight, and statements covered by both bug-inducing and normal tests have moderate (0.1) weights (this strategy follows previous work [11, Sec. 3.2]). On the 105 defects considered here, the total weight of possible fault locations averages 110. Other fault localization schemes could potentially be plugged directly into GenProg [6].

E. Fix Localization

We introduce the term *fix localization* (or *fix space*) to refer to the *source* of insertion/replacement code, and explore ways to improve fix localization beyond blind random choice. As a start, we restrict inserted code to that which includes variables that are in-scope at the destination (so the result compiles) and that are visited by at least one test case (because we hypothesize that certain common behavior may be correct). For a given program and defect we define the function $\mathit{fixloc} : \mathit{Stmt} \rightarrow \mathcal{P}(\mathit{Stmt})$ as follows:

$$\mathit{fixloc}(d) = \left\{ s \;\middle|\; \begin{array}{l} \exists t \in T.\ s \in \mathit{Visited}(t) \wedge \\ \mathit{VarsUsed}(s) \subseteq \mathit{InScope}(d) \end{array} \right\}$$

The previous approach chose an AST node randomly from the entire program. As a result, an average of 32% of generated variants did not compile [11]—usually due to type checking or scoping issues. For larger programs with long compilation times, this is a significant overhead. For the 105 defects considered here, less than 10% of the variants failed to compile using the fix localization function just defined.

F. Mutation Operator

Earlier work used three types of mutation: *delete*, *insert*, and *swap*. However, we found swap to be up to an order of magnitude less successful than the other two [12, Tab. 2]. We thus remove swap in favor of a new operator *replace* (equivalent to a delete followed by an insert to the same location). In a single mutation, a destination statement d is chosen from the fault localization space (randomly, by weight). With equiprobability GenProg either deletes d (i.e., replaces it with the empty block), inserts another source statement s before d (chosen randomly from $\mathit{fixloc}(d)$), or replaces d with another statement s (chosen randomly from $\mathit{fixloc}(d)$). As in previous work, inserted code is taken exclusively from elsewhere in the same program. This decision reduces the search space size by leveraging the intuition that programs contain the seeds of their own repairs.

G. Crossover Operator

The *crossover* operator combines partial solutions, helping the search avoid local optima. Our new *patch subset* crossover operator is a variation of the well-known *uniform* crossover operator [18] tailored for the program repair domain. It takes as input two parents p and q represented as ordered lists of edits (Section III-B). The first (resp. second) offspring is created by appending p to q (resp. q to p) and then removing each element with independent probability one-half. This operator has the advantage of allowing parents that both include edits to similar ranges of the program (e.g., parent p inserts B after A and parent q inserts C after A) to pass any of those edits along to their offspring. Previous uses of a one-point crossover operator on the fault localization space did not allow for such recombination (e.g., each offspring could only receive one edit to statement A).

Program	LOC	Tests	Defects	Description
fbc	97,000	773	3	legacy coding
gmp	145,000	146	2	precision math
gzip	491,000	12	5	data compression
libtiff	77,000	78	24	image processing
lighttpd	62,000	295	9	web server
php	1,046,000	8,471	44	web programming
python	407,000	355	11	general coding
wireshark	2,814,000	63	7	packet analyzer
total	**5,139,000**	**10,193**	**105**	

Table I
SUBJECT C PROGRAMS, TEST SUITES AND HISTORICAL DEFECTS: Tests were taken from the most recent version available in May, 2011; Defects are defined as test case failures fixed by developers in previous versions.

IV. EXPERIMENTAL SETUP

This section describes how we selected a set of subject programs and defects for our systematic evaluation, and it describes the parameter settings used for the experiments.

A. Subject Programs and Defects

Our goal was to select an unbiased set of programs and defects that can run in our experimental framework and is indicative of "real-world usage." We required that *subject programs* contain sufficient C source code, a version control system, a test suite of reasonable size, and a set of suitable subject defects. We only used programs that could run without modification under cloud computing virtualization, which limited us to programs amenable to such environments. We required that *subject defects* be reproducible and important. We searched systematically through the program's source history, looking for revisions that caused the program to pass test cases that it failed in a previous revision. Such a scenario corresponds to a human-written repair for the bug corresponding to the failing test case. This approach succeeds even in projects without explicit bug-test links, and it ensures that benchmark bugs are important enough to merit a human fix and to affect the program's test suite.

Table IV-A summarizes the programs used in our experiments. We selected these benchmarks by first defining predicates for acceptability, and then examining various program repositories to identify first, acceptable candidate programs that passed the predicates; and second, all reproducible bugs within those programs identified by searching backwards from the checkout date (late May, 2011). The next subsection formalizes the procedure in more detail.

B. Selecting Programs for Evaluation

A candidate subject program is a software project containing at least 50,000 lines of C code, 10 viable test cases, and 300 versions in a revision control system. We consider all *viable versions* of a program, defined as a version that checks out and builds unmodified on 32-bit Fedora 13 Linux (a lowest common denominator OS available on the EC2 cloud computing framework). A program *builds* if it produces its primary executable, regardless of the exit status of **make**.

We define *test cases* to be the smallest atomic testing units for which individual pass or fail information is available. For example, if a program has 10 "major areas" which each contain 5 "minor tests" and each "minor test" can pass or fail, we say that it has 50 test cases. We define a *viable test case* as a test that is reproducible, non-interactive, and deterministic in the cloud environment (over at least 100 trials). $testsuite(i)$ denotes the set of viable test cases passed by viable version i of a program. We use all available viable tests, even those added *after* the version under consideration. We exclude programs with test suites that take longer than one hour to complete in the cloud environment.

We say that a *testable bug* exists between viable versions i and j of a subject program when:

1) $testsuite(i) \subsetneq testsuite(j)$ and
2) there is no $i' > i$ or $j' < j$ with the $testsuite(j) - testsuite(i) = testsuite(j') - testsuite(i')$ and
3) the only source files changed by developers to reach version j were `.c`, `.h`, `.y` or `.l`

The second condition requires a minimal $|i - j|$. The set of *positive tests* (i.e., encoding required behavior) is defined as $testsuite(i) \cap testsuite(j)$. The *negative tests* (i.e., demonstrating the bug) are $testsuite(j) - testsuite(i)$. Note that the positive and negative tests are disjoint.

Given a viable candidate subject program, its most recent test suite, and a range of viable revisions, we construct a set of testable bugs by considering each viable version i and finding the minimal viable version j, if any, such that there is a testable bug between i and j. We considered all viable revisions appearing before our start date in late May, 2011 as a potential source of testable bugs. However, we capped each subject program at 45 defects to prevent any one program from dominating the results.

Given these criteria, we canvassed the following sources:

1) the top 20 C-foundry programs on Sourceforge.net
2) the top 20 C programs on Google code
3) the largest 20 non-kernel Fedora 13 source packages
4) programs in other repair papers [13], [14] or known to the authors to have large test suites

Many otherwise-popular projects failed to meet our criteria. Many open-source programs have nonexistent or weak test suites; opaque testing paradigms; non-automated GUI testing; or are difficult to modularize, build and reproduce on our architecture (e.g., `eclipse`, `firefox`, `ghostscript`, `handbrake`, `openjpeg`, `openoffice`). For several programs, we were unable to identify any viable defects according to our definition (e.g., `gnucash`, `openssl`). Some projects (e.g., `bash`, `cvs`, `openssh`) have inaccessible or unusably small version control histories. Other projects were ruled out by our test suite time bound (e.g., `gcc`, `glibc`, `subversion`). Some projects have many revisions but few viable versions that compile and run against recent test cases (e.g., `valgrind`). Earlier versions of certain programs (e.g,. `gmp`) require incompatible versions of `automake` and `libtool`.

The set of benchmark programs and defects appears in Table IV-A. The authors acknowledge that it is not complete and that other additions are possible. While it is certainly "best effort," to our knowledge it also represents the most systematic evaluation of automated program repair to date.

C. Experimental Parameters

We ran 10 GenProg *trials* in parallel for each bug. We chose $PopSize = 40$ and a maximum of 10 generations for consistency with previous work [11, Sec. 4.1]. Each individual was mutated exactly once each generation, crossover is performed once on each set of parents, and 50% of the population is retained (with mutation) on each generation (known as elitism). Each trial was terminated after 10 generations, 12 hours, or when another search found a repair, whichever came first. SampleFit returns 10% of the test suite for all benchmarks.

We used Amazon's EC2 cloud computing infrastructure for the experiments. Each trial was given a "high-cpu medium (c1.medium) instance" with two cores and 1.7 GB of memory.[6] Simplifying a few details, the virtualization can be purchased as *spot instances* at $0.074 per hour but with a one hour start time lag, or as *on-demand instances* at $0.184 per hour. These August–September 2011 prices summarize CPU, storage and I/O charges.[7]

V. Experimental Results

This section reports and analyzes the results of running GenProg on our benchmark suite of defects. We address the following questions:

- How many defects can GenProg repair, and at what cost? (Section V-A)
- What determines the success rate? (Section V-B)
- What is the impact of alternative repair strategies? (Section V-C)
- How do automated and human-written repairs compare? (Section V-D)

A. How many defects can GenProg repair?

Table II reports results for 105 defects in 5.1 MLOC from 8 subject programs. GenProg successfully repaired 55 of the defects (52%), including at least one defect for each subject program. The 50 "Non-Repairs" met time or generation limits before a repair was discovered. We report costs in terms of monetary cost and wall clock time from the start of the request to the final result, recalling that the process terminates as soon as one parallel search finds a repair. Results are reported for cloud computing spot instances, and thus include a one-hour start lag but lower CPU-hour costs.

For example, consider the repaired `fbc` defect, where one of the ten parallel searches found a repair after 6.52

[6] http://aws.amazon.com/ec2/instance-types/
[7] http://aws.amazon.com/ec2/pricing/

Program	Defects Repaired	Cost per Non-Repair Hours	Cost per Non-Repair US$	Cost Per Repair Hours	Cost Per Repair US$
fbc	1 / 3	8.52	5.56	6.52	4.08
gmp	1 / 2	9.93	6.61	1.60	0.44
gzip	1 / 5	5.11	3.04	1.41	0.30
libtiff	17 / 24	7.81	5.04	1.05	0.04
lighttpd	5 / 9	10.79	7.25	1.34	0.25
php	28 / 44	13.00	8.80	1.84	0.62
python	1 / 11	13.00	8.80	1.22	0.16
wireshark	1 / 7	13.00	8.80	1.23	0.17
total	**55 / 105**	**11.22h**		**1.60h**	

Table II
REPAIR RESULTS: 55 of the 105 defects (52%) were repaired successfully and are reported under the "Cost per Repair" columns. The remaining 50 are reported under the "Non-Repair"s columns. "Hours" columns report the wall-clock time between the submission of the repair request and the response, including cloud-computing spot instance delays. "US$" columns reports the total cost of cloud-computing CPU time and I/O. The total cost of generating the results in this table was $403.

wall-clock hours. This corresponds to 5.52 hours of cloud computing CPU time per instance. The total cost for the entire bug repair effort for that to repair that defect is thus $10 \times 5.52 \text{ hours} \times \$0.074/\text{hour} = \$4.08$ (see Section IV-C).

The 55 successful repairs return a result in 1.6 hours each, on average. The 50 unsuccessful repairs required 11.22 hours each, on average. Unsuccessful repairs that reach the generation limit (as in the first five benchmarks) take less than 12+1 hours. The total cost for all 105 attempted repairs is $403, or $7.32 per successful run. These costs could be traded off in various ways. For example, an organization that valued speed over monetary cost could use on-demand cloud instances, reducing the average time per repair by 60 minutes to 36 minutes, but increasing the average cost per successful run from $7.32 to $18.30.

Table II does not include time to minimize a repair, an optional, deterministic post-processing step. This step is a small fraction of the overall cost [11].

We view the successful repair of 55 of 105 defects from programs totaling 5.1 million lines of code as a very strong result for the power of automated program repair. Similarly, we view an average per-repair monetary cost of $7.32 as a strong efficiency result.

B. What determines the success rate?

This section explores factors that may correlate with GenProg's success in repairing a given defect. We first quantitatively analyze the algorithmic changes we made to GenProg's program representation and genetic operators. We next investigate the relationship between GenProg success and defect complexity using several external metrics, including developer reported defect severity and the number of files touched by developers in a repair. We also consider internal metrics such as localization size.

1) Representation and Genetic Operators: We compare our new representation and operators to the previous approach using the benchmarks from [11], first to allow for a direct comparison, and second because the previous approach does not scale to our new benchmarks. We held population size, number of generations, mutation rate and fault localization strategy constant, changing only the internal representation and genetic operators. We ran 100 random repair trials per benchmark. Success rate is the number of trials that find a repair (as in [11, Fig. 5]).

Program	Fault	LOC	Ratio of Repairs Found
gcd	infinite loop	22	1.07
uniqw-utx	segfault	1146	1.01
look-utx	segfault	1169	1.00
look-svr	infinite loop	1363	1.00
units-svr	segfault	1504	3.13
deroff-utx	segfault	2236	1.22
nullhttpd	buffer exploit	5575	1.95
indent	infinite loop	9906	1.70
flex	segfault	18775	3.75
atris	buffer exploit	21553	0.97
average		**6325**	**1.68**

Table III
NEW ALGORITHM: The final column reports the ratio of successful repairs found by our enhanced algorithm to those found by the originally published algorithm on that work's benchmarks [11] (higher is better).

Table III shows results. The new representation outperformed the old on all benchmarks except **atris**, where success drops slightly, and **look**, where both approaches succeed on all trials. Averaged over these benchmarks, the new representation allows GenProg to find repairs 68% more frequently than the original method. This result is consistent with our hypothesis that the new representation would enable a more efficient search for solutions.

2) Correlating Repair with External Metrics: One concern is that GenProg might succeed only on "unimportant" or "trivial" bugs. We investigated this hypothesis by analyzing the relationship between repair success and external metrics such as human time to repair, human repair size, and defect severity. With one exception, we were unable to identify significant correlations with these external metrics.

We manually inspected version control logs, bug databases, and associated history to link defects with bug reports. Although all of our benchmarks are associated with source control and bug-tracking databases, not all defect-associated revisions could be linked with a readily available bug report [19]. We identified publicly accessible bug or security vulnerability reports in 52 out of 105 of our cases. All bug reports linked to a defect in our benchmark set were eventually marked "confirmed" by developers. We measure developer time as the difference between when the bug report was marked "assigned" and when it was "closed", which we know is a rough approximation. We extracted developer-reported defect severities on a 1–5 scale. We assigned **php** security bug reports marked "private" a severity of 4.5. Ultimately, we identified severity information for 28 of the 105 defects. Results on this subset are comparable to those on the full dataset: GenProg repaired 26 of the 52

defects associated with bug reports (50%) and 13 of the 28 (46%) associated with severity ratings.

We investigated both linear and non-linear relationships between repair success and search time and the external metrics. Correlation values are Pearson's unless otherwise noted. We found a significant correlation in only one case. The number of files touched by a human-generated patch is slightly negatively correlated with GenProg success ($r = -0.29$, $p = 0.007$): The more files the humans changed to address the defect, the less likely GenProg was to find a repair (although we note that the correlation is not very strong). We were unable to identify a significant relationship between either "human time to repair" or "human patch size (in `diff` lines)" and GenProg's repair success.

We found no significant correlation between "bug report severity" and "GenProg's ability to repair." Exploring further, we found no significant difference between the mean severity of repaired and unrepaired defects (Student T test and Wilcoxon Rank-Sum test) at $\alpha = 0.95$. These results suggest that the defects that GenProg can and those that it cannot repair are unlikely to differ in human-provided severity. We note that no defect associated with a severity report has lower than "Normal" priority (3 in our scheme). Recall that, by definition, our dataset restricts attention to bugs important enough for developers to fix (see Section IV-A).

3) Correlating Repair with Internal Metrics: We define the space of possible program repairs by both the fault (Section III-D) and fix (Section III-E) space. Previous work reported that the time to repair scaled roughly linearly with the size of the weighted path, or fault localization size [12, Fig. 3]. Fix space size has not been previously studied.

We find a statistically significant, though not very strong, relationship between the log of the fault weight and repair success ($r = -0.36$, $p = 0.0008$) as well as the log of the number of fitness evaluations to repair ($r = 0.28$, $p = 0.01$). As fault space size increases, the probability of repair success decreases, and the number of variants evaluated to a repair increases. This result corroborates our previous findings. We additionally find a significant negative correlation between the log of the fix space size and the log of the number of fitness evaluations required to find a repair ($r = -0.42$, $p < 0.0001$). One possible explanation for these results is that while bad fault localization can preclude a repair (e.g., the variable `x` must be zeroed just before *this* function call), imprecise fix localization may make it difficult but still possible (e.g., there are many ways to set `x` to `0` *without* using "`x=0;`"). A larger fix space may include more candidate repair options, reducing the time to find any one, even if it does not appear to correlate with actual success.

C. What is the impact of alternative repair strategies?

In this subsection we evaluate two alternative repair strategies: searching for multiple repairs and using annotations.

Program	Defects Repaired	Unique Patches	Patches Per Repair	Repaired w/ Annotat.
`fbc`	1 / 3	1	1.0	2 / 3
`gmp`	1 / 2	2	2.0	2 / 2
`gzip`	1 / 5	8	8.0	4 / 5
`libtiff`	17 / 24	115	6.8	19 / 24
`lighttpd`	5 / 9	23	4.6	6 / 9
`php`	28 / 44	157	5.6	33 / 44
`python`	1 / 11	5	5.0	2 / 11
`wireshark`	1 / 7	7	7.0	3 / 7
total	**55 / 105**	**318**	**5.8**	**71 / 105**

Table IV
ALTERNATE DEFECT REPAIR RESULTS. "Unique Patches" counts the number of distinct post-minimization patches produced if each of the 10 parallel searches is allowed to run to completion. The final column reports that 30% more defects can be repaired via our technique if human localization annotations are provided.

1) Search for multiple repairs: Diverse solutions to the same problem may provide multiple options to developers, or enable consideration of multiple attack surfaces in a security context. To investigate GenProg's utility in generating multiple repairs, we allowed each of the ten independent trials per bug to run to completion instead of terminating early when any trial found a repair. To identify unique patches, we convert each repair into a tree-structured expression-level edit script using the DiffX algorithm [20] and minimize the edit script using delta debugging [21] (effectively removing unnecessary edits). We consider a repair unique if the result of using this patch is textually unique.

Table IV shows how many different patches were discovered in this use-case. GenProg produced 318 unique patches for 55 repairs, or an average of 5.8 distinct patches per repaired bug. The unique patches are typically similar, often involving different formulations of guards for inserted blocks or different computations of required values. Because all trials, including successful ones, must now run to completion, the total cost increases from $403 to $502 for all 550 runs.

2) Include human annotations: GenProg is fully automated. However, we might instead use programmer annotations to guide a repair search, similar in spirit to programming by sketching [22]. In sketching, a programmer specifies high-level implementation strategies—a "sketch" of general structure, as well as details such as likely relevant variables, invariants, or function calls—but leaves low-level details to a program synthesizer. The synthesizer uses these inputs to generate the complete code.

In these experiments, we relax our assumption of full automation, and assume that humans provide an unordered superset of statements that may be used to construct a patch (i.e., fix localization information) and pinpoint critical areas where patch actions mights be applied (i.e., fault localization). Such annotations are easier to provide than a concrete patch [22], but are not automatic. We are interested in annotations to explore the upper limits of our fully automated method and to explore what a hybrid human-

machine approach might achieve. We use the actual human repairs for our defect set as the source of our annotations. We say that a defect can be *repaired with annotations* if (1) it can be repaired automatically, or (2) it can be repaired with fault and fix information restricted to those lines and changes made by the human developers.

The final column of Table IV shows results. With annotations, the statement-level repair method can address 71 out of 105 bugs (68%). Annotations also reduce time to first repair by 50% on this dataset (data not shown). This is consistent with the relationship between search space size and repair success (Section V-B3) and suggests that benefits might be gained from improved localization.

These results also illuminate our decision to use only statement-level changes. Human developers used at least one "extra-statement level" change (e.g., introducing a new global variable) in 33 of the 105 subject defects. However, the unannotated statement-level approach can repair 11 of those defects. For example, we observed that humans often introduce new variables to hold intermediate computations or to refactor buggy code while repairing it. GenProg achieves the same effect by reusing existing variable definitions to hold intermediate results. The statement-level technique is less likely to repair such defects, addressing only 33% of them (vs. 52% overall repair rate). Statistically, whether a human repair restricts attention to statement-only changes moderately correlates with whether our technique can repair that same bug: $r = 0.38$, $p < 0.0001$.

Restricting attention to statements reduces the search space by one to two orders of magnitude. These results suggest that is a good trade-off. However, they also suggest that more powerful or finer-grained operators might allow GenProg to address many other real-world defects.

D. How do automated and human-written repairs compare?

In this subsection, we compare the repairs produced by humans with those produced by GenProg for two indicative defects. We have not inspected all 318 unique repairs manually; a user study of patch quality is left as future work.

1) Python Date Handling: In one bug, six `python` tests failed based on whether the date "70" maps to "1970" or "70". The human patch removed a global dictionary, 17 lines of processing using that dictionary, and a flag preserving that dictionary during `y2k` checking. The automated repair removes the 17 lines of special processing but leaves untouched the empty dictionary and unused flag. This retains required functionality but increases run time memory usage by one empty dictionary. The patch is thus as functionally correct as the human patch but degrades some non-functional aspects (maintainability and memory footprint), neither of which are tested.

This "normal" priority issue[8] was open for 7 days and involved 12 developer messages and two different candidate patches submitted for review by human developers.

2) Php Global Object Accessor Crash: `php` uses reference counting to determine when dynamic objects should be freed. `php` also allows user programs to overload internal accessor functions to specify behavior when undefined class fields are accessed. Version 5.2.17 had a bug related to a combination of those features. At a high level, the "read property" function, which handles accessors, always calls a deep reference count decrement on one of its arguments, potentially freeing both that reference and the memory it points to. This is the correct behavior *unless* that argument points to `$this` when `$this` references a global variable—a situation that arises if the user program overrides the internal accessor to return `$this`. In such circumstances, the global variable has its reference count decremented to zero and its memory is mistakenly freed while it is still reachable, causing the interpreter to incorrectly return an error later.

The human-written patch replaces the single line that always calls the deep decrement with a simple if-then-else: in the normal case (i.e., the argument is not a class object), call the deep decrement on it as before, otherwise call a separate shallow decrement function (`Z_DELREF_P`) on it. The shallow decrement function may free that particular pointer, but not the object to which it points.

The GenProg patch adapts code from a nearby "unset property" function. The deep decrement is unchanged, but additional code is inserted to check for the abnormal case. In the abnormal case, the reference count is deeply incremented (through machinations involving a new variable) and then the same shallow decrement (`Z_DELREF_P`) is called.

Thus, at a very high level, the human patch changes `deep_Decr()` to:

```
1  if (normal) deep_Decr(); else shallow_Decr();
```

while the GP-generated patch changes it to:

```
1  deep_Decr();
2  if (abnormal) { deep_Incr(); shallow_Decr(); }
```

The logical effect is the same but the command ordering is not. Both patches increase the file size by four lines. The human patch is perhaps more natural: it avoids the deep decrement rather than performing it and then undoing it.

E. Summary

GenProg repaired 55 of 105 defects from subject programs spanning 5.1 MLOC and 10,193 tests. In a commercial cloud computing setting, GenProg repaired these bugs in 1.6 hours for $7.32 each, on average. This includes a 1-hour start time; paying more for on-demand instances reduces trial time, but increases cost. All defects in this study were at least moderately severe and were important enough for developers to fix. We were unable to identify a significant relationship between human-reported severity (or human time to repair, etc.) and repair success. However, GenProg was less successful at

[8] http://bugs.python.org/issue11930

repairing defects in which humans touched a large number of files or for which the fault could not be precisely localized. Qualitative comparison suggests that GenProg's repairs are often functionally equivalent to the human patches, but deemphasize untested non-functional requirements such as memory usage, readability or maintainability.

Our extensive use of parallelism is novel compared to previous work (cf. [11, Sec 3.4]) and yields an average return time of 96 minutes per successful repair. However, if we continue to search beyond the first repair, GenProg finds 5.8 unique patches per successful repair, which provides developers more freedom and information. When augmented with sketching-inspired annotations [22], GenProg repairs 71 of the 105 defects. The remaining 34 presumably require algorithmic or localization improvements.

Although we report reproducible monetary costs for fixing a defect once a test case is available, it is difficult to directly compare our costs to those for human-generated repairs. Our programs do not report per-issue effort tracking. As an indirect time comparison, Weiß *et al.* [23] survey 567 effort-tracked issues in `jboss` (an open-source Java project of comparable scale to our subject programs). Their mean time taken per issue was 15.3 hours with a median of 5.0 hours. As an indirect cost comparison, the Tarsnap.com bug bounty averaged $21 for each non-trivial repair (Section II). Similarly, an IBM report gives an average defect cost of $25 during coding (rising to $100 at build time, $450 during testing/QA, and $16,000 post-release) [24, p.2]. In personal communication, Robert O'Callahan of Novell, the lead engineer for the Mozilla Gecko layout engine, noted that our costs would be "incredibly cheap if it carried over to our project!" but cautioned that the fix must be the right one to avoid damaging the long-term health of the code.

We note three potential complexities in cost comparisons. First, we require test cases that identify the defects. This is standard practice in some organizations (e.g., at IBM, testing/QA might prepare test cases for particular bugs that separate maintenance developers may then use to fix them). In others (e.g., much open-source development), test case construction may introduce additional costs. Second, candidate patches produced by our technique must be inspected and validated by developers. While even incorrect tool-generated patches have been shown to reduce the amount of time it takes developers to address an issue [25], the exact reduction amount is unknown. Finally, we note that human patches are imperfect: in a survey of 2,000 OS fixes, Yin *et al.* find that 14–24% are incorrect and 43% of those bad fixes led to crashes, hangs, corruption, or security problems [5].

One of the `php` defects that GenProg successfully repairs corresponds to a use-after-free vulnerability with an associated security CVE.[9] The human patch uses intermediate variables to hold deep copies of the function arguments such that when one is destroyed, the others are unaffected. GenProg inserts code that copies the vulnerable argument an additional time, preserving the relevant values when they are converted. We note that all three of the bug bounties surveyed in Section II pay at least $500 for a single security fix, which exceeds the entire cost of our 105 runs ($403) — including the one that obtained this security repair.

[9] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1148

VI. Limitations, Threats to Validity

An important threat to validity involves whether our results generalize to other settings (i.e., whether our benchmarks represent an indicative sample). We attempt to mitigate selection bias (i.e., "cherry-picking") by defining viable subject programs and defects and then including all matching defects found by an exhaustive search. We acknowledge that our benchmark set is "best effort," however. Our requirements limit the scope from which we draw conclusions. For example, using deterministic bugs leaves race conditions out of scope, while using only C code leaves multi-language bugs out of scope. In addition, we only evaluate on open source software, and thus cannot directly speak to industrial development. Finally, using checked-in bugs that trigger checked-in test cases has the advantage that all bugs considered were of at least moderate priority, but our technique cannot be applied to bugs without tests.

An orthogonal threat relates to the sensitivity of our algorithm to GP parameters. We address this issue directly in previous work [12]. Representation choice and genetic operators matter more than the particular parameter values in this setting [26]. For example, increasing the number of generations has a minimal effect [12, Fig. 2].

VII. Related Work

Automated repair. This work extends our previous work [11], [12], [13], [14] in several important ways. We systematically develop a large benchmark set and conduct a significant study of the technique on two orders of magnitude more code; propose novel GP representations and operators to enhance effectiveness and enable scalability; characterize actual, real-world costs; propose and characterize *fix space* as an important consideration in search-based software repair; and explore factors influencing repair success as well as theoretical and practical limitations.

Clearview [8] uses monitors and instrumentation to flag erroneous executions, and generate and evaluate candidate binary patches to address invariant violations. AutoFix-E [9] leverages contracts present in Eiffel code and abstract state diagrams to propose semantically sound candidate bug fixes. AFix [10] uses reports generated by an atomicity-violation detector to automatically generate patches for single-variable atomicity violations. In the space of dynamic error recovery, Jolt [27] assists in the dynamic detection of and recovery from infinite loops; Demsky *et al.* [28] use run-time monitoring and formal specifications to detect

and repair inconsistent data structures; Smirnov *et al.* [29] automatically compile programs with code to detect memory overflow and generate trace logs, attack signatures, and proposed patches; and Sidiroglou and Keromytis [30] use intrusion detection to identify vulnerable memory allocations and enumerate candidate repair patches.

These approaches address particular defect types via repair strategies or templates enumerated *a priori*, whereas GenProg has developed patches for many defect types [14]. Unlike AFix, GenProg is unlikely to repair concurrency errors, although it can repair deterministic bugs in multi-threaded programs. Unlike AutoFix-E, GenProg does not require specifications or annotations. At most 16 of these 105 bugs might be detected by the two monitors with which ClearView is evaluated, establishing an upper bound on what it might repair from this dataset. In addition, our evaluation includes two orders of magnitude more subject code than AutoFix-E, two orders of magnitude more test cases than ClearView, and two orders of magnitude more defects than AFix, and is strictly larger than these projects and our own related work on each of these metrics separately.

Debugging assistance. He and Gupta use weakest preconditions to statically debug fully-specified programs written in a restricted variant of C [31]. BugFix mines user input and common bug-fix scenarios tracked over a project's lifecycle to suggest bug repairs [32]. DebugAdvisor [33] helps programmers query databases of history, source control, etc, to identify context or prior issues potentially relevant to a given bug. BugFix's user annotations may be useful in repair scenarios such as the one we evaluate in Section V-C.

Molnar *et al.* [7] find integer bugs in 1.5 million lines of code using dynamic test generation. They also evaluate costs using cloud computing prices, finding (but not fixing) 77 bugs for $2.24 each. A natural next step would be to combine both approaches to both find and fix defects.

Evolutionary search and GP. Arcuri and Yao [34] proposed to use GP to automatically co-evolve defect repairs and unit test cases, demonstrating on a hand-coded example of bubble-sort. However, the work relies on formal specifications, limiting generalizability and scalability. Orlov and Sipper have experimented with evolving Java bytecode [35], using specially designed operators. However, our work is the first to report substantial experimental results on real defects in real programs. Recently, Debroy and Wong independently validated that mutations targeted to probably-faulty statements can repair bugs without human intervention [36]. White *et al.* use GP to improve non-functional program properties, particularly execution time [37]. Ackling *et al.* recently proposed to encode variants as a list of rewrite rules and a modification table [15], evaluating on 74 lines of code. Our patch representation follows this spirit.

Search-Based Software Engineering (SBSE) has applied evolutionary and related search methods to software concerns such as testing, project management and effort estimation identification of safety violations, and to re-factoring of large software bases. See Harman [38] for a survey.

VIII. Conclusion

We report novel enhancements to *GenProg*, an automated program repair technique based on genetic program, which significantly improve repair success and enable scalability.[10] With new representation, mutation and crossover operators, GenProg finds 68% more repairs than previous work [11]. These changes enable scalability to bugs in large, open-source programs while taking advantage of cloud computing parallelism. We systematically evaluate GenProg on 105 reproducible defects that developers have previously patched. Those defects come from 8 programs including 5.1 million lines of code and 10,193 test cases. This evaluation includes orders of magnitude more code, test cases, and defects than related or previous work [8], [9], [10], [11].

Our overall goal is to reduce the costs associated with defect repair in software maintenance. GenProg requires test cases and developer validation of candidate repairs, but reduces the cost of actually generating a code patch. While these results are only a first step, they have implications for the future of automated program repair. For example, part of the high cost of developer turnover may be mitigated by using the time saved by this technique to write additional tests, which remain even after developers leave, to guide future repairs. GenProg could also be used to generate fast, cheap repairs that serve as temporary bandages and provide time and direction for developers to find longer-term fixes.

We directly measure the time and monetary cost of our technique by using public cloud computing resources. Our 105 runs can be reproduced for $403: this can be viewed as $7.32, and 96 minutes, for each of 55 bug repairs. While we do not have a quantitative theory that fully explains how GenProg works, the systematic benchmark suite presented here will allow us to investigate such issues in the future. We consider our results to be strongly competitive, and hope that they will increase interest in this research area.

Acknowledgments

The authors are grateful to Mark Fox and Steven Halliwell at Amazon Web Services for cloud computing assistance. We thank Cathrin Weiß and coauthors [23] for making their dataset available. We thank Rob O'Callahan of Novell, as well as Mark Wegman and Peter Santhanam of IBM for enlightenment about real-world bug fixing costs. We recognize Anh Nguyen-Tuong and John C. Knight for insightful discussions about early drafts of this work. We gratefully acknowledge the partial support of NSF (SHF-0905236, CCF-0954024, CCF-0905373), AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231) and the Santa Fe Institute.

[10]Our implementation and reproduction information for our benchmarks are available at http://genprog.cs.virginia.edu/

References

[1] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *International Conference on Software Engineering*, 2006, pp. 361–370.

[3] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated Software Engineering*, 2005, pp. 273–282.

[4] D. Saha, M. G. Nanda, P. Dhoolia, V. K. Nandivada, V. Sinha, and S. Chandra, "Fault localization for data-centric programs," in *Foundations of Software Engineering*, 2011.

[5] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *Foundations of Software Engineering*, 2011, pp. 26–36.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Programming Language Design and Implementation*, 2005, pp. 15–26.

[7] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Security Symposium*, 2009, pp. 67–82.

[8] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Symposium on Operating Systems Principles*, 2009.

[9] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *International Symposium on Software Testing and Analysis*, 2010, pp. 61–72.

[10] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Programming Language Design and Implementation*, 2011.

[11] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*, 2009, pp. 364–367.

[12] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference*, 2009, pp. 947–954.

[13] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Genetic and Evolutionary Computation Conference*, 2010.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automated software repair," *Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[15] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Genetic and Evolutionary Computation*, 2011, pp. 1427–1434.

[16] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[17] B. L. Miller and D. E. Goldberg, "Genetic algorithms, selection schemes, and the varying effects of noise," *Evol. Comput.*, vol. 4, no. 2, pp. 113–131, 1996.

[18] G. Syswerda, "Uniform crossover in genetic algorithms," in *International Conference on Genetic Algorithms*, J. D. Schaffer, Ed., 1989, pp. 2–9.

[19] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced? Bias in bug-fix datasets." in *Foundations of Software Engineering*, 2009, pp. 121–130.

[20] R. Al-Ekram, A. Adma, and O. Baysal, "diffX: an algorithm to detect changes in multi-version XML documents," in *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 1–11.

[21] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Foundations of Software Engineering*, 1999.

[22] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *Programming Language Design and Implementation*, 2008, pp. 136–148.

[23] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Workshop on Mining Software Repositories*, May 2007.

[24] L. Williamson, "IBM Rational software analyzer: Beyond source code," in *Rational Software Developer Conference*, Jun. 2008.

[25] W. Weimer, "Patches as better bug reports," in *Generative Programming and Component Engineering*, 2006, pp. 181–190.

[26] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*, 2011, pp. 33–47.

[27] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *European Conference on Object Oriented Programming*, 2011.

[28] 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 *International Symposium on Software Testing and Analysis*, 2006.

[29] A. Smirnov and T.-C. Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks," in *Network and Distributed System Security Symposium*, 2005.

[30] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Security and Privacy*, vol. 3, no. 6, pp. 41–49, 2005.

[31] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *Fundamental Approaches to Software Engineering*, 2004, pp. 267–280.

[32] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *International Conference on Program Comprehension*, 2009.

[33] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: a recommender system for debugging," in *Foundations of Software Engineering*, 2009, pp. 373–382.

[34] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 162–168.

[35] M. Orlov and M. Sipper, "Genetic programming in the wild: Evolving unrestricted bytecode," in *Genetic and Evolutionary Computation Conference*, 2009, pp. 1043–1050.

[36] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference on Software Testing, Verification, and Validation*, 2010, pp. 65–74.

[37] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. on Evolutionary Computation*, vol. 15, no. 4, pp. 515 –538, aug. 2011.

[38] M. Harman, "The current state and future of search based software engineering," in *International Conference on Software Engineering*, 2007, pp. 342–357.