Experience Report: How Do Techniques, Programs, and Tests Impact Automated Program Repair?

Xianglong Kong*†, Lingming Zhang†, W. Eric Wong†, Bixin Li*

*School of Computer Science and Engineering, Southeast University, 211189, China
{xlkong, bx.li}@seu.edu.cn
†Department of Computer Science, University of Texas at Dallas, 75080, USA
{lingming.zhang, ewong}@utdallas.edu

***Abstract*—Automated program repair can save tremendous manual efforts in software debugging. Therefore, a huge body of research efforts have been dedicated to design and implement automated program repair techniques. Among the existing program repair techniques, genetic-programming-based techniques have shown promising results. Recently, researchers found that random-search-based and adaptive program repair techniques can also produce effective results. In this work, we performed an extensive study for four program repair techniques, including genetic-programming-based, random-search-based, brute-force-based and adaptive program repair techniques. Due to the extremely large time cost of the studied techniques, the study was performed on 153 bugs from 9 small to medium sized programs. In the study, we further investigated the impacts of different programs and test suites on effectiveness and efficiency of program repair techniques. We found that techniques that work well with small programs become too costly or ineffective when applied to medium sized programs. We also computed the false positive rates and discussed the ratio of the explored search space to the whole search space for each studied technique. Surprisingly, all the studied techniques except the random-search-based technique are consistent with the 80/20 rule, i.e., about 80% of successful patches are found within the first 20% of search space.**

I. Introduction

Program repair manually is one of the most tedious and time-consuming work in software industry. Therefore, researchers pay much attention to the area of automated program repair. In recent years, various automated program repair techniques have been proposed [2], [5], [13], [27], [29], [33], [35]. GenProg pioneers the *generate-and-validate* techniques in automated program repair [35]. To repair the faulty programs, generate-and-validate techniques first rank the potential suspicious statements that may contain faults using off-the-shelf fault localization techniques; then they generate automated patches by manipulating suspicious statements in the ranked list; finally, the generate-and-validate techniques validate each generated patch by running the patched version with test cases in the original test suite – if test cases pass, the patch is then validated and reported to the user. Since the generate-and-validate approach is easy to apply and fully automated, a number of techniques have been proposed in this line of work after GenProg, such as RSRepair [27], JAFF [1], and so on [4], [30].

Corresponding author: Bixin Li (email: bx.li@seu.edu.cn)

To date, there are four main techniques for general program repair (i.e., not specifically designed for specific types of bugs) using the generate-and-validate approach: (1) genetic-programming-based technique, (2) random-search-based technique, (3) brute-force-based technique, and (4) adaptive technique. Qi *et al.* proposed random-search-based technique in RSRepair [27], while Weimer *et al.* and Le Goues *et al.* proposed all the other three techniques in GenProg and its extensions [16], [17], [34], [35]. Three of the above techniques are reported to produce promising results: the genetic-programming-based technique is reported to fix 55 out of 105 real bugs; the random-search-based technique is reported to fix 24 out of 24 bugs (the 24 bugs is a subset of the 55 bugs reported to be fixed successfully by GenProg); and the adaptive technique is reported to fix 53 out of the 105 bugs. On the contrary, the brute-force-based technique, which is proposed in the second version of GenProg [16], has not been fully evaluated previously due to its high cost.

To comprehensively understand automated program repair techniques based on the generate-and-validate approach, we perform an extensive study by empirically comparing all the four main general program repair techniques on the same set of subject systems. Furthermore, in generate-and-validate program repair techniques, test suites are essential for repair effectiveness and efficiency. First, the fault localization step of program repair techniques is largely influenced by different test suites, e.g., different number of passed or failed test cases may produce different fault localization results, which further influence both the efficiency and effectiveness of the patch generation process. Second, the patch validation process is also largely influenced by different test suites, since a larger test suite may produce more precise validation results while incurring more patch validation time. Due to the essential role of test suites, in this work, we further investigate how the four main techniques perform in terms of both effectiveness and efficiency when given different number of test cases and different sizes of programs. To conduct the study, we used 153 faulty versions of 9 small to medium sized C programs in Software-artifact Infrastructure Repository (SIR) [8]. Note that we were not able to study more and larger subjects since the current experiments already cost us more than 7 months.

The paper makes the following contributions:

- The first empirical study that considers the impacts

of different techniques, programs, and test suites on general automated program repair.

- The first empirical study that quantitatively computes the false positive rates of automated program repair techniques.
- The experimental results provide various interesting findings and practical guidelines regarding the efficiency and effectiveness of automated program repair techniques in different settings, e.g., different programs and different test suites. For example, the results demonstrate that for deterministic techniques increasing the number of passed tests cannot impact the real success rates, while increasing the number of failed tests to some extent can improve the real success rates. In addition, the experimental results also demonstrate that all the studied techniques except the random-search-based technique are consistent with the 80/20 rule, i.e., about 80% of successful patches are found within the first 20% of search space.

II. Background

Generate-and-validate techniques consist of three steps: fault localization, patch generation and patch validation. The fault localization step first analyzes the paths of both passed and failed test cases to generate a ranked list of suspicious statements that may be faulty. Then, the patch generation step generates candidate patches by manipulating suspicious statements in the ranked list. Manipulation rules used in these techniques contain statement addition, replacement and removal. The genetic-programming-based technique also combines simple statement manipulations into more complex manipulations [14]. The patch validation step runs each candidate patch with the test cases in the test suite to validate the outputs. If there is a failed test case, the patch will be falsified. The patch generation and validation steps will be invoked repeatedly until the technique has found a patch which can pass all test cases in the test suite or it has validated all the candidate patches. The costs in a generate-and-validate technique depend on patch generation (how many candidate patches are created) and patch validation (how each one is tested) [34].

There are four main techniques applying the generate-and-validate framework. The random-search-based technique is proposed in RSRepair [27]. The genetic-programming-based, brute-force-based, and adaptive techniques are proposed in GenProg and its extensions [16], [17], [34], [35]. All the four techniques are based on the assumption that repair action already exists in the faulty program [21]. Barr $et\ al.$ proposed and validated "The Plastic Surgery Hypothesis" to support this assumption [3] – changes to a codebase contain snippets that already exist in the codebase at the time of the change, and these snippets can be efficiently found and exploited. Search algorithms used for patch generation are the main source of differences between these techniques. In the following subsections, we will discuss this in detail.

A. Genetic-programming-based technique

GenProg is a genetic-programming-based automated program repair technique [35]. This technique randomly selects a set of candidate patches as the initial population by manipulating suspicious statements. Then it applies genetic programming on the basis of initial population for several generations. For each patch, it needs to be validated against all the test cases to compute fitness in the step of patch validation. The detailed process is described in Algorithm 1. The search space depends on the size of population and number of generations. These parameters can be modified by user. Each patch in the initial population has only one modification compared to the original program. Each patch in the other generations usually has more than one modification.

Algorithm 1: Genetic-programming-based algorithm

```
   Input  : Faulty program P
   Input  : Test cases {PassedT, FailedT}
   Input  : Parameters {Popsize, Generationnum}
   Output: SuccessfulPatch(pass Test cases)
1  RankedList ← FaultLocate(P, PassedT, FailedT) ;
2  CurrentGen ← 0 ;
3  Pop ← RandomSelect(RankedList, P, Popsize, Mutate) ;
4  while Not find SuccessfulPatch do
5      for Patch ∈ Pop do
6          Fitness ←Validate(Patch, PassedT, FailedT);
7          if Patch passes PassedT and FailedT then
8              SuccessfulPatch ← Patch ;
9              return SuccessfulPatch ;
10         end
11     end
12     CurrentGen ← CurrentGen + 1 ;
13     if CurrentGen > Generationnum then
14         break ;
15     end
16     Pop ←
       Genetic(Popsize, Fitness, Mutate, Crossover) ;
17 end
18 return No valid patch
```

B. Random-search-based technique

RSRepair is a random-search-based technique [27]. This technique is built on GenProg and replaces genetic programming with random search. At first, it also selects a set of candidate patches randomly by manipulating suspicious statements. Then, unlike GenProg, it selects a set of candidate patches again. It selects several times to find a patch which can pass all test cases in the test suite. For each patch, it needs to be validated against the test cases one by one until the patch fails on some test case. The test case which $killed$ more patches will be executed earlier [26]. The detailed process is described in Algorithm 2. The search space depends on the size of population and number of iterations. Each candidate patch has only one modification compared to the original program.

C. Brute-force-based technique

GenProg also includes a simpler technique in its source code [16]. It is brute-force-based technique. When it gets the list of suspicious statements by fault localization, it manipulates the statements in the list one by one. For each suspicious statement, this technique tries to remove it, replace it with another statement in the program, or insert another statement in the program. For each patch, it needs to be validated against the test cases one by one until the patch fails on some test case.

Algorithm 2: Random-search-based algorithm

Input : Faulty program P
Input : Test cases $\{PassedT, FailedT\}$
Input : Parameters $\{Popsize, Iterationnum\}$
Output: $SuccessfulPatch$(pass Test cases)

```
RankedList ← FaultLocate(P, PassedT, FailedT) ;
CurrentIteration ← 1 ;
Pop ← RandomSelect(RankedList, P, Popsize, Mutate) ;
while Not find SuccessfulPatch do
    for Patch ∈ Pop do
        while Patch passes test case do
            Validate(Patch, TestPriority) ;
        end
        if Patch passes PassedT and FailedT then
            SuccessfulPatch ← Patch ;
            return SuccessfulPatch ;
        end
    end
    CurrentIteration ← CurrentIteration + 1 ;
    if CurrentIteration > Iterationnum then
        break ;
    end
    Pop ←
    RandomSelect(RankedList, P, Popsize, Mutate) ;
end
return No valid patch
```

Algorithm 3: Brute-force-based algorithm

Input : Faulty program P
Input : Test cases $\{PassedT, FailedT\}$
Output: $SuccessfulPatch$(pass Test cases)

```
RankedList ← FaultLocate(P, PassedT, FailedT) ;
for Statement ∈ RankedList do
    for ValidStatement ∈ P do
        Patch ← Mutate(P, Statement, ValidStatement)
        ;
        while Patch passes test case do
            Validate(Patch, FailedT, PassedT) ;
        end
        if Patch passes PassedT and FailedT then
            SuccessfulPatch ← Patch ;
            return SuccessfulPatch ;
        end
    end
end
return No valid patch
```

The detailed process is described in Algorithm 3. The search space depends on the size of the faulty program and number of test cases. Each candidate patch has only one modification compared to the original program.

D. Adaptive technique

GenProg's extension also implements an adaptive search and program-equivalence-based technique [34]. This technique is an evolution of brute-force-based technique. Since general program equivalence checking is undecideable, the program equivalence checking used in the adaptive technique is a heuristic-based method. The adaptive technique checks semantic equivalence in three ways: considering duplicate statements as one statement; considering manipulations to dead code (determined by dataflow analysis) as equivalent; considering inserting/moving one instruction before/after different adjacent statements which do not share dependencies with the instruction as equivalent. Before patch generation, the adaptive technique can ignore some useless statements, such as header statements. When it selects a candidate patch, it will check whether it has validated a semantically equivalent patch before. The candidate patch can be validated only if there is not yet semantically equivalent patch in $EquivClasses$. After validation, this patch will be added to $EquivClasses$. For each patch, it needs to be validated against the test cases one by one until the patch fails on some test case. The detailed process is described in Algorithm 4. The search space depends on the size of the faulty program and the number of test cases. Each candidate patch has only one modification compared to the original program.

Algorithm 4: Adaptive algorithm

Input : Faulty program P
Input : Test cases $\{PassedT, FailedT\}$
Output: $SuccessfulPatch$(pass Test cases)

```
RankedList ← FaultLocate(P, PassedT, FailedT) ;
EquivClasses ← ∅ ;
for Statement ∈ RankedList do
    ReducedP ← Ignore(P);
    for ValidStatement ∈ ReducedP do
        Patch ← Mutate(P, Statement, ValidStatement)
        ;
        if Patch ∈ EquivClasses then
            continue;
        else
            EquivClasses ←EquivClasses ∪ {Patch};
            while Patch passes test case do
                Validate(Patch, TestPriority) ;
            end
            if Patch passes PassedT and FailedT then
                SuccessfulPatch ← Patch ;
                return SuccessfulPatch ;
            end
        end
    end
end
return No valid patch
```

As shown in Algorithms 1-4, the random-search-based technique and adaptive technique can be considered as optimized brute-force-based technique, and their search space is a subset of that of brute-force-based technique. The brute-force-based technique and adaptive technique are deterministic techniques, i.e., they can report same results on the same test suites and faulty programs. Since there is random selection in the genetic-programming-based technique and random-search-based technique, they are nondeterministic techniques and may report different results on the same test suites and faulty programs. Because each patch in the genetic-programming-based technique needs to be validated against all the test cases to compute fitness, execution time of the genetic-programming-based technique is largely influenced by number of test cases. In other techniques, patch will be falsified whenever it fails on some test case.

III. Study Approach

In this work, we aim to answer the following research questions:

TABLE I: Subject system statistics

	Subjects	NLOC	# Faults	Test-Pool Size	Description
Small sized	Tcas	135	34	1608	Aircraft collision avoidance system
	Totinfo	273	21	1052	Program used to compute statistics of given data
	Schedule	292	7	2650	Priority schedulers
	Schedule2	262	8	2710	Priority schedulers
	Printtokens	342	5	4130	Lexical analyzers
	Printtokens2	355	10	4115	Lexical analyzers
	Replace	512	28	5542	Pattern matching and substitution system
Medium sized	Space	5902	34	13525	Interpreter for array definition language
	Gzip	4006-5094	6	211	Program used for file compression and decompression
Total		26281	153	36176	

- **RQ1:** What's the overall performance of the different studied techniques?
- **RQ2:** How do the four techniques perform with different sizes of subject systems?
- **RQ3:** How do the studied techniques perform on different test suites?
- **RQ4:** Do the tool-reported patches really fix the bugs?
- **RQ5:** Is there any point so that when the techniques cannot find successful patch before it they also cannot succeed after it in most cases?

A. Subject Systems

To investigate the five research questions, we used all the seven programs from Siemens test suite, Space and Gzip as our subject systems. All the subject systems have been widely used in software testing and debugging research [6], [10], [11], [15]. Table I presents the detailed subject systems statistics. In the table, column "Subjects" list all the subject systems that we used. Note that all the used subject systems come from the Software-artifact Infrastructure Repository (SIR) [8]. And we merged different versions of Gzip (i.e., v1.1.2, v1.2.2, v1.2.4 and v1.3) into one whole subject in our statistics. Column "NLOC" presents the net lines of code, i.e., lines of code that are non-blank and non-comment. Column "# Faults" presents the number of faulty versions that we used for evaluating the program repair techniques. The studied subjects have more faulty versions, but we only used the ones that have at least 5 failed test cases and 100 passed test cases in order to control the number of passed and failed test cases. Column "Test-Pool Size" presents the sizes of the all available test cases in the subjects. The test pools are used to construct smaller test suites, and can be used to validate whether the tool-reported patches based on the smaller test suites really fix the bug. Finally, column "Description" presents the basic functionality of each subject.

We used the subjects shown in Table I instead of the subjects used by GenProg and RSRepair due to the following reasons: First, our experimental setting is more complex than the previous work, e.g., executing all our experiments on the 9 small to medium sized programs already cost us more than 7 months in total. Second, Qi *et al.* found that some tool-reported patches do not produce correct outputs for some test cases in the original test suite for the previous subjects [28]. The reason is that some test scripts do not explicitly check output files. Therefore, GenProg and RSRepair cannot check whether the output is correct, they instead check another property that may indicate correctness such as an exit code, which is not completely correct. Unbefitting test scripts make the patch overfit to these test suites [31]. Thus we ensure that our subjects produce explicit outputs to compare with the expected outputs using the *diff* command. Third, our studied subjects should have enough passed and failed test cases for selection since we want to control the sizes of both passed and failed test cases.

B. Experimental Design

We next describe our experimental setup and the data we collected.

1) Independent Variables: We used the following independent variables to control the factors that may influence the study outcome:

IV1: Different Program Repair Techniques. We applied all the four program repair techniques on the subjects to investigate success rate, false positive rate and efficiency. In our experiments, the genetic-programming-based, brute-force-based and adaptive techniques are provided by latest GenProg V3.0 [34]. The random-search-based technique is provided by RSRepair [27].

IV2: Different Test Suites. We constructed test suites with different sizes to investigate the impact of different test suites on program repair. There are totally 153 faulty programs, each program has 25 different test suites (see Section III-C for detailed test suite construction). Every test suite for each faulty program needs to be repaired once by the four techniques. Thus, each technique needs to run 3825 times in total.

IV3: Different Subject Sizes. We also divided our subjects into two sets, i.e., the small sized subjects (i.e., the seven programs from Siemens suite) and medium sized subjects (i.e., Space and Gzip) to consider the performance differences of the studied techniques on subjects with different sizes.

2) Dependent Variables: We used the following dependent variables to measure the output of the experiments:

DV1: Success Rate. We used the repair success rate as the effectiveness metric. Success rate is the most widely used metric for program repair techniques [16], [17], [34], [35]. Success rate denotes the ratio of tool-reported patches to all repair attempts. Tool-reported patches refer to the valid patches (which can pass all test cases in the test suite) generated by certain repair attempts. The other repair attempts are all failed repair actions, i.e., the studied technique has validated all candidate patches in search space and was not able to find a patch

which can pass all test cases in the test suite. We calculated success rate as: $success\ rate = \frac{\#\ tool{-}reported\ patches}{\#\ all\ repair\ attempts}$

DV2: False Positive Rate. We also measured false positive rate for each studied technique. When we get a tool-reported patch, we further validate it with all test cases in the original test pool (much larger than the test suites used for evaluating repair techniques). If a patch can pass all the test cases in the original test pool, there is a high probability that it is a real-successful patch. Since the manual inspection of each tool-reported patch is extremely time-consuming and may be subjective, we simply treat the patch that passes all test cases in the original test pool as real-successful patch. We calculated false positive rate as: $false\ positive\ rate = 1 - \frac{\#\ real{-}successful\ patches}{\#\ tool{-}reported\ patches}$

DV3: Efficiency. We collected time cost to measure the efficiency. Every programmer knows that manual program repair is a time-consuming work. But few people know that how much time automated program repair technique costs. We collected CPU time of both the successful and failed repair executions. We performed the experiments on a server with Intel Xeon E5-1620v2@3.70GHz CPU and 128G Memory, running Ubuntu-64bit 14.04.

C. Experimental Steps

For each studied subject, we performed the following steps:

- For each faulty program, we selected 5 sets of passed test cases randomly, which have 20, 40, 60, 80, 100 test cases. We also selected 5 sets of failed test cases, which have 1, 2, 3, 4, 5 test cases. For both passed and failed sets of test cases, the larger sets of test cases are constructed to subsume the smaller ones. For example, we selected 1 failed test case randomly as the first failed set, then we continued to select another 1 failed test case randomly, combined with the first failed set to be the second failed set. Each faulty program is evaluated against 25 different test suites which are constructed by combining each set of passed test cases with each set of failed test cases (denoted as 20-1, 20-2, ..., 100-5).
- When the sets of test suites are constructed, all the four studied techniques are applied on each test suite and faulty program. We collected the repair time and results for each execution. Note that due to the time limitation, we set a time out bound of 3 hours for each repair attempt.
- For each tool-reported patch, we collected ratio of the explored search space to the whole search space. We further validated its correctness by executing the tool-reported patch against the original test pool (which is much larger than the used test suite, and can be treated as the golden rule for patch correctness).

IV. Result Analysis

A. RQ1: Overall performance

The overall repair results for all the four studied techniques on all the programs are shown in Table II. In the table, Column 1 lists all the subject systems; Columns 2 to 5 present the overall repair success rates for the four techniques, i.e., the ratio of tool-reported patches to all the repair attempts for all faults and test suite constructions. From the table, we have the following observations.

First, the brute-force-based technique and adaptive technique have the higher success rates. Overall, the brute-force-based/adaptive technique can generate valid patches for 1482/1351 of the 3825 repair attempts. On the contrary, the genetic-programming-based/random-search-based technique can only generate valid patches for 673/359 of the 3825 repair attempts. This finding surprises us, since it shows that neither genetic-programming-based nor random-search-based technique performs well for repairing our subjects. In contrast, the more systematic search techniques (i.e., the brute-force-based technique and adaptive technique) are able to perform better. The only exception is Schedule on which the genetic-programming-based technique has higher success rate than the brute-force-based technique and adaptive technique. This benefits from genetic programming, which can generate more complex patches than other techniques. Therefore, although genetic-programming-based technique has lower success rate than brute-force-based technique and adaptive technique totally, it can fix some bugs which cannot be fixed by other techniques in some cases. In our study, there are 21 (out of 3825) repair attempts that can only be patched by the genetic-programming-based technique. In addition, all the valid patches generated by random-search-based technique can also be generated by brute-force-based technique and adaptive technique.

Second, between the brute-force-based technique and adaptive technique, the success rate of the brute-force-based technique is higher on 5 out of 9 subjects, while equivalent on 3 out of the 9 subjects. The reason is that adaptive technique is an optimized brute-force-based technique with reduced search space, but the reduction is not always safe. Therefore, for the majority of subjects, adaptive technique is outperformed by brute-force-based technique in terms of success rate. Space is the only subject on which adaptive technique outperforms brute-force-based technique. The reason is that program equivalence technique works well on Space, it could reduce redundant patches and make repair attempts on Space more efficient. In most cases, the brute-force-based technique has the highest success rate, i.e., there are 179 (out of 3825) repair attempts that can only be patched by the brute-force-based technique. However, the adaptive technique can generate valid patches that other techniques cannot generate for 56 of the 3825 repair attempts.

Third, between the random-search-based technique and genetic-programming-based technique, the success rate of the genetic-programming-based technique is higher than that of random-search-based technique on 8 out of the 9 subjects. This finding is totally different from the findings in the RSRepair work [27]. We think there can be several reasons. Of course, one reason is that the subjects are totally different. In addition, as also reported by Qi *et al.* [28], some tool-reported patches by RSRepair and GenProg cannot produce correct outputs for the test cases in the test suite. The reason is that for some subjects, GenProg and RSRepair do not check whether the output is correct, they instead check another property that may indicate correctness, which is not completely correct. On the

TABLE II: Average tool-reported success rates

	Brute force	Adaptive	Genetic	Random
Tcas	51.76% (440/850)	42.58% (362/850)	25.52% (217/850)	22.58%(192/850)
Totinfo	23.80% (125/525)	23.80% (125/525)	8.00% (42/525)	5.33% (28/525)
Schedule	2.85% (5/175)	2.85% (5/175)	9.14% (16/175)	2.85% (5/175)
Schedule2	31.50% (63/200)	30.00% (60/200)	25.50% (51/200)	27.50% (55/200)
Printtokens	60.00% (75/125)	60.00% (75/125)	4.80% (6/125)	1.60% (2/125)
Printtokens2	72.00% (180/250)	60.00% (150/250)	32.80% (82/250)	5.20% (13/250)
Replace	47.57% (333/700)	43.28% (303/700)	14.42% (101/700)	7.71% (54/700)
Space	19.76%(168/850)	26.00% (221/850)	15.05% (128/850)	1.17% (10/850)
Gzip	62.00% (93/150)	33.33% (50/150)	20.00% (30/150)	0.00% (0/150)
Total	38.74% (1482/3825)	35.32% (1351/3825)	17.59% (673/3825)	9.38% (359/3825)

contrary, in this work, we directly compared the test outputs with the expected outputs to determine whether a patch passes a test case.

We further show the average time of successful and failed repair attempts in Table III and Table IV. In Table III/Table IV, Column 1 lists all the subjects, while Columns 2 to 5 present the average repair time for successful/failed repair attempts. Note that we do not have the average time data for successful executions of RSRepair on Gzip, since RSRepair could not fix any bug for Gzip. In addition, the brute-forced-based and adaptive techniques timed out for each failed repair attempt on Space and Gzip. We can conclude from the table that for both successful and failed repair attempts, random-search-based technique costs the least time, while genetic-programming-based technique costs the most time on small sized subjects. In addition, the adaptive technique costs less time than brute-force-based technique.

B. RQ2: Program repair for different sizes of subjects

As presented in Section IV-A, brute-force-based technique and adaptive technique have higher success rates than genetic-programming-based technique and random-search-based technique on small sized subjects. Success rate of brute-force-based technique is slightly higher than that of adaptive technique, but adaptive technique costs less time. Because time of executions on small sized subjects usually is not too long, brute-force-based technique can be the most excellent technique for small sized subjects. Random-search-based technique costs the least time for both successful and failed executions, but it has the lowest success rate. The genetic-programming-based technique usually takes more than an hour to repair a program which contains hundreds of LOC. That is because each patch needs to be validated against every test case to compute fitness. Execution time of genetic-programming-based technique is mainly due to the number of test cases and the time of each test case running. If the faulty program easily fails into execution hanging by the repair attempt, the execution time will be enormous, such as Printtokens.

For medium sized subjects, brute-force-based technique and adaptive technique cost much more time than genetic-programming-based technique and random-search-based technique. The reason is that search space of deterministic techniques grows rapidly with the increase of program size. We could not run brute-force-based technique and adaptive technique completely on Space and Gzip within timeout bound (i.e., 3 hours in this work). For example, brute-force-based technique can generate millions of candidate patches on Space which has 5902 NLOC. The brute-force-based technique can validate about 70 thousand patches within 3 hours. The adaptive technique can reduce more than half patches, but the remaining patches are still too many to validate. The adaptive technique technique can only validate less than 10 thousand patches within 3 hours because of the equivalence validation process. Average time of successful executions on these subjects is less than 1 hour and the longest time of successful executions is less than 2 hours. Thus, we think that 3 hours is an appropriate time limit. The genetic-programming-based technique applied on Space has higher success rate than brute-force-based technique, but overall brute-force-based technique and adaptive technique have higher success rates. The random-search-based technique still costs the least time and has the lowest success rate. We can conclude that few techniques can work well on even medium sized programs. The two deterministic techniques (the brute-force-based technique and adaptive technique) are too costly, and two nondeterministic techniques (the genetic-programming-based technique and random-search-based technique) are too ineffective.

C. RQ3: Program repair using different test suites

The detailed repair success rates for the studied techniques using different sizes of test suites are shown in Figure 1.

TABLE III: Average time of successful executions

Unit: second

	Brute force	Adaptive	Genetic	Random
Tcas	63.94	59.33	58.42	11.33
Totinfo	275.40	214.53	438.70	31.24
Schedule	47.23	6.46	287.53	21.93
Schedule2	63.23	18.24	1094.69	22.31
Printtokens	355.40	155.59	3913.51	75.53
Printtokens2	274.56	217.50	601.29	64.19
Replace	402.60	30.50	453.71	44.02
Space	2664.69	902.62	363.30	77.15
Gzip	2249.86	1642.36	1591.23	–

TABLE IV: Average time of failed executions

Unit: second

	Brute force	Adaptive	Genetic	Random
Tcas	260.90	231.67	474.25	37.60
Totinfo	1788.13	967.26	3096.51	111.36
Schedule	395.36	219.90	2630.53	115.07
Schedule2	559.19	281.70	5181.53	126.89
Printtokens	1181.08	551.13	7458.03	210.20
Printtokens2	2739.86	908.07	3304.57	147.60
Replace	1024.47	522.96	3304.11	128.19
Space	TimeOut	TimeOut	1082.43	122.51
Gzip	TimeOut	TimeOut	9084.24	739.14

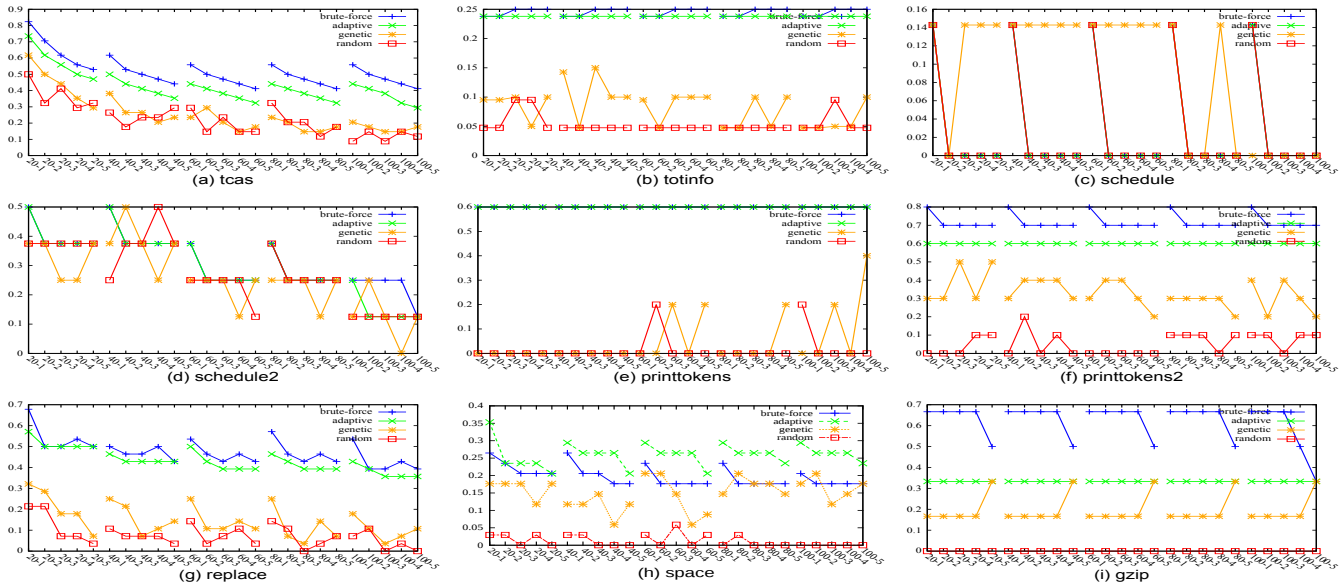

Fig. 1: Tool-reported success rates on each subject with different test suites

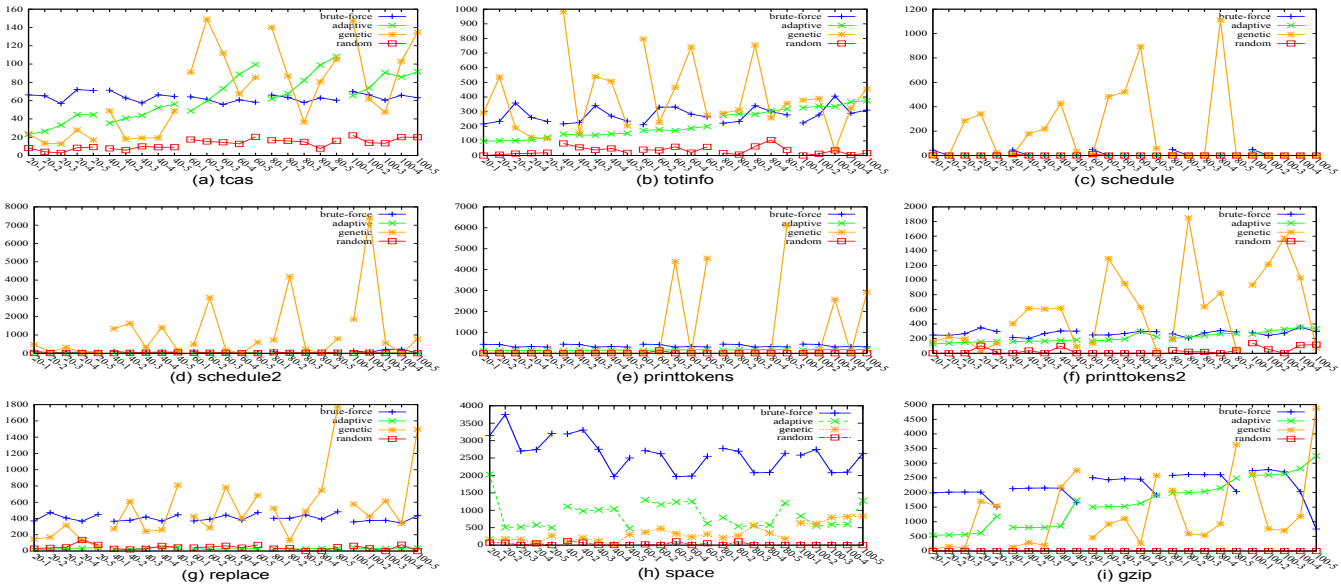

Fig. 2: Average time of successful executions on each subject with different test suites

In the figure, each sub-figure shows the results for each subject. In each sub-figure, the x axis represents the different test suites, the y axis represents the success rates, and the four lines of different patterns represent the results for the four techniques, respectively. From the figure, we have the following observations. First, when the number of failed test cases increases, the success rates of the two deterministic techniques (i.e., the adaptive technique and brute-force-based technique) have a clear decrease trend, while the other two nondeterministic techniques do not have such clear trend. The deterministic techniques have lower success rates since when the number of failed test cases increases it is harder for a patch to pass all the test cases. The nondeterministic techniques do not have clear trend, since the patch execution ordering can be quite random for different test suite settings. Second, when the number of passed test cases increases, the deterministic techniques tend to have a stable or slight decrease trend. The reason is two folds: (1) more test cases make it harder for a patch to pass all of them, thus lowering down the success rate; (2) more passed test cases may improve the fault localization results, thus improving the success rate. The interaction of the two factors tends to make the repair success rates either stable or slightly decreasing.

We further show the average time for successful and failed repair attempts using different sizes of test suites in Figure 2 and Figure 3, respectively. Note that in Figure 3, we do not show the figures for Gzip and Space since the adaptive technique and brute-force-based technique always time out for the failed executions on those two subjects. For the average

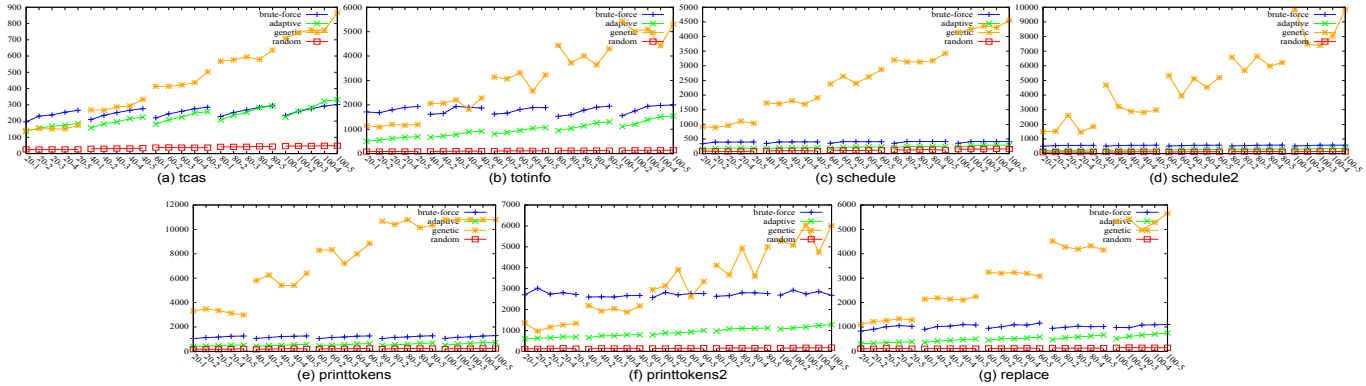

Fig. 3: Average time of failed executions on each subject with different test suites

time of successful repair attempts (Figure 2), we can find that changing the number of neither passed test cases nor failed test cases has clear influence on the repair time. The only observation we can make is that the genetic-programming-based technique tends to be more time-consuming on small subjects, while the brute-force-based technique tends to be more time-consuming on larger subjects. For the average time of failed repair attempts (Figure 3), surprisingly, there are several clear trends. First, when the number of failed test cases increases, the two deterministic techniques (i.e., the adaptive technique and brute-force-based technique) have clear growth trend in repair time. Second, when the number of passed test cases increases, only the genetic-programming-based technique have clear growth trend, while all the other techniques are stable. The reason is that the genetic-programming-based technique needs to run all test cases against each patch and is influenced most by the number of test cases.

D. RQ4: Do the tool-reported patches really fix the bug?

Tool-reported patch is a patch which can pass all test cases in the test suites. It may not fix the bug actually. Qi *et al.* found that some tool-reported patches are equivalent to a single modification that simply deletes functionality [28]. Thus we want to investigate the ratio of tool-reported patches that can really fix the bug. We assume that if a patch can pass all the test cases in original test pool, it is a real-successful patch. We validated all the tool-reported patches against test cases in original test pool. We present both false positive rates and real-success rates for the studied techniques in Figure 4. The x axis represents the different test suites, and the y axis represents the overall false positive rates or real success rates for all studied techniques (represented by four lines with different patterns).

As shown in Figure 4(a), when there are 20 passed test cases and 1 failed test case, false positive rates of brute-force-based technique and adaptive technique are over 50%, those of genetic-programming-based technique and random-search-based technique are over 80%. When test cases increase to 100 passed test cases and 5 failed test cases, false positive rates of brute-force-based technique and adaptive technique are 22% and 24%, those of genetic-programming-based technique and random-search-based technique are 38% and 14%. We can make the following observations. First, false positive rates vary widely with the increase of test cases. In general, the false positive rates for deterministic techniques (i.e., the adaptive technique and brute-force-based technique) have very clear decreasing trend. Second, although false positive rates of genetic-programming-based technique and random-search-based technique are more irregular, the general trend is similar to the brute-force-based technique and adaptive technique. Third, the influence of failed test cases is much larger than that of passed test cases. Fourth, the false positive rates of the adaptive technique and brute-force-based technique are quite close, and the adaptive technique has just a slightly higher false positive rate than the brute-force-based technique on 22 out of 25 different test suites. Fifth, in general, the brute-force-based technique and adaptive technique have lower false positive rates than genetic-programming-based technique and random-search-based technique.

As shown in Figure 4(b), real success rates do not get much influence from the number of passed test cases for all techniques. The real success rates of the two nondeterministic techniques (the random-search-based technique and genetic-programming-based technique) do not have clear trend when the number of failed tests increases. On the contrary, the real success rates of the two deterministic techniques (the brute-force-based technique and adaptive technique) increase when the number of failed test cases increases at the beginning, but then become stable or even drop when the number of failed tests keep increasing. The reason is that it is much harder to repair faults when there are too many failed test cases, especially on medium sized programs. This provides a practical guideline that adding more passed test cases cannot help with program repair effectiveness, and adding more failed tests to some extent (not always) can be beneficial.

E. RQ5: Position of verified patches in search space

For medium or large sized subjects, these techniques usually report nothing after a long time wait. That is a waste of time. So we try to find some point in the whole search space so that we can stop the executions when techniques have validated patches before the point. In other words, most of successful patches are validated before the point. To find the point, we investigated the explored search space of all the tool-reported patches in our experiments. There are totally 1482 brute-force-reported patches, 1351 adaptive-reported patches, 673 genetic-reported patches and 359 random-reported patches. Figure 5 shows the results. The x axis shows the percentage of tool-reported patches,

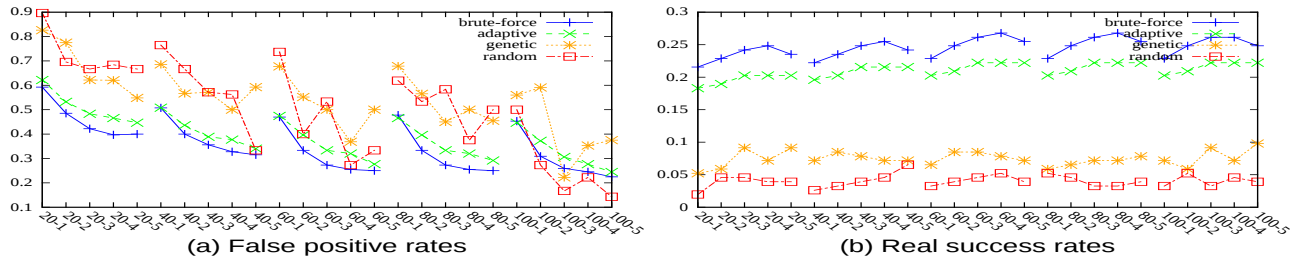

Fig. 4: False positive rates and Real success rates on different test suites

the y axis shows the ratio of the explored search space to the whole search space. Among the tool-reported patches of brute-force-based technique, 75% of the patches are explored within 20% of search space, and 89% of the patches are explored within 50% of search space. For adaptive technique, 80%/81% of the tool-reported patches are explored within 20%/50% of search space. For genetic-programming-based technique, 38% of the tool-reported patches are found in initial population. Genetic-programming-based technique can find 70%/87% of tool-reported patches within 20%/50% of search space. Random-search-based technique selects sets of candidate patches 10 times randomly in each execution, so tool-reported patches distribute more homogeneous than other techniques, 43% of the patches are found within 20% of search space. We found that genetic-programming-based, brute-force-based and adaptive techniques are consistent with the 80/20 rule. About 80% of successful patches are found within the first 20% of search space. The 80/20 rule of brute-force-based technique and adaptive technique benefits from ranked list of suspicious statements. The higher position the faulty statement is in the ranked list, the more suspicious it is. For genetic-programming-based technique, because most bugs can be fixed by simple modifications, most tool-reported patches are found in earlier generations.

We also present the explored search space of real-successful patches in Figure 6. The 80/20 rule also applies to all techniques except the random-search-based technique

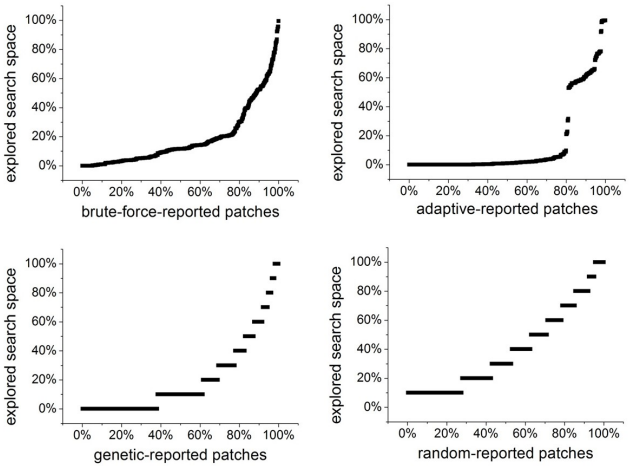

Fig. 5: Explored search space of tool-reported patches

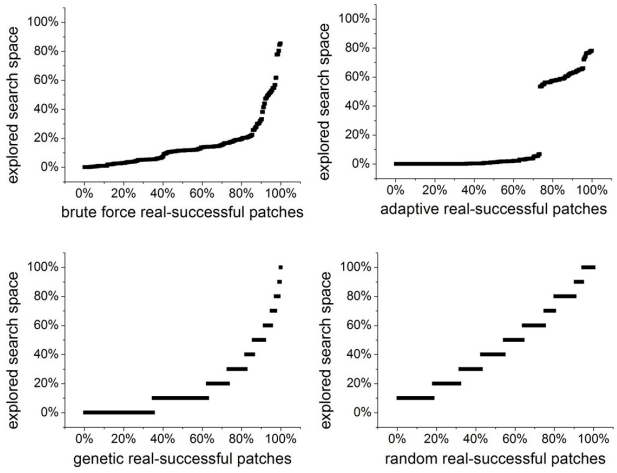

Fig. 6: Explored search space of real-successful patches

here. We have got 2186 real-successful patches in total. Within the first 20% of search space, these techniques can get 83% (brute-force-based technique), 73% (adaptive technique), 73% (genetic-programming-based technique) and 32% (random-search-based technique) of real-successful patches.

F. Threats to Validity

Threats to construct validity. The main threat to construct validity is the metrics that we used to evaluate the effectiveness and efficiency of program repair techniques. To reduce this threat, we used the widely used *success rate* metric [16], [17], [34], [35] (which compute the ratio of tool-reported repairs to all repair attempts) and also record the repair cost in terms of execution time. In addition, in this work, we also consider the *false positive rate* metric, which computes the ratio of actually unsuccessful tool-reported repairs to all tool-reported repairs.

Threats to internal validity. The main threat to internal validity is the potential faults in the configurations of existing tools, in our scripts, or in our data analysis. To reduce this threat, the first author carefully reviewed all the tool configurations, scripts, and data analysis code during the study.

Threats to external validity. The subject systems, faults, test cases, and repair tools used in our study may all pose threats to external validity. To reduce these threats, we plan to conduct the study on more and larger subject systems, more faulty versions, more test paradigms, and more program repair

techniques. Furthermore, recently, Zhong and Su showed that only 30% of real bugs can be fixed by a single repair action [36]. Therefore, we also plan to conduct study on more real bugs with multiple faulty locations.

V. Related Work

General automated program repair. GenProg and its extensions implement brute-force-based technique, genetic-programming-based technique and adaptive technique on automated program repair [16], [34], [35]. They manipulate faulty programs to generate candidate patches by using manipulation rules such as statement addition, replacement and removal, then validate these patches by running passed and failed test cases in the test suites. RSRepair is built on GenProg, it replaces genetic programming with random search [27]. Debroy and Wong mutate faulty programs by replacing operators and negating condition in if or $while$ statements [7]. Schulte $et\ al.$ repair legacy software by using same manipulation rules as GenProg, but it is applied at assembly code level [30]. All these techniques generate candidate patches by using some specific manipulation rules which are not relevant to faults and semantics, they all ignore types of faults and just expect to find a successful patch under aimless manipulation. SPR makes a new try by generating semantics-preserving patches [19]. It uses condition synthesis to instantiate transformation schema to repair faulty program. Unlike above techniques, there are some other techniques which focus on particular repair patterns. PAR manipulates faulty programs by using ten repair patterns which are extracted from thousands of human-written patches [13]. Monperrus found that PAR is good at fixing particular bugs [22]. There are also some techniques which need additional specifications. For example, AutoFix-E can generate semantically sound patches using Eiffel contracts [33]. It needs preconditions, postconditions and intermediate assertions to repair the faulty program. SemFix can repair faulty programs via semantic analysis based on symbolic execution, constraint solving, and program synthesis [24]. It needs to formulate executions of test cases as constraints. In this work, we present an extensive study on the scalable general program repair techniques that do not require manual repair patterns, specifications, or constraint solving.

Automated program repair for specific types of bugs. With static analysis of bug reports generated by atomicity violation detector, AFix generates a suitable patch for each atomicity violation bug [12]. ClearView can generate binary patches to correct buffer overflows and illegal control flow transfers [25]. It observes normal executions to learn invariants which can show safe behavior and monitors the program dynamically. When the invariants are violated, the system tries to restore them. RCV can correct divide-by-zero and null-dereference faults by using recovery shepherding [20]. It discards writing via null-references and returns zero as the result of divides-by-zero. LeakFix can correct memory leak by insert free-memory statements safely [9]. It analyzes memory allocations to guarantee that the patch will not interrupt normal execution of the program. Axis corrects atomicity violations by modeling the concurrent properties of a program as Petri nets and using SBPI control constraints to generate possible patches [18]. Since these techniques are based on some particular repair patterns, they can just fix particular bugs.

Survey of bug fixes. Zhong and Su analyzed more than 9000 real bug fixes to investigate the factors influencing program repair [36], such as fault distribution, fault complexity and so on. They reported that programmers made a single repair action to fix about 30% of real bugs, more than 70% of real bugs are fixed by at least two repair actions. Thung $et\ al.$ investigated hundreds of real faults in several software systems, they found that many faults may not be localizable to a few lines of code [32]. Most of current program repair techniques are on the basis of fault localization, if a bug is not localizable, it cannot be fixed at all. Nguyen $et\ al.$ found that bug fixes repeat frequently for small bugs, fixes repeat less frequently for large bugs [23]. So a good way to fix small bugs is extracting repair patterns from history. Barr $et\ al.$ provided a solid footing to these generate-and-validate techniques that depends on the plastic surgery hypothesis [3]. They found that 30% of repair actions can be found within the same file and 53%/34% of the patches can be composed out of code from 1/2 addresses in the program.

VI. Conclusion

To investigate how different techniques, test suites, and programs impact automated program repair in terms of effectiveness and efficiency, we conduct this empirical study. We have applied each technique on different test suites and faulty programs for more than 3000 times. The results show that success rates of brute-force-based technique and adaptive technique are higher than those of genetic-programming-based technique and random-search-based technique in most cases, while the random-search-based technique usually costs the least time but has the lowest success rate. However, techniques that work well with small programs become too costly or ineffective when applied to medium sized or larger programs. In terms of average time cost, when the number of failed tests increases, both the brute-force-based technique and adaptive technique have clear growth trend for failed repair attempts; when the number of passed tests increases, only the genetic technique has growth trend for failed repair attempts. In terms of success rate, the genetic and random techniques do not have clear trend in tool-reported or real success rates when the number of passed or failed tests changes. On the contrary, for the brute-force-based technique and adaptive technique, in terms of tool-reported success rate, they both have clear decreasing trend when the number of failed tests increases and stable or slightly decreasing trend when the number of passed test cases increases; in terms of real success rate, adding more passed tests does not impact their effectiveness, while adding more failed tests are beneficial to some extent (not always). Finally, we found that all the studied repair techniques except the random-search-based technique are consistent with the 80/20 rule, i.e., about 80% of successful patches are found within the first 20% of search space.

VII. Acknowledgments

We would like to thank W. Weimer, C. Le Goues, and Y. Qi $et\ al.$ for sharing the source code of GenProg and RSRepair with us. This work is sponsored partially by China Scholarship Council No.201406090080, partially by National Natural Science Foundation of China under Grant No. 61572126 and partially by Huawei Innovation Research Program (HIRP) under Grant No. YB2013120195.

REFERENCES

[1] A. Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing Journal*, pages 3494–3514, 2011.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. *Proceedings of 2008 IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[3] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 306–317, 2014.

[4] J. S. Bradbury and K. Jalbert. Automatic Repair of Concurrency Bugs. *Proceedings of the 2nd International Symposium on Search Based Software Engineering*, pages 1–2, 2010.

[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554, 2009.

[6] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.

[7] V. Debroy and W. E. Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, pages 45–60, 2014.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, pages 405–435, 2005.

[9] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe Memory-Leak Fixing for C Programs. *Proceedings of the 37th International Conference on Software Engineering*, In Press, 2015.

[10] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. Fault localization based on failure-inducing combinations. *Proceedings of the 24th International Symposium on Software Reliability Engineering*, pages 168–177, 2013.

[11] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[12] G. Jin and S. Lu. Automated Atomicity-Violation Fixing. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400, 2011.

[13] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. *Proceedings of the 35th International Conference on Software Engineering*, pages 802–811, 2013.

[14] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

[15] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of systems and Software*, pages 2416–2430, 2010.

[16] C. Le 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. *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13, 2012.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, pages 54–72, 2012.

[18] P. Liu and C. Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. *Proceedings of the 34th International Conference on Software Engineering*, pages 299–309, 2012.

[19] F. Long and M. Rinard. Staged Program Repair with Condition Synthesis. *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, In Press, 2015.

[20] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–238, 2014.

[21] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495, 2014.

[22] 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. *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242, 2014.

[23] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 180–190, 2013.

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. *Proceedings of the 35th International Conference on Software Engineering*, pages 772–781, 2013.

[25] J. H. J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, Others, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. *Symposium on Operating Systems Principles*, pages 87–102, 2009.

[26] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 180–189, 2013.

[27] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[28] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, In Press, 2015.

[29] H. Samirni, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287, 2012.

[30] E. Schulte, S. Forrest, and W. Weimer. Automated Program Repair through the Evolution of Assembly Code. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2010.

[31] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, In Press, 2015.

[32] F. Thung, D. Lo, and L. Jiang. Are Faults Localizable ? *Proceedings of Mining Software Repositories*, pages 74–77, 2012.

[33] Y. Wei, Y. Pei, C. a. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Debugging of Programs with Contracts. *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, 2010.

[34] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366, 2013.

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

[36] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. *Proceedings of the 37th International Conference on Software Engineering*, In Press, 2015.