Should We Add Repair Time to an Unfixed Bug?

An Exploratory Study of Automated Program Repair on 2980 Small-Scale Programs

Chuanqi Xu, Yisen Xu, Yingqi Zhang, Jifeng Xuan
School of Computer Science
Wuhan University
Wuhan, China
{legendxu, xuyisen, 2016301500318, jxuan}@whu.edu.cn

***Abstract*—The goal of automated program repair is to automate patch generation for buggy programs to reduce the manual effort by developers. A generate-and-validate method, such as GenProg, is a kind of typical repair methods that continuously generate potential patches and then validate the patches with a given test suite. A generate-and-validate method can accumulate patches when the execution time of repair methods increases. However, how many buggy programs can be newly patched when the time increase? In this paper, we conducted an exploratory study of repairing 2980 small-scale buggy programs from the CODEFLAWS benchmark with three repair methods GENPROG, SPR, and PROPHET. The aim of this study is to understand the execution time of repair methods via investigating four research questions. Experimental results show that the time of patch generation correlates with the number of executable lines of code and the Cyclomatic complexity. That is, a complex program is difficult to be repaired. This motivates us to explore a new repair method that can weaken such correlation with the lines of code and the complexity. We designed VANFIX, a simple and effective repair method for small-scale C programs. VANFIX leverages the probability of exploring the search space to conduct a variable search neighborhood for potential patches, rather than patching suspicious statements one by one. The comparison among repair methods shows that VANFIX can generate patches for 653 buggy programs, which contains 408 correctly patched buggy programs. This makes VANFIX achieve 24% to 30% better precision than GENPROG, SPR, and PROPHET.**

***Index Terms*—program repair, execution time, exploratory study, correlation, Codeflaws**

I. INTRODUCTION

Automated program repair has been proposed to automate patch generation for buggy programs to reduce the effort of manual debugging by developers. In program repair, generate-and-validate methods, such as GENPROG [16], are a kind of typical repair methods that are widely studied in the research community. A generate-and-validate method continuously generates patches and validates the patches with a given test suite until a patch that makes the whole test suite pass is found. GENPROG [53], a pioneering repair method since 2009, converts the buggy program into Syntax Abstract Trees (ASTs) and searches a potential patch by mutating the AST to pass the test suite. In this paper, we focus on the study of generate-and-validate repair methods [10], [13], [16], [24], [25], [28], [52], [54], [56].

Increasing the execution time of a generate-and-validate repair method can increase the opportunity of finding a patch. In this paper, we refer to the execution time of a repair method as *repair time*. On the one hand, given a limited time budget, a repair method may fail in patch generation due to the huge search space of patches; on the other hand, exhausting all patches is expensive and requires extremely long execution time. For generate-and-validate repair methods, longer repair time may lead to more generated patches for buggy programs.

In this paper, we conducted an exploratory study on the repair time of automated program repair. *If a buggy program is not fixed in short time, should we add repair time to enlarge the probability of generating new patches*? This question reveals whether we should spend long repair time in applying an automated repair method. This study evaluates the repair time of repairing 2980 small-scale buggy programs from the CODEFLAWS benchmark with three off-the-shelf methods, GENPROG, SPR, and PROPHET. We do not aim at comparing which repair method generates patches in the shortest time since existing repair tools are diversely designed and implemented. Instead, for a particular repair method, we investigate the incremental number of patched buggy programs over time.

We leveraged the experimental results to understand the repair time via four research questions. We find out the following results: 1) The number of newly patched buggy programs does not highly increase when the execution time increases; after the execution of 600 seconds, correctly patches buggy program are hardly increased. 2) The repair time of generating plausible or correct patches for buggy programs correlates with the number of executable lines of code, the Cyclomatic complexity, and the mutation score; the repair time of finding out a correct patch can be affected by the scale and the complexity of the source code as well as the test suite. 3) Three repair methods GENPROG, SPR, and PROPHET under evaluation correctly patched 32% to 38% of plausibly patched buggy programs. Among all buggy programs, GENPROG correctly patched 2% while SPR and PROPHET correctly patched around 11%.

The correlation between the repair time and the number of executable lines of Code (called *ExecLoC*) and the Cyclomatic complexity motivates us to explore a new method of program repair. We designed VANFIX, a simple and effective

repair method for small-scale C programs. Different from patching suspicious statements one by one, VANFIX leverages the probability of exploring the search space to conduct a variable search neighborhood for potential patches. The variable search neighborhood can help VANFIX avoid to be caught by exhausting one suspicious statement and reduce the dependency between the repair time and the LoC and the code complexity. The comparison among repair methods shows that VANFIX can generate patches for 653 buggy programs, which contains 408 correctly patched buggy programs. This makes VANFIX achieve 24% to 30% of improvement of precision over GENPROG, SPR, and PROPHET.

This paper makes the following major contributions:

- We conducted an exploratory study of repair time on 2980 buggy programs in the CODEFLAWS dataset.
- We empirically compared existing repair tools on C programs, including GENPROG, SPR, and PROPHET, via analyzing generated patches and correct patches.
- We designed a new and effective method, VANFIX, which can achieve the precision of 0.62, i.e., 408 correct patches out of 653 generated patches.

The remainder of this paper is organized as follows. Section II presents the background and motivation. Section III presents the experimental setup, including research questions, data preparation, and evaluation setup. Section IV reports the experimental results in our study. Section V describes a new repair approach, VANFIX, as well as the evaluation. Section VI discusses the threats to the validity. Section VII lists the related work and Section VIII concludes this paper.

II. BACKGROUND

We present the terminology, the background, and the motivation of this study.

A. Terminology

A *generate-and-validate repair method* is a method that aims to continuously generate patches and validate the patches with a given test suite [37].

A *plausible patch* is a patch that passes the whole test suite of validation [26], [43].

A *correct patch* is a plausible patch that do not introduce any latent or new defect. Existing works identify the correct patch as a semantically equivalent patch to the manual patch that was written by human developers [27], [57]. If a generated patch is exact the same as the manual one, the patch is labeled as correct; if not, validating the correctness requires the manual analysis by human developers [43], [62], [63].

The *repair time* is the execution time that is used to generate a patch with a limited time budget (also called *timeout*) for a given repair method. Note that we do not limit *repair time* to particular plausible patches or correct patches. The repair time relates the actual execution time of a repair method.

B. Background

Automated program repair. Automated program repair has attracted many attentions since the birth of GENPROG [53]. GENPROG, as well as other existing repair methods, aims at fixing a buggy program via generating a potential patch that can pass a manually designed test suite. Such repair methods are also called test suite based repair [27], [37]. Existing methods of test suite based repair can be roughly divided into two categories: generate-and-validate repair methods or synthesis-based repair methods [60]. In this study, we focus on the category of generate-and-validate methods, which employ continuously heuristic or rule-based search to generate candidate patch lists and then validate whether a patch passes the test suite or not. Besides generate-and-validate methods, synthesis-based repair methods transform patch generation into the problem of constraint solving and convert back the solution into a code patch [34], [38], [57], [60].

Search space for patches. Long and Rinard [26] have analyzed the search space of generate-and-validate repair methods on 69 buggy C programs. They found that correct patches are sparse in the search space and plausible patches are relatively abundant. Thus, it is challenging to find out a correct patch among plausible ones. The fact that correct patches is sparse leads to the expansion of the current search space. Nguyen et al. [39] leveraged test input generation to enlarge the possibility of patch generation. Hua et al. [6] translated buggy programs into partial programs with holes that may consists of thousands of candidate patches. Martinez and Monperrus [29] conducted patches with parametrized fix ingredients and led to the explosion of the search space. Their approach finds out over 8900 patches for the Astor tool for Java patch generation [30]. The rich search space provides the potential of finding out correct patches as well as hiding correct patches among plausible ones.

Correlation analysis. Yi et al. [59] have investigated the correlation between program repair and test suite metrics on 142 buggy C programs. Test suite metrics under evaluation include statement coverage, branch coverage, the size of the test suite, the mutation score, and the ratio of capable test cases. Their experiments show that the reliability of patches generally tends to increase when the test suite metrics increase. Correlation with the repair time is also involved: inconclusive results about the correlation between test suites and the repair time [59].

C. Motivation

Fig. 1 presents an excerpt of a buggy program in the CODEFLAWS benchmark [50] with ID 109-A-bug-13092782-13092815. The source code is expected to find out an integer that contains the digits of 4 and 7 and the sum of all digits equals to a given number z. The buggy program fails to output the parsed digits due to the lack of exiting the loop. The manually written patch is at Line 28, i.e., adding a `break` statement. An existing repair method SPR [24] can modify Line 21 into `if(i%4==0 && flag==0)`, which is equivalent to the manual patch. Our trials show that SPR failed in patching the program.

According to the design and implementation, SPR [24] could correctly patched the example in Fig. 1. However,

```
15  if(z%7 != 0 && flag == 0)
16  {
17      k = z/7;
18      for(i = z-(k-7); k != 0; k --)
19      {
20          i = z-(k*7);
21          if(i%4 == 0)
22          {
23              flag = 1;
24              for(b = 1; b <= i/4; b ++)
25                  printf("4");
26              for(j = 1; j <= k; j ++)
27                  printf("7");
28  +           break;
29          }
30      }
31  }
32  if(z%4 == 0 && flag == 0)
33      [...]
34  if(flag == 0)
35      [...]
```

Fig. 1. Excerpt of a buggy program (ID 109-A-bug-13092782-13092815) in the CODEFLAWS benchmark.

SPR conducts patches according to its ranking of faulty statements: statements are examined and patched one by one. The statement at Line 21 is ranked over half the ranking list, which makes SPR caught by other statements which may not provide patches. Even the repair time is increased, SPR cannot further generate patches for the buggy program.

This example motivates our study: given a repair method, should we add repair time to a not-patched buggy program? Which factor can affect the repair time and how can we reduce the repair time? In this paper, we conducted an exploratory study on the repair time of GENPROG, SPR, and PROPHET on 2980 buggy C programs from the CODEFLAWS benchmark.

III. EXPERIMENTAL SETUP

We present the experimental setup of our study, including the research questions, the data preparation, the studied repair methods, and the evaluation setup.[1]

A. Research Questions

Our work aims to investigate the repair time of generate-and-validate methods of patch generation. We designed the following four Research Questions (RQs) to understand the repair time.

RQ1. *How Many Buggy Programs are Newly Patched When the Repair Time Increases?* A generate-and-validate method of patch generation can try to find out different patches over time. In RQ1, we aim at studying the incremental number of newly patched buggy programs when the repair time increases.

RQ2. *Which Metric Correlates with the Time of Plausibly Patching a Buggy Program?* In automated program repair, a generate-and-validate method is to generate a patch that makes all given test cases pass. The number of generated patches can be used to measure the repairability of repair methods. We evaluate the correlation between repair time and metrics for patched buggy programs in RQ2.

RQ3. *Which Metric Correlates with the Time of Correctly Patching a Buggy Program?* A generated patch may be inconsistent with the manually written patch by developers. This leads to the manual evaluation of automatically generated patches. We investigate RQ3 to evaluate the repair time of correctly patched buggy programs.

RQ4. *How Effective can Existing Methods Patch Buggy Programs in an Hour?* A developer cannot allow a repair method to exhaustively generate all patches. In a given limited time, how many buggy programs can be patched or correctly patched? In RQ4, we investigate the number of patched buggy programs in one hour.

B. Experimental Dataset Preparation

We evaluate the repair time on CODEFLAWS, a defect benchmark of C programs.[2] The CODEFLAWS dataset consists of 3902 defects, which are derived from the submitted source code by 1653 users of an online programming contest [50]. Buggy programs in CODEFLAWS are categorized into 39 defect classes, including defects relating to control flows, function calls, arithmetics, arrays, etc. The growth of research on program repair benefits from the publicly available defect benchmarks, such as IntroClass with 998 C defects and ManyBugs with 185 C defects [15], Defects4J with 438 Java defects [9], [27], bugs.jar with 1158 Java defects [46], and Bears with 251 Java defects [2].

Two major reasons for choosing CODEFLAWS as the dataset are as follows. First, CODEFLAWS provides 3902 buggy programs, which diversifies the evaluation. To date, it may be impossible to directly understand the nature of repair time. Thus, we leverage a large number of buggy programs to support the observation. Second, the correctness evaluation of patch generation requires manual check by developers. In CODEFLAWS, the number of executable lines of code buggy programs is less than 322. The small-scale source code can ease the check of generated patches and reduce the threat of bias that derives from the manual check.

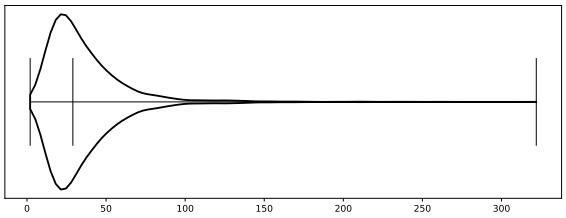
In program repair, several repair methods share the assumption that the patch locates inside one line. Since the time evaluation of program repair is computationally expensive, we compared each buggy program with its manually-written patch and filtered out the bug, whose patch crosses multiple lines. Finally, 2980 buggy programs that contain patches inside one line of code are kept as the experimental dataset. Fig. 2 presents the violin-plots of the number of executable lines of code and the number of test cases in each buggy programs. Since our study is to investigate the repair time, we used the whole test suite to maximize the potential of repair methods.

C. Repair Methods under Evaluation

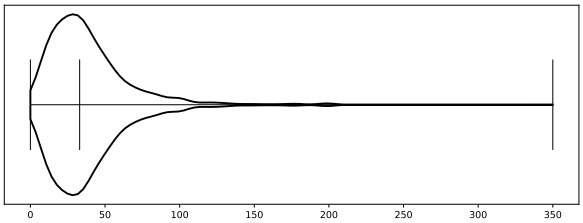
To evaluate the repair time of generate-and-validate methods on C programs, we selected three off-the-shelf repair methods in the experiments, i.e., GENPROG, SPR, and PROPHET.

[1] Experimental data, tools, and results in our study, as well as patch correctness, are publicly available at http://cstar.whu.edu.cn/p/vanfix/.

[2] CODEFLAWS, http://codeflaws.github.io/.



(a) Number of **executable lines of code** in buggy programs. The most common value is 21 lines of code, which appears in 94 buggy programs. The minimum, the median, and the maximum lines of code are 2, 29, and 322.



(b) Number of **test cases in the test suite** in buggy programs. The most common value is 25 test cases, which appears in 111 buggy programs. The minimum, the median, and the maximum numbers of test cases are 1, 33, and 320.

Fig. 2. Violin-plots of the number of executable lines of code and the number of test cases in each buggy programs.

- GENPROG [14], [16] is a pioneering repair tool that evolves the Abstract Syntax Tree (AST) of a buggy program via genetic programming. The implementation of GENPROG returns the first found patch as output [16].
- SPR [24] is a staged repair method with condition synthesis that can find patches from a rich search space. The implementation of SPR can return more than one plausible patches. We refer to such SPR as SPR-ALL. Meanwhile, we define another version of SPR, called SPR-FIRST, which only counts the first plausible patch if SPR-ALL returns two or more patches; that is, for one buggy program, SPR-FIRST outputs zero or one plausible patch, which may be correct or not. Given a buggy program, SPR-ALL can provide no less than plausible or correct patches than SPR-FIRST.
- PROPHET [25] is a probabilistic repair model that learns patch generation from successful human patches. Similar to SPR, PROPHET return more than one plausible patches for one buggy program. We similarly defined PROPHET-ALL and PROPHET-FIRST, respectively.

Existing studies have evaluated the effectiveness of these selected methods [11], [26], [31], [47], [59]. In our study, we followed the same parameter configuration of each selected method [16], [24], [25].

D. Evaluation Setup

Testbed and repair time. The experiment was conducted on four servers with the processor of Intel Core i7 3.60GHz and the memory of 16GB. For all methods under evaluation, we set the execution timeout to one hour, i.e., 3600 seconds. If a method returns its patch before reaching the timeout, we recorded the execution time as the repair time.

Implementation. Besides three off-the-shelf tools of GENPROG, SPR, and PROPHET, all the experiments are implemented with LLVM 4.0.1.[3] The Spearman's rank correlation coefficient and Kullback–Leibler divergence are implemented with the Python Scipy library 1.2.1. [4] The mutation score is calculated with Mull.[5]

Labeling of patch correctness. A repair method can generate and output a patch that makes the whole test suite pass. However, the correctness of a generated patch cannot be automatically checked. In our study, three of the authors, called *checkers*, have manually checked the correctness of each generated patches by all repair methods. The protocol of manually patch checking consists the following steps. First, we automatically compared each generated patch with the manual patch that was recorded in CODEFLAWS. We labeled the exactly same patch as a correct patch (177 correct patches in this step). Second, the rest of generated patches are sent to three checkers; the manual patches are also sent as reference. Each checker is required to individually examine the correctness of generated patches. Then each checker should label the generated patches with one of three candidate labels: *correct*, *incorrect*, or *unknown*. The timeout of labeling one patch is set to 10 minutes: a patch is labeled as *unknown* if the checker cannot make a decision before the timeout. Third, for each generated patch, if three checkers labeled the same *correct* or *incorrect*, we set the final label accordingly; if not, we required the three checkers to discuss and make the decision. After the discussion, if checkers cannot agree with each other or the patch is labeled with *unknown*, we finally labeled the patch with *incorrect*.

Spearman's rank correlation coefficient. We employ Spearman's rank correlation coefficient to measure the correlation between the repair time and a metric of buggy programs, such as the number of test cases or the mutation score. *Spearman's rank correlation coefficient* is widely used to measure the correlation between the rankings of two variables [51]. Spearman's rank correlation coefficient is a nonparametric statistical method that can be configured to be robust to tied values.[6] In the following experimental result, we presented the correlation coefficient and the p-value between each metric and the repair time.

Kullback–Leibler divergence. We used Kullback–Leibler divergence to calculate the distance between two discrete one-dimensional probability distributions, i.e., the relative entropy [51]. The Kullback–Leibler divergence equals to zero indicates that two distributions are identical.

[3]LLVM, http://llvm.org/.

[4]Scipy, http://www.scipy.org/.

[5]Mull, http://github.com/mull-project/mull.

[6]Note that we have not followed Yi et al. [59] to use the Kendall's ranking correlation coefficient. Compared with the Spearman's method, the major difference with the Kendall's method is the support of categorical variables, which are not involved in our study.

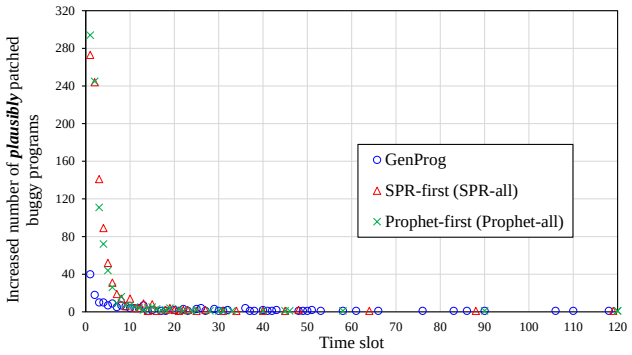


Fig. 3. Increased number of **plausibly** patched buggy programs when the execution time of a repair method increases. Each plot denotes the number of newly patched buggy programs in a new period of 30 seconds. We omit the result of SPR-ALL (or PROPHET-ALL) since its result is identical with SPR-FIRST (or PROPHET-FIRST).

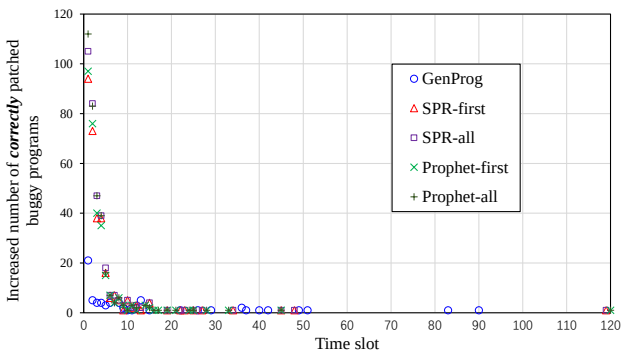


Fig. 4. Increased number of **correctly** patched buggy programs when the execution time of a repair method increases. Each plot denotes a time slot of 30 seconds.

IV. EXPERIMENTAL RESULTS

A. RQ1. How Many Buggy Programs are Newly Patched When the Repair Time Increases?

In generate-and-validate methods, the accumulation of execution time leads to the increase of trials for patch generation. We investigate how many buggy programs can be newly patched if the execution time of a repair method increases.

As mentioned in Section III-D, we set the timeout of executing a repair method to one hour. Fig. 3 illustrates the number of newly patched buggy programs by three repair methods, GENPROG, SPR, and PROPHET. To simplify the explanation, we set one time slot to 30 seconds; that is, each plot in the figure denotes the number of newly patched buggy programs in a new period of 30 seconds.

As shown in Fig. 3, all three methods under evaluation, GENPROG, SPR-FIRST, and PROPHET-FIRST, can increase the number of plausibly patched buggy programs when the repair time increases. In the first time slot (30 seconds), GENPROG plausibly patched 40 buggy programs; SPR-FIRST and PROPHET-FIRST plausibly patched 273 buggy programs and 294 buggy programs, respectively. In the second time slot, the increased numbers for GENPROG, SPR-FIRST, and PROPHET-FIRST are 18, 244, and 245, respectively; in the third time slot, the increased numbers are 10, 141, and 111. However, after 10 time slots (executing for 300 seconds), the increased number of plausibly patched buggy programs sharp decreased to nine or less; after 30 time slots, the increased number is two, one, or zero.

From the illustration in Fig. 3, adding more repair time has not highly increased the number of newly patched buggy programs. For a limited time budget, even executing for 10 time slots, i.e., 5 minutes, can lead to a high proportion of plausibly patched buggy programs, which reaches 62.4% ($= 116/186$), 94.4% ($= 883/935$), and 94.5% ($= 831/879$) for GENPROG, SPR-FIRST, and PROPHET-FIRST, respectively.

A plausible patch may not be correct due to the inadequacy of test oracle in test suite based repair [43], [55]. Thus, we further illustrate the increased number of correctly patched buggy programs over time in Fig. 4.

TABLE I
ACCUMULATED PERCENT OF PATCHED BUGGY PROGRAMS BY EACH REPAIR METHOD WITH INCREASING TIME SLOTS (%)

	Method	Percent of patched buggy programs				
		Top-1	Top-5	Top-10	Top-15	Top-20
Plausibly patched buggy programs	GENPROG	21.51	45.70	62.37	72.04	74.19
	SPR-FIRST	29.20	85.45	94.44	97.22	98.18
	PROPHET-FIRST	33.45	87.14	94.54	96.81	98.07
Correctly patched buggy programs	GENPROG	30.43	53.62	69.57	82.61	82.61
	SPR-FIRST	31.13	85.76	93.71	97.02	97.35
	SPR-ALL	31.07	86.69	94.08	97.34	97.63
	PROPHET-FIRST	32.01	86.80	93.73	96.70	97.69
	PROPHET-ALL	33.14	87.87	94.08	97.04	97.93

In the first time slot (30 seconds), GENPROG correctly patched 21 buggy programs; SPR-FIRST and SPR-ALL correctly patched 94 and 105 buggy programs, respectively; PROPHET-FIRST and PROPHET-ALL correctly patched 97 and 112 buggy programs. In the second time slot, the increased numbers of correctly patched buggy programs are 5, 73, 84, 76, and 83, respectively. However, after 15 time slots (executing for 450 seconds), the increased number in each time slot of correctly patched buggy programs for all repair methods decreased to one or zero; in the final 60 time slots (after 1800 seconds), GENPROG correctly patched two buggy programs while the other four methods only correctly patched one buggy programs.

Table I presents the accumulated percent of patched buggy programs by each repair method. For plausibly patched buggy programs, the first time slot has contributed to over 20%. In top-10 time slots, i.e., 300 seconds, GENPROG accumulates 62% of plausibly patched ones; SPR-FIRST and PROPHET-FIRST contributed to 94%. After the top-20 time slots, i.e., 600 seconds, the percent of plausibly patched buggy programs has accumulated to 98%. For correctly patched buggy programs, top-10 time slots have contributed to 70% to 94% while top-20 time slots have contributed to 83% to 98%. As shown in Table I, increasing the execution time of a repair methods under evaluation cannot highly improve the percent of patched

TABLE II
SPEARMAN'S RANK CORRELATION COEFFICIENT BETWEEN REPAIR TIME AND METRICS FOR **PLAUSIBLY** PATCHED BUGGY PROGRAMS

Method		#Tests	#FailingTests	FailingRatio	ExecLoC	Complexity	MutationScore	Ochiai			Tarantula		
								Score	AbsPos	RelPos	Score	AbsPos	RelPos
GENPROG	Correlation	0.1303	0.0837	-0.0075	0.0853	0.1287	0.0242	-0.1013	0.0917	0.0696	-0.0981	0.0890	0.0703
	p-value	0.0763	0.2560	0.9190	0.2470	0.0800	0.7430	0.1690	0.2130	0.3450	0.1830	0.2270	0.3410
SPR-FIRST	Correlation	0.2185	0.1344	0.0323	0.3769	0.3518	0.2280	-0.0259	0.0943	-0.1013	0.0272	0.0847	-0.0905
	p-value	1.4e-11	3.8e-05	0.3230	6.3e-33	1.3e-28	1.7e-12	0.4280	3.9e-03	1.9e-03	0.4070	9.6e-03	5.6e-03
PROPHET-FIRST	Correlation	0.2212	0.1342	0.0219	0.3333	0.3186	0.1971	-0.0257	0.0943	-0.0816	0.0315	0.0648	-0.0913
	p-value	3.3e-11	6.5e-05	0.5170	3.0e-24	3.4e-22	3.8e-09	0.4470	5.1e-03	0.0155	0.3510	0.0548	6.8e-03

TABLE III
SPEARMAN'S RANK CORRELATION COEFFICIENT BETWEEN REPAIR TIME AND METRICS FOR **CORRECTLY** PATCHED BUGGY PROGRAMS

Method		#Tests	#FailingTests	FailingRatio	ExecLoC	Complexity	MutationScore	Ochiai			Tarantula		
								Score	AbsPos	RelPos	Score	AbsPos	RelPos
GENPROG	Correlation	0.4027	0.2177	-0.0337	-0.0263	0.1406	-0.0557	-0.1079	0.0244	0.0630	-0.1225	-0.0108	0.0215
	p-value	6.0e-04	0.0723	0.7835	0.8300	0.2491	0.6495	0.3777	0.8420	0.6073	0.3161	0.9297	0.8607
SPR-ALL	Correlation	0.1978	0.2349	0.1564	0.4449	0.3731	0.2305	0.1220	0.1396	-0.0969	0.0381	0.0420	-0.1562
	p-value	2.5e-04	1.3e-05	4.0e-03	7.9e-18	1.3e-12	1.9e-05	0.0249	0.0102	0.0751	0.4846	0.4420	4.0e-03
SPR-FIRST	Correlation	0.1939	0.2744	0.2036	0.4746	0.4066	0.2480	0.1629	0.1612	-0.0983	0.0210	0.0357	-0.1793
	p-value	7.1e-04	1.3e-06	3.7e-04	2.3e-18	1.9e-13	1.3e-05	4.5e-03	5.0e-03	0.0882	0.7156	0.5362	1.8e-03
PROPHET-ALL	Correlation	0.2078	0.2205	0.1411	0.4411	0.3978	0.2342	0.1426	0.1535	-0.0868	0.0674	0.0601	-0.1429
	p-value	1.2e-04	4.3e-05	9.4e-03	1.6e-17	2.9e-14	1.4e-05	8.7e-03	4.7e-03	0.1113	0.2165	0.2704	8.5e-03
PROPHET-FIRST	Correlation	0.2071	0.2615	0.1841	0.4647	0.4088	0.2387	0.1934	0.1577	-0.1061	0.0733	0.0599	-0.1565
	p-value	2.8e-04	3.9e-06	1.3e-03	1.2e-17	1.2e-13	2.7e-05	7.1e-04	5.9e-03	0.0652	0.2034	0.2991	6.3e-03

buggy programs.

Finding 1. The number of newly patched buggy programs does not highly increase when the execution time increases. For the repair methods under evaluation, after top 20 time slots, i.e., 600 seconds, correctly patches buggy program are hardly increased.

B. RQ2. Which Metric Correlates with the Time of Plausibly Patching a Buggy Program?

We explored which factor correlates to the repair time of plausible patched buggy programs. To examine potential factors, we introduce 12 metrics that may relate the repair time as follows:

- *#Tests* is the number of test cases in a given test suite. *#FailingTests* is the number of failing test cases in the test suite. *FailingRatio* is the ratio that *#FailingTests* divides *#Tests*.
- *ExecLoC* is the number of executable lines of source code in a buggy program. Blank lines or lines with only comments are removed from the calculation of *ExecLoC*.
- *Complexity* of the buggy program is measured with the Cyclomatic complexity, which is a software metric of linearly independent paths [32].
- *MutationScore* measures how many versions of mutated programs can be detected by the given test suite [7].
- Two widely used techniques of fault localization, Ochiai [44] and Tarantula [8] are used. For either Ochiai or Tarantula, we explored the impact of *Score* (i.e., the suspicious score of the faulty statement) and *AbsPos* and *RelPos* (the absolute position and the relative position of the faulty statement in the ranking, respectively). For two or more statements with the same suspicious scores in the ranking, we followed existing techniques [12], [19], [58] to use the median position.

Table II present the Spearman's rank correlation coefficient (see Section III-D) between the repair time of plausibly patched buggy programs with 12 defined metrics. As shown in Table II, the repair time of SPR-FIRST and PROPHET-FIRST correlates with *ExecLoC* and *Complexity* with the coefficient over 0.30 and the statistically significance (we consider p-value < 0.05 statistically significant in this paper). The repair time of SPR-FIRST and PROPHET-FIRST also correlates with *#Test* and MutationScore the coefficient between 0.20 to 0.30. In the correlation result of GENPROG, there exist coefficients for *#Test*, *#Complexity*, and *Ochiai score* over 0.10.

Finding 2. The repair time of generating plausible patches for buggy programs correlates with the number of executable lines of code and the Cyclomatic complexity. The number of test cases in the given test suite also correlates with the repair time.

C. RQ3. Which Metric Correlates with the Time of Correctly Patching a Buggy Program?

We further investigate the correlation between the repair time of correctly patched buggy programs and the 12 metrics. Table III shows the correlation of five methods under evaluation. The repair time of SPR-FIRST, PROPHET-FIRST, SPR-ALL, and PROPHET-ALL correlates with *ExecLoC* and *Complexity* with the coefficient over 0.35 and the statistically

TABLE IV
KULLBACK–LEIBLER DIVERGENCE OF METRIC DISTRIBUTIONS BETWEEN **PLAUSIBLY PATCHED AND UNPATCHED** BUGGY PROGRAMS

Method	#Tests	#FailingTests	FailingRatio	ExecLoC	Complexity	MutationScore
GENPROG	0.4714	0.3457	0.5523	0.3962	0.2626	0.3522
SPR-FIRST	0.3207	0.3783	0.2859	0.4289	0.2161	0.1403
PROPHET-FIRST	0.2685	0.3821	0.2719	0.4614	0.2221	0.1099

TABLE V
KULLBACK–LEIBLER DIVERGENCE OF METRIC DISTRIBUTIONS BETWEEN **CORRECTLY AND INCORRECTLY PATCHED** BUGGY PROGRAMS

Method	#Tests	#FailingTests	FailingRatio	ExecLoC	Complexity	MutationScore
GENPROG	3.4039	3.9393	5.0790	2.9518	0.5854	0.6781
SPR-ALL	0.6162	1.2074	0.7855	0.5745	0.2841	0.4462
SPR-FIRST	0.6386	0.7255	0.7143	0.5991	0.2753	0.4959
PROPHET-ALL	0.8835	1.2314	0.7078	0.6252	0.2768	0.4630
PROPHET-FIRST	0.9315	0.9200	0.6413	0.6569	0.2648	0.5109

significance; The repair time of GENPROG correlates with the number of test cases with the coefficient of 0.40. In GENPROG, *#FailingTests* also correlates with the repair time with the coefficient of 0.22. Besides GENPROG, the other four methods correlate with *#Test*, *#FailingTests*, and *MutationScore* with the coefficient over 0.20.

The correlation analysis between the repair time and the mutation score is -0.06 for GENPROG and 0. 23 to 0.25 for SPR and PROPHET. This result is inconsistent with the correlation in existing work [59]. Yi et al. indicated that there is no conclusive result about the correlation between mutation score and the repair time (RQ4, *p*. 2966, in [59]). One possible reason for such inconsistency is the number of buggy programs under evaluation: our work used 2980 buggy programs and the work by Yi et al. [59] used 142 buggy programs. Different datasets may result in the change of correlation analysis.

Finding 3. The repair time of generating correct patches for buggy programs correlates with the number of executable lines of code, the Cyclomatic complexity, the number of test cases, the number of failing tests, and the mutation score. We conclude that the repair time of finding out a correct patch can be affected by the scale and the complexity of the source code as well as the test suite.

The observation in Table II and Table III motivates us to further investigate the difference between plausibly patched buggy programs and unpatched ones. We evaluate whether the distribution of each metric between plausibly patched buggy programs and unpatched ones are different. Table IV presents the distance between plausibly patched unpatched buggy programs based on Kullback–Leibler divergence (see Section III-D). A smaller divergence (no less than zero) indicates that a lower distance; the divergence of zero means two distributions are the same. For all methods under evaluation, plausibly patched buggy programs and the unpatched buggy programs have different distributions for each given metric, such as *#Tests*. The values for *MutationScore* in SPR-FIRST and PROPHET-FIRST are the lowest. This observation reveals that it is possible to directly isolate patchable buggy programs with unpatchable ones with specific metrics or their combinations.

We further explore the difference between the distributions of correctly and incorrectly patched buggy programs among all plausibly patched ones. Table V presents whether the distribution of each metric between correctly patched buggy programs and incorrectly ones are identical. Only plausible patched buggy programs are considered in this table. Table V shows similar results to Table IV: among all plausibly patched buggy programs, correctly patched buggy programs and uncorrectly patched buggy programs have different distributions for each given metric. The distribution distance for each metric of SPR or PROPHET is lower than that of GENPROG.

Finding 4. We employed Kullback–Leibler divergence to measure the distribution difference of a specific metric between plausibly patched buggy programs and unpatched ones (or between correctly patched buggy programs and incorrectly ones. We observed that selected metrics in our study could be used to isolate the plausible ones or the correct ones. This requires further study in the future.

D. RQ4. How Effective can Existing Methods Patch Buggy Programs in an Hour?

The calculation of repair time highly relates to the effectiveness of a repair method, i.e., how many plausible patches are actually correct.

TABLE VI
EFFECTIVENESS OF REPAIR METHODS UNDER EVALUATION

Method	#Plausible	#Correct	Precision	Recall	F-measure
GENPROG	186	71	0.3817	0.0238	0.04489
SPR-FIRST	935	302	0.3230	0.1013	0.15416
SPR-ALL	935	338	0.3615	0.1133	0.17254
PROPHET-FIRST	879	303	0.3447	0.1016	0.15691
PROPHET-ALL	879	338	0.3845	0.1133	0.17504

Table VI shows the effectiveness of patch generation with *#Plausible*, i.e., the number of plausibly patched buggy programs, *#Correct*, i.e., the correctly patched buggy programs, and the precision, recall, F-measure measurements [56] that are defined accordingly.

- *Precision* is defined as $\frac{\#Correct}{\#Plausible}$;
- *Recall* is defined as $\frac{\#Correct}{\#All}$, where $\#All$ denotes the number of all buggy programs under evaluation;
- F-measure is defined as $\frac{2 \times precision \times recall}{precision + recall}$.

As shown in Table VI, the precision value of all five methods under evaluation ranges from 0.32 to 0.38; that is, all methods can find out over 30% of correctly patched buggy programs from plausibly ones. The recall value, i.e., the ratio of correctly patched ones among all buggy programs, ranges from 0.02 to 0.11. SPR-ALL can correctly patched more buggy programs than SPR-FIRST because SPR-FIRST identifies the first output patch as its only output (in Section III-C); PROPHET-ALL and PROPHET-FIRST behave in a

similar way. Thus, the F-measure ranges from 0.04 to 0.18. We can observe that SPR-FIRST and PROPHET-FIRST reach the similar effectiveness with F-measure of 0.17; SPR-ALL and SPR-ALL also behave similarly.[7]

Finding 5. Each of repair methods under evaluation correctly patched 32% to 38% of plausibly patched buggy programs. Among all buggy programs, GENPROG correctly patched 2% while SPR and PROPHET correctly patched around 11%.

V. PROPOSED REPAIR METHOD – VANFIX

As mentioned in Table II and Table III, the repair time correlates with *ExecLoC* (i.e., the number of executable lines of code) and the Cyclomatic complexity of a buggy program. That is, a buggy program with long and complex source code may require long time for patch generation. Motivated by such correlations, we designed a new repair method VANFIX.

A. Design of VANFIX

The key idea of the proposed repair method VANFIX is to reduce the correlation between repair time and the ExecLoC or the complexity. Similar to GENPROG, VANFIX is a pool-based search algorithm that iteratively mutates the program until generating a plausible patch. The major difference with GENPROG is that VANFIX leverages the probability of exploring the search space to conduct a variable search neighborhood for potential patches, rather than patching suspicious statements one by one. Algorithm 1 shows the steps of VANFIX. The output of VANFIX is one or zero plausible patch for the buggy program. We present three major techniques in the design of VANFIX as follows.

1) Probability guided search: VANFIX does not focus on one statement in each iteration of program mutation; instead, VANFIX leverages the probability distribution of suspicious scores to select different statements inside one iteration. As shown at Line 8 in Algorithm 1, in each iteration, VANFIX selects the i-th statement $stmt_i$ according to the probability p_i. Comparing with mutating statements case by case [16], VANFIX can randomly walk through different statements to explore potential patches for each candidate statement. Since existing techniques of fault localization may not rank the actually faulty position to the top [22], [61], the probability guided search can enhance the chance of finding patches in the early stage.

2) Variable search neighborhood: In Section V-A1, crossly searching patches for each statement can result in the possibility of failing in generating patches for the statement with the highest suspicious score. Therefore, VANFIX updates the probability of mutating the program based on the current search result. We referred to the set of potential patches as the neighborhood [4]. Then updating probabilities of patch generation can dynamically change the search space. We employ such dynamic change to enhance the chance of finding patches. This conducts the variable search neighborhood of patch generation via probability updating [36], [40].

[7] We note that GENPROG is open available since 2009 [53], SPR is since 2015 [24], and PROPHET is since 2016 [25]. The comparison of execution time among repair methods may be unfair to methods since the early stage.

Input: buggy program $prog$, test suite $suite$, fault localization technique FL, change step $step$ ($0 < step < 1$), size of the variant pool $size$
Output: $patch$

```
   /* Preparation.                                          */
1  Execute fault localization on prog with suite and
     compute suspicious score s_i with FL for each
     statement stmt_i in prog (1 ≤ i ≤ n);
2  Compute probability p_i for stmt_i, p_i = s_i / Σ_{i=1}^{n} s_i;
3  Uniformly generate probability q_j of selecting
     available mutation operator oper_j (1 ≤ j ≤ m),
     q_j = 1/m;
4  while not timeout and patch is not found do
       /* Initialization.                                 */
5      pool = Φ;
6      Clean binary flags flag_i^{stmt} = 0, flag_j^{oper} = 0;
       /* Patch generation.                               */
7      while |pool| ≤ size and patch is not found do
8          Select stmt_i according to probability p_i;
9          Generate variant v_{i,j} by applying oper_j to
             stmt_i according to probability q_j;
10         flag_i^{stmt} = 1, flag_j^{oper} = 1;
11         Execute test suite suite on v_{i,j};
12         if v_{i,j} passes suite then patch = v_{i,j} ;
13     end
       /* Probability update.                             */
14     for i ← 1 to n do
15         p'_i = p_i − step * flag_i^{stmt};
16         if p'_i < 0 then p'_i = 0;
17     end
18     for i ← 1 to n do p_i = p'_i / Σ_{i=1}^{n} p'_i ;
19 end
```

Algorithm 1: Algorithm of VANFIX, a repair method based on variable search neighborhood.

Each statement is mutated based on the probability of selecting statements, i.e., p'_i at Line 15 in Algorithm 1. The probability p'_i is reduced by a pre-defined constant $step$ if the statement has been selected but fails in patch generation. Note that p'_i is then normalized at Line 18. The reason for reducing the probability of selected statements is that a statement is tried without patch generation should be tentatively abandoned.

3) Mutation on both statements and expressions: VANFIX employs two types of mutation operators to generate patches: statement mutation and expression mutation.

Statement mutation. We followed GENPROG [17] to configure STMTINSERT, STMTREMOVE, and STMTREPLACE operators of statements in VANFIX (STMT and EXPR are short for the operator for statements and expressions, respectively). To speed up the processing of statements, we also added a STMTMOVE operator, which is the directly combination of STMTREMOVE and STMTINSERT.

TABLE VII
SPEARMAN'S RANK CORRELATION COEFFICIENT BETWEEN REPAIR TIME OF VANFIX AND METRICS FOR PLAUSIBLY AND CORRECTLY PATCHED BUGGY PROGRAMS

VANFIX		#Tests	#FailingTests	FailingRatio	ExecLoC	Complexity	MutationScore	Ochiai			Tarantula		
								Score	AbsPos	RelPos	Score	AbsPos	RelPos
Plausibly patched buggy programs	Correlation	0.1140	-0.1415	-0.2221	0.3131	0.2764	0.2246	-0.1275	0.0794	-0.0741	0.1842	0.1892	0.0387
	p-value	3.6e-03	2.9e-04	9.9e-09	2.7e-16	6.8e-13	6.7e-09	1.1e-03	0.0427	0.0587	2.2e-06	1.1e-06	0.3243
Correctly patched buggy programs	Correlation	0.0577	-0.1087	-0.1649	0.3288	0.2918	0.2205	-0.1223	0.0687	-0.0852	0.1453	0.1664	0.0119
	p-value	0.2451	0.0281	8.3e-04	9.6e-12	1.9e-09	6.9e-06	0.0134	0.1662	0.0856	3.3e-03	7.4e-04	0.8099

TABLE VIII
EFFECTIVENESS OF VANFIX AND DIFFERENCE WITH OTHER REPAIR METHODS UNDER EVALUATION

	Method	#Patched	#Correct	Precision	Recall	F-measure
	VANFIX	653	408	0.6248	0.1368	0.2244
Improvement	VANFIX - GENPROG	467	337	0.2431	0.1130	0.1796
	VANFIX - SPR-FIRST	-282	106	0.3018	0.0355	0.0703
	VANFIX - SPR-ALL	-282	70	0.2633	0.0235	0.0519
	VANFIX - PROPHET-FIRST	-226	105	0.2801	0.0352	0.0675
	VANFIX - PROPHET-ALL	-226	70	0.2403	0.0235	0.0494

Expression mutation. We configured mutation operators for expressions, including EXPRREPLACE (e.g., replacing `a && b` with `a || b`), EXPRREVERSE (e.g., reversing `c >= 0` with `!(c >= 0)`), TYPEEXPAND (e.g., expanding `int d` with `long d`), and SIZEEXPAND (e.g., expanding `int e[10]` with `int e[100]`).

In VANFIX, given one statement, several mutation operators may be not available and have the probability of zero. For instance, for a statement `a = 0;`, the expression mutation operator of EXPRREVERSE is not available and its probability can be set to zero. Each mutation operator with the probability over zero is set to q_j at Line 3 in Algorithm 1.

B. Evaluation on VANFIX

We evaluate VANFIX based on the experimental setup in Section III-B. We set parameters of VANFIX as follows: $step$ is set to 0.01 and $size$ is set to 40, and FL is Ochiai [44].

In Table VII, we followed Table II to show the correlation between each metric and the repair time. The repair time of plausibly and correctly patched buggy programs are listed. VANFIX has lower correlations with *ExecLoC* and *Complexity* than SPR and PROPHET in Table II. VANFIX also has lower correlations with *ExecLoC* and *Complexity* than SPR and PROPHET in Table III.

We also evaluated the effectiveness under the same experimental setup as Section IV-D. Table VIII shows the precision, recall, and F-measure of VANFIX. VANFIX can generate plausible patches for 653 buggy programs, which contains 408 patched buggy programs. This leads to the precision of 0.62. That is, VANFIX reaches the highest precision, which achieves 24% to 30% better than existing methods GENPROG, SPR, and PROPHET. We can conclude that 62% of patched buggy programs by VANFIX are correct.

As shown in Table VIII, the recall of VANFIX is 0.13, which is 0.02 to 0.11 better than other methods. The F-measure of VANFIX is 0.22, which achieves 0.05 to 0.18 better than other methods.

Finding 6. VANFIX correctly patched 408 out of 653 plausibly patched buggy programs. This leads to the precision of 0.62; that is, 62% of patched buggy programs by VANFIX are correct.

VI. THREATS TO VALIDITY

We present the threats to the validity of our study in three categories.

Threats to construct validity. We manually examined all plausible patches to evaluate the correctness of patch generation (Section III-D). This manual examination consists of three major steps: automatically matching, individually labeling, and final discussion. However, similar to any manual labeling of patch correctness [26], [27], there exists a threat that the manual labeling involves mistakes due to the bias of human knowledge. Meanwhile, we have extracted 12 metrics to evaluate the correlation with the repair time. Among software metrics, there is a large number of metrics that may relate to the program repair, e.g., the statement coverage or the capable ratio of test cases in [59]. We cannot exhaustively examine all potential metrics. In our study, the 12 selected metrics can be viewed as a subset of factors under evaluation.

Threats to internal validity. In our evaluation, three existing repair methods are employed to evaluated the repair time of C programs, including GENPROG, SPR, and PROPHET. Existing methods such as AE [52] and TrpAutoRepair [41] can also be used to evaluate the repair method. Our future work will include other existing methods to extend the study. In this study, repair methods GENPROG and VANFIX relies on the iterative search based on random seeds. The experiment should be conducted for several times to avoid the disturbance by the random seeds. However, our experiment sets the timeout of repairing one buggy program to one hour; that is, a repair method continuously executes for an hour on a server (in Section III-D) unless a plausible patch is generated before reaching the timeout. This leads to extremely long time of

execution, i.e., around 45 days on four servers. As mentioned in Section I, we do not aim at comparing the repair time among different methods. Therefore, the number of 2980 buggy programs can reduce the disturbance of randomness.

Threats to external validity. This study is conducted on buggy programs from the CODEFLAWS benchmark. As shown in Fig. 2, the median of the number of executable lines of code is 29; that is, most of buggy programs are small-scale ones. In Section III-B, we have claimed two reasons for choosing such small-scale programs: leveraging the large number of buggy programs to support the statistics and to reduce the risk in manual evaluation of patch correctness. This adds the threat that our experimental result may be not generalized to other datasets. Since our study has no theoretical guarantee, changing repair methods or datasets may lead to different observation or results.

VII. RELATED WORK

We list the related work to our study in three categories: program repair methods, empirical studies, and search space of patch generation.

Program repair methods. In this paper, we have introduced many existing works of automated program repair, including generate-and-validate methods [10], [16], [24], [25], [30], [42], [52], [54], [56] and synthesis-based methods [34], [38], [57].

For the sake of space, we introduced several recent works on program repair as follows. Liu et al. [23] have proposed the Avatar system to generate patches via leveraging the fix pattern of static analysis violations as fix ingredients in the generation process. Avatar generates correct patches to fix 36 out of 39 buggy programs in Defects4J. Hua et al. [5] designed the SketchFix approach, which utilizes lazy candidate patch generation to reduce the number of re-compilations and re-executions of candidate fixes. SketchFix can correctly fix 19 out of 357 buggy programs within 23 minutes on average in Defects4J. To automatically fix concurrency buggy programs, Lin et al. [20] proposed PFix to fix concurrency buggy programs by inferring locking policies. PFix can fix 19 out of 23 concurrency buggy programs and achieves better results than the state-of-the-art tool in concurrency bug fixing. Recent work by Lee et al. [18] presents MemFix, an automated technique for fixing memory deallocation errors in C programs. MemFix can fix all errors related to the memory leak in the Juliet dataset. Gopinath et al. [3] presented a data-driven approach for automatically repairing buggy programs in the selection condition of database. Tan et al. [49] employed the evolutionary search to generate patches for Android crashes. Stocco et al. [48] focused on visual analysis of web applications and designed Vista to support automated repair of web test breakages.

Empirical studies. Empirical studies are widely conducted to understand the repair behaviors of automated program repair. Martinez et al. [31] and Barr et al. [1] examined the redundancy of patch generation to investigate whether generate-and-validate methods can be applied to real-world patch generation. Le Goues et al. [15] have conducted two datasets of buggy programs and evaluated the performance of existing repair methods. Martinez et al. [27] have employed the Defects4J dataset to empirically explore the results of fixing Java bugs. Liu et al. [21] examined real-world patches with a detailed analysis. Saha et al. [45] have conducted a study to understand the process of assigning and fixing the long lived bugs.

Many studies have focus on the patch overfitting problem in generation process. Yu et al. [60] have analyzed the overfitting problem and classified the issues into incomplete fixing and regression introduction. Xiong et al. [55] proposed a novel method to automatically identify the overfitting issues of generated patches. As mentioned in Section II-B, Yi et al. [59] have investigated the correlation between the reliability of patch generation and test suite metrics.

This paper leverages the correlation techniques to analyze the execution time of existing repair methods. Different from the correlation with patch generation in [59], we aim at understanding factors that affect the repair time and have designed a simple and effective method, VANFIX.

Search space of patch generation. As mentioned in Section II-B, the search space of generate-and-validate methods leads to the potentials of patch generation [6], [28], [29], [39]. Mechtaev et al. [33] proposed a method of patch generation based on the test-equivalence relation to reduce the number of test executions. Wen et al. [54] propose CapGen, a context-aware patch generation technique to reduce the huge search space. Mehne et al. [35] have recently proposed two techniques, location selection and test case pruning to accelerating the generate-and-validate repair.

Our proposed method VANFIX also leverage the expansion of the search space to enlarge the probability of finding out correct patches. VANFIX can be considered as searching patches in variable search neighborhood of expressions and statements.

VIII. CONCLUSION

In this paper, we investigated the execution time of patch generation by existing automated program repair methods, including GENPROG, SPR, and PROPHET. Our experimental results on 2980 buggy programs in the CODEFLAWS benchmark indicate that adding the repair time cannot highly enlarge the number of patched buggy programs. Meanwhile, the repair time of plausibly or correctly patched buggy programs correlates to *ExecLoC*, i.e., the number of executable lines of code and the code complexity. To weaken such correlation, we designed a new repair method with variable search neighborhood, which can dynamically update the search probability for effective patch generation.

In future work, we plan to investigate the repair time and the design of repair methods to enhance existing repair methods. We aim to further analyze the program behavior to understand the drawbacks and the potentials of current generate-and-validate repair methods. Conducting a large dataset based on continuous code changes is also one of the future works.

REFERENCES

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

[2] F. M. Delfim, S. Urli, M. de Almeida Maia, and M. Monperrus. BEARS: an extensible java bug benchmark for automatic program repair studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 468–478, 2019.

[3] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 243–253, 2014.

[4] P. Hansen, N. Mladenovic, and J. A. Moreno-Pérez. Variable neighbourhood search: methods and applications. *Annals OR*, 175(1):367–407, 2010.

[5] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 888–891, 2018.

[6] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 12–23, 2018.

[7] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering ASE 2005, November 7-11, Long Beach, CA, USA*, pages 273–282, 2005.

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

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

[11] X. Kong, L. Zhang, W. E. Wong, and B. Li. Experience report: How do techniques, programs, and tests impact automated program repair? In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 194–204, 2015.

[12] T. B. Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 177–188, 2016.

[13] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 213–224, 2016.

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

[15] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256, 2015.

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

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

[18] J. Lee, S. Hong, and H. Oh. Memfix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 95–106, 2018.

[19] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *PACMPL*, 1(OOPSLA):92:1–92:30, 2017.

[20] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei. Pfix: fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 589–600, 2018.

[21] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. L. Traon. A closer look at real-world patches. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 275–286, 2018.

[22] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of ICST 2019, Xi'an, China, 2019*, 2019.

[23] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 456–467, 2019.

[24] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178, 2015.

[25] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312, 2016.

[26] F. Long and M. C. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 702–713, 2016.

[27] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[28] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[29] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, pages 65–86, 2018.

[30] M. Martinez and M. Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, 151:65–80, 2019.

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

[32] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[33] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury. Test-equivalence analysis for automatic patch generation. *ACM Trans. Softw. Eng. Methodol.*, 27(4):15:1–15:37, 2018.

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

[35] B. Mehne, H. Yoshida, M. R. Prasad, K. Sen, D. Gopinath, and S. Khurshid. Accelerating search-based program repair. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 227–238, 2018.

[36] N. Mladenović, J. Petrović, V. Kovačević-Vujčić, and M. Čangalović. Solving spread spectrum radar polyphase code design problem by tabu search and variable neighbourhood search. *European Journal of Operational Research*, 151(2):389–399, 2003.

[37] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[38] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013*

International Conference on Software Engineering, pages 772–781. IEEE Press, 2013.

[39] T. Nguyen, W. Weimer, D. Kapur, and S. Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 301–318, 2017.

[40] J. Pei, Z. Dražić, M. Dražić, N. Mladenović, and P. M. Pardalos. Continuous variable neighborhood search (c-vns) for solving systems of nonlinear equations. *INFORMS Journal on Computing*, 2019.

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

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

[43] Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.

[44] A. Rui, P. Zoeteweij, and A. J. C. V. Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation. Taicpart-Mutation*, pages 89–98, 2007.

[45] R. K. Saha, S. Khurshid, and D. E. Perry. Understanding the triaging and fixing processes of long lived bugs. *Information & Software Technology*, 65:114–128, 2015.

[46] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 10–13, 2018.

[47] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[48] A. Stocco, R. Yandrapally, and A. Mesbah. Visual web test repair. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 503–514, 2018.

[49] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in android apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 187–198, 2018.

[50] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 180–182, 2017.

[51] R. E. Walpole, S. L. Myers, K. Ye, and R. H. Myers. *Probability and statistics for engineers and scientists*. Pearson, 2007.

[52] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 356–366, 2013.

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

[54] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1–11, 2018.

[55] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 789–799, 2018.

[56] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 2017*, 2017.

[57] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Software Eng.*, 43(1):34–55, 2017.

[58] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 2014*, pages 191–200, 2014.

[59] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering*, 23(5):2948–2979, 2018.

[60] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24(1):33–67, 2019.

[61] Y. Yuan and W. Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[62] L. Zemín, S. G. Brida, A. Godio, C. Cornejo, R. Degiovanni, G. Regis, N. Aguirre, and M. F. Frias. An analysis of the suitability of test-based patch acceptance criteria. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 14–20, 2017.

[63] H. Zhong and Z. Su. An empirical study on real bug fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 913–923, 2015.