Using Automated Program Repair for Evaluating the
Effectiveness of Fault Localization Techniques

Yuhua Qi, Xiaoguang Mao∗, Yan Lei, and Chengsong Wang
School of Computer
National University of Defense Technology, Changsha, China
{yuhua.qi, xgmao, yanlei}@nudt.edu.cn jameschen186@gmail.com

ABSTRACT

Many techniques on automated fault localization (AFL) have been introduced to assist developers in debugging. Prior studies evaluate the localization technique from the viewpoint of developers: measuring how many benefits that developers can obtain from the localization technique used when debugging. However, these evaluation approaches are not always suitable, because it is difficult to quantify precisely the benefits due to the complex debugging behaviors of developers. In addition, recent user studies have presented that developers working with AFL do not correct the defects more efficiently than ones working with only traditional debugging techniques such as breakpoints, even when the effectiveness of AFL is artificially improved.

In this paper we attempt to propose a new research direction of developing AFL techniques from the viewpoint of fully automated debugging including the program repair of automation, for which the activity of AFL is necessary. We also introduce the NCP score as the evaluation measurement to assess and compare various techniques from this perspective. Our experiment on 15 popular AFL techniques with 11 subject programs shipping with real-life field failures presents the evidence that these AFL techniques performing well in prior studies do not have better localization effectiveness according to NCP score. We also observe that Jaccard has the better performance over other techniques in our experiment.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; D.2.3 [Software Engineering]: Software Engineering—Coding Tools and Techniques

General Terms

Experimentation, Measurement

Keywords

Fault localization, automated program repair, automated debugging

1. INTRODUCTION

Automated fault localization (AFL) techniques are developed to reduce the effort of software debugging, which is well known to be frustrating, and often time-consuming [34]. Rather than attempting to pinpoint and suggest a fix for a bug, most existing techniques on AFL give the programmer suggestions about likely fault locations, such as a suspect list (ranking program entities according to their likelihood of containing the fault) produced by statistical approaches, or a small fraction of code (to which failure cause is narrowed down) generated by experimental approaches [25]. Since lots of techniques have been presented in the past decade, one popular research area is to evaluate and compare the effectiveness of various AFL techniques. In addition, when a novel AFL technique is proposed, it is a common practice to evaluate the effectiveness of this technique, and present the advantage by comparing it with other AFL techniques [3].

Much existing literature [32, 20, 3, 14] evaluates the effectiveness of AFL from the viewpoint of developers, that is, measuring how many benefits that developers can obtain from the AFL technique used when debugging. To quantify the benefits, much existing work evaluates the effectiveness according to the percentage of the program code that needs to be examined before the faults are identified, which is referred to as the EXAME score [32, 3, 33]. Obviously, a lower EXAME score indicates a better localization effectiveness.

This kind of evaluation measurement, however, is based on the strong assumption that developers are always suitable in practice, because understanding the root cause of a failure typically involves complex activities [21]. Hence, existing evaluation approaches are not always suitable in practice due to this strong assumption.

What is worse, given the maturity of the field of AFL, the fact that most developers are still unwilling to debug faulty programs through existing fault localizers [34] results in the natural doubt about the rationality of current evaluation measurement. Actually, to our knowledge most developers are more likely to proceed using traditional debugging such as breakpoints (which can provide data values for interesting statements to assist fault understanding), and then analyze the possible reason through complex human experience abil-
ity. This mechanism is most probably much more efficient, when compared to debugging through long lists of unrelated suspicious locations with small chances of spotting the defect. Furthermore, the real-life user studies presented in [21] has suggested that statistical approaches were no more effective than traditional debugging for complex tasks, even when using an artificially-high rank. The current evaluation approach preferring a lower EXAM to benefit the developers (most of who, in practice, does not use AFL techniques at all so far), however, still guides many researchers to just focus on statement selection and ranking.

In contrast, for fully automated debugging including the program repair of full automation [11], the activity of AFL is most often necessary especially when automating the repair of large-scale programs. Recent advances in the area of automated program repair [16, 30, 5] enable the faulty programs to be automatically repaired by modifying some statements according to specified repair rules. For a concrete faulty program, the effectiveness of AFL techniques, which is used to track down the possible faulty statements, will be the dominant factor affecting the repair effectiveness when the repair rules have been already specified.

Although recent work [16, 30, 5] on automated program repair has noticed the important influence caused by the AFL activity, there is no detail discussion on this kind of information. For instance, GenProg [17, 16], a state-of-the-art tool for automated C program repair, uses one very simple scheme to compute the weight value, which is similar to the term of suspiciousness value in statistical localization techniques, for each statement of target program; the probability of each statement being selected for modification depends on its weight value. Although the authors in most recent work on GenProg [16] declared that “Other fault localization schemes could potentially be plugged directly into GenProg”, they did not discuss the effectiveness difference between popular AFL techniques such as Tarantula and Jaccard.

Given the increasing advances on automated program repair and the distinct way of using localization information against developers (see Section 3.2), developing the AFL technique aiming to assist effectively program repair (not developers) is pressing. As the first step, in this paper we present an evaluation measurement to evaluate the effectiveness of AFL from the viewpoint of fully automated debugging (including the repair process of full automation). Specifically, we evaluate the effectiveness of one AFL technique according to the repair effectiveness guided by the localization technique. We refer to the repair effectiveness as the number of candidate patches generated before a valid patch is found in the repair process (NCP); the lower NCP score, the better effectiveness of the used AFL technique. Based on NCP, we can justify the advantage of one specified AFL technique by presenting the improved effect size over other techniques through rigorous statistic approach such as A-test.

For maximum applicability we built GenProg-FL, a tool which can automatically repair faulty programs like GenProg and have the ability of accepting the guidance of existing AFL techniques, and then evaluated the effectiveness of 15 popular AFL techniques by running GenProg-FL to repair 11 real-life programs coming with field failures [13], the failures occur after deployment. The result data indicate that we can identify the effectiveness difference between various AFL techniques according to the NCP score. We also observed that 1) Ochiai did not always perform better over either Jaccard or Tarantula, which is inconsistent with prior empirical studies [1, 2] neither Optimal_p or Russel_Rao performed better over other 13 techniques, although the two techniques have been theoretically proven to be optimal under the evaluation measurement of EXAM [32, 20].

Overall, the contributions of this paper can be summarized as follows:

- The first to present the effectiveness evaluation for AFL techniques from the viewpoint of fully automated debugging (Section 3). Unlike developers who are more likely to prefer traditional debugging techniques such as breakpoints, for automated program repair (at the source code level) the activity of tracking down the faulty code snippet is most often necessary. Thus, developing the corresponding AFL techniques to improve the repair effectiveness is more pressing.

- The evaluation measurement in term of NCP for evaluating and comparing the effectiveness of various AFL techniques (Section 3.4). A lower NCP score indicates a better localization effectiveness.

- An effectiveness evaluation of 15 popular AFL techniques by running GenProg-FL to automatically repair 11 subject programs shipping with real-world field failures (Section 4). Result data show that the techniques that perform well in existing studies do not have the lower NCP score, justifying the necessity of studying the AFL from the viewpoint of fully automated debugging. In addition, we also find that Jaccard performs at least as good as the other 14 investigated techniques, with the effectiveness improvement up to “large” effect size using A-test.

2. BACKGROUND

2.1 Automated Fault Localization (AFL)

AFL techniques are originally proposed to provide some information to reduce the effort of debugging activity. Although there are many techniques on AFL presented in the past decades, these techniques can be divided into two categories – statistical approaches and experimental approaches [25]. Statistical approaches, also called spectrum-based approaches [15, 19, 20, 32], rank each program entity according to their suspiciousness of being faulty. Although the program entity can be referred to as different terms such as statement, branch, and basic block, the term of statement is most often studied. The suspiciousness value of each program entity is computed by analyzing the program profile, which can be observed by executing a large number of failed and passed test cases. Based on either different designs of suspiciousness computation formulas under the same program profile or the different program profiles collected, many AFL techniques on statistical approaches, such as Tarantula [15], Jaccard [9], and predicate-based technique [10], are introduced.

Unlike statistical approaches requiring lots of testing executions to collect the program profile information, experimental approaches [35] attempt to narrow down the defective code to a small set of program entity by generating additional executions. For instance, the technique of delta debugging [35], one representative of experimental approaches, can narrow down failure causes up to 0.2% of the original source code [8]. Although experimental approaches have the ability
of precisely locating the defective code, there exists a risk that program behaviors are changed in a way that the original program cannot perform forever due to, for example, unsuitable input mutation.

Both statistical and experimental approaches are developed from the viewpoint of developers. Especially for statistical approaches, they focus on only the rank of faulty statements regardless of the absolute suspiciousness values of program statements. The effectiveness of these AFL techniques at assisting the technique on automated program repair has been not studied so far. Our work in this paper preliminarily investigates this question.

2.2 The Effectiveness Evaluation of AFL

With more and more AFL techniques being proposed, one popular research area is to evaluate and compare the effectiveness of various AFL techniques. In all these studies, both empirical approaches [3, 14] and theoretical approaches [20, 32] were conducted to investigate and evaluate the AFL techniques. For empirical approaches, researchers used the same experimental setup and benchmarks, such as Siemens programs and Space program, to compare the effectiveness of several AFL techniques. For theoretical approaches, which are the recent advances on the effectiveness evaluation, researchers attempted to study the performance of various AFL techniques from a formally theoretical perspective; in particular, studies in [32] have formally proved that some AFL techniques, such as Naish1 and Naish2, are equivalent, and some AFL techniques such as Tarantula always perform better than other techniques such as Jaccard.

Majority of these studies, either through empirical approaches or through theoretical approaches, used the same evaluation measurement of EXAM or its equivalent. Namely, they prefer these AFL techniques that can give the faulty statement higher priority under the assumption of “perfect bug detection”, which means that examining a faulty statement in isolation is enough for a developer to understand and eliminate the bug. Although this assumption has actually been used by the AFL community, it does not normally hold in practice, because understanding the fault causing a failure typically involves many complex activities [21]. In addition, AFL techniques are originally introduced to provide assistant information to the developers, to what extent can existing techniques actually benefit the developers has been ignored [32]. In fact, although AFL techniques have been developed almost 30 years since one of the first AFL techniques was presented, developers still rarely use the AFL tools to debug faulty programs. Thus, given the maturity of AFL [34], it is reasonable to doubt about the advantage of AFL compared to traditional debugging techniques from the viewpoint of developers.

In addition, although the activity of AFL is necessary for automated program repair, to our knowledge there exists no AFL technique developed with goal of improving the assistant effectiveness for it. In this paper we propose another research direction on AFL: developing techniques from the viewpoint of fully automated debugging: we also introduce the NCP measurement to evaluate the localization effectiveness accordingly.

2.3 Automated Program Repair

In this paper, we refer to the concept of automated program repair as automatically fixing faults in source code, not binary files or run-time patching. Automated program repair, in general, is a three-step procedure: fault localization, patch generation and patch validation. Given a bug reported by some user, in order to repair the faulty program automatically, suspicious faulty code snippet causing the bug is first identified by AFL techniques. Once the faulty code snippet is located, lots of candidate patches can be produced by modifying that code snippet, according to specific repair rules based on either evolutionary computation [16, 17, 5, 4] or code-based contacts [22, 29]. To check that whether one candidate patch is valid or not, there are two main approaches on assessing patched program correctness: formal specifications and test suites. For specifications, although precise and complete formal specifications have the ability of guaranteeing that the valid patch does work well on the defective program, formal specifications are rarely available despite recent advances in specification mining [18]. Thus, test suits are most often used to validate the patch effectiveness.

Automated repair techniques have received considerable recent research attention. Guided by genetic programming, GenProg has the ability to repair programs requiring not any specifications [16]. AutoFix-E [30] can repair programs but requires for the contracts in terms of pre- and post-conditions. JAFF [5] tries to automatically correct the faulty java programs using an evolutionary approach. AFix [12] focuses on the repair for single-variable atomicity violations. PHPRepair [26] can automatically fix HTML generation Errors in php applications through string constraint solving.

Collectively, these advances on automated program repair have pushed forward the state of the art in completely automated debugging. Unlike manual debugging by developers, for automated program repair, the activity of AFL, identifying the program statement(s) responsible for the corresponding fault, is necessary to constrain the fixing operations to operate only on the region of the program that is relevant to the fault. Intuitively, the accuracy of used AFL technique will drastically influence the repair effectiveness, but existing work only focuses on the effectiveness evaluation of AFL techniques from the viewpoint of developers, and there is no work on the influence analysis of these techniques from the viewpoint of fully automated debugging, to our knowledge.

3. OUR APPROACH

In this section we first give a description of insight overview on our evaluation approach, and then describe the evaluation detail about our approach. Finally, we present the evaluation measurement of NCP to assess the effectiveness of AFL techniques.

3.1 Overview

The insight of our approach is motivated by the fact that the effectiveness of used AFL techniques has the important impact on the repair effectiveness in automatically repairing the faulty programs. As described in Section 2.3, for automated program repair, the repair process is a three-step procedure: locating the program fault (fault localization); generating one candidate patch in light of some repair rules (patch generation); validating the patch through test suits in term of regression testing (patch validation). For the second step (patch generation), according to specified fix rules, such as evolutionary computation [16, 5] and code-based contacts [29], lots of candidate patches are produced by modifying
the constrained code region. This code region, in fact, is considered as the defective code snippet causing the fault, and is located through AFL techniques. Obviously, the more accuracy the used fault localization is, the better effectiveness the repair process will perform when the repair rules have been specified.

In the context of automated program repair, the better effectiveness can be referred to as that the fewer invalid patches are produced before a valid patch is found. We consider a patch invalid when the patch fails to pass the regression testing in the patch validation step. Since AFL with better effectiveness should have the ability to enhance the repair effectiveness by providing more useful localization information, we can also evaluate the effectiveness of AFL by measuring the corresponding repair effectiveness.

Evaluating the effectiveness of AFL technique from the viewpoint of fully automated debugging is very important and useful for the area of AFL and automated program repair. Given recent advances on automated program repair, it is pressing that developing the AFL techniques aiming at providing assistant information when automating the program repair. Our work in this paper will address the problem of how to evaluate the effectiveness when some new AFL techniques are proposed in the future.

3.2 Evaluation Framework

Before the description of our evaluation framework in detail, we first discuss about how the fault information provided by AFL is used in the process of automated program repair. Most existing techniques on automated program repair use randomized algorithms, such as evolutionary computation [16, 5], to select some statements as the target for modification. The probability \( pb \) of a statement being selected is computed according to the suspiciousness value of the statement provided by AFL. Take GenProg, a state-of-the-art tool for automated C program repair, for instance, the suspiciousness value \( sp \) of each statement \( s \) is computed through a simple technique similar to statistical approaches: for a program \( P \), a statement never visited by any test case has the \( sp \) value of \( 0 \); a statement visited only by negative test case which reproduces the failure has the high value of \( 1.0 \); a statement visited both by positive and negative test cases is given the moderate value of \( 0.1 \). For each statement \( s \in P \), the corresponding probability \( pb \) of \( s \) being selected is computed through the formula: \( pb = sp \times mute \), where \( mute \) represents the global mute probability set for initialization.

Obviously, AFL techniques guide the technique on automated program repair in a distinct way against manually debugging by developers. For developers, they center on only the ranking of faulty statements rather than the absolute suspiciousness values \([32]\). In contrast, for automated program repair, the absolute suspiciousness value of each statement is one determinant of the repair effectiveness. The bigger suspiciousness values of faulty statements as well as the smaller values of other statements, the better repair effectiveness, because the probability \( pb \) of a statement \( s \) being selected is computed according to \( sp \) rather than the ranking.

In light of mentioned discussion, we present our approach of evaluating the accuracy of fault localization through automated program repair. To fairly compare each presented AFL technique, we should first construct a repair framework with some specific repair rules \([16, 5, 29]\); the framework should provide the interface for accepting the outputs of various AFL techniques. Second, each fault localization technique, with the goal of assisting the techniques on automated program repair rather than developers, should ensure that the final output meets the interface format provided by the repair framework. Specifically, suppose that the source code of a faulty program \( P \) is constructed from a set of statements \( S = \{s_1, s_2, \cdots, s_n\} \), then, the right output format should be a list like that:

\[
List = \{(s_1, sp_1), (s_2, sp_2), \cdots, (s_n, sp_n)\}, 0 \leq sp \leq 1,
\]

where \( sp \) represents the suspiciousness value of a statement \( s \), the constraint of \( 0 \leq sp \leq 1 \) is used to simplify the computation of probability \( pb \). In fact, this output format is similar as the normalized ranking list outputted by statistical fault localization techniques. For the experimental techniques which output the defective code snippet \( S_{sub} = \{s_d, s_g, \cdots, s_w\} \), we can obtain this kind of \( List \) by assigning the value 1 to \( sp \) of each statement \( s \in S_{sub} \) and 0 for the remaining statements. At last, the repair framework tries to repair automatically the faulty program by modifying some statements in light of specific repair rules; the statements with higher suspiciousness values will have more chances of being modified. We rank various fault localization techniques according to the guide effectiveness in term of the repair effectiveness.

3.3 Framework Implementation: GenProg-FL

To demonstrate the feasibility of our evaluation approach, we have implemented the framework by building GenProg-FL, a tool that has the ability of repairing automatically C programs. GenProg-FL is the modification version of GenProg\(^1\) [16] by adding the interface for accepting the localization information on the failure. We selected GenProg for the reason that it is almost the only state-of-the-art automated repair tool having the ability of fixing real-world, large-scale C faulty programs; literature [10] also used GenProg as the sole tool to study patch maintainability.

3.4 Evaluation Measurement

As described earlier, we can evaluate the effectiveness of one AFL technique by measuring the repair effectiveness of GenProg-FL guided by this AFL technique. In general, the bigger suspiciousness values of faulty statements as well as the smaller values of other statements in the \( List \) produced by the AFL technique, the better repair effectiveness of GenProg-FL. Theoretically, we can evaluate the effectiveness of AFL by analyzing only the \( List \); this way of evaluation measurement is similar to the traditional measurement of \( EXAM \), which evaluates the effectiveness by observing the rank of faulty statements in ranking list produced by AFL.

Unfortunately, this evaluation measurement, however, suffers from two problems. First, finding precisely the faulty statements is often time-consuming or impossible [27], especially for real-life, complex faults. For instance, for the real-life \( \text{php} \) bug presented in [10, p.178], there are two distinct patches affecting even different functions code, resulting in the difficulty in making the decision on where the actually faulty statements locate.

Second, it is still difficult to compare the effectiveness between two AFL techniques in some cases, even when we have supposed that faulty statements could be precisely tracked down. For instance, suppose that for the same faulty

\(^1\)Available: http://dijkstra.cs.virginia.edu/genprog/
program $P$ with the actually faulty statement of $s_3$, the ranking lists:

$$Rank_1 = \{(s_j, 1), (s_p, 0.9), (s_3, 0.8), \cdots\},$$

$$Rank_2 = \{(s_j, 1), (s_p, 0.8), (s_3, 0.7), \cdots\},$$

which are produced by two statistical AFL techniques of AFL1 and AFL2, respectively. The two techniques are equivalent according to the EXAM measurement. Although AFL1 produces bigger suspiciousness value of $s_3$, which improves the probability of $s_3$ being selected for modification, the bigger value of $s_p$ will also improves the probability of $s_p$ being wrongly selected for modification, increasing the risk of introducing unexpected side effect.

In our work we measure the repair effectiveness of GenProg-FL according to the Number of Candidate Patches (NCP) generated before a valid patch is found in the repair process. When the global mute rate mute is specified, the lower NCP score can be observed in the repair process, the better repair effectiveness can be performed by GenProg-FL, and the better repair effectiveness indicates a better localization effectiveness of AFL technique used by GenProg-FL. Measuring precisely the NCP for one AFL technique, however, is difficult because the NCP is most probably different for each run of GenProg-FL due to the use of randomized algorithm (i.e., genetic programming). Hence, we use statistical analysis, including nonparametric Mann-Whitney-Wilcoxon test and A-test, to evaluate and compare various AFL techniques. We defer discussing the details to Section 5.1.

4. EXPERIMENTAL DESIGN

To justify the ability of GenProg-FL to identify the effectiveness of various AFL techniques, in our experiment we use GenProg-FL to repair 11 real-life faulty programs coming with field failures, under the separate guidance of 15 popular statistical AFL techniques. This experiment also plans to investigate whether the conclusions drawn under the evaluation measurement of EXAM still hold according to NCP measurement. In addition, experimental results present some useful advices on selecting some AFL techniques from existing ones to effectively guide the repair process before some more effective AFL techniques, which can have better guidance effectiveness on automated program repair, occur in the future.

4.1 Research Questions

Our experimental evaluation plans to addresses the following Research Questions:

**RQ1:** Do the AFL techniques performing well under the evaluation measurement of EXAM still have good performance according to NCP measurement?

Prior empirical studies [1] have shown that Ochiai always performs better than the techniques of Jaccard and Tarantula. Furthermore, recent theoretical analysis proved that Optimal_p (also called Naish in [32]) and Russel_rao are the maximal AFL techniques in the sense that the two AFL techniques are strictly “better” than other statistical techniques according to the EXAM measurement. Then, RQ1 asks whether these “better” AFL techniques behave similarly in our experiment based on the NCP measurement.

**RQ2:** Does there exist some AFL technique(s) performing better over other techniques, according to NCP score?

Majority of existing AFL techniques were developed from the viewpoint of developers on the strong assumption of “perfect bug detection”. Namely, most of these techniques aim to improve the rank of faulty statement, regardless of the absolute suspiciousness value, which plays an important part on the effectiveness evaluation based on NCP. Nevertheless, RQ2 plans to investigate whether there is any existing AFL technique performing better according to the NCP measurement. If so, we will suggest that the techniques on automated program repair should use the AFL technique(s) to guide the repair process before some more effective AFL techniques are proposed in the future.

**RQ3:** If so, to what extent can the AFL technique(s) perform better on the improvement than other techniques?

If there exists some AFL technique(s) having better performance, then RQ3 plans to investigate whether there is any existing AFL technique(s) having better performance. If so, we will suggest that the AFL technique(s) can be used to guide the repair process before some more effective AFL techniques are proposed in the future.

4.2 Subject Programs

In this experiment, we selected the subject programs used in the most recent work [16] on GenProg as the experimental benchmarks, all of which are written in C language and are different real-world systems with real-life bugs from different domains. These programs come with different sizes of positive test cases ranging from 12 to 8,471, which facilitates the application of AFL. In [2] Abreu and colleagues found that including more than 20 positive test cases has minimal effects on the effectiveness of AFL techniques. That is, chance is that the effectiveness of studied AFL techniques is unstable when target programs come with less than 20 positive test cases. To remove this instability and compare each technique fairly, we exclude gzip and lighttpd programs, because there is too few positive test cases (no more than 10 ones) actually used in both gzip and lighttpd programs, although more test cases were listed in [16].

For the fbc program we have the compilation trouble when we try to compile the program. We also exclude the gmp programs and 9 versions of libtiff programs, because we found that there exists much less similarity between negative and positive test cases, which will also affect the AFL accuracy [7]. At last, we select the remaining programs, i.e., libtiff, python, php, and wireshark, as target programs used in the experiment.

For php which equips with too many test cases, several minutes are needed for validating one patched program, which will result in time-consuming repair process. Thus, complete experimental evaluation on all the php programs can take too much time (see [16, Table II]); the authors in [16] used Amazon’s EC2 cloud computing infrastructure including 10 trials in parallel for their experiment. What is worse, in our experiment, for each subject program we need 15 times more computing resource compared to the experiment.

RQ3: If so, to what extent can the AFL technique(s) perform better on the improvement than other techniques?

If there exists some AFL technique(s) having better performance, then RQ3 plans to investigate whether there is any existing AFL technique(s) having better performance. If so, we will suggest that the AFL technique(s) can be used to guide the repair process before some more effective AFL techniques are proposed in the future.

4.2 Subject Programs

In this experiment, we selected the subject programs used in the most recent work [16] on GenProg as the experimental benchmarks, all of which are written in C language and are different real-world systems with real-life bugs from different domains. These programs come with different sizes of positive test cases ranging from 12 to 8,471, which facilitates the application of AFL. In [2] Abreu and colleagues found that including more than 20 positive test cases has minimal effects on the effectiveness of AFL techniques. That is, chance is that the effectiveness of studied AFL techniques is unstable when target programs come with less than 20 positive test cases. To remove this instability and compare each technique fairly, we exclude gzip and lighttpd programs, because there is too few positive test cases (no more than 10 ones) actually used in both gzip and lighttpd programs, although more test cases were listed in [16].

For the fbc program we have the compilation trouble when we try to compile the program. We also exclude the gmp programs and 9 versions of libtiff programs, because we found that there exists much less similarity between negative and positive test cases, which will also affect the AFL accuracy [7]. At last, we select the remaining programs, i.e., libtiff, python, php, and wireshark, as target programs used in the experiment.

For php which equips with too many test cases, several minutes are needed for validating one patched program, which will result in time-consuming repair process. Thus, complete experimental evaluation on all the php programs can take too much time (see [16, Table II]); the authors in [16] used Amazon’s EC2 cloud computing infrastructure including 10 trials in parallel for their experiment. What is worse, in our experiment, for each subject program we need 15 times more computing resource compared to the experiment.
We select the 14 techniques from the literature [20, 32], in which 30 statistical AFL techniques are studied; other 16 techniques are eliminated because each of them is equivalent to some technique included by the 14 ones [32]. The more details on the definition of selected techniques in our experiment can be found in [20, 32]. As described earlier, all the 14 techniques are developed to improve the ranking of faulty statements from the viewpoint of developers. We also use the AFL technique used by GenProg as the baseline technique in our experiment.

In addition, GenProg-FL requires that the output of used AFL technique have the format like List described in Section 3.2. Namely, the suspiciousness value $sp$ of each statement must meet the constraint of $0 \leq sp \leq 1$. Some AFL techniques, however, do not meet the constraint. For instance, the value of $sp$ in ranking list produced by Wong3 technique can be negative for some statements. Hence, to ensure that investigated AFL techniques work well with GenProg-FL, we preprocess the output of each AFL technique in a way of normalizing these suspiciousness values. Specifically, suppose that the rank list $Rank$ is produced by one AFL technique, and $Rank = \{ (s_j, sp_j), (s_p, sp_p), \ldots, (s_n, sp_n) \}$, where $s$ and $sp$ represent the program statement and the corresponding suspiciousness value, respectively. Then, we can get the List by preprocessing $Rank$:

$$List = \{ (s_j, \frac{sp_j - \min(sp)}{\max(sp) - \min(sp)}), (s_p, \frac{sp_p - \min(sp)}{\max(sp) - \min(sp)}) \cdots, (s_n, \frac{sp_n - \min(sp)}{\max(sp) - \min(sp)}) \},$$

where $\min(sp)$ and $\max(sp)$ represent the minimal and maximal value of $sp$ in $Rank$. At last, $List$ meeting the format requirement of GenProg-FL is used to assist GenProg-FL to track down the faulty statements. Note that this normalization process does not change the rank order of each statement in $Rank$. That is, $List$ is equivalent to $Rank$ in term of EXAM measurement.

### 4.4 Experimental Setup

For the purposes of comparison, for each AFL technique, we separately ran GenProg-FL to repair 11 subject programs with the guidance of AFL. All the experimental parameters for GenProg-FL in our experiment are similar to those settings in [16]: we limited the size of the population of first generation to 40 and remaining generations to 80, and a maximum of 10 generations for each repair process; the global mutate rate is set to 0.01. We considered that GenProg-FL fails to repair one subject program for one repair trial if the valid patch is not found within 10 generations. We recorded only these trials which are successful to find the valid patches. As described in Section 3.4, we use statistical analysis, which requires many runs to ensure high confidence level, to evaluate and compare the effectiveness of the 15 AFL techniques. Hence, in our experiment, for one concrete subject program, we have 100 runs of GenProg-FL according to the same AFL technique. Namely, for each AFL technique, totaling 100 runs of GenProg-FL are required for one subject program; each run is most probably different from other runs due to the application of genetic programming, a kind of randomized algorithm (hence, statistical analysis is necessary), in GenProg-FL. Our experiment ran on three Ubuntu 10.04 machines in parallel.

### 5. RESULT DATA AND ANALYSIS

Experimental results are presented in Figure 1, which shows the box-plots of repair effectiveness in term of $NCP$ when running GenProg-FL to repair each subject program according to different AFL techniques. Note that there are a total of 100 repair trials for one AFL technique on each program. Hence, there are $n$ successful trials if the success rate is $n \%$, which is shown in Table 2; the statistics in Figure 1 and Table 2 are computed according to the $n$ successful trials.

#### 5.1 Statistical Analysis

Since randomized algorithm (i.e., genetic programming) is applied in GenProg-FL, we use statistical analysis to compare the localization effectiveness of various AFL techniques. Specifically, the localization effectiveness (in term of $NCP$ score) of each AFL technique is summarized using the mean, because the mean gives a more meaningful statistic located over the median when the result data are constructed from similarly sized clusters around two (or more) widely separated values [23].

Then, we use nonparametric statistical tests to analyze the result data. Although parametric statistical tests can be

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Test Cases</th>
<th>Version</th>
<th>Bug Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>python</td>
<td>407,000</td>
<td>303</td>
<td>69783-69784</td>
<td>Year 2000 issues</td>
</tr>
<tr>
<td>php</td>
<td>1,046,000</td>
<td>4,986</td>
<td>309992-309910</td>
<td>Incorrect output</td>
</tr>
<tr>
<td>wireshark</td>
<td>2,814,000</td>
<td>53</td>
<td>37112-37111</td>
<td>Memory leak</td>
</tr>
</tbody>
</table>

Table 1: Subject Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Test Cases</th>
<th>Version</th>
<th>Bug Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lisp.tiff</td>
<td>77,000</td>
<td>78</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

in [16], because the effectiveness of all the 15 investigated AFL techniques need to be statistically observed. Given the expensive testing computation, we randomly select one faulty version without any bias. For python and wireshark, there exists only one version having been repaired successfully by GenProg in [16] for each program.

In total, Table 1 describes our subject programs in detail, and we spent about one and a half month on running our experiment on three computers in parallel without break. The LOC column lists the scale of each subject program in term of lines of code, and the last three columns give the size of positive test cases, the version information, and bug description. Note that we got these test cases by running the original test cases (provided by [16]) against the corresponding programs and eliminating bad ones for which some failure occurs. We assigned one negative test case for each subject program to reproduce the bug.

#### 4.3 Investigated AFL Techniques

In our experiment, we investigate 15 statistical AFL techniques, which includes most of popular techniques on AFL. We select the 14 techniques from the literature [20, 32], in which 30 statistical AFL techniques are studied; other 16 techniques are eliminated because each of them is equivalent to some technique included by the 14 ones [32]. The more details on the definition of selected techniques in our experiment can be found in [20, 32]. As described earlier, all the 14 techniques are developed to improve the ranking of faulty statements from the viewpoint of developers. We also use the AFL technique used by GenProg as the baseline technique in our experiment.

In addition, GenProg-FL requires that the output of used AFL technique have the format like List described in Section 3.2. Namely, the suspiciousness value $sp$ of each statement must meet the constraint of $0 \leq sp \leq 1$. Some AFL techniques, however, do not meet the constraint. For instance, the value of $sp$ in ranking list produced by Wong3 technique can be negative for some statements. Hence, to ensure that investigated AFL techniques work well with GenProg-FL, we preprocess the output of each AFL technique in a way of normalizing these suspiciousness values. Specifically, suppose that the rank list $Rank$ is produced by one AFL technique, and $Rank = \{ (s_j, sp_j), (s_p, sp_p), \ldots, (s_n, sp_n) \}$, where $s$ and $sp$ represent the program statement and the corresponding suspiciousness value, respectively. Then, we can get the List by preprocessing $Rank$:

$$List = \{ (s_j, \frac{sp_j - \min(sp)}{\max(sp) - \min(sp)}), (s_p, \frac{sp_p - \min(sp)}{\max(sp) - \min(sp)}) \cdots, (s_n, \frac{sp_n - \min(sp)}{\max(sp) - \min(sp)}) \},$$

where $\min(sp)$ and $\max(sp)$ represent the minimal and maximal value of $sp$ in $Rank$. At last, $List$ meeting the format requirement of GenProg-FL is used to assist GenProg-FL to track down the faulty statements. Note that this normalization process does not change the rank order of each statement in $Rank$. That is, $List$ is equivalent to $Rank$ in term of EXAM measurement.

### 4.4 Experimental Setup

For the purposes of comparison, for each AFL technique, we separately ran GenProg-FL to repair 11 subject programs with the guidance of AFL. All the experimental parameters for GenProg-FL in our experiment are similar to those settings in [16]: we limited the size of the population of first generation to 40 and remaining generations to 80, and a maximum of 10 generations for each repair process; the global mutate rate is set to 0.01. We considered that GenProg-FL fails to repair one subject program for one repair trial if the valid patch is not found within 10 generations. We recorded only these trials which are successful to find the valid patches.

As described in Section 3.4, we use statistical analysis, which requires many runs to ensure high confidence level, to evaluate and compare the effectiveness of the 15 AFL techniques. Hence, in our experiment, for one concrete subject program, we have 100 runs of GenProg-FL according to the same AFL technique. Namely, for each AFL technique, totaling 100 runs of GenProg-FL are required for one subject program; each run is most probably different from other runs due to the application of genetic programming, a kind of randomized algorithm (hence, statistical analysis is necessary), in GenProg-FL. Our experiment ran on three Ubuntu 10.04 machines in parallel.

### 5. RESULT DATA AND ANALYSIS

Experimental results are presented in Figure 1, which shows the box-plots of repair effectiveness in term of $NCP$ when running GenProg-FL to repair each subject program according to different AFL techniques. Note that there are a total of 100 repair trials for one AFL technique on each program. Hence, there are $n$ successful trials if the success rate is $n \%$, which is shown in Table 2; the statistics in Figure 1 and Table 2 are computed according to the $n$ successful trials.

#### 5.1 Statistical Analysis

Since randomized algorithm (i.e., genetic programming) is applied in GenProg-FL, we use statistical analysis to compare the localization effectiveness of various AFL techniques. Specifically, the localization effectiveness (in term of $NCP$ score) of each AFL technique is summarized using the mean, because the mean gives a more meaningful statistic located over the median when the result data are constructed from similarly sized clusters around two (or more) widely separated values [23].

Then, we use nonparametric statistical tests to analyze the result data. Although parametric statistical tests can be
Figure 1: Experimental Results. We present the 15 SBFL techniques in ascending order according to “Mean”. 
more accuracy in some cases, these tests requires additional assumptions, such as normality, for the studied data [6]. Since there may exist small deviations from the assumptions, we use nonparametric tests to guarantee the analysis validity. Specifically, suppose that the technique AFL1 has the higher mean value over AFL2, the nonparametric Mann-Whitney-Wilcoxon test [31] and A-test [28] are separately used to qualitatively and quantitatively analyze the improvement of AFL1 over AFL2.

For nonparametric Mann-Whitney-Wilcoxon test, the null hypothesis is that result data from AFL1 and AFL2 share the same distribution; the alternate hypothesis is that the two group data have different distributions. In this case, we say the improvement of AFL1 over AFL2 is statistical significance when we reject the null hypothesis at a 5 percent significance level.

To further assess the improvement quantitatively, we use the nonparametric Vargha-Delaney A-test, which is recommended in [6] and was also used in [23], to evaluate the magnitude of the improvement by measuring effect size (scientific significance)—in this case, a difference in the localization effectiveness in term of NCP score. For A-test in this case, the bigger deviation of A-statistic from the value of 0.5, the greater improvement of AFL1 over AFL2. In [28] Vargha and Delaney suggest that A-statistic of greater than 0.64 (or less than 0.36) is indicative of “medium” effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising “large” effect size.

5.2 RQ1
As described in [1], Ochiai performs better than the techniques of Jaccard and Tarantula according to the EXAM measurement. Based on a empirical evaluation, Rui Abreu et al. [1] found that Ochiai always performed at least as good as the other studied techniques, with the improvement of 4% against the second-best technique, on average. As shown in Figure 1, however, Ochiai did not always perform better over other techniques according to the mean on NCP score. Take php case in Figure 1, Jaccard improved the localization effectiveness over Ochiai in term of lower NCP score; the improvement is statistically significant using Mann-Whitney-Wilcoxon test ($p = 0.015568$).

In addition, Figure 1 also clearly shows that neither Optimalp nor Russel_rao has the lowest mean value for any subject program. Namely, neither Optimalp nor Russel_rao is the maximal AFL technique according to the NCP score, though they have been theoretically proved to be the maximal according to EXAM measurement [32] among the 15 investigated techniques.

Overall, although more experiments are needed before strong conclusions can be drawn, Figure 1 shows that these AFL techniques performing well under the evaluation measurement of EXAM do not have good performance according to NCP measurement. That is, existing AFL techniques aiming to benefit developers do not always perform better in the context of automated program repair, justifying the necessity of developing AFL techniques for automated program repair.

5.3 RQ2
As described in Section 5.1, we statistically rank the 15 techniques according to the mean. Note that in Figure 1 the investigated AFL techniques have been presented in ascending order according to the mean value for each sub-figure. The smaller mean indicates the better localization effectiveness of the corresponding AFL technique. We are pleased to find that the Jaccard technique has the better effectiveness over most techniques in term of the smaller mean value. Specifically, Jaccard has the best or second best effectiveness in 8 of 11 faulty programs. For the remaining faulty programs, Jaccard, at least, has the better performance than most techniques. We then check whether the effectiveness improvement of Jaccard over other techniques is statistically significant.

Table 2 gives the significance information (based on Mann-Whitney-Wilcoxon test) on the comparisons of Jaccard and the other techniques for each subject programs. We can observe that for the first 10 subject programs Jaccard significantly has the smaller NCP score than most studied techniques. For the wireshark program, Table 2 indicates that Jaccard performs at least as good as the other techniques, even if it does not have the best performance.

5.4 RQ3
We have qualitatively reported that Jaccard most often has the best performance over other techniques, in this subsection we further quantitatively study the magnitude of the performance improvement of Jaccard over other techniques. Table 2 quantitatively shows the improvement in term of A-test representing the effect size, if the improvement is statistical significance using Mann-Whitney-Wilcoxon test. Recall that A-statistic of greater than 0.64 (or less than 0.36) can represent “medium” effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising “large” effect size. As described in Table 2, for most programs except wireshark, Jaccard most often has the “medium” effect size over other techniques. The effect size can be promising “large” in some cases.

In summary, result data in our experiment show that 1) these AFL techniques performing well under the evaluation measurement of EXAM do not have good performance from the viewpoint of fully automated program repair, 2) Jaccard performs at least as good as the other 13 investigated techniques, with the effectiveness improvement up to “large” effect size using A-test. Hence, although more experiments are needed before strong conclusions can be drawn, we suggest that Jaccard should be used with high priority to provide fault information for these techniques on automated program repair before some more effective AFL techniques are proposed in the future.

5.5 Threats to Validity
The main threats to the validity of our result belong to the internal, external, and construct validity threat categories. Internal validity threats corresponds to the relationship between the independent and dependent variables in the study. One such threat in our experiment is the distinct inconsistency on the output among various AFL techniques. For example, the suspiciousness value of each statement outputted by Jaccard ranges from 0 to 1; the value outputted by Wong3, however, does not suffer from the limitation, and even can be negative. The inconsistency may cause unfair comparison between various techniques. To mitigate this threat we normalize the suspiciousness value of statements outputted by each AFL techniques in the way described in Section 4.3.
Table 2: The statistical comparisons between Jaccard and the other fault localization techniques. The "Sig." column shows the Mann-Whitney-Wilcoxon test on the comparison difference between various SBFL technique and Jaccard (1 represents statistically significant difference). We give the magnitude of "A-test" if the difference is statistical significance. The "SR" column gives the Success Rate of each AFL techniques.

<table>
<thead>
<tr>
<th>SBFL Techniques</th>
<th>libttf-bug-0a80855-5b02f3b</th>
<th>libttf-bug-069c8f0a-541b0a</th>
<th>libttf-bug-002f0a-0a96b5d</th>
<th>libttf-bug-052a179-3b02f6b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sig.</td>
<td>A-test p-value SR</td>
<td>Sig.</td>
<td>A-test p-value SR</td>
<td>Sig.</td>
</tr>
<tr>
<td>jaccard</td>
<td>- 0.5 0.00000 100%</td>
<td>- 0.5 0.00000 100%</td>
<td>- 0.5 0.00000 100%</td>
<td>- 0.5 0.00000 100%</td>
</tr>
<tr>
<td>tarantula</td>
<td>0 * 0.42362 100%</td>
<td>0 * 0.47277 100%</td>
<td>1 0.73 0.00000 100%</td>
<td>0 * 0.64209 100%</td>
</tr>
<tr>
<td>kulczynski2</td>
<td>0 * 0.29670 100%</td>
<td>0 * 0.91896 100%</td>
<td>1 0.68 0.00001 100%</td>
<td>0 * 0.90846 100%</td>
</tr>
<tr>
<td>ample</td>
<td>1 0.87 0.00000 99%</td>
<td>1 0.94 0.00000 98%</td>
<td>1 0.33 0.00004 99%</td>
<td>1 0.82 0.00000 99%</td>
</tr>
<tr>
<td>ochiai</td>
<td>0 * 0.14653 100%</td>
<td>0 * 0.75270 100%</td>
<td>1 0.63 0.00016 100%</td>
<td>0 * 0.37788 100%</td>
</tr>
<tr>
<td>cohen</td>
<td>1 0.84 0.00000 100%</td>
<td>0 * 0.08009 100%</td>
<td>1 0.73 0.00000 100%</td>
<td>1 0.68 0.00001 100%</td>
</tr>
<tr>
<td>m2</td>
<td>0 * 0.57444 100%</td>
<td>0 * 0.17638 100%</td>
<td>1 0.64 0.000092 100%</td>
<td>0 * 0.30604 100%</td>
</tr>
<tr>
<td>russel_rao</td>
<td>1 0.88 0.00000 99%</td>
<td>0 0.91 0.00000 100%</td>
<td>1 0.76 0.00000 100%</td>
<td>1 0.74 0.00000 100%</td>
</tr>
<tr>
<td>fleiss</td>
<td>1 0.86 0.00000 99%</td>
<td>1 0.85 0.00000 100%</td>
<td>1 0.78 0.00000 100%</td>
<td>1 0.77 0.00000 100%</td>
</tr>
<tr>
<td>ample2</td>
<td>1 0.87 0.00000 100%</td>
<td>1 0.83 0.00000 100%</td>
<td>1 0.74 0.00000 100%</td>
<td>1 0.78 0.00000 100%</td>
</tr>
<tr>
<td>optimap</td>
<td>1 0.77 0.00000 100%</td>
<td>1 0.87 0.00000 99%</td>
<td>1 0.78 0.00000 100%</td>
<td>1 0.80 0.00000 100%</td>
</tr>
<tr>
<td>rogers_animoto</td>
<td>1 0.88 0.00000 97%</td>
<td>1 0.87 0.00000 100%</td>
<td>1 0.82 0.00000 100%</td>
<td>1 0.80 0.00000 100%</td>
</tr>
<tr>
<td>arithmetic_mean</td>
<td>1 0.83 0.00000 96%</td>
<td>1 0.80 0.00000 100%</td>
<td>1 0.75 0.00000 100%</td>
<td>1 0.77 0.00000 100%</td>
</tr>
<tr>
<td>wong3</td>
<td>1 0.79 0.00000 98%</td>
<td>0 * 0.15153 100%</td>
<td>0 * 0.40496 100%</td>
<td>0 * 0.20630 100%</td>
</tr>
</tbody>
</table>

6. DISCUSSION

Experimental results suggest that Jaccard can better help program repair than other investigated techniques such as Ochiai, Optimap, and Russel_rao, which are considered to have the ability of ranking faulty statements higher in prior studies. In this section, we try to explain why Jaccard has the advantage of better helping program repair, and discuss the possible limitations of Jaccard.

After investigating the outputs of 15 AFL techniques, we find that Jaccard has more sharp score than other techniques. Take php for instance, Table 3 presents the rank lists produced by 4 AFL techniques. Suppose that $S = \{s_1, s_2, \ldots, s_{id}, \ldots\}$ represents the set of statements constructing the studied faulty program $P$, $B \subset S$ represents
the set of statements which have the bigger suspiciousness values, and $\overline{B} = S \setminus B$, the complement set of $S$ and $B$, consists of the remaining statements. Then, we consider $M \subseteq S$ as the key statements$^3$ in the sense that GenProg-FL can produce some valid patch by modifying one statement $s \in M$ or some statements $M_{sub} \subseteq M$. Obviously, in Table 3 Jaccard has the more sharp score in term of the smaller suspiciousness of $B$. On one hand, when the size of $BM = B \cap M$ is big enough, the more sharp score enables Jaccard to keep bigger suspiciousness of key statements and smaller suspiciousness of most normal statements, resulting in more chances of generating valid patches. Although some techniques may rank faulty statements higher, they also produce not too low suspiciousness of many normal statements (i.e., more flat score), which drastically increases the risk of modifying normal statements in $\overline{B}$, resulting in new faults introduced.

On the other hand, the sharp score produced by Jaccard may hurt the repair effectiveness when the size of $BM = B \cap M$ is much smaller (i.e., the size of $BM = B \cap M$ is much larger than the size of $BM$). The reason for that is the smaller suspiciousness of $BM$ (which includes most statements of $M$ in this context) reduces the chance of $M$ being selected for modification, which can compromise the advantage provided by sharp score itself.

In fact, result data in Figure 1 are the trade-off between the two above contexts. Fortunately, Jaccard, in most cases (8/11), has the ability of getting not too small size of $BM$, and thus has better performance in our experiment. Meanwhile, we also notice that the advantage of Jaccard is not too evident in the remaining cases (3/11), i.e., bug-439d1b2c-4cd598e-bug-4a24508-cc79e2b, and bug-69783-69784, due to a relatively small size of $BM$. We will further empirically investigate the impact of $BM$ and $\overline{BM}$ on the repair effectiveness of GenProg-FL in the future.

$^3$Note that the key statements are not equivalent to the faulty statements, and there may exist many distinct patches affecting code in even different functions [10].

### 7. CONCLUSIONS

Existing AFL techniques have been developed with the goal of helping developers fix bugs in code. Recent studies, however, have shown that the advantages of AFL techniques over traditional debugging techniques on manual fault correction are not as evident as expected even when the effectiveness of AFL techniques is artificially improved. In contrast, the activity of AFL is most often necessary for the techniques on automated program repair, an indispensable part for the implementation of fully automated debugging: the effectiveness of used AFL techniques can drastically influence the repair effectiveness. In this paper, we propose to develop AFL techniques from the viewpoint of fully automated debugging, and present the NCP score as the evaluation measurement to assess and compare the effectiveness of various techniques.

In our experiment, we investigated 15 popular AFL techniques under the NCP score on 11 subject programs shipping with real-life field failures. By analyzing the result data using statistical test, we have two important observations: 1) these AFL techniques performing well in prior studies do not have good performance from the viewpoint of fully automated program repair, 2) Jaccard performs at least as good as the other 13 investigated techniques, with the effectiveness improvement up to “large” effect size. The first observation presents the evidence of the necessity of studying the AFL from the viewpoint of fully automated debugging. The second observation suggests us that Jaccard should be used with high priority before some more effective AFL techniques specially proposed for automated program repair occur in the future.

Complete experimental results in this paper are available at:

http://qiyuhua.github.com/projects/afl/

### 8. ACKNOWLEDGMENTS

The authors wish to thank W. Weimer et al. for their noteworthy study on GenProg, based on which the GenProg-FL system was built. The authors would also like to thank the anonymous reviewers for their constructive comments on this paper. This research was supported in part by grants from National Natural Science Foundation of China (Nos. 61133001 and 91118007), National High Technology Research and Development Program of China (Nos. 2011AA010106 and 2012AA011201), and Program for New Century Excellent Talents in University.

### 9. REFERENCES


