Every Mutation Should Be Rewarded: Boosting Fault Localization with Mutated Predicates

Xuezheng Xu, Changwei Zou and Jingling Xue
School of Computer Science and Engineering, UNSW Sydney, Australia
{xuezhengxu, changweiz, jingling}@cse.unsw.edu.au

Abstract—Many fault localization (FL) techniques have been proposed to facilitate software debugging. Due to being lightweight, spectrum-based fault localization (SBFL) is one of the most popular FL families and widely deployed in program repair tools. SBFL ranks program elements by recording the program coverage under a test suite and calculates the suspiciousness score of each element with a ranking formula. Despite numerous formulae proposed, SBFL still suffers from providing no new sources of information other than program coverage. Mutation-based fault localization (MBFL) iteratively mutates a faulty program and suggests fault locations through mutants that overturn failed test cases. However, due to its explosive search space, MBFL has been adopted by only few program repair tools.

In this paper, we aim at exploiting the advantages of MBFL and boosting SBFL with low overhead by building a practical FL tool. We propose FLIP, an FL technique with inferences from mutated predicates. Based on SBFL, we leverage and extend the predicate switching technique to infer fault locations no matter whether the mutated predicate can overturn a failed test case or not. Finally, we compute a new ranking list with a joint inference that combines program coverage and mutation inferences.

We use Defects4J (version 1.5.0), containing 438 real-world faults from six projects to evaluate FLIP. All the seven state-of-the-art SBFL techniques benefit from FLIP (e.g., by ranking up to 46.4% more faults in top-1) with low overhead (e.g., by incurring less than 2-minute average overhead for each fault). We also offer some insights on how to further improve FL on real-world faults based on the empirical results.

Index Terms—fault localization, testing, debugging

I. INTRODUCTION

Debugging software is unavoidable, tedious, and time-consuming in software engineering. To save developers from this heavy work and reduce the cost of software maintenance, advanced fault localization (FL) techniques, such as spectrum-based \cite{18, 47, 56}, slicing-based \cite{12, 59}, mutation-based \cite{32, 34} and machine-learning-based \cite{49, 55} techniques, have been proposed. FL techniques aim at automatically finding the faulty elements in a buggy program by collecting and analyzing its static or dynamic information. Recently, automated program repair (APR) has attracted researchers’ attentions with the majority of APR tools employing FL tools in the first stage of an APR workflow \cite{31, 33, 54}.

Problem Statement. In this paper, we aim at designing an FL approach to finding real-world faults. To meet the needs of modern software debugging, an FL technique should be precise (e.g., able to localize many faulty elements in top-1) and efficient (e.g., able to do so in minutes instead of hours). Besides, APR requires an FL tool to be fine-grained (by, e.g., recognizing faulty elements at the statement level) and applicable to general-type faults (e.g., without relying on expert knowledge or program specifications).

Prior Work and Limitations. Spectrum-based fault localization (SBFL) represents one of the most popular FL families \cite{48}. For a given test suite, SBFL first records the execution trace, i.e., program coverage for each test case, together with the test result, and then calculates the suspiciousness score for each program element (e.g., class, method, or statement) using a ranking formula. Elements that are covered by more failed test cases and less passed test cases will receive a higher suspiciousness score. In this paper, we investigate FL at the statement level, which is quite fine-grained and compatible with APR tools. Figure 1 lists seven state-of-the-art SBFL techniques, in which \( s(\ell) \) denotes the suspiciousness score of a statement \( \ell \), is calculated by the corresponding ranking formula. SBFL does not rely on a model for the tested system and can be easily integrated with existing testing procedures. Because of its relatively small overhead, SBFL is the most commonly used technique in APR \cite{31, 33, 54}. However, SBFL only considers program coverage, which is usually collected at the block level. Due to the lack of extra information, the effectiveness of SBFL is limited even with a learning model combining different SBFL techniques \cite{61}.

Another popular FL family is mutation-based fault localization (MBFL) \cite{32, 54}, which iteratively mutates a faulty program and tests mutants against the test cases. MBFL suggests fault locations based on the following assumptions:

\[ Ochiai \quad s(\ell) = \frac{N_F(\ell)}{\sqrt{N_P(\ell)(N_P(\ell) + N_F(\ell))}} \]

\[ Tarantula \quad s(\ell) = \frac{N_F(\ell) / N_P(\ell)}{N_F(\ell) / N_P(\ell) + N_P(\ell) / N_F(\ell)} \]

\[ Jaccard \quad s(\ell) = \frac{N_F(\ell)}{N_F(\ell) + N_P(\ell)} \]

\[ Kulczynski2 \quad s(\ell) = \frac{1}{2} \left( \frac{N_F(\ell)}{N_P(\ell)} + \frac{N_P(\ell)}{N_F(\ell)} \right) \]

\[ DStar2 \quad s(\ell) = \frac{N_P(\ell) - N_F(\ell)}{N_P(\ell) + N_F(\ell)} \]

\[ Zoltar \quad s(\ell) = \frac{N_F(\ell)}{N_F(\ell) + N_P(\ell) + 1000(N_F(\ell) - N_P(\ell)) / (N_F(\ell) + N_P(\ell))} \]

\[ Goodman \quad s(\ell) = \frac{3N_F(\ell) - N_F(\ell) - N_P(\ell)}{N_F(\ell) + N_P(\ell)} \]
selecting & mutating predicates

Step 1.

testing

√

√
critical predicates

uncritical predicates

faulty program

test suite

pre-ranking with SBFL

program mutants

testing

critical predicates

uncritical predicates

inferring

program

mutants

selecting mutating
test suite

pre-ranking with SBFL

~

Step 3. reranking with the enhanced program spectra

program coverage

mutation inferences

Fig. 2. The workflow of FLIP.

- If mutating a statement $\ell$ makes more test cases pass, $\ell$ is likely to be the faulty statement.
- If no matter how a statement $\ell$ is mutated, the test result will not be better, then $\ell$ is more likely to be correct.

Compared to SBFL, MBFL works at a more fine-grained level, observing the effects of each statement on testing results. However, mutating even just one statement will yield a large number of possible mutants, resulting in the search space explosion problem. Thus, MBFL may take hours per fault [27, 36, 61], finding few adoptions in APR tools [29].

Challenges. There are two key challenges in developing an effective and efficient FL technique. First, the effectiveness of FL with a single-information source (e.g., program coverage or mutation testing) is limited by the lack of other kinds of information. Approaches from different FL families may complement each other, but it is challenging to combine them to achieve a better effectiveness. Second, the efficiency and effectiveness dilemma arises when we combine different FL techniques. Even with machine learning, we will still need to collect information from different sources by recording program coverage or running mutation testing, inevitably increasing time overhead. Integrating more sources may achieve a better effectiveness, at the expense of a greater time overhead.

Insights. When combining FL techniques from different families, we should make full use of their respective advantages during the analysis process instead of running them independently before integrating their results. One possibility is to apply a pre-analysis to SBFL so that we can then subject only the statements with high suspiciousness scores to mutation testing. Another possibility is to reap the benefits from mutation testing, but in a time-controlled manner.

We build our work on top of *predicate switching* [57], a lightweight MBFL technique. However, this technique considers only the predicates asmutable elements and can thus dramatically reduce the mutation search space, since a predicate has only one of the two possible states (*true* and *false*). If switching a predicate $p$ makes a failed test case pass, $p$ is called a *critical predicate* and the faulty code may be found by manually examining its root cause, which can be assisted by dynamic slicing to reduce the number of potential faulty statements found [57]. However, if the actual faulty statement is far away from the critical predicate, quite a few related statements may be sliced but without a relative ranking. Besides, the critical predicate may not exist for a general fault. Therefore, predicate switching has a limited application in FL and so far only works for predicate-related faults in APR (e.g., ACS [51] uses predicate switching to locate faulty predicates).

Our Solution. In this paper, we propose FLIP, a lightweight FL tool that combines two FL families, SBFL and MBFL, by reaping both of their benefits. Figure 2 shows the workflow of FLIP. First, we follow a standard SBFL workflow to collect program coverage and compute a ranking list. Then we select relatively highly ranked predicates for mutation testing. For every mutated predicate, we make an inference as follows:

- For a critical predicate $p$, we collect all the statements that may affect the values of $p$ and regard them to be suspicious statements executed by a failed test case.
- For an uncritical predicate $p'$, we collect all the statements that only affect the values of $p'$ and regard them to be less suspicious statements executed by a passed test case.

Through the above mutation inferences, FLIP can obtain extra information from every mutated predicate no matter whether it is critical or not, which can be further integrated into the original program spectra seamlessly. Then we make a joint inference with the original SBFL formula and rerank the suspicious statements based on the enhanced program spectra. This approach applies to all faults except for the program without executed predicates, which is either rare or trivial.

In summary, this paper makes three main contributions:

- **Novel Approach.** We propose FLIP, a novel FL approach for boosting SBFL by leveraging mutation testing, in which non-program-coverage information can be inferred from every mutation to improve the effectiveness of FL.
- **Lightweight Technique.** When leveraging mutation testing, we only mutate selected predicates, with each producing only one mutant for testing. On average, FLIP spends less than 2 minutes for handling a single fault.
- **Extensive Evaluation on Real-World Faults.** We evaluate FLIP on Defects4J (version 1.5.0), which contains 438 real-world faults from six projects. All the seven SBFL techniques benefit from FLIP, e.g., by being now able to rank up to 46.4% more faults in top-1. In addition,
we provide some insights on how to further improve the effectiveness of FL on real-world faults.

II. A Motivating Example

We use a real-world example to illustrate how FLIP combines SBFL and MBFL to locate a faulty statement than either approach alone. Figure 3(a) illustrates the fault, Chart-17 in Defects4j, found in the class TimeSeries.

First, we go through a standard SBFL workflow, running test cases and recording the corresponding program coverage. Since all the seven formulae in Figure 1 generate the same ranking results for this fault, we use Ochiai as a representative for illustrations below. As shown in Figure 3(a), there is one failed test case \( t_1 (N_F = 1) \) and five passed test cases \( t_2 \sim t_6 (N_P = 5) \). Figure 3(b) gives the call graph generated from \( t_1 \). The method main() calls the constructor TimeSeries(), and then clone(), which calls createCopy() in \( \ell_{857} \) with the return value of getCount() as one of its arguments. The faulty statement is \( \ell_{857} \), which calls createCopy() with wrong arguments and thus triggers an illegal argument exception in \( \ell_{880} \). In the original program spectra comprising program coverage only, \( \ell_{857} \) is executed by one failed test case \( N_F(\ell_{857}) = 1 \) and two passed test cases \( N_P(\ell_{857}) = 2 \), resulting in a suspiciousness score of 0.58, ranking behind \( \ell_{880} \), which has the highest suspiciousness score 0.71.

To improve SBFL, FLIP works (by leveraging MBFL) as follows. We explain its three steps below.

**Step 1: Selecting and Mutating Predicates.** First, we select predicates for mutation testing. Instead of exhausting all the predicates in the program, we will settle with only relatively highly ranked ones in the original ranking list, since they are more likely to be related to the fault. For the fault in the example, only two predicates in \( \ell_{876} \) and \( \ell_{879} \) exist and both happen to be selected and mutated, producing one mutant for each (e.g., with the predicate in \( \ell_{876} \) mutated to \( s \geq 0 \)). Then we test each mutant by rerunning the failed test case \( t_1 \).

**Step 2: Inferring from Mutated Predicates.** After mutation testing, \( t_1 \) remains failed with the mutated \( \ell_{876} \) but passes with the mutated \( \ell_{879} \). We can infer that the outcome of the predicate in \( \ell_{876} (\ell_{879}) \) is most likely wrong (correct) when the original program runs under \( t_1 \). For the predicate in \( \ell_{879} \), FLIP collects all the statements that may affect its values and will regard them as suspicious. Figure 3(c) gives the data dependences on the predicate in \( \ell_{876} \). In addition to \( \ell_{876} \), three other suspicious statements, \( \ell_{175}, \ell_{238}, \ell_{857} \), make it possible for the faulty statement \( \ell_{857} \) to be captured successfully. FLIP also makes an inference from the mutated predicate in \( \ell_{876} \), which fails to overturn \( t_1 \), by assuming itself and the other statements that only affect the outcome of \( \ell_{876} \) to be less suspicious. For the fault in the example, only \( \ell_{876} \) is relevant. We will elaborate the details in Section III-B2.

**Step 3: Reranking with the Enhanced Program Spectra.** By mutation testing and inference, we have successfully distilled not only program coverage but also some dependence-related information from the program. To integrate all into the original program spectra, we treat the statements inferred from the critical (uncritical) predicates as the statements executed by failed (passed) test cases. We then rerank suspicious statements based on their suspiciousness scores calculated using the ranking formula but on the enhanced program spectra. As shown in the last column of Figure 3(a), the statements that are assumed to be suspicious (less suspicious) found in mutation inferences rank up (down), as desired, causing the actual faulty statement \( \ell_{857} \) to rise from the second (when SBFL is applied alone) to first in the ranking list (when MBFL is also exploited), made possibly by our new approach.

With the enhanced program spectra, \( \ell_{857} \) will rank the first as the most suspicious statement by all the seven SBFL
formulæ given in Figure 1 a result that is difficult to achieve with program coverage or mutation inferences alone. FLIP combines two complementary information sources to achieve efficiently better results without any learning process.

III. APPROACH

We describe FLIP’s three steps for selecting and mutating predicates (Section III-A), making inferences from mutated predicates (Section III-B), and reranking suspicious statements with the enhanced program spectra (Section III-C).

A. Selecting and Mutating Predicates

Once a faulty program has been pre-ranked by SBFL, we only select top $K$ predicates as mutation candidates. Currently, we set $K = 10$ by default and will discuss how the value of $K$ affects the FL effectiveness in Section V-B1. This mechanism increases our chances in obtaining fault-related information through mutation testing efficiently (in a limited time).

Given a set of candidate predicates $P = \{p_1, p_2, ..., p_K\}$, we generate a set of program mutants $M = \{m_{p_1}, m_{p_2}, ..., m_{p_K}\}$, in which the mutant $m_{p_i}$ is generated by transforming $p_i$ to $\neg p_i$ in the original program. Then, we rerun the set of failed test cases, $T_F$, on each $m_{p} \in M$ and divide $P$ into the following two sets according to the test results as follows:

\[
\exists t \in T_F, \text{ if } t \text{ passes on } m_p \rightarrow p \in P_c \\
\forall t \in T_F, \text{ if } t \text{ fails on } m_p \rightarrow p \in P_u
\]

where $t$ denotes a test case in $T_F$. $P_c$ and $P_u$ denote the set of critical and uncritical predicates obtained, respectively.

As pointed out in [57], a predicate may be executed multiple times in a test case, when, for example, it appears in a loop or recursion. To find out whether such a predicate $p$ is critical or not, we need to enumerate all the possible states of $p$. Figure 4 illustrates the search space of $p$ that is executed $n$ times, where $n > 1$, by a failed test case. Here, $p^i$ represents the outcome (true or false) of $p$ during its $i$-th execution. For every $p^i$ ($1 \leq i \leq n$), we may switch it or not, resulting in a huge search space with a total of $2^n - 1$ possible combinations. Without exhausting its search space, the predicate $p$ is not necessarily uncritical even if no failed test case passes. However, such an exhaustive search is nevertheless impractical.

To balance performance and precision, we choose to switch all the states of $p$ (during all its possible executions) by statically negating $p$. Then we keep $p$ during the further inference if it is critical and discard it if no failed test case has passed. This mutation strategy can be extended by selectively switching some states of $p$, by, e.g., selecting its first $k$ states.

B. Inferring from Mutated Predicates

After mutation testing, we apply two different algorithms to make mutation inferences from critical predicates (Section III-B1) and uncritical predicates (Section III-B2).

1) Inferring from Critical Predicates: For any critical predicate $p \in P_c$, we apply Algorithm 1 to collect a set of statements $S_p$ that may affect the values of $p$ and will classify them as being suspicious. For simplicity, our algorithm assumes the absence of static variables in a program. However, global variables can be handled in the standard manner [15].

Let us start with CriticalSlicer in lines 1-6. First, we have $ex$, a set of statements executed by the test cases failing on the original program but passing on $m_p$. Then we call InterSlicer to compute interprocedurally a backward slice starting from $ex$, where the critical predicate $p$ resides.

In InterSlicer (lines 7-25), we compute a backward slice from the statement $t$, as is done traditionally [15][45], except that we consider only the data dependences and the statements in $ex$ in line 18. CallGraph in line 22 represents the call graph of the program built using Soot [41].

Algorithm 1: Inferring from critical predicates

Input: a statement $t_p$

Output: a set of statements $S_p$

1 Function CriticalSlicer($t_p$)
2 Let $ex$ be an empty set;
3 foreach $t \in T_F$ do
4 if $t$ passes on $m_p$ then
5 Add statements executed by $t$ to $ex$;
6 return InterSlicer($t_p$, $\emptyset$, $ex$);
7 Function InterSlicer($\ell$, $S_p$, $ex$)
8 Let $mtd$ be the method containing $\ell$;
9 Let $cfg$ be the control flow graph of $mtd$;
10 Let $visited$ be an empty set;
11 Let $workList$ be an empty stack;
12 workList.push($\ell$);
13 while $workList$ is not empty do
14 $\ell = workList.pop();$
15 if $\ell \in visited$ then continue;
16 visited.add($\ell$);
17 foreach $\ell'$ in $cfg$ do
18 if $\ell'$ in $ex$ and $\ell'$ data $\ell$ then
19 workList.push($\ell'$);
20 $S_p.addAll(visited);$
21 if any statement in $visited$ uses any of $mtd$'s parameters then
22 foreach $\ell$ in $mtd$'s call sites in $CallGraph$ do
23 if $\ell \in ex$ and $\ell \notin S_p$ then
24 $S_p.addAll(InterSlicer(\ell, S_p, ex));$
25 return $S_p;
Example 1. We use some examples in Figure 5 to explain the reasons why we consider only data dependences while ignoring control and potential dependences [5] [12].

```
1 a = x + y;
2 if(a > 0){
3   b = x - y;
4       }
5 }
6 ...
7
(a) Control dependence
8 a = x + y;
9 if(a > 0)
10 b = x - y;
11 else
12 b = x * y;
13 if(a <= b)
14 ...
15
(b) Potential dependence
```

Fig. 5. Examples of control and potential dependences.

Figure 5(a) gives an example for illustrating control dependences. The mutated predicate \( p \) resides in \( \ell_4 \), which has a control dependence on \( \ell_2 \). However, \( \ell_2 \) only affects the execution of \( \ell_4 \) but not the value of \( p \). Since switching \( p \) can produce a correct execution, we mark \( \ell_2 \) as less suspicious and do not include \( \ell_2 \) in the slice. Note that the slice computed by our mutation inferences is not required to be executable.

Figure 5(b) gives an example for illustrating potential dependences. If the outcome of a branch guarded by a predicate \( p \) does not affect the execution of a statement \( \ell \), i.e., \( \ell \) has no control dependence on \( p \), but affects the values of the variables used in \( \ell \), \( \ell \) has a potential dependence on \( p \). As shown, the value of the mutated predicate \( p \) in \( \ell_{13} \) is potentially affected by the outcome of the predicate in \( \ell_9 \). Accounting for such potential dependences will result in conservatively an over-sized slice according to relevant slicing [5] [12] [59]. To obtain a more precise slice and locate execution omission errors better, Zhang et al. [59] introduced the notion of implicit dependences. For a potential dependence, if the outcome of predicate \( p \) does affect the test result, by e.g., making a failed test case pass, it is called an implicit dependence.

In Figure 5(b), if switching the predicate in \( \ell_9 \) makes a failed test case pass, \( \ell_{13} \) has an implicit dependence on \( \ell_9 \). Keeping track of only implicit dependences instead of all the potential dependences will significantly reduce the size of a slice without missing the actual faulty statement. Thus, Algorithm 1 captures only such implicit dependences. If the actual faulty statement is \( \ell_8 \) or \( \ell_9 \), it can be captured when we mutate \( \ell_9 \) and infer from it, even if it is missed when we mutate \( \ell_{13} \). As explained earlier, we have exploited predicate switching [57] to improve SBFL by increasing the chances for the predicate in \( \ell_9 \) to be selected within a given time budget.

2) Inferring from Uncritical Predicates: For any uncritical predicate \( p \in P_u \) that is executed only once, we apply Algorithm 2 to collect a set of statements \( S_p \) that only affect the value of \( p \) and regarded them as being less suspicious.

We use UncriticalSlicer to compute intraprocedurally a slice \( S_p \) starting from \( \ell_p \), where the uncritical predicate \( p \) resides. First, we compute \( ex \), the set of statements executed by all the failed test cases, in lines 2-4. As in the case of critical predicates, we consider only data dependences in computing the slice in lines 10-16. From \( S_p \), we remove iteratively the statements that are dependent on by any executed statement outside \( S_p \) in lines 17-25, until a fixed point has been achieved.

Example 2. Figure 6 gives an example for illustrating how Algorithm 2 works. Figure 6(a) gives the code and Figure 6(b) depicts the corresponding data dependence graph.

The candidate predicate \( p \) appears in \( \ell_3 \). After mutation testing, \( p \) is marked as uncritical. Then we conduct a backward slicing from \( p \), collecting the statements that \( \ell_3 \) depends on \( (\ell_1 \) and \( \ell_2 \)). However, we keep \( \ell_1 \) but remove \( \ell_2 \):

- Since \( \ell_1 \) affects only the value of \( p \), \( \ell_1 \) is more likely to be correct. Otherwise, switching \( p \) would have removed the effects of the faulty \( \ell_1 \), causing \( p \) to be critical.
- The above inference does not apply to \( \ell_2 \) because \( \ell_2 \)
C. Reranking with the Enhanced Program Spectra

To integrate two information sources, we can think of mutation inferences as also a process of discovering program coverage so that we can obtain a joint program spectra, together with SBFL, as motivated in Section IV. Inspired by [32, 33], we focus on how to assign different weights to the mutation inferences from two different types of predicates in order to maximize the effectiveness of our overall FL approach. We use \( w_c(p) \) to represent the weight of the mutation inference from a critical (an uncritical) predicate \( p \) and integrate them into the original program spectra as follows:

\[
\begin{align*}
N^*_F &= N_F + \sum_{p \in P_c} w_c(p) N_F(p) = N_F + \sum_{p \in P_c} w_c(p) \cdot 1_{S_p}(\ell) \\
N^*_P &= N_P + \sum_{p \in P_u} w_u(p) N_P(p) = N_P + \sum_{p \in P_u} w_u(p) \cdot 1_{S_p}(\ell)
\end{align*}
\]

where \( X^* \) is used to replace the corresponding symbol \( X \) in each SBFL formula in Figure 1. \( 1_{S_p}(\ell) \) is an indicator function that indicates whether the statement \( \ell \) exists in the slice \( S_p \) (\( 1_{S_p}(\ell) = 1 \) if \( \ell \in S_p \) and \( 1_{S_p}(\ell) = 0 \) otherwise).

We propose to calculate an adaptive weight \( w_c \) by:

\[
w_c(p) = \frac{N_F}{|P|} e^{\frac{N^*_F(p)}{N_F}}
\]

where \( N_F \) represents the number of failed test cases for the original program, \( |P| \) is the number of its critical predicates found, \( N^*_F(p) \) is the number of test cases that have changed from failed to passed on the mutant \( m_p \).

Let us consider \( N_F / |P| \) first. We use this component to balance the contributions made by SBFL and mutation inferences in the enhanced program spectra. Essentially, \( w_c(p) \) is proportional to \( N_F \) except that it is scaled by a factor \( |P| \). We have also included a second component \( e^{N^*_F(p)/N_F} \) to measure the confidence of the mutation inference according to the percentage of failed test cases that are overturned after mutation. The more test cases that the mutant turns over, the higher weight we assign to the corresponding mutation inference.

For \( w_u \), we adopt a fixed value, i.e., \( w_u(p) = 1 \) for every uncritical predicate \( p \in P_u \). For the mutation inferences from uncritical predicates, we treat them equally as the statements executed by passed test cases since their corresponding mutants fail to overturn any failed test case.

IV. EXPERIMENTAL SETUP

Since FLIP is designed as a practical FL tool, we evaluate it on real-world faults and compare its effectiveness and overhead with state-of-the-art SBFL techniques. To investigate the contribution of each component in FLIP, we evaluate FLIP with several configurations and take a further study on the complementarity of mutation inferences to the information obtained by SBFL and predicate switching. In summary, our evaluation aims to answer the following research questions:

- **RQ1**: Does FLIP outperform SBFL on real-world faults?
- **RQ2**: How does FLIP behave in terms of its effectiveness under different configurations?
- **RQ3**: Does FLIP provide complementary information to improve SBFL and predicate switching?
- **RQ4**: What is the overhead of FLIP?

A. Implementation

We have implemented FLIP in SOOT [41] on its Jimple IR by also using its CFGs and call graphs provided. For each fault, we first run its test suite and use GZoltar [8] to collect the corresponding program coverage. Following Figure 1, seven SBFL techniques have been implemented to generate the suspiciousness score for each statement. Then, we use SOOT to transform the selected predicates in bytecode and make mutation inferences (as described in Section IV).

Our experiments were done on a machine with an Intel Core i5 3.20 GHz CPU and 8GB memory, running Ubuntu 18.04 operating system with JDK 1.6.0_45 with the maximum heap size of JVM set as 4 GB. The time budget is 1 minute for a single test case and 10 minutes for the whole test suite (the default setting in GZoltar) and the test cases running over the time budget will not be added into the program spectra.

B. Dataset

For benchmarks, we use Defects4j [19] (version 1.5.0), containing 438 real-world faults from six open-source projects (Table I), to evaluate FLIP. Defects4j is one of the most popular datasets for APR [17, 32, 54]. In recent years, researchers [27, 36, 61] also use Defects4j to evaluate the effectiveness of FL techniques on real-world faults.

To measure the effectiveness of FL techniques, we first determine the faulty statements for each fault by referring to the report in [39] and manual analysis. The modified or deleted statements are marked as being faulty. For the code insertion, we follow [36] and mark the statement immediately following the inserted code as being faulty. The faults incompatible with the statement-level FL (e.g., those requiring new methods or classes to be inserted in a patch) are not considered.

C. Metrics

1) Expected First Rank: Following [61], we use \( E_{first} \), the expected rank of the first faulty element, to measure the effectiveness of an FL technique as follows:

\[
E_{first} = R_{start} + \sum_{k=1}^{T-F} k \left( \frac{T-k-1}{T-1} \right) \left( \frac{T-F}{T_F} \right)
\]

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>KLOC</th>
<th>#Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>Chart Library</td>
<td>85.1</td>
<td>26</td>
</tr>
<tr>
<td>Closure</td>
<td>Javascript Compiler</td>
<td>85.6</td>
<td>176</td>
</tr>
<tr>
<td>Lang</td>
<td>Java Utility</td>
<td>18.5</td>
<td>65</td>
</tr>
<tr>
<td>Math</td>
<td>Mathematics Library</td>
<td>50.6</td>
<td>106</td>
</tr>
<tr>
<td>Mockito</td>
<td>Mocking Framework</td>
<td>8.9</td>
<td>38</td>
</tr>
<tr>
<td>Time</td>
<td>Calendar System</td>
<td>27.3</td>
<td>27</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>438</td>
</tr>
</tbody>
</table>
This metric can handle real-world faults with multiple faulty statements and tied ranks. In a ranking list, suppose that the first faulty statement appears in \( T \) statements with the same tied rank from the position \( R_{\text{start}} \) to the position \( (R_{\text{start}} + T - 1) \), with \( T_F \) faulty statements among the \( T \) statements. We measure the expected rank of the first faulty statement by calculating the summation of the probability for the first faulty statement appearing in every \( k \)-th location after \( R_{\text{start}} \). \((\frac{T}{T_F})\) is the number of combinations of all \( T \) tied statements containing \( T_F \) faulty statements and \((\frac{T-k-1}{T_F-1})\) is the number of combinations of the remaining tied statements after the first faulty statement with the remaining \( T_F-1 \) faulty statements.

According to Equation 3, \( E_{\text{first}} \) is equal to \( R_{\text{start}} \) when all the \( t \) tied statements are faulty \((T_F = T)\) and will thus reduce to the average rank if \( T_F = 1 \) \([7, 27]\):

\[
E_{\text{first}} = R_{\text{start}} + \frac{T - 1}{2}
\]

We use Equation 2 instead of Equation 3 to handle multiple faulty statements and tied statements because Equation 3 may unnecessarily lower their ranks \([61]\). For example, if all the first 3 statements in the list are faulty and tied, we will obtain rank 1 by Equation 2 but rank 2 by Equation 3. However, the first statement that we check must be faulty.

**Top-N**: We use the number of faults reported in Top-N according to \( E_{\text{first}} \) to measure the effectiveness of an FL technique, where \( N \in \{1,3,5\} \). In practice, Top-N can be quite important because (1) only a few elements are manually checked during debugging (e.g., over 70% developers only check the top-5 elements \([22]\)) and (2) an FL technique with the higher Top-N value will improve both the efficiency and correctness of APR tools, since a plausible patch may be found from a few repair operations \([6]\).

**MEF**: We adopt MEF, the mean value of \( E_{\text{first}} \) for all faults, as another metric. The smaller MEF is, the more effective the corresponding FL technique will be.

2) **Mean Average Rank**: If multiple faulty statements exist, \( E_{\text{first}} \) will be unable to precisely measure the distribution of each faulty statement. We use \( R_{\text{average}} \) to calculate the average rank of all faulty statements as follows:

\[
R_{\text{average}} = \sum_{i=1}^{F} \frac{R_{i}^f + R_{i}^l}{2F}
\]

where \( R_{i}^f \) and \( R_{i}^l \) are the ranks of the first and the last statements in the list with the same suspiciousness score to the \( i \)-th faulty statement and \( F \) is the total number of faulty statements in the program considered.

**MAR**: We adopt MAR, the mean value of all \( R_{\text{average}} \) for all faults, as another metric. Again, the smaller MAR is, the more effective the corresponding FL technique will be.

V. RESULTS AND ANALYSIS

A. RQ1: Overall Effectiveness Comparison

We compare the effectiveness of seven SBFL techniques in Figure 1 with and without FLIP. For each SBFL technique, we apply mutation inferences to the top 10 predicates in the enhanced program spectra using the same formula. We present the results in Table II, where its columns represent all the metrics used and its rows the seven SBFL formulae. For each formula, the first and the second rows give the results without and with FLIP, respectively.

<table>
<thead>
<tr>
<th>Technique</th>
<th>@1</th>
<th>@3</th>
<th>@5</th>
<th>MEF</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>32</td>
<td>112</td>
<td>158</td>
<td>22.22</td>
<td>30.27</td>
</tr>
<tr>
<td>FLIP-Ochiai</td>
<td>46</td>
<td>134</td>
<td>166</td>
<td>20.96</td>
<td>29.47</td>
</tr>
<tr>
<td>Tarantula</td>
<td>28</td>
<td>110</td>
<td>156</td>
<td>19.90</td>
<td>27.80</td>
</tr>
<tr>
<td>FLIP-Tarantula</td>
<td>34</td>
<td>133</td>
<td>167</td>
<td>18.67</td>
<td>26.90</td>
</tr>
<tr>
<td>Jaccard</td>
<td>29</td>
<td>109</td>
<td>155</td>
<td>19.79</td>
<td>27.75</td>
</tr>
<tr>
<td>FLIP-Jaccard</td>
<td>37</td>
<td>130</td>
<td>166</td>
<td>18.63</td>
<td>26.97</td>
</tr>
<tr>
<td>Kulkowicz12</td>
<td>28</td>
<td>108</td>
<td>155</td>
<td>22.37</td>
<td>30.89</td>
</tr>
<tr>
<td>FLIP-Kulkowicz12</td>
<td>41</td>
<td>128</td>
<td>161</td>
<td>21.20</td>
<td>30.06</td>
</tr>
<tr>
<td>DStar2</td>
<td>31</td>
<td>107</td>
<td>149</td>
<td>22.58</td>
<td>30.72</td>
</tr>
<tr>
<td>FLIP-DStar2</td>
<td>41</td>
<td>126</td>
<td>161</td>
<td>21.71</td>
<td>30.34</td>
</tr>
<tr>
<td>Zoltar</td>
<td>28</td>
<td>108</td>
<td>153</td>
<td>22.18</td>
<td>29.86</td>
</tr>
<tr>
<td>FLIP-Zoltar</td>
<td>41</td>
<td>128</td>
<td>159</td>
<td>21.23</td>
<td>28.89</td>
</tr>
<tr>
<td>Goodman</td>
<td>31</td>
<td>109</td>
<td>157</td>
<td>19.54</td>
<td>27.41</td>
</tr>
<tr>
<td>FLIP-Goodman</td>
<td>44</td>
<td>132</td>
<td>165</td>
<td>18.39</td>
<td>26.65</td>
</tr>
</tbody>
</table>

Overall, FLIP outperforms all the seven SBFL techniques in all the metrics. In terms of the Top-N metric, FLIP outperforms seven SBFL techniques by 21.4% – 46.4% for Top-1, 17.8% – 21.1% for Top-3, and 3.9% – 7.1% for Top-5. Once enhanced with mutation inferences, Ochiai, which is the best performer in terms of the Top-N metric among these seven techniques, has been improved by 43.8%, 19.6%, and 5.1% for Top-1, Top-3 and Top-5, respectively. Besides, FLIP outperforms these SBFL techniques by 3.9% – 6.2% for MEF and 1.2% – 3.3% for MAR. Goodman, which is the best performer in terms of MEF and MAR, is further improved by 5.9% for MEF and 2.8% for MAR with our FLIP. The improvements made by FLIP in terms of the average-based metrics, MEF and MAR, are less than Top-N. This is because not all the faults benefit from FLIP, which will be discussed in Section V-E.

Since Ochiai is one of the most commonly used techniques for FL \([36, 61]\) and APR \([20, 23, 24, 46, 54]\), we will use it as the main formula to evaluate FLIP further below.

B. RQ2: Impacts of Configurations

In this section, we evaluate the impacts of different configurations on FLIP’s effectiveness in terms of Top-N. We focus on two factors, the number of mutated predicates \( K \) and the weights of mutation inferences for critical \( (w_c) \) and uncritical \( (w_u) \) predicates. We will use the default setting (Ochiai, \( K=10 \), \( w_c \) and \( w_u \)), as described in Section III-C.

1) **Number of Mutated Predicates**: Figure 7(a) presents the impacts of mutating different numbers of highly ranked predicates \( K \in \{0, 1, 2, 4, 8, 10, 15\} \) in the original ranking list. Overall, Top-N initially increases and then becomes flat...
Fig. 7. Analyzing impacts of different configurations on the effectiveness of FLIP. K denotes the number of mutated predicates. FLIP’s default settings are displayed in bold. \( \sim \) represents the baseline, SBFL without mutation inferences. \( \sim \) denotes the adaptive weight computed according to Equation [1].

when more predicates are selected. For \( K \in \{1, 2\} \), however, FLIP is the most impressive. For example, Top-1 increases by 15.6% for \( K = 1 \) and 13.5% for \( K = 2 \). When \( K > 2 \), Top-N gradually stabilizes, though. These results indicate that SBFL techniques can be used to select predicates to be mutated and the predicates with relatively high suspiciousness scores are more likely to help improve the effectiveness of FL.

2) Weights of Mutation Inferences: Now we separately investigate the impacts of weights of mutation inferences from critical predicates \( w_c \) and uncritical predicates \( w_u \).

To evaluate the impacts of \( w_c \), we set \( w_u = 1 \) and compare our adaptive approach for computing \( w_c \) in Equation [1] which varies from 0.3 to 30 for different faults, with a naive approach for picking a set of fixed \( w_i \)'s in different orders of magnitude \( (w_c \in \{0, 10^{-1}, 1, 10^1, 10^2\}) \). Figure 7(b) gives FLIP’s effectiveness for Top-N. Overall, the Top-N value first increases and then drops, indicating that both program coverage and mutation inference are effective and should be combined with balanced weights. Our adaptive approach outperforms the naive approach. For example, we obtain 46 faults but the naive approach finds at most 44 with \( w_c = 10 \) for Top-1. Note that when \( w_c = 0 \), FLIP still outperforms the baseline. For example, the Top-1 value increases by 12.5%, indicating that mutation inferences from uncritical predicates make considerable contributions to the overall effectiveness.

Similarly, we compare the impacts of \( w_u \) when \( w_u \) ranges over \( \{0, 10^{-1}, 1, 10^1, 10^2\} \). Figure 7(c) gives the results. When \( w_u \) increases from 0 to \( 10^{-1} \), all the Top-1, Top-3 and Top-5 values increase. However, increasing \( w_u \) any further has little effectiveness impact. This is because the mutation inference from an uncritical predicate only discovers less suspicious statements, and continuously lowering their ranks hardly affects the statements at the top of the list.

In summary, FLIP benefits from mutation inferences for both critical and uncritical predicates and computing \( w_c \) adaptively is more effective than using some fixed values.

C. RQ3: Complementarity with SBFL and Predicate Switching

We have designed two experiments to show that FLIP provides complementary information to improve the effectiveness of SBFL and predicate switching.

We first build an oracle model combining Ochiai and predicate switching (PS). When PS has successfully identified all the faulty statements and obtained a better result than Ochiai for \( E_{first} \), the oracle model will use PS. Otherwise, the oracle will use Ochiai instead. We present the results in the top half of Table III. For Top-1, Top-3, MEF and MAR, FLIP outperforms the oracle model, since FLIP can find the root cause of a fault with a joint inference from program coverage and mutation inferences. Besides, the enhanced program spectra with an adaptive \( w_c \) are quite effective for information integration. It is not surprising that FLIP fails to beat oracle in Top-N for some N (e.g., Top-5) because FLIP considers both information sources in a heuristics-based manner but the oracle model always chooses the technique with a better effectiveness.

Second, we build three oracle models, ORACLE (SBFL) combining seven SBFL techniques, ORACLE (SBFL+PS) combining seven SBFL techniques and PS, and ORACLE (FLIP) combining seven SBFL techniques equipped with FLIP, respectively. Similarly, each oracle model opts to use the technique with the best effectiveness among all its supported techniques. For ORACLE (FLIP), we run FLIP with different SBFL technique independently before combining their results with an oracle. We present the results in the bottom half of Table III. Overall, ORACLE (FLIP) outperforms the other two models in all the metrics. For example, compared with ORACLE (SBFL), ORACLE (FLIP)’s effectiveness is 40.5%, 25.4% and 11.4% higher in Top-1, Top-3, and Top-5, respectively. ORACLE (FLIP) achieves this by leveraging mutation inferences for both critical and uncritical predicates, which is complementary information to program coverage, while SBFL techniques, even assisted by many different formulae, obtain highly correlated information. Besides, ORACLE (FLIP) outperforms ORACLE (SBFL+PS). For example, ORACLE (FLIP)’s effectiveness is 13% higher for Top-1, indicating that the mutation inferences are also complementary to PS.

D. RQ4: Overhead

On average, FLIP takes 85 seconds to handle each fault. Table IV presents the overheads in different projects with a different number of test cases. FLIP spends 128 and 106 seconds on Closure-1 and Mockito-1, respectively, since both have a lot of failed test cases executed multiple times during mutation testing. Closure-1, a fault in the compiler application, costs more time in mutation inferences due to its relatively complex dependence graphs. Time-1 has a low

table

<table>
<thead>
<tr>
<th>Technique</th>
<th>@1</th>
<th>@3</th>
<th>@5</th>
<th>MEF</th>
<th>MAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai</td>
<td>32</td>
<td>112</td>
<td>158</td>
<td>22.22</td>
<td>30.27</td>
</tr>
<tr>
<td>ORACLE (Ochiai+PS)</td>
<td>41</td>
<td>125</td>
<td>171</td>
<td>21.62</td>
<td>29.67</td>
</tr>
<tr>
<td>FLIP-Ochiai</td>
<td>46</td>
<td>134</td>
<td>166</td>
<td>20.96</td>
<td>29.47</td>
</tr>
<tr>
<td>ORACLE (SBFL)</td>
<td>37</td>
<td>118</td>
<td>166</td>
<td>17.03</td>
<td>25.04</td>
</tr>
<tr>
<td>ORACLE (SBFL+PS)</td>
<td>46</td>
<td>131</td>
<td>179</td>
<td>16.44</td>
<td>24.45</td>
</tr>
<tr>
<td>ORACLE (FLIP)</td>
<td>52</td>
<td>148</td>
<td>185</td>
<td>15.59</td>
<td>24.08</td>
</tr>
</tbody>
</table>
Table IV: Analyzing FLIP’s overhead.

<table>
<thead>
<tr>
<th>Fault</th>
<th>#Test</th>
<th>#Fail</th>
<th>B-T</th>
<th>S-T</th>
<th>D-T</th>
<th>T-T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart-1</td>
<td>436</td>
<td>1</td>
<td>1m 59s</td>
<td>39s</td>
<td>8s</td>
<td>47s</td>
</tr>
<tr>
<td>Lang-1</td>
<td>173</td>
<td>1</td>
<td>41s</td>
<td>13s</td>
<td>23s</td>
<td>36s</td>
</tr>
<tr>
<td>Math-1</td>
<td>313</td>
<td>2</td>
<td>1m 13s</td>
<td>15s</td>
<td>41s</td>
<td>56s</td>
</tr>
<tr>
<td>Time-1</td>
<td>1126</td>
<td>1</td>
<td>7m 56s</td>
<td>15s</td>
<td>23s</td>
<td>38s</td>
</tr>
<tr>
<td>Closure-1</td>
<td>374</td>
<td>8</td>
<td>2m 13s</td>
<td>57s</td>
<td>1m 11s</td>
<td>2m 8s</td>
</tr>
<tr>
<td>Mockito-1</td>
<td>969</td>
<td>26</td>
<td>7m 28s</td>
<td>14s</td>
<td>1m 32s</td>
<td>1m 46s</td>
</tr>
</tbody>
</table>

B-T is the baseline time with only SBFL. S-T is the time of static analysis, including mutating predicates and mutation inferences. D-T is the time of dynamic testing of mutants. T-T is the total overhead time.

overhead (38 seconds), since there is only one failed test case in the test suite with over one thousand test cases.

Compared with SBFL techniques, FLIP spends more time on rerunning failed test cases for each mutant and applying static analysis for mutation inferences. While MBFL generates a tremendous number of mutants [27][50][52], FLIP is quite efficient because only predicates are mutated, resulting in a limited number of mutants. Although this constraint makes it difficult for FLIP to directly locate faulty statements that are irrelevant to the values of predicates, FLIP still has the opportunity to raise their rankings by mutation inferences, which can be applied to both critical and uncritical predicates. As shown in Section V-B2, FLIP still outperforms SBFL even in the absence of critical predicates in the program.

E. Discussion

Although FLIP outperforms SBFL in all the metrics, we find that there are some faults that cannot benefit from our mutation inferences. We have manually analyzed most of them and will discuss below the main reasons and some possible solutions.

Figure 8 presents the percentages of faults that have caused \( E_{first} \) to increase, stay unchanged and decrease with mutation inferences. Figure 8(a) shows the overall percentages of ranking changes, with 46.9% up, 37.1% unchanged, and 16.0% down. Figure 8(b) gives the changes in different projects.

When the faulty statement is already in the top of the ranking list or no predicate has been executed by a failed test case, FLIP fails to further improve the FL effectiveness. Besides, we present other three main reasons why a fault cannot benefit from FLIP and their possible solutions.

1) FL with Insufficient Granularity: In this paper, our FL technique operates at the statement level, which is fine-grained for manual inspection and commonly used in repair tools. However, it may still be too coarse-grained in some cases.

Figure 9 presents the fault LANG-8 in Defects4j and its corresponding patch. This fault is due to the code omission, missing a statement before \( \ell_{1134} \). To measure the FL effectiveness, we follow \[ 36 \] and mark the immediately following statement \( \ell_{1134} \) as the faulty statement. However, FLIP marks \( \ell_{1134} \) as an uncritical predicate and lowers its ranking by mutation testing and inferences. FLIP has indeed succeeded in obtaining the correct information about \( \ell_{1134} \) but its statement-level abstraction is too coarse to model the missed code.

One possible solution is to distinguish the omitted and faulty statements (by, e.g., reporting locations between two statements instead of only existing statements). This will enable FLIP to guide APR tools better in, e.g., selecting repair operators (e.g., replacement or insertion).

\[
1133 \text{ public void appendTo(...)} \{
1134 \quad + \text{TimeZone zone = calendar getTimeZone();}
1135 \quad \text{if(zone.useDaylightTime())}
1136 \quad \quad \&\& \text{calendar.get(...)} \neq \emptyset \{ ...
1137 \}\}
\]

Fig. 9. The patch for the fault LANG-8 in Defects4j.

Fig. 10. A failed test case for LANG-8.

2) Test Case Merging: To make test code more readable and maintainable, developers usually merge several test cases or assertions with the same test goal into one [9][60]. However, such test case merging may affect the effectiveness of FL techniques [27] since the failure of a sub-case will result in the entire case marked as failed. Figure 10 presents a failed test case with three sub-cases for LANG-58, in which an exception occurs during the execution of \( \ell_{1371} \). In mutation testing, one mutant passes the sub-case in \( \ell_{1371} \) but fails the one in \( \ell_{1369} \), which ultimately leads to the test to remain failed.

One solution is to transform a test case before testing [27]. After we have manually split testLang300() into three individual test cases, FLIP has successfully found a critical predicate with its \( E_{first} \) rising from 24.5 to 2.5. SBFL also benefits from this due to the more fine-grained program spectra used (e.g., with \( E_{first} \) of Ochiai rising from 14 to 13.5).

3) Multiple Fault Locations: When a fault is caused by multiple correlated faulty statements, finding critical predicates may require switching multiple predicates at the same time [57], which will dramatically increase the search space. The search space explosion is a common challenge for existing FL techniques. For example, SBFL will mistakenly raise the ranking of the convergence of branches if faulty statements reside in multiple branches. MBFL has to mutate multiple statements at the same time to overturn test results. Locating and fixing multiple faults may require heuristics (e.g., code similarities [38] or specific bug models [14]).
F. Threats to Validity

First, we are concerned about whether the metrics used reflect real-world situations. We mitigate this threat to validity by using the expected ranking of the first faulty statement ($E_{first}$) and the average ranking of all faulty statements ($R_{average}$). However, as discussed in Section V-E1, how to model some tricky situations, e.g., code omission, needs further studies.

Second, we are concerned about the effects of configurations on FLIP’s effectiveness. As discussed in Section V-B, an inappropriate configuration will impair FLIP’s effectiveness. To reduce this threat to validity, we have evaluated FLIP as comprehensively possible under a range of configurations.

VI. RELATED WORK

A. Fault Localization with a Single Information Source

Spectrum-based fault localization (SBFL) techniques usually rely on program coverage, the execution traces generated by passed and failed test cases. Jones et al. first proposed Tarantula [18] to rank statements by distinguishing the executions of passed and failed test cases. Abreu et al. then introduced Jaccard [11], Ochiai [2] and Barinel [3]. Wong et al. [47] proposed DStar, one of the state-of-the-art SBFL techniques. These techniques make different assumptions on programs, test cases, and their relationships with faults [30]. However, SBFL only focuses on the coverage information without considering how each statement affects test results [35]. Due to its relative effectiveness and efficiency, FLIP adopts SBFL as the basic information source and uses it to guide the selection of candidate predicates, effectively as evaluated in Section V-B1.

Mutation-based fault localization (MBFL) techniques, such as MUSE [32] and METALLAXIS [34], repeatedly transform the statements in a buggy program and rank them by analyzing how the mutation affects the test results. Given an unlimited time budget and a sufficient number of mutation operators, MBFL can precisely locate the faulty statement. However, in reality, it needs to re-execute the test cases for every single mutation, leading to an unaffordable time overhead [27, 30, 61]. Therefore, we did not use MBFL as the the basis in FLIP.

Zhang et al. [57] introduced predicate switching, which can be regarded as a lightweight MBFL since the outcome of a predicate is only true or false, to locate the fault that triggers a wrong program execution. Wang and Roychoudhury [43] also toggle the outcomes of some predicates to generate a successful run from the failing run. Besides, predicate switching can also be used to increase program coverage in vulnerability detection [28, 37, 50]. However, predicate switching only works for predicate-related faults in fault localization and only repair tools that focus on condition synthesis (used, in e.g., ACS [51]). While inspired by [57], FLIP distills useful information through different mutation inferences no matter how the predicate transformation affects the test results.

Dynamic slicing [41, 13, 42, 44, 58] can assist software debugging by reducing the number of suspicious statements but fail to further rank them. To improve its precision and make it compatible with repair tools, slicing is usually integrated with other FL techniques. FLIP uses two slicing algorithms (modified for our purposes) to assist mutation inferences for critical and uncritical predicates.

B. Fault Localization with Multiple Information Sources

Xuan and Monperrus [55] proposed MULTRIC, a model learning to combine different formulae in SBFL in order to outperform individual formula. Le et al. [7] augmented SBFL with Daikon [10] invariants. Sohn and Yoo proposed FLUCCS [40] that learns to rank by using multiple SBFL formulae and code change metrics as features. Li and Zhang [27] extended MBFL and learn to rank by transforming test code and error messages. Zou et al. [61] observed that the FL techniques from one family contain strongly correlated information and proposed COMBINEFL to improve the effectiveness of previous learning-to-rank techniques by combining the FL techniques from different families (e.g., SBFL and MBFL).

FLIP is inspired by prior work with multiple information sources but does not require a learning process. Besides, FLIP makes mutation inferences, providing complementary information to SBFL and predicate switching (Section V-C).

C. Program Repair

APR [25, 33] aims at generating a patch automatically to reduce the cost of software maintenance. FL techniques are usually employed to first find the faulty statement. SBFL is the most commonly used FL technique in APR due to its lightweightness. Ochiai, one of the most popular FL formulae, is used widely in the APR literature [20, 23, 24, 46, 54].

The effectiveness of FL directly affects the efficiency and correctness of APR tools [6, 25, 53]. For example, Nguyen et al. [53] find that when using Ochiai instead of Tarantula, SEMFix fixes two more faults in one project and one less in another. SBFL has also been extended to meet their needs, by assisting semantic code search [21] and combining SBFL and predicate switching to locate faulty predicates [51].

FLIP is compatible with most of the APR tools because it is lightweight, fine-grained, and designed for general faults (i.e., requiring neither a specification nor a bug model).

VII. CONCLUSION

We have presented an effective and efficient FL approach, FLIP, to improve SBFL by applying mutation inferences on critical and uncritical predicates in the program. FLIP outperforms seven state-of-the-art SBFL techniques in terms of their effectiveness in locating faults with small performance overheads. In future work, we plan to deploy FLIP in APR tools to provide richer information about fault locations and guide their selection about repair operators used.

VIII. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by Australian Research Grants ((DP170103956 and DP180104069).


