 X. Xu, C. Zou and J. Xue, Every Mutation Should Be Rewarded: Boosting Fault Localization with Mutated Predicates, 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 2020, pp. 196-207, doi: 10.1109/ICSME46990.2020.00028.

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 [1–3, 18, 47, 56], slicing-based [12, 59], mutation-based [32, 34] and machine-learning-based [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 [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).

Ochiai [2]:	$s(\ell) = \frac{\mathcal{N}_\mathcal{F}(\ell)}{\sqrt{\mathcal{N}_\mathcal{F} \cdot (\mathcal{N}_\mathcal{P}(\ell)+\mathcal{N}_\mathcal{F}(\ell))}}$
Tarantula [18]:	$s(\ell) = \frac{\mathcal{N}_\mathcal{F}(\ell)/\mathcal{N}_\mathcal{F}}{\mathcal{N}_\mathcal{F}(\ell)/\mathcal{N}_\mathcal{F}+\mathcal{N}_\mathcal{P}(\ell)/\mathcal{N}_\mathcal{P}}$
Jaccard [1]:	$s(\ell) = \frac{\mathcal{N}_\mathcal{F}(\ell)}{\mathcal{N}_\mathcal{F}+\mathcal{N}_\mathcal{P}(\ell)}$
Kulczynski2 [26]:	$s(\ell) = \frac{1}{2}(\frac{\mathcal{N}_\mathcal{F}(\ell)}{\mathcal{N}_\mathcal{F}} + \frac{\mathcal{N}_\mathcal{F}(\ell)}{\mathcal{N}_\mathcal{F}(\ell)+\mathcal{N}_\mathcal{P}(\ell)})$
DStar2 [47]:	$s(\ell) = \frac{\mathcal{N}_\mathcal{F}(\ell)^2}{\mathcal{N}_\mathcal{P}(\ell)+(\mathcal{N}_\mathcal{F}-\mathcal{N}_\mathcal{F}(\ell))}$
Zoltar [16]:	$s(\ell) = \frac{\mathcal{N}_\mathcal{F}(\ell)}{\mathcal{N}_\mathcal{F}+\mathcal{N}_\mathcal{P}(\ell)+\frac{10000(\mathcal{N}_\mathcal{F}-\mathcal{N}_\mathcal{F}(\ell))\mathcal{N}_\mathcal{P}(\ell)}{\mathcal{N}_\mathcal{F}(\ell)}}$
Goodman [11]:	$s(\ell) = \frac{3\mathcal{N}_\mathcal{F}(\ell)-\mathcal{N}_\mathcal{F}-\mathcal{N}_\mathcal{P}(\ell)}{\mathcal{N}_\mathcal{F}(\ell)+\mathcal{N}_\mathcal{F}+\mathcal{N}_\mathcal{P}(\ell)}$

Fig. 1. Seven state-of-the-art SBFL formulae. $\mathcal{N}_\mathcal{P}$ ($\mathcal{N}_\mathcal{F}$) represents the total number of passed (failed) test cases in the test suite. $\mathcal{N}_\mathcal{P}(\ell)$ ($\mathcal{N}_\mathcal{F}(\ell)$) represents the number of passed (failed) test cases that cover the statement ℓ.

Prior Work and Limitations. Spectrum-based fault localization (SBFL) represents one of the most popular FL families [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 $s(\ell)$, the suspiciousness score of a statement ℓ, 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 [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 [61].

Another popular FL family is mutation-based fault localization (MBFL) [32, 34], which iteratively mutates a faulty program and tests mutants against the test cases. MBFL suggests fault locations based on the following assumptions:

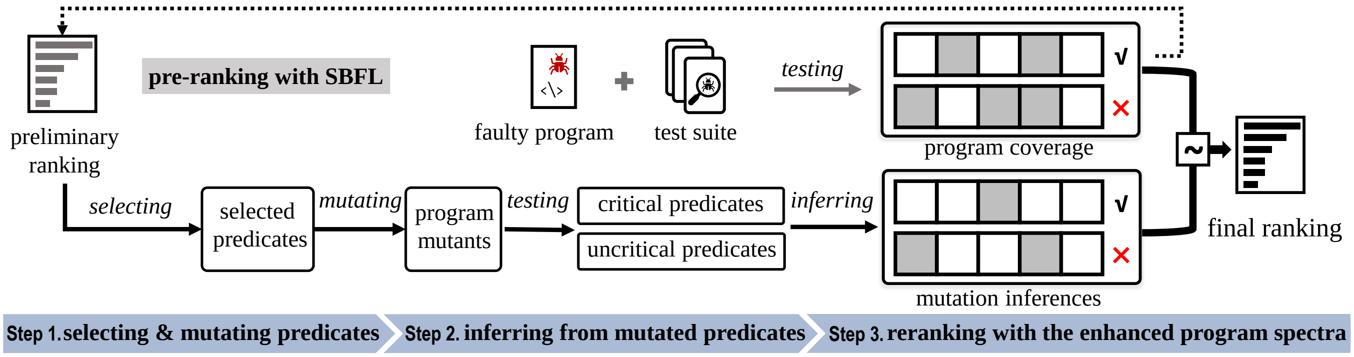


Fig. 2. The workflow of FLIP.

- If mutating a statement ℓ makes more test cases pass, ℓ is likely to be the faulty statement.
- If no matter how a statement ℓ is mutated, the test result will not be better, then ℓ 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 as mutable 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:

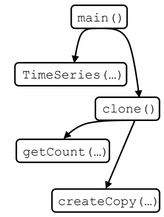
- For a critical predicate p, we collect all the statements that *may* affect the values of p and regard them to be *suspicious* statements executed by a *failed* test case.
- For an uncritical predicate p', we collect all the statements that *only* affect the values of p' and regard them to be *less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

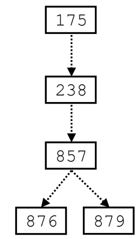
- *Novel Approach*. We propose FLIP, a novel FL approach for boosting SBFL by leveraging mutation testing, in which non-program-coverage information can be inferred from every mutation to improve the effectiveness of FL.
- *Lightweight Technique*. When leveraging mutation testing, we only mutate selected 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,

	Faulty Program	Program Coverage								Mutation Inference			
		t_1	t_2	t_3	t_4	t_5	t_6	Score	Rank	ℓ_{879}	ℓ_{876}	Score	Rank
-	`public TimeSeries(...){`												
-	`  ...;`												
175	`  this.data=new ArrayList();`	✗	✓	✓	✓	✓	✓	0.41	8	✗		0.54	5 ↑
176	`  this.maxCount=Integer.MAX_VALUE;`	✗	✓	✓	✓	✓	✓	0.41	8			0.29	8
177	`  this.maxAge=Long.MAX_VALUE;`	✗	✓	✓	✓	✓	✓	0.41	8			0.29	8
-	`}`												
-	`public int getCount(...){`												
238	`  return this.data.size();`	✗	✓	✓	✓	✓	✓	0.41	8	✗		0.54	5 ↑
-	`}`												
-	`public Object clone(){`												
857	`  Object clone=createCopy(0,getCount()-1);`	✗		✓	✓			0.58	2	✗		0.71	1 ↑
858	`  return clone;`			✓	✓			0.00	11			0.00	11
-	`}`												
-	`public TimeSeries createCopy(int s,int e){`												
876	`  if(s < 0){`	✗		✓	✓	✓	✓	0.45	4		✓	0.29	8 ↓
877	`    ...;//IllegalArgumentException is thrown`						✓	0.00	11			0.00	11
-	`  }`												
879	`  if(e < s){`	✗		✓	✓	✓	✓	0.45	4	✗		0.58	2 ↑
880	`    ...;//IllegalArgumentException is thrown`	✗					✓	0.71	1			0.50	3 ↓
-	`  }`												
882	`  TimeSeries copy=(TimeSeries)super.clone();`			✓	✓	✓	✓	0.00	11			0.00	11
-	`  ...;}`												

(a) The fault `Chart-17` in `Defects4j` with its original and added program spectra. t_i denotes the i-th test case. ✓(✗) indicates that the statement is executed by a passed (failed) test case.



(b) Call Graph of t_1



(c) Data dependences on mutated predicates

Fig. 3. A motivating example (with simplifications).

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 ($\mathcal{N}_\mathcal{F} = 1$) and five passed test cases t_2 – t_6 ($\mathcal{N}_\mathcal{P} = 5$). 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 ℓ_{857} with the return value of `getCount()` as one of its arguments. The faulty statement is ℓ_{857}, which calls `createCopy()` with wrong arguments and thus triggers an illegal argument exception in ℓ_{880}. In the original program spectra comprising program coverage only, ℓ_{857} is executed by one failed test case ($\mathcal{N}_\mathcal{F}(\ell_{857}) = 1$) and two passed test cases ($\mathcal{N}_\mathcal{P}(\ell_{857}) = 2$), resulting in a suspiciousness score of 0.58, ranking behind ℓ_{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 ℓ_{876} and ℓ_{879} exist and both happen to be selected and mutated, producing one mutant for each (e.g., with the predicate in ℓ_{876} mutated to $s >= 0$). 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 ℓ_{876} but passes with the mutated ℓ_{879}. We can infer that the outcome of the predicate in ℓ_{879} (ℓ_{876}) is most likely wrong (correct) when the original program runs under t_1. For the predicate in ℓ_{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 ℓ_{879}. In addition to ℓ_{879}, three other suspicious statements, ℓ_{175}, ℓ_{238}, and ℓ_{857}, make it possible for the faulty statement ℓ_{857} to be captured successfully. FLIP also makes an inference from the mutated predicate in ℓ_{876}, which fails to overturn t_1, by assuming itself and the other statements that only affect the outcome of ℓ_{876} to be less suspicious. For the fault in the example, only ℓ_{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 ℓ_{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, ℓ_{857} will rank the first as the most suspicious statement by all the seven SBFL

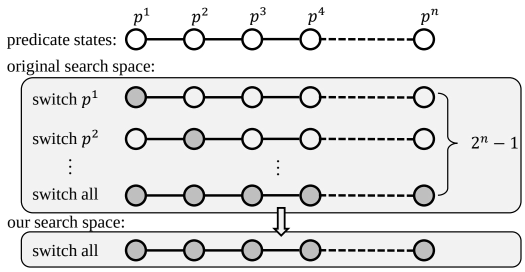


Fig. 4. Search space of predicate states.

formulae 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, $T_{\mathcal{F}}$, on each $m_p \in M$ and divide P into the following two sets according to the test results as follows:

$$\exists\ t \in T_{\mathcal{F}}, t \text{ passes on } m_p \Longrightarrow p \in P_c$$
$$\forall\ t \in T_{\mathcal{F}}, t \text{ fails on } m_p \Longrightarrow p \in P_u$$

where t denotes a test case in $T_{\mathcal{F}}$. 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 $(1 \leqslant i \leqslant n)$, 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.

Algorithm 1: Inferring from critical predicates

```
   Input: a statement ℓ_p
   Output: a set of statements S_p
1  Function CriticalSlicer(ℓ_p)
2      Let ex be an empty set;
3      foreach t ∈ T_F do
4          if t passes on m_p then
5              Add statements executed by t to ex;
6      return InterSlicer(ℓ_p, ∅, ex);
7  Function InterSlicer(ℓ, S_p, ex)
8      Let mtd be the method containing ℓ;
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(ℓ);
13     while workList is not empty do
14         ℓ = workList.pop();
15         if ℓ ∈ visited then continue;
16         visited.add(ℓ);
17         foreach ℓ' in cfg do
18             if ℓ' ∈ ex and ℓ' --data--> ℓ then
19                 workList.push(ℓ');
20     S_p.addAll(visited);
21     if any statement in visited uses any of mtd's
        parameters then
22         foreach ℓ in mtd's call sites in CallGraph do
23             if ℓ ∈ ex and ℓ ∉ S_p then
24                 S_p.addAll(InterSlicer (ℓ, S_p, ex));
25     return S_p;
```

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 ℓ_p, where the critical predicate p resides.

In `InterSlicer` (lines 7-25), we compute a backward slice from the statement ℓ, 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].

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     if(a <= b){
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     ...;
```

(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 ℓ_4, which has a control dependence on ℓ_2. However, ℓ_2 only affects the execution of ℓ_4 but not the value of p. Since switching p can produce a correct execution, we mark ℓ_2 as less suspicious and do not include ℓ_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 ℓ, i.e., ℓ has no control dependence on p, but affects the values of the variables used in ℓ, ℓ has a potential dependence on p. As shown, the value of the mutated predicate p in ℓ_{13} is potentially affected by the outcome of the predicate in ℓ_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 ℓ_9 makes a failed test case pass, ℓ_{13} has an implicit dependence on ℓ_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 ℓ_8 or ℓ_9, it can be captured when we mutate ℓ_9 and infer from it, even if it is missed when we mutate ℓ_{13}. As explained earlier, we have exploited predicate switching [57] to improve SBFL by increasing the chances for the predicate in ℓ_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.

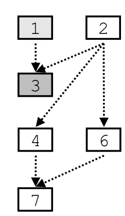
We use `UncriticalSlicer` to compute intraprocedurally a slice S_p starting from ℓ_p, where the uncritical predicate p resides. First, we compute ex, the set of statements executed by all the failed test cases, in lines 2-4. As in the case of critical predicates, we consider only data dependences in 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.

Algorithm 2: Inferring from uncritical predicates

Input: a statement ℓ_p
Output: a set of statements S_p
1 **Function** `UncriticalSlicer`(ℓ_p)
2 Let ex be an empty set;
3 **foreach** $t \in T_{\mathcal{F}}$ **do**
4 Add statements executed by t to ex;
5 Let mtd be the method containing ℓ_p;
6 Let cfg be the control flow graph of mtd;
7 Let S_p be an empty set;
8 Let $workList$ be an empty stack;
9 $workList$.push(ℓ_p);
10 **while** $workList$ *is not empty* **do**
11 $\ell = workList$.pop();
12 **if** $\ell \in S_p$ **then** continue;
13 S_p.add(ℓ);
14 **foreach** ℓ' *in* cfg **do**
15 **if** $\ell' \in ex$ *and* $\ell' \xrightarrow{data} \ell$ **then**
16 $workList$.push(ℓ');
17 **do**
18 Let rm be an empty set;
19 $others = ex - S_p$;
20 **foreach** $\ell \in S_p$ **do**
21 **foreach** ℓ' *in* cfg **do**
22 **if** $\ell' \in others$ *and* $\ell \xrightarrow{data} \ell'$ **then**
23 rm.add(ℓ); break;
24 S_p.removeAll(rm);
25 **while** $rm \neq \emptyset$;
26 **return** S_p;

```
1 a = x + y;
2 b = x - y;
3 if(a < b)
4     c = b;
5 else
6     c = -b;
7 return c;
```

(a) The example code



(b) The data dependence graph

Fig. 6. An example of mutation inference from uncritical predicates.

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 ℓ_3. After mutation testing, p is marked as uncritical. Then we conduct a backward slicing from p, collecting the statements that ℓ_3 depends on (ℓ_1 and ℓ_2). However, we keep ℓ_1 but remove ℓ_2:

- Since ℓ_1 affects only the value of p, ℓ_1 is more likely to be correct. Otherwise, switching p would have removed the effects of the faulty ℓ_1, causing p to be critical.
- The above inference does not apply to ℓ_2 because ℓ_2

affects not only p but also ℓ_4 and ℓ_6, and switching p cannot guarantee that the effects of ℓ_2 will be removed.

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 II. Inspired by [32, 34], 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 $w_c(p)$ ($w_u(p)$) 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:

$$\mathcal{N}^*_{\mathcal{F}} = \mathcal{N}_{\mathcal{F}} + \sum_{p \in P_c} w_c(p) \qquad \mathcal{N}^*_{\mathcal{F}}(\ell) = \mathcal{N}_{\mathcal{F}}(\ell) + \sum_{p \in P_c} w_c(p) \cdot \mathbf{1}_{S_p}(\ell)$$
$$\mathcal{N}^*_{\mathcal{P}} = \mathcal{N}_{\mathcal{P}} + \sum_{p \in P_u} w_u(p) \qquad \mathcal{N}^*_{\mathcal{P}}(\ell) = \mathcal{N}_{\mathcal{P}}(\ell) + \sum_{p \in P_u} w_u(p) \cdot \mathbf{1}_{S_p}(\ell)$$

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

We propose to calculate an adaptive weight w_c by:

$$w_c(p) = \frac{\mathcal{N}_{\mathcal{F}}}{|P_c|} \cdot \mathbf{e}^{\frac{\mathcal{N}_{\widetilde{\mathcal{F}}}(p)}{\mathcal{N}_{\mathcal{F}}}} \qquad (1)$$

where $\mathcal{N}_{\mathcal{F}}$ represents the number of failed test cases for the original program, $|P_c|$ is the number of its critical predicates found, $\mathcal{N}_{\widetilde{\mathcal{F}}}(p)$ is the number of test cases that have changed from failed to passed on the mutant m_p.

Let us understand the two components in Equation 1. Let us consider $\mathcal{N}_{\mathcal{F}}/|P_c|$ 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 $\mathcal{N}_{\mathcal{F}}$ except that it is scaled by a factor $|P_c|$. We have also included a second component $\mathbf{e}^{\mathcal{N}_{\widetilde{\mathcal{F}}}(p)/\mathcal{N}_{\mathcal{F}}}$ 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:

TABLE I
DEFECTS4J (VERSION 1.5.0). KLOC IS THE AVERAGE NUMBER OF LINES OF SOURCE CODE FOR EACH BUGGY PROGRAM.

Project	Description	KLOC	#Faults
Chart	Chart Library	85.1	26
Closure	Javascript Compiler	85.6	176
Lang	Java Utility	18.5	65
Math	Mathematics Library	50.6	106
Mockito	Mocking Framework	8.9	38
Time	Calendar System	27.3	27
Total		-	**438**

- **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 III).

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, 52, 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:

$$E_{first} = R_{start} + \sum_{k=1}^{\mathcal{T}-\mathcal{T}_{\mathcal{F}}} k \frac{\binom{\mathcal{T}-k-1}{\mathcal{T}_{\mathcal{F}}-1}}{\binom{\mathcal{T}}{\mathcal{T}_{\mathcal{F}}}} \qquad (2)$$

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 $\mathcal{T}$ statements with the same tied rank from the position R_{start} to the position $(R_{start} + \mathcal{T} - 1)$, with $\mathcal{T}_\mathcal{F}$ faulty statements among the $\mathcal{T}$ 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_{start}. $\binom{\mathcal{T}}{\mathcal{T}_\mathcal{F}}$ is the number of combinations of all $\mathcal{T}$ tied statements containing $\mathcal{T}_\mathcal{F}$ faulty statements and $\binom{\mathcal{T}-k-1}{\mathcal{T}_\mathcal{F}-1}$ is the number of combinations of the remaining tied statements after the first faulty statement with the remaining $\mathcal{T}_\mathcal{F} - 1$ faulty statements.

According to Equation 2, E_{first} is equal to R_{start} when all the t tied statements are faulty ($\mathcal{T}_\mathcal{F} = \mathcal{T}$) and will thus reduce to the average rank if $\mathcal{T}_\mathcal{F} = 1$ [7, 27]:

$$E_{first} = R_{start} + \frac{\mathcal{T} - 1}{2} \tag{3}$$

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_{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_{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_{first} will be unable to precisely measure the distribution of each faulty statement. We use $R_{average}$ to calculate the average rank of all faulty statements as follows:

$$R_{average} = \sum_{i=1}^{\mathcal{F}} \frac{R^i_{start} + R^i_{end}}{2\mathcal{F}} \tag{4}$$

where R^i_{start} and R^i_{end} 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 $\mathcal{F}$ is the total number of faulty statements in the program considered.

MAR: We adopt *MAR*, the mean value of all $R_{average}$ 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 original ranking list. After mutation inferences, we rerank the statements with 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 II
COMPARING FLIP WITH SBFL IN SEVEN STATE-OF-THE-ART RANKING FORMULAE ON DEFECTS4J. @N REPRESENTS THE NUMBER OF FAULTS FALLING INTO TOP-N ACCORDING TO E_{first} RANKING.

Technique	E_{first} @1	@3	@5	MEF	MAR
Ochiai	32	112	158	22.22	30.27
FLIP-Ochiai	**46**	**134**	**166**	**20.96**	**29.47**
Tarantula	28	110	156	19.90	27.80
FLIP-Tarantula	**34**	**133**	**167**	**18.67**	**26.90**
Jaccard	29	109	155	19.79	27.75
FLIP-Jaccard	**37**	**130**	**166**	**18.63**	**26.97**
Kulczynski2	28	108	155	22.37	30.89
FLIP-Kulczynski2	**41**	**128**	**161**	**21.20**	**30.06**
DStar2	31	107	149	22.58	30.72
FLIP-DStar2	**41**	**126**	**161**	**21.71**	**30.34**
Zoltar	28	108	153	22.18	29.86
FLIP-Zoltar	**41**	**128**	**159**	**21.23**	**28.89**
Goodman	31	109	157	19.54	27.41
FLIP-Goodman	**44**	**132**	**165**	**18.39**	**26.65**

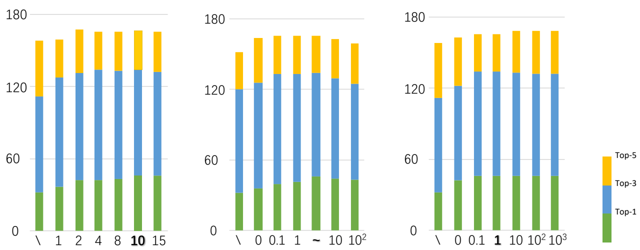
Overall, FLIP outperforms all the seven SBFL techniques in all the metrics. In terms of the *Top-N* metric, FLIP outperforms seven SBFL techniques by 21.4% – 46.4% for *Top-1*, 17.8% – 21.1% for *Top-3*, and 3.9% – 7.1% for *Top-5*. Once enhanced with mutation inferences, Ochiai, which is the best performer 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 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



(a) Impact of K (b) Impact of w_c (c) Impact of w_u

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. \ 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_c'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 $\{0, 10^{-1}, 1, 10^1, 10^2, 10^3\}$). 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 is 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.

TABLE III
ANALYZING THE COMPLEMENTARITY BETWEEN FLIP AND THE ORACLE MODEL COMBINING SBFL AND PREDICATE SWITCHING.

Technique	E_{first} @1	@3	@5	MEF	MAR
Ochiai	32	112	158	22.22	30.27
ORACLE (Ochiai+PS)	41	125	**171**	21.62	29.67
FLIP-Ochiai	**46**	**134**	166	**20.96**	**29.47**
ORACLE (SBFL)	37	118	166	17.03	25.04
ORACLE (SBFL+PS)	46	131	179	16.44	24.45
ORACLE (FLIP)	**52**	**148**	**185**	**15.59**	**24.05**

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 IV
ANALYZING FLIP'S OVERHEAD.

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

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, 36, 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

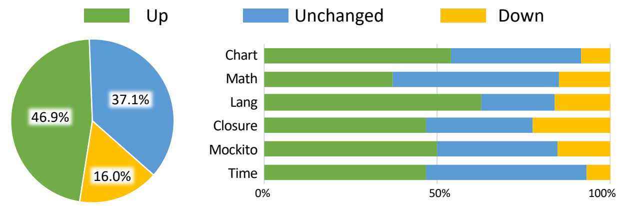
Although FLIP outperforms SBFL in all the metrics, we find that there are some faults that cannot benefit from our mutation inferences. We have manually analyzed most of them and will discuss below the main reasons and some possible solutions.

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 ℓ_{1134}. To measure the FL effectiveness, we follow [36] and mark the immediately following statement ℓ_{1134} as the faulty statement. However, FLIP marks ℓ_{1134} as an uncritical predicate and lowers its ranking by mutation testing and inferences. FLIP has indeed succeeded in obtaining the correct information about ℓ_{1134} but its statement-level abstraction is too coarse to model the missed code.



(a) All projects (b) Individual projects

Fig. 8. Analyzing the changes of E_{first} with FLIP.

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 public void appendTo(...){
       + TimeZone zone = calendar.getTimeZone();
1134     if(zone.useDaylightTime()
                 && calendar.get(...) != 0){
            ...}
1141 }
```

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

```
1368  public void testLang300(){
1369      NumberUtils.createNumber("-1l");
1370      NumberUtils.createNumber("01l");
1371      NumberUtils.createNumber("1l");
1372  }
```

Fig. 10. A failed test case for `Lang-58`.

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 ℓ_{1371}. In mutation testing, one mutant passes the sub-case in ℓ_{1371} but fails the one in ℓ_{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 [1], 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, 36, 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 [4, 13, 42, 44, 58] can assist software debugging by reducing the number of suspicious statements. We usually use the statements that trigger the assertion violations or crashes as the slicing criterion. Although sliced-based FL techniques can effectively reduce 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. [33] 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).

References

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE Computer Society, 2009.

[4] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPlan Notices*, volume 25, pages 246–256. ACM, 1990.

[5] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *1993 Conference on Software Maintenance*, pages 348–357. IEEE, 1993.

[6] F. Y. Assiri and J. M. Bieman. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal*, 25:171–199, 2017.

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

[8] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *ASE '12*, pages 378–381, 2012.

[9] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie. How do assertions impact coverage-based test-suite reduction? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 418–423. IEEE, 2017.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27:99–123, 2001.

[11] L. A. Goodman and W. H. Kruskal. Measures of association for cross classifications. In *Measures of association for cross classifications*, pages 2–34. Springer, 1979.

[12] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Software Engineering—ESEC/FSE'99*, pages 303–321. Springer, 1999.

[13] C. Hammacher. Design and implementation of an efficient dynamic slicer for Java. *Bachelor's Thesis, November*, 2008.

[14] S. Hong, J. Lee, J. Lee, and H. Oh. Saver: Scalable, precise, and safe memory-error repair. In *ICSE*, 2020.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:26–60, 1990.

[16] T. Janssen, R. Abreu, and A. J. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE Computer Society, 2009.

[17] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *ISSTA '18*, pages 298–309, 2018.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[19] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA '14*, pages 437–440, 2014.

[20] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276. ACM, 2014.

[21] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306. IEEE, 2015.

[22] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.

[23] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. Jfix: semantics-based repair of Java programs via symbolic pathfinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 376–379. ACM, 2017.

[24] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM, 2017.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38:54–72, 2011.

[26] H. J. Lee. *Software debugging using program spectra*. PhD thesis, The University of Melbourne, Australia, 2011.

[27] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. In *OOPSLA*, volume 1, page 92. ACM, 2017.

[28] J. Liu, D. Wu, and J. Xue. TDroid: Exposing app switching attacks in Android with control flow specialization. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 236–247. ACM, 2018.

[29] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE, 2019.

[30] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26:172–219, 2014.

[31] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[32] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[33] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[34] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25:605–628, 2015.

[35] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[36] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[37] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by

program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[38] S. Saha et al. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2019.

[39] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of SANER*, 2018.

[40] J. Sohn and S. Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283. ACM, 2017.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON '99*, pages 13–, 1999.

[42] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering*, pages 512–521. IEEE Computer Society, 2004.

[43] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351. ACM, 2005.

[44] T. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30:10, 2008.

[45] M. Weiser. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. *PhD thesis, University of Michigan*, 1979.

[46] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.

[47] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63:290–308, 2013.

[48] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42:707–740, 2016.

[49] W. E. Wong and Y. Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19:573–597, 2009.

[50] D. Wu, D. He, S. Chen, and J. Xue. Exposing Android event-based races by selective branch instrumentation. In *Proceedings of the 31st IEEE International Symposium on Software Reliability Engineering*, ISSRE '20, 2020.

[51] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[52] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE '17*, pages 416–426, 2017.

[53] X. Xu, Y. Sui, H. Yan, and J. Xue. VFix: Value-flow-guided precise program repair for null pointer dereferences. In *ICSE '19*, pages 512–523. IEEE Press, 2019.

[54] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *TSE*, 43:34–55, 2017.

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

[56] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(1):1–30, 2017.

[57] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM, 2006.

[58] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 319–329. IEEE, 2003.

[59] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards locating execution omission errors. In *PLDI'07*, volume 42, pages 415–424. ACM, 2007.

[60] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 214–224. ACM, 2015.

[61] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 2019.