Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs

YUNHO KIM, SEOKHYEON MUN, SHIN YOO, and MOONZOO KIM, School of Computing, KAIST, South Korea

Finding the root cause of a bug requires a significant effort from developers. Automated fault localization techniques seek to reduce this cost by computing the suspiciousness scores (i.e., the likelihood of program entities being faulty). Existing techniques have been developed by utilizing input features of specific types for the computation of suspiciousness scores, such as program spectrum or mutation analysis results. This article presents a novel learn-to-rank fault localization technique called *PRecise machINe-learning-based fault loCalization tEchnique (PRINCE)*. PRINCE uses genetic programming (GP) to combine multiple sets of localization input features that have been studied separately until now. For dynamic features, PRINCE encompasses both Spectrum Based Fault Localization (SBFL) and Mutation Based Fault Localization (MBFL) techniques. It also uses static features, such as dependency information and structural complexity of program entities. All such information is used by GP to train a ranking model for fault localization. The empirical evaluation on 65 real-world faults from CoREBench, 84 artificial faults from SIR, and 310 real-world faults from Defects4J shows that PRINCE outperforms the state-of-the-art SBFL, MBFL, and learn-to-rank techniques significantly. PRINCE localizes a fault after reviewing 2.4% of the executed statements on average (4.2 and 3.0 times more precise than the best of the compared SBFL and MBFL techniques, respectively). Also, PRINCE ranks 52.9% of the target faults within the top ten suspicious statements.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: Fault localization, machine learning, mutation analysis, source file characteristics

ACM Reference format:
Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 23 (October 2019), 34 pages.
https://doi.org/10.1145/3345628

This work is supported by Next-Generation Information Computing Development Program through National Research Foundation (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2017M3C4A7068177 and NRF-2017M3C4A7068179), Basic Science Research Program through NRF funded by MSIT (NRF-2019R1A2B5B01069865), Basic Science Research Program through NRF funded by the Ministry of Education (MOE) (NRF-2017R1D1A1B03035851), and Engineering Research Center Program through NRF funded by MSIT (NRF-2018R1A5A1059921).
Authors' address: Y. Kim, S. Mun, S. Yoo, and M. Kim (corresponding author), School of Computing, KAIST, 291 Daehak-ro, Daejeon 34141, South Korea; emails: {yunho.kim03, seokhyeon.mun}@gmail.com, shin.yoo@kaist.ac.kr, moonzoo@cs.kaist.ac.kr.

1049-331X/2019/10-ART23 $15.00
https://doi.org/10.1145/3345628

1 INTRODUCTION

Software developers spend a large amount of time debugging software failures. The first step in debugging is to identify the location of the root cause of the software failure, called fault localization (FL). FL is an expensive task [71, 84], as it usually involves human effort to understand the complex internal logic of the Program Under Test (PUT), as well as to reason about the differences between passing and failing test runs. Consequently, automated FL has received much attention [74]. To reduce human effort in localizing a fault, various automated FL techniques have been proposed. The automated FL evaluates suspiciousness of a program element (i.e., a statement or a function) that indicates a likelihood of having a fault using a specific suspiciousness evaluation formula.

So far, most of the FL techniques utilize a limited set of program features. For example, Spectrum Based Fault Localization (SBFL) techniques use only the number of passing and failing test cases that cover a given element. As another example, Mutation Based Fault Localization (MBFL) techniques utilize the number of killed mutants generated by mutating a given element. In addition, the traditional automated FL techniques use a fixed form of a suspiciousness evaluation formula and lose the opportunity to improve FL precision through learning from a given target program.

We propose a novel learn-to-rank FL technique utilizing various program features called *PRecise machINe learning-based fault loCalization tEchnique (PRINCE)*. PRINCE utilizes various program features such as suspiciousness formulas of SBFL and MBFL and the dependency and complexity of a file, a function, and a statement. Some of these are utilized by previous learn-to-rank FL techniques [5] (e.g., SBFL features) but the other features are newly adopted to improve precision of PRINCE (e.g., MBFL and file features). PRINCE adopts these new features for the following reasons. First, recent MBFL techniques achieve high FL precision [25, 26, 49, 54]. Second, the complexity and the dependency of a file can be good indicators of a fault [61, 62]. This is because a complex file implements a complex module, which is likely to have a fault. Also, a source code file that many other files depend on (i.e., a large degree of fan-in) is likely to be complicated, because it serves many different users and has a high chance of containing faults. Based on these features, PRINCE applies genetic programming as a learning algorithm to learn a ranking model.

We empirically evaluate PRINCE using 65 real-world faults from CoREBench [6], 84 artificial faults from the SIR benchmark [16], and 310 real-world faults from Defects4J [33]. We compare the precision of PRINCE with the existing SBFL, MBFL, and learn-to-rank techniques in terms of the expense metric, mean average precision (MAP), and acc@n metrics. Furthermore, we evaluate the importance of the each group of features by comparing how much the precision of PRINCE is affected when each group of the features is excluded from the entire feature set [15]. The empirical study results show that PRINCE outperforms SBFL, MBFL, and learn-to-rank techniques in terms of precision. In addition, the experimental results show that newly adopted MBFL and file features significantly improve the precision of FL. For CoREBench and SIR, a group of the MBFL features improves the FL precision most significantly. The file feature group, which this article newly proposes, is the third-most significant feature group to increase the precision for CoREBench and Defects4J.

The contributions of this article are as follows:

(1) We have demonstrated that PRINCE can localize a fault *precisely* (e.g., 2.4% in the expense metric and 0.49 in MAP on average over the target faults and 52.9% in acc@10) using a machine-learning technique on various dynamic and static program features such as testing result changes through mutation and source file characteristics (e.g., a size of a file, a fan-in and a fan-out degree of a file in a file dependency graph).
(2) We have proposed to use a set of representative static and dynamic features on a target program, which improves FL precision to a large degree. We have studied 55 features in

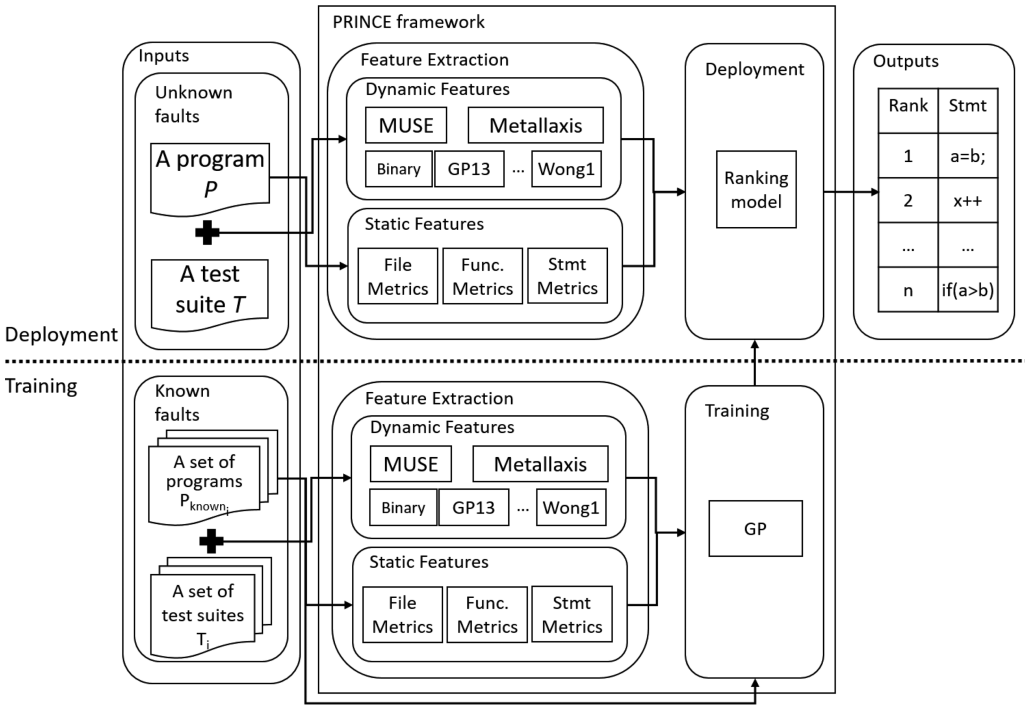

Fig. 1. Overall process of PRINCE.

five groups including entirely new ones for FL such as fan-in and fan-out of a file dependency graph.

(3) We have demonstrated our study results through extensive empirical study on real-world C program faults in CoREBench and artificial faults in SIR as well as real-world Java program faults in Defects4J.

The rest of the article is organized as follows. Section 2 overviews the PRINCE framework. Section 3 explains the dynamic and static features on a target program utilized by PRINCE. Section 4 describes the experimental setting. Section 5 demonstrates and analyzes the experimental results. Section 6 discusses observations through the experiment. Section 7 discusses related work. Finally, Section 8 summarizes this article with future work.

2 PRINCE FRAMEWORK

Figure 1 shows the overall process of the PRINCE framework. The process mainly consists of the two phases: *training phase* (see the bottom half of Figure 1) and *deployment phase* (see the top half of Figure 1). In the training phase, PRINCE learns a ranking model from the ground truth (i.e., knowledge on the locations of faults). Then, in the deployment phase, PRINCE utilizes the ranking model to generate a ranked list of target statements in which a statement in high rank is likely a fault. The inputs and output of PRINCE are as follows:

- Inputs:
  - —A target program P that has a fault but whose location is unknown
  - —A test suite T for P where T has at least one failing test case
  - —A set of programs P_{known_i}, each of which has a fault whose location is known
  - —A set of test suites T_i, each of which has at least one failing test case for P_{known_i}

- Output:
  PRINCE produces a ranked list of the target statements of P. The higher rank (i.e., the first rank) a statement has, the more likely the statement is a faulty one.

2.1 Training Phase

In the training phase, PRINCE takes as inputs a set of programs P_{known_i} each of which has a fault whose location is known and a set of test suites T_i, each of which has at least one failing test case for P_{known_i}.

First, PRINCE extracts both *dynamic* features, which depend on the dynamic behaviors of the target programs and *static* features. For example, PRINCE extracts dynamic features such as SBFL features (e.g., the number of passing and failing test cases execute a given statement s) and MBFL features (e.g., the number of failing tests that become passing tests on mutants generated by mutating s). The static features include dependency and complexity information of files, functions, and statements. PRINCE extracts such static features by using lightweight static analyses. The details of the features PRINCE uses and the method to extract such features are explained in Section 3. Note that PRINCE is the first learn-to-rank FL framework that utilizes a new diverse set of features including MBFL features and dependency of file features.

Then, PRINCE learns a ranking model from the extracted features. To learn a ranking model using a learning algorithm such as genetic programming or rankSVM, PRINCE needs a label of a statement s, which shows if s is faulty or not and its feature values. We assume that all the faults in P_{known_i} are already known and fixed so that we can assign a label to each statement precisely (i.e., statements that are modified to fix a fault are faulty).

To generate a ranking model, PRINCE uses *Genetic Programming (GP)*. GP evolves a ranking function that takes features and produces scores. We use a tree-based GP and a single-point crossover with a rate of 1.0. We use 0.1 as a mutation rate and 40 as the size of population. The maximum tree depth is nine and the number of generations of GP is set as 100. The fitness function is the average ranking of the faulty statements calculated from all faulty programs in the training set. We use various dynamic and static code features for FL (see Section 3), and a constant 1.0 as variables for the evolving ranking function.[1]

2.2 Deployment Phase

In the deployment phase, PRINCE takes as inputs a program P, which has a fault but whose location is unknown and a test suite T, which has at least one failing test case for P.

First, as PRINCE did in the training phase, PRINCE extracts both dynamic and static features. The feature extraction process in the deployment phase is same as the one in the training phase. Then, PRINCE applies the ranking model learned in the training phase to the extracted features and generates a ranked list of statements as an output. The statement with the higher rank is more likely to be faulty.

3 DYNAMIC AND STATIC CODE FEATURES FOR FAULT LOCALIZATION

This section describes the features used by PRINCE. PRINCE uses features classified into five feature groups: MBFL, SBFL, File, Function, and Statement. MBFL and SBFL feature groups represent dynamic characteristics of a faulty program using test execution results on the faulty program and mutants of the faulty program. File, function, and statement feature groups represent static characteristics of a faulty program. Table 1 shows the full list of the features PRINCE uses. For

[1]We used the same parameter setting used in the experiment in Sohn and Yoo [68], since the experiment with these parameter setting produces precise FL results.

Table 1. List of All Features PRINCE Uses

Type	Feature group	List of features
Dynamic	MBFL	• Metllaxis: $\max_{m \in mut_{killed}(s)}(kill(m))$, $\max_{m \in mut_{killed}(s)}(\frac{1}{\sqrt{kill(m)}})$, $\max_{m \in mut_{killed}(s)}(\frac{1}{\sqrt{kill(m)+notkill(m)}})$, $\max_{m \in mut_{killed}(s)}(\frac{kill(m)}{\sqrt{(kill(m))(kill(m)+notkill(m))}})$ where - $mut_{killed}(s)$ is a set of killed mutants generated at statement s - $kill(m)$ represents the number of test cases that kill m - $notkill(m)$ represents the number of test cases that do not kill m • MUSE: $\frac{1}{(\lvert mut(s)\rvert+1)}$, $\Sigma_{m \in mut(s)} \lvert p_P(s) \cap f_m \rvert$, $\Sigma_{m \in mut(s)} \lvert f_P(s) \cap p_m \rvert$ $\frac{1}{(\lvert mut(s)\rvert+1)(f2p+1)} \times \sum_{m \in mut(s)}(\lvert f_P(s) \cap p_m \rvert)$, $\frac{1}{(\lvert mut(s)\rvert+1)(p2f+1)} \times \sum_{m \in mut(s)}(\lvert p_P(s) \cap f_m \rvert)$, $(\frac{1}{(\lvert mut(s)\rvert+1)(f2p+1)} \times \sum_{m \in mut(s)}(\lvert f_P(s) \cap p_m \rvert) -$ $\frac{1}{(\lvert mut(s)\rvert+1)(p2f+1)} \times \sum_{m \in mut(s)}(\lvert p_P(s) \cap f_m \rvert))$ where - $mut(s)$ is # of mutants generated on s - $f_P(s)$ (or $p_P(s)$) is the set of tests that cover s and fail (or pass) on a target program P - f_m (or p_m) is the set of tests that fail (or pass) on a mutant m. - $f2p$ (or $p2f$) is the number of test result changes from fail to pass(or pass to fail) for all mutants of P
	SBFL	• Basic terms: $e_p(s)$, $e_f(s)$, $n_p(s)$, $n_f(s)$ - $e_p(s)$ (or $e_f(s)$) is the the number of passing (or failing) tests that execute s - $n_p(s)$ (or $n_f(s)$) is the the number of passing (or failing) tests that do not execute s • Binary: $0\ if 0 < n_f(s)$, $1\ if 0 = n_f(s)$ • GP13: $e_f(s)$, $\frac{1}{2e_p(s)+e_f(s)}$, $\frac{e_f(s)}{2e_p(s)+e_f(s)}$, $e_f(s) + \frac{e_f(s)}{2e_p(s)+e_f(s)}$ • Jaccard: $e_f(s)$, $\frac{1}{e_f(s)+n_f(s)+e_p(s)}$, $\frac{e_f(s)}{e_f(s)+n_f(s)+e_p(s)}$ • Naish1: $n_p(s)$, $-1\ if 0 < n_f(s)$, $n_p(s)\ if 0 = n_f(s)$ • Naish2: $e_f(s)$, $e_p(s)$, $\frac{1}{e_p(s)+n_p(s)+1}$, $\frac{e_p(s)}{e_p(s)+n_p(s)+1}$, $e_f(s) - \frac{e_p(s)}{e_p(s)+n_p(s)+1}$ • Ochiai: $e_f(s)$, $\frac{1}{\sqrt{e_f(s)+n_f(s)}}$, $\frac{1}{\sqrt{e_f(s)+e_p(s)}}$, $\frac{e_f(s)}{\sqrt{(e_f(s)+n_f(s))(e_f(s)+e_p(s))}}$ • Russell and Rao: $e_f(s)$, $\frac{1}{e_p(s)+n_p(s)+e_f(s)+n_f(s)}$, $\frac{e_f(s)}{e_p(s)+n_p(s)+e_f(s)+n_f(s)}$ • Wong1: $e_f(s)$
Static	File	• Fan-in and Fan-out of a file dependency graph • # of defined file scope functions, # of defined file scope variables • # of defined global functions, # of defined global variables • # of defined functions, # of defined variables • Compile time(s), compiler memory usage(KB), # of compiler warnings, LOC
	Function	• Fan-in and Fan-out of a static function call graph • LOC, Cyclomatic complexity, # of parameters • # of global variables a function reads, # of global variables a function writes • # of local variables a function reads, # of local variables a function writes
	Statement	• Length of statements (bytes) • # of operators that a statement uses • # of variables that a statement uses

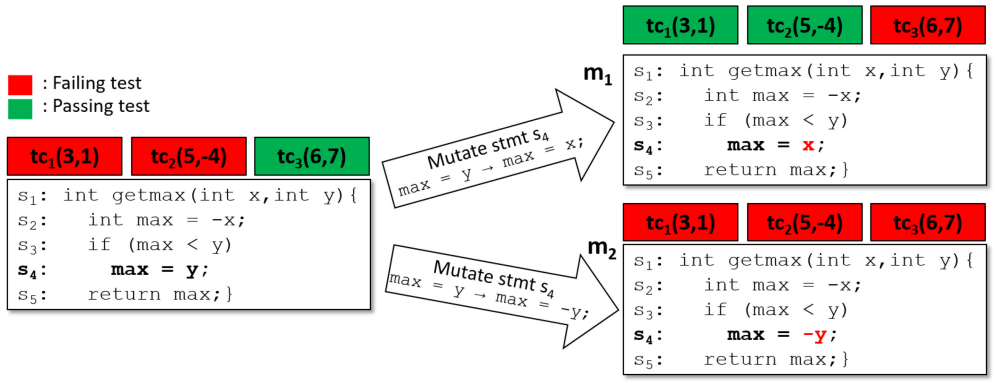

Fig. 2. An example of extracting the MUSE features.

data-driven learning techniques such as GP, selecting features is important for effectiveness and efficiency of the learning techniques [22, 43].

3.1 Dynamic Features

As dynamic features for FL, we choose the suspiciousness formulas and their sub-formulas of the widely used FL techniques. We use not only the whole formula but also their sub-formulas, because PRINCE can learn efficient ranking models by composing sub-formulas in a flexible way.

3.1.1 MBFL Features. MBFL features represent the relation between the mutant execution results of the faulty program and a faulty statement. As MBFL features, we use sub-formulas of the suspiciousness formulas of the two mutation-based FL techniques: Metallaxis [54] and MUSE [49]. We decompose a formula into sub-formulas as follows recursively until the sub-formulas cannot be decomposed further:

(1) If a formula f is composed using $\times$ such as $f_1 \times \cdots \times f_n$ where each of f_i is not composed using $\times$, then we use f and each of f_i which is parameterized by s as features.
(2) If a formula f is composed using $+$ or $-$ such as $f_1 + \cdots - f_n$, where each of f_i is not composed using $+$ nor $-$, then we use f and each of f_i which is parameterized by s as features.

For example, MUSE's suspiciousness formula, $\frac{1}{(|mut(s)|+1)(f2p+1)} \times \sum_{m \in mut(s)}(|f_P(s) \cap p_m|) - \frac{1}{(|mut(s)|+1)(p2f+1)} \times \sum_{m \in mut(s)}(|p_P(s) \cap f_m|)$ is composed of two sub-formulas $\frac{1}{(|mut(s)|+1)(f2p+1)} \times \sum_{m \in mut(s)}(|f_P(s) \cap p_m|)$ and $\frac{1}{(|mut(s)|+1)(p2f+1)} \times \sum_{m \in mut(s)}(|p_P(s) \cap f_m|)$(see Table 1 for details of the terms used in the formulas). Then, MUSE's suspiciousness formula and its two sub-formulas are used as features. Next, we apply the above rule to each of the sub-formulas. The first sub-formula, $\frac{1}{(|mut(s)|+1)(f2p+1)} \times \sum_{m \in mut(s)}(|f_P(s) \cap p_m|)$ is composed of the three sub-formulas $\frac{1}{(|mut(s)|+1)}$, $\frac{1}{(f2p+1)}$, and $\sum_{m \in mut(s)}(|f_P(s) \cap p_m|)$. These sub-formulas are also included in the MBFL feature group except $\frac{1}{(f2p+1)}$, which is not parameterized by s (i.e., all statements in a program have the same value).

Figure 2 shows an example of how to extract the MUSE features. The target function `getmax` should return a bigger value of the two integer parameters `x` and `y`. Suppose that we have three test cases tc_1, tc_2, and tc_3 (where tc_1 and tc_2 fail and tc_3 passes) and generate two mutants m_1 and m_2 by mutating s_4 (`max = y`) to `max = x` and `max = -y`, respectively. m_1 changes the test results of

	`void getmax2(int x, int y){`	Coverage of Test Cases (x, y)					$e_p(s)$	$e_f(s)$	$n_p(s)$	$n_f(s)$	Naish2 Susp.
		tc_1 (3,1)	tc_2 (5,-4)	tc_3 (0,-4)	tc_4 (0,7)	tc_5 (-1,3)					
1:	`int max = -x; //should be max = x;`	•	•	•	•	•	3	2	0	0	1.25
2:	`if(max < y){`	•	•	•	•	•	3	2	0	0	1.25
3:	`max = y;`	•	•		•	•	2	2	1	0	1.50
4:	`if(x*y<0){`	•	•		•	•	2	2	1	0	1.50
5:	`print("diff.sign");}`		•			•	1	1	2	1	0.75
6:	`return max;}`	•	•	•	•	•	3	2	0	0	1.25
	Test Results	Fail	Fail	Pass	Pass	Pass					

Fig. 3. An example of extracting the SBFL features.

tc_1 and tc_2 from failing to passing and the test result of tc_3 from passing to failing. m_2 changes the test result of tc_3 from passing to failing. So both of $f2p$ and $p2f$ are two. For s_4, we can compute the feature values of MUSE sub-formulas as follows:

- $|mut(s_4)|$ is two (m_1 and m_2),
- $\sum_{m \in mut(s_4)}(|f_P(s_4) \cap p_m|)$ is two, because
  $|f_P(s_4) \cap p_{m_1}|)$ is two (the two failing tests tc_1 and tc_2 becomes passing on m_1) and
  $|f_P(s_4) \cap p_{m_2}|)$ is zero (no failing test becomes passing on m_2)
- $\sum_{m \in mut(s_4)}(|p_P(s_4) \cap f_m|)$ is two, because
  $|p_P(s_4) \cap f_{m_1}|)$ is one (the passing test tc_3 becomes failing on m_1) and
  $|p_P(s_4) \cap f_{m_2}|)$ is one (the passing test tc_3 becomes failing on m_2).

Using this computation, PRINCE can extract MBFL feature values of MUSE.

Metallaxis defines suspiciousness of a statement s as the maximum suspiciousness value of the killed mutants $m \in mut_{killed}(s)$ generated by mutating a statement s. Suspiciousness value of the killed mutant, $susp(m)$, is defined as $\frac{kill(m)}{\sqrt{(kill(m))(kill(m)+notkill(m))}}$ where $kill(m)$ and $notkill(m)$ are the number of test cases that do and do not kill m, respectively, and suspiciousness value of the statement s is defined as $\max_{m \in mut_{killed}(s)}(susp(m))$. To compute feature values of Metallaxis on a target statement s, first we decompose the suspiciousness formula of Metallaxis on the killed mutants m into sub-formulas $f_i(m)$s. Then, for each decomposed sub-formula $f_i(m)$, PRINCE defines the feature value formulas on s as $\max_{m \in mut_{killed}(s)}(f_i(m))$.

3.1.2 SBFL Features. SBFL features capture a relation between the test execution results and a faulty statement. SBFL suspiciousness formulas use the four atomic terms $e_p(s)$, $e_f(s)$, $n_p(s)$, and $n_f(s)$, which are the number of passing tests that execute s, the number of failing tests that execute s, the number of passing tests that do not execute s, and the number of failing tests that do not execute s, respectively. Figure 3 shows an example of how to extract the SBFL features. The target function `getmax2` should return a bigger value of the parameters `x` and `y`. Suppose that we have five test cases tc_1 to tc_5 (where tc_1 and tc_2 fail and tc_3 to tc_5 pass). PRINCE computes $e_p(s)$, $e_f(s)$, $n_p(s)$, and $n_f(s)$ for each statement s using the coverage information of each test case. SBFL techniques compose the suspicious metric using these four atomic terms (e.g., Naish2's suspicious metric is $e_f(s) - \frac{e_p(s)}{e_p(s)+n_p(s)+1}$).

For SBFL features, PRINCE uses the suspiciousness formulas of the eight SBFL techniques. Of these formulas, six formulas (Binary, GP13, Naish1, Naish2, Russell and Rao, and Wong1) are proven to be maximal in theory [77, 83]. Note that these SBFL techniques form the two biggest maximal groups of SBFL formulas, ER1 and ER5 [83]. These SBFL features include or dominate the SBFL formulas discussed in Perez et al. [58] and Pearson et al. [57]. Also, Perez et al. [58]

empirically showed that Naish2 is optimal to localize a single fault. In addition, we add Jaccard [27] and Ochiai [51] that are widely studied in literature.[2] Similarly to the MBFL features (see Section 3.1.1), we use the sub-formulas of the eight SBFL suspiciousness formulas as SBFL features.

3.2 Static Features

From a program P, PRINCE extracts various static features on files, functions, and statements. PRINCE also uses various source code metrics to measure code *dependency* and *complexity* (e.g., fan-in and fan-out degrees of code dependency [10, 14], source code size [4], Halstead complexity metrics [24], McCabe complexity metrics [48], and Object-Oriented (OO) complexity metrics [10, 17]).[3] These two characteristics can be good indicators for faulty code segments, because a code segment that is highly complex or associated with many other code segments can be a fault with high probability [61]. The static features used by PRINCE are mostly programming language agnostic.

3.2.1 File Features. File features represent the *dependency* and the *complexity* of a source code file, which can be interpreted as indicators of a faulty file.

Dependency: The dependency between files is measured by *fan-in* and *fan-out* of a target file in a file dependency graph. Using fan-in and fan-out of a source code file is inspired by Chidamber and Kremerer's work [10] and D'Ambros et al.'s work [14].

A file dependency graph $G = (N, E)$ consists of a set of nodes N representing files and a set of edges E. An edge from a file A to a file B represents that a function defined in A accesses a global variable defined in B or invokes a function defined in B.[4] *Fan-in* of a file A is defined as a number of edges from other files to A. *Fan-out* of a file A is defined as a number of edges from A to other files.

Complexity: PRINC measures the complexity of a file using the number of defined functions and variables in file scope (i.e., `static` type quantifier in C and `private` or `protected` access modifier in Java) and global scope. These metrics are inspired by D'Ambros et al.'s work [14], which uses the number of `private`/`public` attributes and methods as complexity measure of a class. Also, PRINCE computes a weight for each defined variable according to its type as not every variable contributes to the complexity of a file equally. We assign more weight to array, `struct` and object variables than primitive type variables, because a `struct` or object variable is a collection of multiple primitive type variables (and objects); the complexity of handling one `struct` or object variable increases proportionally to the number of the primitive variables included in the `struct` or object variable. Similarly, the complexity of handling an array variable is proportional to the size of the array. A weight of a variable is computed as follows:

- primitive type variables (in C and Java): PRINCE assigns a weight value of one.
- pointer variables pointing to a primitive type variable (in C): PRINCE assigns a weight value of one.
- array type variables: PRINCE assigns a weight as multiplication of the length of the array and the weight of its element variable.[5]

[2]We do not include DStar [57, 73], because it can be theoretically shown that Naish1 dominates DStar, i.e., Naish1 can always rank an arbitrary faulty statement at equal or higher ranks when compared to DStar under the single fault scenarios.
[3]The static code features used by PRINCE can cover most metric-based bad code smells [47, 66] (e.g., god class, long parameter list, long method).
[4]We do not consider variable references and function calls through a pointer, because existing pointer analysis may generate spurious point-to relations, which can decrease the precision of a file dependency graph.
[5]If the size of an array is not known in static analysis (e.g., an array passed as a function parameter via a pointer), then we conservatively consider the size of the array as one, because precisely identifying the array size using static analysis is technically difficult.

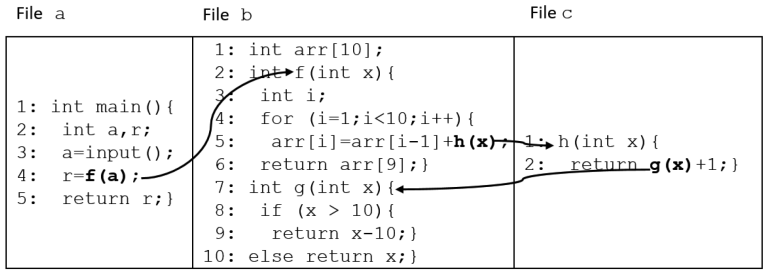

Fig. 4. An example of extracting static features.

- `struct` type variables and pointers (in C): PRINCE assigns the sum of weights of all fields of the `struct` type variable recursively. A weight of a `struct` type pointer variable is equal to the weight of the corresponding pointee variable. When PRINCE calculates a weight of the same variable type again through recursive weight calculation, it assigns a weight value one to prevent infinite recursion (i.e., base cases).
- object variables (in Java): PRINCE assigns the sum of weights of all fields of the object variable recursively. When PRINCE calculates a weight of the same variable type again through recursive weight calculation, it assigns a weight value of one to prevent infinite recursion (i.e., base cases). If the definition of class of the object variable is not known (e.g., `Object` class in Java), then PRINCE assigns a weight value of 1.

Figure 4 shows an example of measuring the dependency and complexity features of files. The example has three source code files: *a*, *b*, and *c*. The file *a* has a `main` function, which invokes `f` defined in the file *b*. The file *b* has `f`, which invokes `h` in the file *c*, which invokes a function `g` in the file *b*. Fan-in of the file *b* is two, because `main` in the file *a* calls `f`, and `h` in the file *c* calls `g`, but they do not access variables defined in the file *b*. The weight of `arr` defined in the file *b* is 10 because of `arr`, which has 10 integer elements.

Also, PRINCE measures compile time, compiler memory usage, the number of compiler warnings generated, and LOC for a target file as complexity metrics.

3.2.2 Function Features. Function features represent the dependency and the complexity of a function similarly to the file features (Section 3.2.1).

Dependency: Dependency of a function f is measured by fan-in and fan-out of f in the static call graph. A static call graph $G = (N, E)$ consists of a set of nodes N representing functions and a set of edges E. An edge from a function f to another function g represents that f invokes g. *Fan-in* of a function f is defined as a number of edges from other functions to f. *Fan-out* of a function f is defined as a number of edges from f to other functions. Using fan-in and fan-out of a function is inspired by Chidamber and Kremerer's work [10] and D'Ambros et al.'s work [14].

Complexity: Complexity of f is measured by LOC [4], cyclomatic complexity [48], the number of parameters [14], the number of global and local variables accessed by f [14] with variable weights as defined in Section 3.2.1.

3.2.3 Statement Features. PRINCE uses three statement features to represent the complexity of statement *s* (inspired by the Halstead complexity measures [23], which use the number of operators and operands in program code):[6]

[6]We did not use a dependency feature for statement, because the dependency feature may not be effective to precisely localize faults. This is because, unlike the file and function dependency, which have a wide range of fan-in and fan-out, the

(1) the number of operators in s: We use Clang to count the number of operators including the arithmetic operators, logical operators, relational operators, bit-wise operators, pointer operators, array element access operator (i.e., `[]`), assignment operator, compound assignment operators, `sizeof` operator, and the function calls.

    For example, in Figure 4, the number of operators of the statement in Line 5 of file b is six: two array element access operators (one at the left-hand side and the other at the right-hand side of the assignment operator), one assignment operator, one minus operator in the array subscription, one plus operator, and one function call.
(2) the number of variables accessed in s with their weights (as defined in Section 3.2.1). If one variable is accessed multiple times in s, then we count the weight of the variable only once.
(3) the length of s in text bytes: When we measure a length of a statement s, we remove all whitespace characters (e.g., spaces, tabs, and newlines) in s.

4 EMPIRICAL STUDY SETUP

This section describes the research questions to evaluate the precision and efficiency of PRINCE. Also, we explain target programs and the experiment setting in detail. Finally, we describe threats-to-validity of our experiment.

4.1 Research Questions

RQ1. Precision of PRINCE: *How precise is PRINCE in localizing a target fault in terms of expense metric, mean average precision, and acc@n compared to SBFL, MBFL, and machine-learning-based FL techniques?*

RQ1 is about evaluating the precision of PRINCE, which uses the proposed features (see Section 3) in terms of the expense, mean average precision (MAP), and acc@n metrics. Expense metric [63] measures % of the executed code elements (i.e., statements or functions) to be examined to localize a faulty element, should a human developer inspect a target program following the order of the elements in the ranking generated by a FL technique. MAP [46] is a metric that evaluates the effectiveness of a ranking technique. MAP is widely used in information retrieval research field. acc@n metric counts the number of faults localized within top n elements of the ranking. We use 1, 5, and 10 for the value of n.

We compare PRINCE with eight spectrum-based fault localization techniques (Naish1 [50], Naish2 [50], GP13 [82], Wong1 [75], Russell and Rao [64], Binary [50], Ochiai [51], and Jaccard [27]), two mutation-based FL techniques (MUSE [49] and Metallaxis [54]), and a machine-learning-based FL technique Savant [5] which is a state-of-the-art learn-to-rank FL technique. We compare PRINCE with Savant on Defects4J in terms of the expense metric, MAP, acc@1, acc@3, and acc@5 in a function level as reported in the Savant paper [5]. We used the function-level aggregation technique (as used in Sohn and Yoo [68]), which assigns a function m with the highest suspiciousness score of m's statements.

RQ2. Efficiency of PRINCE: *How much runtime cost does PRINCE incur, compared to SBFL and MBFL techniques?*[7]

statement dependency has only limited range of fan-in and fan-out (i.e., all statements (except entry or exit statements of basic blocks) have the same fan-in and fan-out value of 1).

[7]We do not compare the efficiency of PRINCE with Savant, because the reported execution time in the Savant paper [5] was measured on the machine setting different from our machine setting.

RQ2 is about evaluating the efficiency of PRINCE, compared to SBFL and MBFL. We measure feature value computation time, training time of genetic programming, and suspiciousness score computation time to compare the efficiency of these techniques.

RQ3. Importance of Program Feature Groups for FL precision: *How much does each of the program feature groups contribute to precision of FL?*

RQ3 is about evaluating the contribution of each of the five feature groups (see Section 3) to the precision of FL in terms of the expense metric. We compare PRINCE using all features with PRINCE using all features except the feature group α where $\alpha \in$ {MBFL, SBFL, File, Function, Statement} to evaluate the contribution of the feature group α to the precision of FL. This method is called one-factor-at-a-time (OFAT) method [15], which is widely used to analyze the impact of a factor.

RQ4. Importance of Program Feature Groups for FL precision with Different Machine Learning Techniques—Genetic Programming, Linear RankSVM, and Non-linear RankSVM: *How much does each of the program feature groups contribute to precision of FL with different machine-learning techniques?*

RQ4 is about investigating importance of the program feature groups with different machine-learning techniques. We use three different machine-learning techniques (genetic programming, linear rankSVM, and non-linear rankSVM) to evaluate the importance of the program feature groups. *Linear RankSVM* is a variation of Support Vector Machine [12] algorithm that performs pairwise learning to rank [42]. Through training, linear rankSVM learns the linear weights to input features and the weighted sum of the input features produces suspiciousness scores. *Non-linear RankSVM* is another variation of SVM algorithm. As the linear rankSVM learns the linear weights to input features, non-linear rankSVM learns non-linear weights to input features.

To answer the research questions, we performed a series of experiments by applying spectrum-based FL techniques, mutation-based FL techniques, and PRINCE to the 65 real-world C program faults in CoREBench [6], 84 injected C program faults in the SIR benchmark [16], and 310 real-world Java program faults in Defects4J [33] that were targeted by Pearson et al. [57]. Also, Savant, which is a learn-to-rank FL technique, is applied to the 282 faults in Defects4J that belong to both 357 bugs used in the Savant paper [5] and the 310 bugs targeted by Pearson et al. [57].[8] We compare PRINCE with Savant on the 282 faults.

4.2 Subject Programs

We used the following three benchmark suites as subject programs—CoREBench, SIR, and Defects4J.

- CoREBench [6] is a collection of 70 real-world bugs in the four non-trivial real-world programs: `coreutils`, `find`, `grep`, and `make`. CoREBench provides a docker [1] container that contains 70 buggy versions of the four programs and a failing test case for each buggy version (CoREBench itself does *not* contain any passing test cases, which a user has to download from the target program repository separately). We used the failing test cases provided in CoREBench and all passing test cases provided in the regression test suites in the public repository of the four programs.
- The SIR benchmark [16] is a collection of artificially injected faults of real-world open-source programs and smaller Siemens programs. We excluded Siemens programs, because they are too small (less than 500 LoC) to represent real-world programs. We also excluded `space`, because the faults of `space` are real-world faults, which have different characteristic

[8] Since the experiment in the Savant paper [5] did not target Mockito, we also excluded Mockito in the experiment.

Table 2. Subject Programs, # of Buggy Versions Used, a Number of Their Source Code Files, a Number of Their Functions, Their Average Sizes in Lines of Code (kLOC), the Average Number of Failing and Passing Test Cases, and Brief Description

Bench-mark	Subject program	# of ver. used	# of source code files	# of funcs	Size (kLOC)	$\lvert f_P \rvert$	$\lvert p_P \rvert$	Description
Core-bench	`Coreutils`	19	27.3	49.5	83.1	1.1	485.1	Command line utilities
	`find`	14	31.0	213.2	18.0	1.1	51.3	File searcher
	`grep`	14	26.0	149.3	9.4	1.0	87.6	Pattern matcher
	`make`	18	27.0	174.6	35.3	1.2	471.1	Source builder
SIR	`bash`	6	35.0	1214.0	32.7	167.3	832.7	Shell interpreter
	`flex`	19	1.0	147.0	7.4	137.1	429.9	Parser generator
	`grep`	18	1.0	132.0	6.0	97.6	711.4	Text matcher
	`gzip`	16	1.0	82.0	3.0	34.3	179.7	File archiver
	`make`	19	1.0	555.0	28.7	87.8	955.2	Build script interpreter
	`sed`	3	1.0	73.0	4.0	58.8	301.2	Stream editor
	`vim`	3	35.0	1749.0	66.2	67.1	907.9	Text editor
Defects4J	Chart	18	583.7	7789.8	96.3	4.3	2200.6	Chart library
	Closure	111	394.9	7587.2	90.2	3.2	7923.2	Closure compiler
	Lang	55	89.3	2151.6	22.1	2.2	2242.5	Apache commons-lang
	Math	83	512.4	4900.7	85.5	2.4	3599.3	Apache commons-math
	Mockito	28	270.4	1301.8	37.2	2.8	2431.9	Mocking library
	Time	15	156.4	4087.5	28.4	3.3	4126.8	Date and time library

from the artificial faults in the other SIR programs. As a result, we target 84 faults in the SIR benchmark. We used all tests in the universe test suite provided in the SIR benchmark.

- Defects4J is a collection of 395 real-world bugs in six Java programs with failing and passing unit tests. We used the 310 faults selected by Pearson et al. [57] to compare MBFL's FL precision with the results reported by Pearson et al.

Table 2 describes the target programs including the numbers of the buggy versions used, the number of their source code files, the number of their functions, their sizes in LOC, the average numbers of failing and passing test cases (i.e., $\lvert f_P \rvert$ and $\lvert p_P \rvert$, respectively) for each buggy version, and brief description. We have targeted 65 of 70 buggy versions in CoREBench (we excluded the following five buggy versions, because they cannot execute a failing test case or passing test cases due to conflict problems between the CoREBench docker container and the environment of the target buggy version[9]: `core.51a8f707`, `find.f7197f3a`, `core.b8108fd2`, `core.a860ca32`, `grep.074842d3`), and all 84 buggy versions in the SIR benchmark and 310 of 395 buggy versions in Defects4J.

4.3 Configuration

4.3.1 Mutation. PRINCE generates mutants of a target program, each of which is obtained by mutating exactly one statement covered by at least one failing test case. To generate mutants for C programs, we have used MUSIC [60], which implements the mutation operators defined in Agrawal el al. [3].[10] For Defects4J, we used Major 1.3.4 [32] to generate mutants for Java programs,

[9]Other papers report the same problems on the aforementioned target versions of CoREBench [52, 70].

[10]We tried but failed to use popular mutations tools for C such as Proteum [45] and Milu [30] for the experiments. Proteum (last release on 2001) does not recognize C99 standard and often fails to parse target C programs in CoREBench. Milu also frequently generated stillborn mutants from programs in CoREBench.

because Major is integrated with Defects4J benchmark framework. For high FL accuracy of MBFL, we generated more mutants by enabling all possible mutation operators of Conditional Operator Replacement (COR) and Relational Operator Replacement (ROR) in Major (for example, by default, ROR operator of Major does *not* mutate ">" to "<" nor "<" to ">"). Also, Major 1.3.4 supports Expression Value Replacement (EVR) mutation operator, which is not supported by Major 1.2.1 used by Pearson et al. [57].

To reduce a huge amount of mutant execution time, PRINCE randomly selects 20% of the mutants generated per statement. For example, if 85 mutants are generated at one statement, then PRINCE randomly selects 17 (= $\lceil 85 \times 20\% \rceil$) mutants.

4.3.2 Training and Validation. To avoid over-fitting problem, we use standard 10-fold cross validation. To evaluate PRINCE on the 65 target faults in CoREBench, we perform 10-fold cross validation using the 65 target faults as follows:

Step 1. We randomly partition the target 65 faults into 10 equal sized groups $g_1, \ldots, g_{10}$.[11]
Step 2. We select the kth ($0 < k \leq 10$) fault group of the 10 fault groups as a test set and use the other nine groups as a training set.
Step 3. PRINCE learns a ranking model using the faults in the training set.
Step 4. PRINCE applies the ranking model learned from the training set in Step 3 to the faults in the test set selected in Step 2.
Step 5. We repeat Step 2 to Step 4 for each k from 1 to 10.

Similarly, we apply 10-fold cross validation for SIR and Defects4J.

For the GP experiments, we use FLUCCS [68]. Regarding the rankSVM experiments, we use version 2.11 of LIBLINEAR [19], which is a linear rankSVM implementation based on version 3.22 of libSVM [9]. We choose LIBLINEAR, because it has been used to learn ranking models for FL in the literature [68]. We use default parameter values for rankSVM, because our preliminary experiments showed that using the default parameter values produce the most-precise FL results on average. For the non-linear rankSVM experiments, we use 3.22 of libSVM [9] for non-linear rankSVM. For non-linear rankSVM, we use a radial basis function (i.e., $e^{-\gamma(|c-v|^2)}$), where c is a constant weight to be learned, v is a feature variable, and γ is a parameter constant set as $\frac{1}{\#features}$, which is default in libSVM. For other parameters, we use the default values of libSVM.

4.3.3 Fault Localization Setup. Because of PRINCE's random mutation sampling, random shuffling for 10-fold cross validation, and random effects in GP, we repeated the experiment 30 times to minimize the random effect.

We implemented PRINCE in 4,300 lines of Python and Bash shell scripts using FLUCCS [68], LIBLINEAR [19], and LibSVM [9] as GP, linear rankSVM, and non-linear rankSVM engines, respectively. To measure the statement coverage achieved by a given test case, we used `gcov` for C programs and `Cobertura` for Java programs. All experiments were performed on 100 machines equipped with Intel i5 3.6GHz CPU, NVIDIA Geforce 1060 GPU, and 16GB of memory running Debian Linux 8 64 bits. PRINCE uses GPU to boost the speed of the GP computation implemented in FLUCCS. We ran one CoREBench docker instance on each machine.

4.4 Threats to Validity

The primary threat to external validity for our study involves the representativeness of our subject programs, since we have examined C programs in CoREBench and SIR, and Java programs in

[11] The size of each group can be slightly different when the number of target faults is not divisible by 10 (e.g., the size of g_1 to g_5 of CoREBench is 7 while the size of g_6 to g_{10} is 6).

Table 3. Statement and Branch Coverage of Given Test Cases, the Number of Target Statements, and the Number of the Generated Mutants

Benchmark	Target programs	Stmt Cov	Br. Cov	#Target Stmts	#Mutants
CoREBench	coreutils	83.4%	76.5%	557.5	12813.7
	find	71.5%	75.3%	914.0	11406.1
	grep	78.9%	72.4%	1180.7	8690.0
	make	63.9%	65.9%	1229.0	13820.4
SIR	bash	49.0%	46.2%	3108.2	16742.3
	flex	50.3%	45.7%	2107.2	12018.4
	grep	60.4%	50.3%	1508.0	9563.3
	gzip	64.2%	55.8%	398.6	5007.6
	make	67.1%	64.5%	2618.9	10795.0
	sed	56.8%	47.3%	1713.8	9758.9
	vim	40.8%	35.8%	4512.7	14356.4
Defects4J	Chart	65.1%	56.3%	7657.1	9781.7
	Closure	68.5%	55.6%	32974.0	33796.7
	Lang	85.0%	70.3%	1400.8	3078.0
	Math	71.0%	59.9%	4870.9	12070.7
	Mockito	70.4%	56.5%	4680.8	3273.5
	Time	78.2%	66.9%	10642.1	16081.9

Defects4J. While this may be the case, CoREBench, SIR, and Defects4J are constructed using the real-world C and Java programs. Also, CoREBench and Defects4J provide various real-world faults (i.e., faults that were identified during testing and operational use of the programs, not synthetic ones). Thus, we believe that this threat to external validity is limited. A primary threat to internal validity is the existence of possible faults in the tools that implement PRINCE. We controlled this threat through extensive testing of our tool. A threat to construct validity is the measure of FL precision metric we use. We controlled this threat by reporting not only the widely used expense metric to show the percentages of a program to be examined but also MAP and acc@n metric, which shows the absolute rank of a faulty statement as recommended by Parnin and Orso [55].

5 RESULT OF THE EMPIRICAL STUDY

We used the Wilcoxn test [72] to test whether the difference of experimental results is statistically significant. The detailed experiment data are available at http://swtv.kaist.ac.kr/data/PRINCE.

5.1 Experiment Data

Table 3 shows the statement and branch coverage of the target programs with the given test cases. The table also shows the number of target statements and the generated mutants. For example, as shown in the third row of the table, `find` covers 75.3% of its branches with the given test cases and 914.0 statements of `find` are executed by at least one failing test case (i.e., #Target Stmts) on average per faulty version. MUSIC generates 11406.1 mutants for `find` on average per faulty version.

5.2 RQ1: Precision of PRINCE

5.2.1 Expense Metric. Regarding the expense metric, PRINCE localizes faults very precisely and its expense in FL is only 2.4% for CoREBench, SIR, and Defects4J on average per faulty version.

Table 4. Expense Metric of the Eight Spectrum-based FL Techniques, the Two Mutation-based FL Techniques, and PRINCE in a Statement Level

Metric	Benchmark	Target programs	Naish1	Naish2	GP13	Wong1	Russell and Rao	Binary	Ochiai	Jaccard	MUSE	Metallaxis	PRINCE
Expense metric (%)	CoREBench	coreutils	33.4	33.4	33.4	87.4	33.4	87.4	33.4	33.4	10.3	12.2	3.3
		find	33.9	33.9	33.9	92.1	33.9	92.1	33.9	33.9	7.2	8.8	2.8
		grep	26.5	26.5	26.5	100.0	26.5	100.0	26.5	26.5	3.7	3.5	3.1
		make	41.4	41.4	41.4	77.9	41.4	77.9	41.4	41.4	9.2	9.2	3.4
		Average	**34.2**	**34.2**	**34.2**	88.5	**34.2**	88.5	**34.2**	**34.2**	**7.9**	8.7	3.2
	SIR	bash	22.7	21.5	21.5	33.9	21.9	35.6	24.9	21.9	8.7	11.7	2.5
		flex	20.3	19.4	20.0	40.5	22.1	44.4	23.3	23.7	13.7	21.3	3.2
		grep	2.8	2.6	2.7	3.6	3.0	5.0	2.7	2.8	1.3	1.6	0.8
		gzip	6.9	6.7	6.9	9.5	7.9	12.4	6.7	6.8	5.3	5.4	0.8
		make	17.7	15.5	18.6	33.9	18.5	24.4	15.7	16.5	3.9	5.3	1.3
		sed	12.6	11.4	11.4	16.9	13.6	18.8	11.6	11.6	1.2	1.8	0.4
		vim	13.4	11.6	11.9	21.1	13.8	17.8	13.7	12.6	2.5	3.0	1.3
		Average	13.1	**12.1**	13.0	23.2	13.9	22.8	13.3	13.4	**6.0**	8.4	1.6
	Defects4J	Chart	4.7	4.9	5.2	23.0	5.1	23.4	2.5	8.0	8.7	6.4	2.4
		Closure	6.3	6.5	5.3	26.6	5.3	13.9	4.8	6.9	12.7	7.8	2.4
		Lang	4.9	4.4	5.5	26.0	6.6	26.6	4.8	8.9	12.4	6.7	3.1
		Math	3.6	3.4	5.1	14.4	6.1	26.0	2.8	7.7	12.8	4.6	2.5
		Mockito	5.2	3.1	6.8	23.8	7.7	23.2	4.4	9.3	9.5	6.7	2.2
		Time	5.4	5.8	4.5	26.0	8.6	12.1	5.2	7.9	12.5	3.6	1.9
		Average	5.1	4.8	5.4	22.7	6.1	20.7	**4.1**	7.8	12.1	**6.4**	2.5
	Average		10.7	10.3	10.9	32.1	11.5	30.7	**10.0**	12.6	10.4	**7.1**	2.4

Table 4 presents the expense metric (i.e., the proportion of the executed statements to examine to localize a target fault) for the applied FL techniques for CoREBench, SIR, and Defects4J. The most precise average results per faulty version in the SBFL techniques and the MBFL techniques are marked in a bold font. The average precision of PRINCE per faulty version is 3.0 (=7.1%/2.4%) times higher than the best fixed-formula fault localization technique (i.e., Metallaxis) (see the last row showing the average expenses of Table 4). The difference between PRINCE and Metallaxis in terms of the expense metric is statistically significant and the effect size is 3.41. Also the average precision of PRINCE per faulty version is 4.2 (=10.0%/2.4%) times higher than the best SBFL technique (i.e., Ochiai).[12] The difference between PRINCE and Ochiai in terms of the expense metric is statistically significant and the effect size is 3.92.

For Defects4J, MBFL is less precise than Ochiai unlike CoREBench and SIR. The difference between the best MBFL technique (i.e., Metallaxis) and Ochiai in terms of the expense metric is statistically significant and the effect size is 0.66. This is because FL accuracy of MBFL depends on

[12]For CoREBench, six SBFL techniques (Naish1, Naish2, GP13, Russell and Rao, Ochiai, and Jaccard) showed the same expense because 58 of 65 target CoREBench faulty versions have only one failing test case and the remaining seven target versions have two failing test cases that have the same statement coverage. Thus, $e_f(s)$ and $n_f(s)$ are 1 and 0 for all target statements s in the 58 versions, respectively (similarly, $e_f(s)$ and $n_f(s)$ are 2 and 0 for all target statement s in the remaining 7 versions, respectively). Then, the suspiciousness formulas of the six SBFL techniques become *monotonically* decreasing functions of $e_p(s)$. Thus, although the suspiciousness scores for s may be different among the six SBFL techniques, the obtained suspiciousness rankings of s (including a faulty statement) are all the same.

Table 5. Mean Average Precision (MAP) of the Eight Spectrum-based FL Techniques, the Two Mutation-based FL Techniques, and PRINCE in a Statement Level

Metric	Benchmark	Target programs	Russell	Naish1	Naish2	GP13	Wong1	and Rao	Binary	Ochiai	Jaccard	MUSE	Metallaxis	PRINCE
MAP	CoREBench	coreutils		0.0300	0.0300	0.0300	0.0114	0.0300	0.0114	0.0300	0.0300	0.0971	0.0823	0.3067
		find		0.0295	0.0295	0.0295	0.0109	0.0295	0.0109	0.0295	0.0295	0.1389	0.1138	0.3623
		grep		0.0377	0.0377	0.0377	0.0100	0.0377	0.0100	0.0377	0.0377	0.2703	0.2845	0.3211
		make		0.0242	0.0242	0.0242	0.0128	0.0242	0.0137	0.0242	0.0242	0.1087	0.1087	0.2967
		Average		**0.0299**	**0.0299**	**0.0299**	0.0114	**0.0299**	0.0116	**0.0299**	**0.0299**	**0.1466**	0.1399	0.3190
	SIR	bash		0.0440	0.0466	0.0466	0.0295	0.0457	0.0281	0.0402	0.0457	0.1144	0.0854	0.4078
		flex		0.0492	0.0516	0.0501	0.0247	0.0453	0.0225	0.0430	0.0423	0.0728	0.0470	0.3157
		grep		0.3622	0.3876	0.3691	0.2769	0.3341	0.2008	0.3663	0.3623	0.7634	0.6107	0.7016
		gzip		0.1453	0.1497	0.1439	0.1054	0.1258	0.0805	0.1497	0.1479	0.1894	0.1857	0.6883
		make		0.0565	0.0644	0.0537	0.0295	0.0541	0.0410	0.0638	0.0607	0.2591	0.1877	0.7937
		sed		0.0796	0.0876	0.0876	0.0592	0.0736	0.0531	0.0864	0.0864	0.6065	0.5679	0.6915
		vim		0.0744	0.0864	0.0838	0.0474	0.0726	0.0561	0.0732	0.0792	0.3953	0.3294	0.7463
		Average		0.1378	**0.1474**	0.1394	0.0976	0.1265	0.0786	0.1397	0.1383	**0.3258**	0.2575	0.6128
	Defects4J	Chart		0.2151	0.2041	0.1934	0.0436	0.1957	0.0427	0.4065	0.1258	0.1149	0.1062	0.4167
		Closure		0.1587	0.1550	0.1898	0.0376	0.1883	0.0720	0.2092	0.1453	0.0787	0.1143	0.4167
		Lang		0.2062	0.2268	0.1805	0.0385	0.1524	0.0377	0.2101	0.1125	0.0806	0.1297	0.3226
		Math		0.2809	0.2950	0.1953	0.0693	0.1629	0.0385	0.3521	0.1297	0.0781	0.1776	0.4000
		Mockito		0.1934	0.3226	0.1466	0.0420	0.1300	0.0432	0.2257	0.1073	0.1053	0.1307	0.4545
		Time		0.1845	0.1718	0.2247	0.0384	0.1163	0.0828	0.1912	0.1261	0.0800	0.2193	0.5263
		Average		0.2075	0.2240	0.1876	0.0470	0.1668	0.0532	**0.2597**	0.1298	0.0835	**0.1401**	0.4042
	Average			0.1696	0.1825	0.1565	0.0512	0.1401	0.0519	**0.2052**	0.1172	0.1368	**0.1615**	0.4858

diverse mutant generation [54] (i.e., MBFL achieves higher FL precision by using more diverse mutants) and Major generates much fewer mutants per line for Defects4J than MUSIC for CoREBench and SIR. For example, only 1.58 mutants per target line were generated for Defects4J, while 9.91 mutants per target line were generated for CoREBench and SIR, on average per faulty version.[13]

5.2.2 Mean Average Precision Metric. Regarding the mean average precision (MAP) metric, PRINCE localizes faults more precisely than the other FL techniques for all subject programs.

Table 5 presents MAP for the applied FL techniques for CoREBench, SIR, and Defects4J. The average MAP of PRINCE per faulty version is 0.4858, which is 2.4 (=0.4858/0.2052) and 3.0 (=0.4858/0.1615) times greater than that of the best SBFL (Ochiai) and MBFL (Metallaxis) techniques. The difference between PRINCE and Ochiai in terms of MAP is statistically significant and the effect size is 2.41. In addition, the difference between PRINCE and Metallaxis in terms of MAP is statistically significant and the effect size is 3.08.

5.2.3 Acc@n Metrics. Regarding the acc@1, acc@5, and acc@10 metrics, PRINCE again outperforms the SBFL and the MBFL techniques for CoREBench, SIR, and Defects4J.

[13]Our experiment on Defects4J uses 39.8% more mutants than the experiment in Pearson et al. [57] (i.e., 1.58 vs 1.13 mutants per target line on average per faulty version). Thus, in contrast to Pearson et al. [57] which reported the expense metrics of MUSE and Metallaxis as 20.6% and 7.5% for the 310 faults in Defects4J on average, our experiment result shows that the expense metrics of MUSE and Metallaxis are 12.1% and 6.4% for the same 310 faults in Defects4J on average per faulty version, respectively.

Table 6 shows that PRINCE locates a faulty statement at the top rank for 13.1% (=60/(65+84+310)) of the target faults (see the last column of the last row of the acc@1 section in Table 6). Compared to the best fixed formula FL technique (i.e., MUSE), PRINCE locates 1.7 times (=60/35) more faults at the top rank. Compared to Ochiai, which is the best SBFL technique, PRINCE locates 2.5 (=60/24) times more faults at the top rank. For acc@10, PRINCE locates 52.9% (=243/(65+84+310)) of the target faults in the top 10 ranks (see the last column of the last row of the acc@10 section in Table 6).

5.2.4 Comparison with Other Learning-based FL Techniques. We have compared PRINCE with Savant [5], which is a state-of-the-art learn-to-rank FL technique that localizes a fault in a function level. Table 7 shows the comparison results of PRINCE with Savant. The table shows the expense metric, MAP, acc@1, acc@3, and acc@5 of Savant and PRINCE in a function level. We compare the function-level FL precision of PRINCE with the experiment results reported in the Savant paper [5].

The results in Table 7 show that PRINCE outperforms Savant in terms of the expense metric, MAP, acc@1, acc@3, and acc@5 for Defects4J in a function level. PRINCE achieved 94.9% lower expense metric than Savant. The difference between PRINCE and Savant in terms of the expense metric is statistically significant and the effect size is 4.83. In addition, the difference between PRINCE and Savant in terms of MAP is statistically significant and the effect size is 2.45. Also, PRINCE localizes a fault in the top one, three, and five functions for 58.7%, 56.6%, and 38.5% more faulty program versions than Savant, respectively.

5.3 RQ2: Efficiency of PRINCE

Table 8 shows the time spent by the SBFL techniques, the MBFL techniques, and PRINCE for FL per faulty version. The SBFL techniques spend most of the FL time for running test cases, and the time taken to calculate suspiciousness scores is negligible (i.e., less than 3s). Therefore, we can consider that the runtime costs of the different SBFL techniques are same. The SBFL techniques spend up to 230.9s (i.e., Closure in Defects4J) for localizing a fault depending on the number of test cases and execution time of test cases. The MBFL techniques (i.e., Metallaxis and MUSE) spend most of the FL time for generating mutants and running test cases on the mutants. Since the time taken to calculate the MBFL suspiciousness scores is negligible, we can consider that the runtime costs of the MBFL techniques are same.

The runtime cost of PRINCE mainly consists of the followings: feature value computation time, training time to learn ranking models, and suspiciousness score computation time using the generated ranking models.

- *Feature value computation time*: This consists of the execution times to compute the SBFL features, the MBFL features, file/function/statement features of a target program P.
  - —The execution time to compute SBFL features is the sum of the time to execute the test cases on P and the time to evaluate SBFL feature formulas with the test execution results on P.
  - —The execution time to compute MBFL is the sum of the time to generate mutants from the target statements of P, the time to compile the randomly sampled 20% of the generated mutants, the time to execute test cases on the sampled mutants, and the time to evaluate MBFL feature formulas with the test execution results on the sampled mutants.[14] Computing MBFL features for PRINCE is faster than applying MBFL techniques, because

[14] Although it takes a large amount of time to compute MBFL feature values, running test cases on mutants can easily parallelized on distributed computing nodes so that the execution time of PRINCE can be acceptable in practice.

Table 6. acc@1, acc@5, and acc@10 of the Eight Spectrum-based FL Techniques, the Two Mutation-based FL Techniques, and PRINCE in a Statement Level

Metric	Bench-mark	Target programs	Naish1	Naish2	GP13	Wong1	Russell and Rao	Binary	Ochiai	Jaccard	MUSE	Metallaxis	PRINCE
acc@1	CoRE-Bench	coreutils	0	0	0	0	0	0	0	0	1	1	3
		find	0	0	0	0	0	0	0	0	1	0	2
		grep	0	0	0	0	0	0	0	0	1	1	3
		make	0	0	0	0	0	0	0	0	4	1	6
		Total	0	0	0	0	0	0	0	0	**7**	3	14
	SIR	bash	0	0	0	0	0	0	0	0	0	0	0
		flex	0	0	0	0	0	0	0	0	2	0	3
		grep	4	4	4	3	4	3	4	4	6	5	8
		gzip	3	3	3	1	3	2	3	3	8	7	10
		make	0	0	0	0	0	0	0	0	2	0	3
		sed	0	0	0	0	0	0	0	0	1	0	1
		vim	0	0	0	0	0	0	0	0	0	0	0
		Total	**7**	**7**	**7**	4	**7**	5	**7**	**7**	**19**	12	25
	Defects4J	Chart	0	0	0	0	0	0	0	0	0	0	1
		Closure	3	4	3	1	4	1	6	4	3	4	7
		Lang	1	2	0	0	1	0	2	1	1	1	3
		Math	3	5	3	0	2	0	5	5	2	5	5
		Mockito	1	1	4	1	3	1	4	4	3	4	4
		Time	0	0	0	0	0	0	0	0	0	0	1
		Total	8	12	10	2	10	2	**17**	14	9	14	21
	Total		15 (3.3%)	19 (4.1%)	17 (3.7%)	6 (1.3%)	17 (3.7%)	7 (1.5%)	**24 (5.2%)**	21 (4.6%)	**35** (7.6%)	29 (6.3%)	60 (13.1%)
acc@5	CoRE-Bench	coreutils	0	0	0	0	0	0	0	0	3	2	5
		find	0	0	0	0	0	0	0	0	1	0	3
		grep	0	0	0	0	0	0	0	0	3	1	3
		make	0	0	0	0	0	0	0	0	4	3	7
		Total	0	0	0	0	0	0	0	0	**11**	6	18
	SIR	bash	0	0	0	0	0	0	0	0	0	0	1
		flex	0	0	0	0	0	0	0	0	2	1	5
		grep	6	7	6	3	4	3	4	5	8	6	9
		gzip	3	3	3	1	3	2	3	3	9	7	10
		make	0	0	0	0	0	0	0	0	2	0	4
		sed	0	0	0	0	0	0	0	0	1	0	1
		vim	0	0	0	0	0	0	0	0	0	0	0
		Total	9	**10**	9	4	7	5	7	8	**22**	14	30
	Defects4J	Chart	5	5	4	1	4	1	6	4	5	5	7
		Closure	21	22	21	11	21	8	24	21	19	18	28
		Lang	26	24	26	9	27	11	29	25	24	25	32
		Math	28	28	27	9	28	8	32	27	25	26	34
		Mockito	11	12	11	3	13	4	15	11	9	7	18
		Time	8	9	9	2	9	2	11	9	8	7	15
		Total	99	100	98	35	102	34	**117**	97	**90**	88	134
	Total		108 (23.5%)	110 (24.0%)	107 (23.3%)	39 (8.5%)	109 (23.7%)	39 (8.5%)	**124 (27.0%)**	105 (22.9%)	**123 (26.8%)**	108 (23.5%)	182 (39.7%)
acc@10	CoRE-Bench	coreutils	0	0	0	0	0	0	0	0	4	3	8
		find	0	0	0	0	0	0	0	0	2	1	6
		grep	1	1	1	0	1	0	1	1	4	5	7
		make	0	0	0	0	0	0	0	0	5	4	10
		Total	1	1	1	0	1	0	1	1	15	13	31
	SIR	bash	0	0	0	0	0	0	0	0	0	0	2
		flex	2	3	4	2	2	2	2	2	3	2	7
		grep	6	7	6	3	4	3	4	5	9	7	12
		gzip	3	3	3	1	3	2	3	3	11	9	11
		make	1	2	2	1	1	1	2	1	5	3	7
		sed	0	0	0	0	0	0	0	0	1	0	1
		vim	0	0	0	0	0	0	0	0	0	0	0
		Total	14	**15**	**15**	7	10	8	11	11	**29**	21	40
	Defects4J	Chart	10	9	11	3	11	2	13	9	12	13	13
		Closure	27	28	28	15	30	11	35	28	31	32	38
		Lang	28	26	29	13	31	15	35	27	28	29	39
		Math	34	35	38	14	36	11	42	33	33	35	47
		Mockito	10	12	12	5	11	6	16	12	11	13	19
		Time	8	10	11	3	9	5	13	10	10	12	16
		Total	117	120	129	53	128	50	**154**	119	125	**134**	172
	Total		132 (28.8%)	136 (29.6%)	145 (31.6%)	60 (13.1%)	139 (30.3%)	58 (12.6%)	**166** (36.2%)	131 (28.5%)	**169** (36.8%)	168 (36.6%)	243 (52.9%)

Table 7. Expense Metric, MAP, acc@1, acc@3, and acc@5 of Savant and PRINCE in Method Level for Defects4J

Target	Expense metric (%)		MAP		acc@1		acc@3		acc@5	
Programs	Savant	PRINCE	Savant	PRINCE	Savant	PRINCE	Savant	PRINCE	Savant	PRINCE
Chart	18.9	1.1	0.0529	0.6091	5	6	9	9	11	13
Closure	15.3	1.2	0.0654	0.5333	2	8	13	39	21	45
Lang	23.3	0.8	0.0429	0.6806	29	28	41	51	45	52
Math	15.4	0.9	0.0649	0.6711	22	51	47	78	57	81
Time	8.6	0.7	0.1163	0.6831	5	7	12	14	14	14
Average	17.5	0.9	0.0535	0.5886						
Total					63	100	122	191	148	205
					22.3%	35.5%	43.3%	67.7%	52.5%	72.7%

Table 8. Fault Localization Time of SBFL, MBFL, and PRINCE in Seconds

				PRINCE			
Benchmark	Target programs	SBFL	MBFL	Avg. feature value computation	Training	Susp. computation	Total
CoREBench	coreutils	7.2	7410.7	1951.0	70.7	0.6	2022.3
	find	13.3	14182.8		74.5	1.3	2026.9
	grep	12.1	11685.6		63.0	0.5	2014.5
	make	6.8	6763.4		74.2	1.2	2026.5
	Average	9.5	9610.8	1951.0	70.8	0.9	2022.8
SIR	bash	21.6	21683.1	2751.9	66.0	0.6	2818.4
	flex	10.7	11022.5		71.4	1.1	2824.4
	grep	14.6	13153.8		74.0	1.2	2827.0
	gzip	7.5	7977.4		71.3	0.8	2824.0
	make	11.5	12313.6		77.4	0.8	2830.0
	sed	9.4	8972.6		71.8	1.5	2825.1
	vim	29.1	28105.4		62.0	1.2	2815.0
	Average	12.5	12489.6	2751.9	72.6	1.0	2825.4
Defects4J	Chart	28.9	3509.4	2768.5	246.7	1.9	3015.2
	Closure	230.9	23658.5		190.5	1.6	2959.0
	Lang	28.6	3751.6		192.6	3.7	2961.1
	Math	91.5	11196.5		200.4	2.0	2968.9
	Mockito	55.3	6399.5		184.0	2.9	2952.5
	Time	31.3	4642.1		224.8	3.8	2993.3
	Average	120.4	13141.1	2768.5	197.9	2.3	2966.3

PRINCE used only 20% of the generated mutants while MBFL techniques used all mutants.[15]

—The execution time to compute file, function, and statement features is time to perform static analysis.

We report an average of feature value computation time per faulty version for each benchmark. For example, the reported feature value computation time for CoREBench (i.e.,

[15]If PRINCE uses all generated mutants, then it increases the execution time almost 5 times but decreases the expense only 0.5%p (1.9% vs. 2.4%) compared to PRINCE using only 20% of the mutants.

Table 9. Expense Metric of PRINCE Using All Features and PRINCE Using All Except Each of the Five Feature Groups

Benchmark	Target programs	PRINCE using all features	PRINCE using all features except				
			MBFL features	SBFL features	Stmt. features	Func. features	File features
CoREBench	coreutils	3.3	17.5	6.8	7.2	4.7	6.1
	find	2.8	12.7	6.3	5.1	4.3	6.5
	grep	3.1	9.4	5.8	4.9	4.6	7.2
	make	3.4	13.8	6.6	5.7	5.2	4.8
	Average	3.2	13.7	6.4	5.8	4.7	6.1
SIR	bash	2.5	12.8	3.6	4.4	3.9	3.6
	flex	3.2	19.7	6.5	6.3	4.3	3.9
	grep	0.8	3.1	1.1	0.9	1.0	0.7
	gzip	0.8	2.9	1.3	1.5	0.8	0.7
	make	1.3	5.9	2.3	1.9	1.7	1.2
	sed	0.4	1.8	0.6	0.5	0.4	0.4
	vim	1.3	5.0	2.7	1.4	1.4	2.5
	Average	1.6	8.2	2.8	2.7	2.1	1.8
Defects4J	Chart	2.4	2.1	8.8	2.6	2.4	3.2
	Closure	2.4	4.2	8.2	2.5	2.2	2.8
	Lang	3.1	4.3	6.8	3.6	3.7	3.9
	Math	2.5	2.5	5.1	3.2	2.7	3.1
	Mockito	2.2	4.1	6.8	2.6	2.6	2.7
	Time	1.9	4.4	4.2	2.1	1.8	2.4
	Average	2.5	3.6	6.8	2.9	2.6	3.1

1951.0s) is the sum of the feature value computation time of the all faulty versions of CoREBench divided by the number of the all target faulty versions of CoREBench (i.e., 65). This is because PRINCE needs feature value computation of not only a target faulty version, but also the other faulty versions together to learn a ranking model. Also, the feature values are computed once and reused for FL of every target faulty version. We report average feature value computation time for SIR and Defects4J similarly.

- *Training time*: Training time is the execution time of genetic programming to learn a ranking model from known faults data. Since we have 10 groups of faulty target program versions (e.g., $g_1, g_2, \ldots g_{10}$) in 10-fold cross validation setup for each benchmark, the training time of g_i is the execution time of genetic programming to learn a ranking model from the other nine fault groups. For each faulty version $P_j \in g_i$, the training time is $\frac{\text{training time of } g_i}{|g_i|}$, because the faulty versions in the same fault group share the same ranking model. We report the average training time per faulty version.
- *Susp. computation time*: We report the execution time of applying the learned ranking model to the target statements of each target faulty program version on average per faulty version.

5.4 RQ3: Importance of Program Feature Groups for FL Precision

Table 9 shows the expense metric of PRINCE using all feature groups (see the third column) and PRINCE using all except one of the five feature groups (see the fourth to the last columns). We treat the increase ratio of expense metric caused by excluding a feature group as the contribution of the feature group to the precision of FL.

For CoREBench and SIR, among the five feature groups, the MBFL feature group contributes to the precision of FL most significantly. Removing the MBFL features increases the expense metric value by 4.3 times for CoREBench (=13.7%/3.2% (see the 3rd and 4th columns of the 6th row)) and 5.1 times (=8.2%/1.6% (see the 3rd and 4th columns of the 14th row)) for SIR, on average per faulty version. The second-most effective feature group is SBFL. Removing SBFL features increases the expense metric value by 2.0 times (=6.4%/3.2% (see the 3rd and 5th columns of the 6th row)) for CoREBench, and 1.8 times (=2.8%/1.6% (see the 3rd and 5th columns of the 14th row)) for SIR, on average per faulty version.

For CoREBench, the file features are the third-most important feature group to the precision. Removing the file features increases the expense metric value by 1.7 times (=6.1%/3.2% (see the third and the last columns of the sixth row)) for CoREBench, on average per faulty version. The importance of the statement features and the function features follow that of the file features in order.

For the SIR benchmark, the statement features are the 3rd-most important feature group to the precision. Removing the statement features increase the expense metric value by 1.7 times (=2.7%/1.6% (see the 3rd and 6th columns of the 14th row)) on average per faulty version. The importance of the function features and the file features follow that of the statement features. The file features (the 3rd-most important for CoREBench) are the least important for SIR, because 5 of 7 SIR benchmark programs consist of a single source code file.

For Defects4J, among the five feature groups, the SBFL feature group contributes to the precision of FL most significantly. Removing the SBFL increases expense metric value by 2.7 times (=6.8%/2.5% (see the third and fifth columns of the last row)) on average per faulty version. The second-most-effective feature group is MBFL. Removing MBFL features increases expense metric value by 1.4 times (=3.6%/2.5% (see the third and fourth columns of the last row)) on average per faulty version. Note that MBFL may contribute to the FL precision more than SBFL for Defects4J if a sufficient number of diverse mutants are provided (see Section 5.2).

5.5 RQ4: Importance of Program Feature Groups for FL Precision with Different Machine Learning Techniques

We can confirm that the importance of the feature groups to FL precision is preserved with three different machine-learning algorithms. Table 10 shows the expense metric of PRINCE using all features (see the fourth column) and PRINCE using all except one of the five feature groups (see the fifth to ninth columns) with linear rankSVM and non-linear rankSVM instead of GP.

For CoREBench and SIR, the most important feature group of all the three learning algorithms (i.e., GP, linear rankSVM, and non-linear rankSVM), is still the MBFL. For example, excluding the mutation features increases the expense for CoREBench as follows:

- 4.3 times for PRINCE with genetic programming (=13.7%/3.2%)
- 4.5 times for PRINCE with linear rankSVM (=17.9%/4.0%)
- 4.4 times for PRINCE with non-linear rankSVM (=16.2%/3.7%)

The order of the importance to FL precision among the five feature groups is the same for the all three learning algorithms for CoREBench. For CoREBench, the SBFL features and file features are the second- and the third-most important features for all three learning algorithms, respectively. The importance of the statement and the function features follow that of the file features with all the three learning algorithms, too.

For SIR, the order of the importance to FL precision among the five feature groups is also the same as all three learning algorithms. For SIR, the SBFL features are the second-most-important

Table 10. Expense Metric of PRINCE Using All Features and PRINCE Using All Except Each of the Five Feature Groups with the Linear rankSVM and Non-linear rankSVM Learning Algorithms

Learning Algorithms	Benchmark	Target programs	PRINCE using all features	PRINCE using all features except				
				MBFL features	SBFL features	Stmt. features	Func. features	File features
Linear RankSVM	CoREBench	coreutils	4.1	24.3	8.5	9.0	5.9	9.0
		find	3.6	16.8	7.6	6.2	5.2	9.5
		grep	3.7	11.6	7.4	6.9	5.9	8.5
		make	4.3	17.0	9.0	8.0	6.3	6.7
		Average	4.0	17.9	8.2	7.7	5.8	7.9
	SIR	bash	3.5	16.1	4.3	5.5	4.4	4.5
		flex	4.6	23.6	8.4	8.2	5.0	4.5
		grep	0.9	3.8	1.4	1.0	1.2	0.8
		gzip	1.0	3.3	1.6	1.7	0.9	0.8
		make	1.5	6.5	3.0	2.5	2.1	1.5
		sed	0.4	2.2	0.7	0.6	0.4	0.5
		vim	1.5	6.4	3.5	1.8	1.8	2.9
		Average	2.1	9.7	3.6	3.4	2.5	2.1
	Defects4J	Chart	2.5	2.6	10.4	3.2	3.0	3.5
		Closure	2.3	5.0	10.0	2.9	2.4	3.1
		Lang	3.5	5.0	8.1	4.6	4.8	5.0
		Math	2.6	2.9	6.4	4.0	3.1	3.6
		Mockito	2.4	4.8	8.1	3.2	3.0	3.1
		Time	2.0	5.4	5.1	2.6	2.3	3.0
		Average	2.6	4.3	8.3	3.5	3.1	3.6
Non-linear RankSVM	CoREBench	coreutils	3.7	20.5	8.7	8.6	5.4	8.1
		find	3.6	14.1	8.1	6.4	4.9	7.8
		grep	3.8	11.0	6.4	6.1	5.8	7.6
		make	3.8	17.3	7.4	6.5	6.7	6.0
		Average	3.7	16.2	7.6	7.0	5.8	7.3
	SIR	bash	3.4	14.3	4.5	5.7	4.5	4.4
		flex	4.1	25.2	8.3	7.1	5.0	4.9
		grep	0.9	3.5	1.3	1.2	1.1	0.8
		gzip	1.0	3.5	1.5	1.7	1.0	0.9
		make	1.5	7.6	2.9	2.3	1.9	1.5
		sed	0.5	2.3	0.8	0.6	0.5	0.5
		vim	1.4	5.8	3.3	1.8	1.6	3.1
		Average	2.0	10.1	3.6	3.2	2.4	2.2
	Defects4J	Chart	2.3	2.4	9.5	3.0	2.7	3.8
		Closure	2.3	4.7	9.3	2.8	2.5	3.3
		Lang	3.5	5.0	7.9	4.3	4.3	4.3
		Math	2.9	3.1	6.0	3.7	3.4	3.9
		Mockito	2.3	4.4	7.8	2.8	2.8	3.1
		Time	1.8	4.9	4.7	2.3	2.0	2.8
		Average	2.6	4.2	7.8	3.3	3.1	3.6

features followed by the statement features and the function features for all three learning algorithms. For SIR, the file features are the least important as described in Section 5.4.

For Defects4J, the most important feature group of all the three learning algorithms is SBFL. For example, excluding the SBFL features increases the expense as follows:

- 2.7 times for PRINCE with genetic programming (=6.8%/2.5%)
- 3.2 times for PRINCE with linear rankSVM (=8.3%/2.6%)
- 3.0 times for PRINCE with non-linear rankSVM (=7.8%/2.6%)

The order of the importance to FL precision among the five feature groups is also the same as all three learning algorithms for Defects4J. The MBFL features are the second-most important followed by the file, statement, and function features. The difference between the learners is not statistically significant.

6 DISCUSSIONS

6.1 Advantage of Adopting Learn-to-Rank Technique

The use of the learn-to-rank technique with genetic programming allows PRINCE to be flexible in adapting the target faulty program. PRINCE learns FL formula and weights for the program features by actively optimizing them for given subjects. Compared to pre-determined weights, this adaptive approach has significant practical advantages. If PRINCE uses a pre-defined formula, then it would fail to achieve high precision of FL of CoREBench, SIR, and Defects4J benchmark programs, because the CoREBench and Defects4J programs have largely different characteristics on files from those of SIR programs (i.e., CoREBench and Defects4J programs consist of multiple source files while 5 of 7 SIR programs have only a single source file). If file features have high weights, then an FL technique with a fixed formula may be precise for CoREBench and Defects4J programs but imprecise for the SIR benchmark programs. Similarly, if file features have low weights, then a FL technique with a fixed formula may be imprecise for the CoREBench and Defects4J programs, since it misses a chance to utilize important file features. To overcome such limitation, PRINCE actively optimizes FL formula for given data sets (e.g., 149 known faults for C programs and 310 known faults for Java programs). In practice where programs usually have long lifetime through many revisions, a user can train PRINCE with data sets obtained from already fixed faults and apply PRINCE for effective FL.

6.2 Effective Features for Precise Fault Localization

6.2.1 General Applicability of the Proposed Features for Fault Localization. Note that our experiments show that the proposed features are generally effective to improve FL, since these features improve precision of fault localization with three different learning algorithms (i.e., genetic programming, linear rankSVM, and non-linear rankSVM) in a similar degree. For example of CoREBench, the expense metric of PRINCE using genetic programming, linear rankSVM, and non-linear rankSVM are 3.2%, 4.0%, and 3.7% on average per faulty version, respectively. This is far more precise than Naish2 (34.2% on average per faulty version), the best SBFL technique, and MUSE (7.9% on average per faulty version), the best MBFL technique for C benchmark programs. For Defects4J, the expense metric of PRINCE using genetic programming, linear rankSVM, and non-linear rankSVM are 2.5% 2.6%, and 2.6% on average per faulty version, respectively. This is more precise than Ochici (4.1% on average per faulty version), the best SBFL technique, and Metallaxis (6.4% on average per faulty version), the best MBFL technique for Java benchmark programs.

Also note that relative significance of each feature group for the fault localization precision still remains the same with the three different learning algorithms. For example, mutation features

Table 11. The Number of LoC of Patches, Expense Metric, and Correlation between the LoC of Patches and Expense Metric

Benchmark	Target programs	LoC of patches		Expense (%)			Correlation		
		Avg.	Max.	MUSE	Metallaxis	PRINCE	MUSE	Metallaxis	PRINCE
CoREBench	coreutils	4.8	15	10.3	12.2	3.3	0.26	0.11	0.14
	find	6.7	17	7.2	8.8	2.8	0.30	0.25	0.17
	grep	5.9	13	3.7	3.5	3.1	0.19	0.38	0.18
	make	10.5	58	9.2	9.2	3.4	0.27	0.34	0.11
SIR	bash	1.8	5	8.7	11.7	2.5	−0.03	0.00	0.03
	flex	1.2	3	13.7	21.3	3.2	0.01	−0.02	−0.05
	grep	1.7	4	1.3	1.6	0.8	−0.07	−0.04	−0.02
	gzip	1.5	3	5.3	5.4	0.8	−0.04	0.00	−0.04
	make	1.5	3	3.9	5.3	1.3	−0.03	−0.07	−0.02
	sed	1.4	2	1.2	1.8	0.4	−0.01	0.02	0.01
	vim	2.1	5	2.5	3.0	1.3	−0.07	−0.03	0.01
Defects4J	Chart	7.3	39	8.7	6.4	2.4	0.15	0.18	−0.01
	Closure	6.0	43	12.7	7.8	2.4	0.20	0.35	0.05
	Lang	7.6	43	12.4	6.7	3.1	0.35	0.23	0.13
	Math	6.5	54	12.8	4.6	2.5	0.28	0.26	0.14
	Mockito	7.1	31	9.5	6.7	2.2	0.21	0.34	0.03
	Time	8.5	26	12.5	3.6	1.9	0.28	0.24	−0.04

improve the precision of fault localization on CoREBench most significantly, followed by SBFL, file, statement, and function features for all three different learning algorithms. These observations imply that the proposed features of PRINCE are not only effective for a specific learning algorithm for FL, but also generally effective for various learning algorithms. Thus, we can conclude that the proposed set of program features studied in this article are valuable to learn-to-rank techniques in general.

6.2.2 Mutation Feature Group. Through the experiment (Table 9), we have found that mutation feature group contributes to increase the precision of FL (in terms of expense metric) most significantly among the five feature groups for CoREBench and SIR, and second-most significantly for Defects4J. Since FL can localize a fault precisely with diverse passing and failing test cases, the mutation features contribute in precisely localizing a fault by utilizing *diverse mutant executions* even with a few given test cases. In contrast, SBFL features may fail to contribute in precisely localizing a fault if only a few test cases are available. Note that, in a real-world environment, generating diverse failing test cases is highly challenging. Also, compared to SBFL, which may suffer from the same basic block problem (i.e., every statement in a same basic block has the same coverage, which may lower a rank of a target fault), the mutation feature can assign different suspiciousness values to statements in a same block.

Furthermore, we found that MBFL has weak correlation between the complexity of a patch and FL precision. Table 11 shows the patch complexity (in LoC), expense of FL techniques, and correlation between the patch complexity and the expense. We use the added/removed/changed lines of code of a patch as a complexity measure of a patch and the expense metric as a FL precision. We use Pearson correlation [56] as a correlation measure. The experimental results show that MUSE and Metallaxis have almost no correlation between the patch complexity and FL precision for SIR benchmark programs, which have only simple patches (i.e., the sizes of the patches are no

more than five lines). For CoREBench and Defects4J, which have complex patches (up to 58 lines long patch for `make`), the correlation is still weak (ranging from 0.11 to 0.38). PRINCE has even weaker correlation between the patch complexity and FL precision than MBFL (i.e., the correlation ranges from −0.04 to 0.18).

6.2.3 File Feature Group. A salient observation from the experiments is that the file feature group contributes to increase FL significantly.[16] Our conjecture is that the file features (e.g., fan-in of a source code file and a number of defined functions and variables in a source code file) have high correlation to the number of faults in a file, because such features indicate complexity of interactions with other files that often become causes of failures. Among the 12 file features, the fan-in in a file dependency graph, the number of functions defined in a file, and the number of variables defined in a file are the top 3 effective features. For each of the four target programs in CoREBench and six target programs of Defects4J, we compute the Pearson correlation coefficient [56] between the values of each of the top three effective features and the number of faults in a source code file.[17] The correlation between the fan-in in a file dependency graph and the number of faults is 0.72 on average, which indicates high correlation. The correlation between the number of functions and the number of faults is 0.61 and the one between the number of variables and the number of faults is 0.59, which is also reasonably high. During the learning phase, PRINCE captures this correlation between file features and the number of faults and generates a suspiciousness formula, which utilizes the file features effectively to increase precision of FL.

6.3 General Applicability of PRINCE's Ranking Model

We performed a simple case study to investigate whether or not a ranking model learned by PRINCE from one project can be effective in precisely localizing faults in other projects. We compare the expense of PRINCE using the 10-fold cross validation and inter-benchmark training and validation in Table 12. The third column shows the expense of PRINCE using the 10-fold cross validation. The fourth to ninth columns show the expense of PRINCE using inter-benchmark training and validation. The column name A to B means that PRINCE uses all the faults in benchmark A as a training set and all the faults in benchmark B as a test set where A, B ∈ {CoREBench, SIR, Defects4J} and A ≠ B.

The experiment results show that the ranking model learned by PRINCE using one project can be effective in precisely localizing the faults in other projects. For example, PRINCE using Defects4J (D) to CoREBench (C) and C to D only increases the expense by 0.7% and 0.9%, respectively. Using the faults in SIR as a training set decreases the FL precision of PRINCE compared to the 10-fold cross validation (i.e., SIR (S) to C and S to D increase expense of PRINCE 1.8 and 3.0 times compared to 10-fold cross validation). This is because the faults in SIR are artificial ones, which have different characteristics from the real-world faults in CoREBench and Defects4J (e.g., the average LoC of a patch for SIR faults is only 1.6 lines while that for CoREBench and Defects4J faults are 7.0 and 6.7 lines, respectively, as shown in the third column of Table 11). Also, a ranking model obtained from SIR does not utilize file features much since five of seven SIR benchmark programs have only one file.

[16] File feature group is meaningless for SIR benchmarks, since most of the SIR benchmark programs have only one source file

[17] We calculate correlation coefficient in a following way. We assign an average feature value and a number of faults to a source code file over the different faulty versions.

Table 12. Expense Metric of PRINCE Using 10-fold Cross Validation and Inter-benchmark Training and Validation (C=CoREBench, D=Defects4J, and S=SIR)

Benchmark	Target programs	Expense (%) of PRINCE						
		10-fold cross validation	S to C	D to C	C to S	D to S	C to D	S to D
C	coreutils	3.3	6.5	3.7				
	find	2.8	4.2	3.9				
	grep	3.1	6.4	3.4				
	make	3.4	6.0	4.4				
	Average	3.2	5.8	3.9				
S	bash	2.5			3.4	4.3		
	flex	3.2			4.6	6.4		
	grep	0.8			1.2	1.7		
	gzip	0.8			1.3	1.5		
	make	1.3			1.9	2.9		
	sed	0.4			0.7	1.0		
	vim	1.3			2.1	2.5		
	Average	1.6			2.3	3.2		
D	Chart	2.4					3.2	6.5
	Closure	2.4					3.4	7.1
	Lang	3.1					3.4	11.0
	Math	2.5					3.7	6.6
	Mockito	2.2					3.2	5.3
	Time	1.9					2.7	5.8
	Average	2.5					3.4	7.4

6.4 PRINCE in Function-level Fault Localization

Existing work suggests that the statement level FL may not be the most ideal approach. Parnin and Orso reported, from their seminal human evaluation of FL, that developers sometimes preferred higher level overview at function or file granularity, rather than statement level rankings [55]. Subsequently, many techniques have been evaluated at function or method level [5, 81].

Table 13 shows the FL results of Ochiai (the most-precise SBFL technique), Metallaxis (the most-precise MBFL technique), and PRINCE in function level. The first and second columns show the benchmark and target programs, respectively. The third column shows the number of target functions. The 4th to 6th and 7th to 9th columns show the expense and MAP metric of the three FL techniques, respectively. The 10th to 12th, 13th to 15th, and 16th to the last columns show the acc@1, acc@5, and acc@10 of the three FL techniques, respectively.

PRINCE localizes faults with 1.1% of expense and 0.6037 of MAP on average per faulty version(see the sixth column and ninth column of the second last row in Table 13). Also, PRINCE localizes more than half of the target faulty functions (=304 faults = 66.2% of the target faults) within the top five ranks. In addition, PRINCE localizes 39.7% and 74.7% of the target faulty functions at the top one and ten ranks (see the 12th and the last columns of the second last row in Table 13), respectively. Again, PRINCE outperforms Ochiai and Metallaxis in expense, MAP, and acc@n as it did in statement-level fault localization in Section 5.2. The results suggest that findings in this article can be generalized to other granularity levels.

Table 13. Expense Metric, MAP, acc@1, acc@5, and acc@10 of Ochiai (Best of SBFL), Metallaxis (Best of MBFL), and PRINCE in Function Level

Benchmark	Target Programs	#target funcs.	Expense (%)			MAP			acc@1			acc@5			acc@10		
			O	M	P	O	M	P	O	M	P	O	M	P	O	M	P
CoRE-Bench	coreutils	40.6	12.2	7.0	3.3	0.0740	0.1548	0.3030	2	4	9	5	7	12	7	9	15
	find	181.2	11.4	2.3	0.7	0.0951	0.3991	0.6825	1	3	10	3	6	13	6	8	13
	grep	122.4	9.4	2.4	1.6	0.1016	0.4630	0.5985	2	2	11	4	4	13	6	8	14
	make	141.4	13.2	2.6	1.0	0.0713	0.4038	0.6317	1	2	10	2	5	11	3	9	14
	Average	116.4	11.7	3.8	1.7	0.0837	0.3428	0.5394									
	Total								6	11	40	14	22	49	22	34	56
									9.2%	16.9%	61.5%	21.5%	33.8%	75.4%	33.8%	52.3%	86.2%
SIR	bash	910.5	6.9	4.4	0.5	0.1618	0.3291	0.6546	0	0	1	0	1	2	1	2	4
	flex	127.9	8.4	5.1	1.2	0.1322	0.1990	0.5397	0	4	8	2	5	10	3	8	16
	grep	113.5	4.2	4.8	1.3	0.2824	0.3502	0.5160	1	9	10	4	10	11	7	13	14
	gzip	57.4	5.8	5.0	2.1	0.2046	0.2139	0.4783	1	7	13	4	9	14	6	12	16
	make	488.4	5.6	7.3	0.5	0.1918	0.1108	0.6612	0	1	4	1	2	6	2	5	12
	sed	60.6	10.3	4.3	1.8	0.0789	0.2206	0.5666	1	2	3	2	3	6	2	3	3
	vim	1469.2	3.7	5.6	0.3	0.2939	0.2287	0.6806	0	0	0	0	0	1	1	1	2
	Average	294.3	6.2	5.5	1.2	0.1977	0.2254	0.5646									
	Total								3	23	39	13	30	50	22	44	67
									3.6%	27.4%	46.4%	15.5%	35.7%	59.5%	26.2%	52.4%	79.8%
Defects4J	Chart	713.4	2.6	4.2	1.1	0.2238	0.1438	0.6364	3	3	6	8	7	13	10	10	15
	Closure	1326.7	4.1	3.8	1.2	0.1443	0.1993	0.5833	5	4	8	17	16	45	25	22	52
	Lang	413.1	2.6	3.4	0.8	0.3097	0.1411	0.6725	24	23	28	38	40	52	48	49	53
	Math	377.3	2.4	3.7	0.9	0.3393	0.1131	0.6444	29	24	51	67	41	81	74	58	82
	Mockito	567.1	2.1	3.4	0.8	0.4348	0.1382	0.6525	3	3	7	7	8	14	10	10	14
	Time	261.4	3.5	2.9	0.7	0.1778	0.2403	0.6435	3	3	3	8	8	14	11	10	15
	Average	754.6	3.1	3.7	1.0	0.2583	0.1591	0.6278									
	Total								67	60	103	145	120	219	178	159	231
									21.6%	19.4%	33.2%	46.8%	38.7%	70.6%	57.4%	51.3%	74.5%
Average		580.0	4.9	4.0	1.1	0.2225	0.1973	0.6037									
Total									76	94	182	172	172	318	222	237	354
									16.6%	20.5%	39.7%	37.5%	37.5%	69.3%	48.4%	51.6%	77.1%

O, M, and P denote Ochiai, Metallaxis, and PRINCE, respectively.

7 RELATED WORK

7.1 Fault Localization Techniques

7.1.1 Learn-to-Rank Fault Localization. Learning a ranking model from multiple sources of fault localization has been proposed recently. Xuan and Monperrus combined scores from different SBFL formulas using linear weights [81]. Le et al. used rankSVM to learn ranking models from multiple SBFL scores as well as the invariant violation features [5]. Finally, Sohn and Yoo [68] used rankSVM as well as GP to learn ranking models from SBFL scores and program change metrics. In this article, we propose a new set of features on a target program including features of MBFL and the source code files which significantly improve precision of fault localization.

7.1.2 Spectrum-based Fault Localization. Spectrum-based fault localization is widely studied [74]. SBFL formulas have been designed and proposed, both manually [2, 27, 28, 31, 51, 75] and

automatically using genetic programming [82]. It also has been extended in various ways by considering inputs other than structural coverage including call sequence [13], dataflow analysis [65], and specification [21]. One strength of SBFL is its relatively low cost compared to techniques that uses program state analysis [11, 84] or machine learning [5, 68]. SBFL does not require any input other than structural coverage, which tends to be collected as a measure of test adequacy regardless of localization purpose.

There are well-known limitations and critiques to SBFL. Steinmann et al. [69] point out that the inherent limitations in block structure prevents statement level SBFL from achieving precision over certain level. PRINCE avoids this problem by using the MBFL features that use statement level mutation analysis and the statement features, which captures different characteristic of each statement in the same basic block. There are mixed claims about the practicality of SBFL in the literature. Parnin and Orso presented a human study that showed that SBFL does not improve developer productivity in practice [55], and called for a more realistic evaluation metric such as absolute ranking. However, Xia et al. recently reported that using even a mediocre localization tool (i.e., those that can rank the faulty statement within the top 10 places) can improve the developer productivity [76]. We report both the traditional expense metric and the absolute ranking: PRINCE can rank 182 of 459 (39.7%) target faults within the top five places and 243 of 459 (52.9%) target faults within the top 10 places. Also, several research work have been proposed to increase the precision of existing SBFL techniques by generating effective test cases for SBFL [44, 59] These techniques are orthogonal to PRINCE and can be used to improve the effectiveness of SBFL features of PRINCE to precisely localize faults.

7.1.3 Mutation-based Fault Localization. Moon et al. [49] proposed a mutation-based fault localization technique MUSE. MUSE focuses on the difference introduced by the mutation using the conjectures: (a) the failing tests are more likely to pass on the mutants generated by mutating the faulty statement than the mutants generated by mutating the correct statement, (b) the passing tests are more likely to fail on the mutants generated by mutating the correct statement than the mutants generated by mutating the faulty statement. Papadakis and Le-Traon proposed Metallaxis, whichs use mutation analysis for fault localization [53, 54]. Metallaxis depends on the similarity between mutants in an attempt to detect unknown faults: Variations of existing SBFL formulas were used to identify suspicious mutants. Gong et al. [20] extends Metallaxis to improve mutant execution speed without loss of precision. Zhang et al. [85], however, use mutation analysis to identify a fault-inducing commit from a series of commits to a source code repository: Their intuition is that mutating the statement updated by a faulty commit is likely to generate test results similar to those of the faulty commit. Both of Metallaxis and Zhang et al.'s work are based on similarity between the code mutation and the fault. Those MBFL techniques can overcome the limitation in block structure of SBFL and more precisely locate a fault than the SBFL techniques. However, the MBFL techniques require a large amount of time, because the MBFL techniques need to execute a large amount of various mutants to precisely locate a fault. In contrast, PRINCE can locate a fault more precisely and faster than Metallaxis and MUSE by using the SBFL features, static code features, and only 20% of the generated mutants altogether. Pearson et al. [57] compared the precision of the MBFL techniques (Metallaxis and MUSE) with the SBFL techniques on real-world Java program faults in Defects4J. However, the comparison is limited, because Pearson et al. used a limited number of mutation operators, which can significantly decrease the precision of MBFL techniques. We demonstrated that using more mutation operators can improve the precision of MBFL techniques (Section 5.2).

7.1.4 Fault Localization by Altering Program States. There exists several other fault localization techniques that do not use ranking metrics. Cleve and Zeller [11] use delta debugging to search

for a program state that can cause a failed execution by replacing the state of a passing execution with that of a failing execution. Zhang et al. [86] change outcomes of branch predicates of a failing execution to find suspicious branch predicates. If the changed outcomes make the failing execution pass, then the corresponding branch predicate is considered as the suspiciousness one. In a similar way, Jeffrey et al. [29] changed the value of a program variable in a failing execution to the values in other passing executions. Chandra et al. [8] use symbolic execution to find such value changes that make a failing execution pass. All those techniques depend on the state change by mutating the value of variables and the outcome of predicates, which can be mimicked by mutation in PRINCE's MBFL features. In addition to the MBFL features, PRINCE utilizes SBFL and various static features to increase precision of fault localization.

7.2 Source Code Features Used by Bug Prediction Techniques

Various source code metrics to measure code complexity are used by bug prediction techniques as features to learn a bug prediction model. Widely adopted source code metrics for bug prediction include source code size [4], Halstead complexity metrics [24], McCabe complexity metrics [48], and Object-Oriented (OO) complexity metrics [10, 17].

The size metrics measure lines of code, the number of functions, the number of variables, and so on, as the complexity metrics. Halstead [24] proposed several size metrics using the number of operators and operands. McCabe [48] proposed the cyclomatic complexity to measure the complexity of a control flow structure by using the number of nodes and their connections. OO metrics measure the complexity of class structures in object-oriented programming languages. Chidamber and Kemerer [10] proposed metrics to measure complexity of classes using inheritance depth, the number of parent and children classes in an inheritance graph, and coupling and cohesion of classes. Abreu and Carapuça [17] proposed other metrics for complexity of classes using the number of attributes and methods defined in target classes.

We have applied most of the proposed complexity metrics as the static code features for fault localization. PRINCE utilizes some metrics as they are (e.g., size metrics, the cyclomatic complexity, the number of operators and operands) and some metrics after adaptation as a language agnostic form (e.g., the number of attributes and methods defined in classes are modified to the number of variables and functions defined in files). Several metrics cannot be used by PRINCE, because the metrics measure the complexities that exist only in the object-oriented programming language (e.g., the number of parent and children classes in an inheritance graph), which is not possible to be language agnostic.

8 CONCLUSION AND FUTURE WORK

We have presented PRINCE, a new learn-to-rank fault localization technique that utilizes various dynamic and static code features. PRINCE learns ranking models from existing known faults using dynamic features from MBFL and SBFL as well as static code features from a statement, a function, and a file and applies the ranking models to precisely localize unknown faults. The results of empirical evaluation on 65 real-world faults in CoREBench, 84 artificial faults in SIR, and 310 real-world faults in Defects4J show that PRINCE outperforms the state-of-the-art MBFL, SBFL, and learn-to-rank FL techniques significantly and provides a practical fault localization solution. PRINCE is 3.0 and 4.2 times more precise than Metallaxis and Ochiai in terms of expense metric, each of which is the best MBFL and SBFL technique, respectively. Also, PRINCE ranks the faulty statement among the top 5 statements for 182 of 459 target faults.

Future work includes performance optimization of dynamic future extraction from mutation testing as well as developing new features for PRINCE such as identifier-based features [18] and function relation-based features [35] to improve the precision of PRINCE. Also, we will develop a

test suite augmentation technique [78, 80] for effective fault localization by using automated test generation techniques such as a compositional symbolic analysis [37, 67] from unit-level analysis [35, 39, 40], a mutation-based test generation technique [36, 38], a hybrid test generation technique of concolic testing and genetic algorithm [41, 79], and a distributed concolic testing technique [7, 34]. Finally, we plan to evaluate various other learning methods to improve PRINCE.

ACKNOWLEDGMENTS

The authors thank Jeongju Sohn for assistance with FLUCCS and Phan Duy Loc for assistance with MUSIC and Defects4J experiments. The authors also thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] [n.d.]. Docker. Retrieved from https://www.docker.com/.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *The Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[3] Hiralal Agrawal, Richard DeMillo, R. Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J. Martin, Aditya Mathur, and Eugene Spafford. 1989. *Design of Mutant Operators for the C Programming Language*. Technical Report SERC-TR-120-P. Purdue University.

[4] Fumio Akiyama. 1971. An example of software system debugging. In *IFIP Congress (1)*, Vol. 71. 353–359.

[5] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, New York, NY, 177–188.

[6] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, New York, NY, 105–115. DOI: https://doi.org/10.1145/2610384.2628058

[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*. ACM, New York, NY, 183–198. DOI: https://doi.org/10.1145/1966445.1966463

[8] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 121–130. DOI: https://doi.org/10.1145/1985793.1985811

[9] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages. DOI: https://doi.org/10.1145/1961189.1961199

[10] Shyam R. Chidamber and Chris F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.

[11] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 342–351. DOI: https://doi.org/10.1145/1062455.1062522

[12] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Mach. Learn.* 20, 3 (1 Sep 1995), 273–297. DOI: https://doi.org/10.1023/A:1022627411411

[13] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight bug localization with AMPLE. In *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*. ACM, New York, NY, 99–104. DOI: https://doi.org/10.1145/1085130.1085143

[14] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Emp. Softw. Eng.* 17, 4 (1 Aug. 2012), 531–577. https://doi.org/10.1007/s10664-011-9173-9

[15] Cuthbert Daniel. 1973. One-at-a-time plans. *J. Am. Stat. Assoc.* 68, 342 (1973), 353–360.

[16] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. Softw. Eng.* 10, 4 (1 Oct. 2005), 405–435. DOI: https://doi.org/10.1007/s10664-005-3861-2

[17] Fernando Brito e Abreu and Rogério Carapuça. 1994. Candidate metrics for object-oriented software within a taxonomy framework. *J. Syst. Softw.* 26, 1 (1994), 87–96.

[18] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. 2019. Improving source code readability: Theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19)*. IEEE Press, Piscataway, NJ, 2–12. DOI: https://doi.org/10.1109/ICPC.2019.00014

[19] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *J. Mach. Learn. Res.* 9 (Jun. 2008), 1871–1874. http://dl.acm.org/citation.cfm?id=1390681.1442794.

[20] Pei Gong, Ruilian Zhao, and Zheng Li. 2015. Faster mutation-based fault localization with a novel mutation execution strategy. In *Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*. 1–10. DOI: https://doi.org/10.1109/ICSTW.2015.7107448

[21] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2012. Improving the effectiveness of spectra-based fault localization using specifications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. ACM, New York, NY, 40–49. DOI: https://doi.org/10.1145/2351676.2351683

[22] Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *J. Mach. Learn. Res.* 3 (Mar. 2003), 1157–1182. http://dl.acm.org/citation.cfm?id=944919.944968

[23] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY.

[24] Maurice H. Halstead et al. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY.

[25] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongseok Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Inf. Softw. Technol.* 82 (2017), 80–95. DOI: https://doi.org/10.1016/j.infsof.2016.10.002

[26] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE Computer Society, Washington, DC, 464–475. DOI: https://doi.org/10.1109/ASE.2015.14

[27] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull. Soc. vaud. Sci. nat* 37 (1901), 547–579.

[28] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. 2009. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. IEEE Computer Society, Washington, DC, 662–664.

[29] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, 167–178. DOI: https://doi.org/10.1145/1390630.1390652

[30] Yue Jia and Mark Harman. 2008. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Proceedings of the 3rd Testing: Academic and Industrial Conference—Practice and Research Techniques (TAIC PART'08)*. IEEE, 94–98. DOI: https://doi.org/10.1109/TAIC-PART.2008.18

[31] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, New York, NY, 273–282. DOI: https://doi.org/10.1145/1101908.1101949

[32] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. 433–436.

[33] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, New York, NY, 437–440. DOI: https://doi.org/10.1145/2610384.2628055

[34] Moonzoo Kim, Yunho Kim, and Gregg Rothermel. 2012. A scalable distributed concolic testing approach: An empirical evaluation. In *Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation*. 340–349. DOI: https://doi.org/10.1109/ICST.2012.114

[35] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise concolic unit testing of C programs using extended units and symbolic alarm filtering. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 315–326. DOI: https://doi.org/10.1145/3180155.3180253

[36] Yunho Kim and Shin Hong. 2019. DeMiner: Test generation for high test coverage through mutant exploration. (submitted).

[37] Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-driven compositional concolic testing with function summary refinement for effective bug detection. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*.

[38] Yunho Kim, Shin Hong, Bongseok Ko, Duy Loc Phan, and Moonzoo Kim. 2018. Invasive software testing: Mutating target programs to diversify test exploration for high test coverage. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST'18)*. 239–249. DOI: https://doi.org/10.1109/ICST.2018.00032

[39] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. 2013. Automated unit testing of large industrial embedded software using concolic testing. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, 519–528. DOI: https://doi.org/10.1109/ASE.2013.6693109

[40] Yunho Kim, Dongju Lee, Junki Baek, and Moonzoo Kim. 2019. Concolic testing for high test coverage and reduced human effort in automotive industry. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'10)*. IEEE Press, Piscataway, NJ, 151–160. DOI: https://doi.org/10.1109/ICSE-SEIP.2019.00024

[41] Yunho Kim, Zihong Xu, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. 2014. Hybrid directed test suite augmentation: An interleaving framework. In *Proceedings of the 2014 IEEE 7th International Conference on Software Testing, Verification and Validation*. 263–272. DOI: https://doi.org/10.1109/ICST.2014.39

[42] Ching-Pei Lee and Chih-Jen Lin. 2014. Large-scale linear Ranksvm. *Neur. Comput.* 26, 4 (Apr. 2014), 781–817.

[43] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P. Trevino, Jiliang Tang, and Huan Liu. 2017. Feature selection: A data perspective. *ACM Comput. Surv.* 50, 6, Article 94 (Dec. 2017), 45 pages. DOI: https://doi.org/10.1145/3136625

[44] Bing Liu, Shiva Nejati, Lucia, and Lionel C. Briand. 2019. Effective fault localization of automotive Simulink models: Achieving the trade-off between test oracle effort and fault localization accuracy. *Emp. Softw. Eng.* 24, 1 (1 Feb. 2019), 444–490. DOI: https://doi.org/10.1007/s10664-018-9611-z

[45] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Auri Marcelo Rizzo Vincenzi, and Paulo Cesar Masiero. 2001. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation Testing for the New Century*, W. Eric Wong (Ed.). Springer US, Boston, MA, 113–116. DOI: https://doi.org/10.1007/978-1-4757-5939-6_19

[46] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY.

[47] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance 2004*. 350–359. DOI: https://doi.org/10.1109/ICSM.2004.1357820

[48] Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering* 4 (1976), 308–320.

[49] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST'14)*. IEEE Computer Society, Washington, DC, 153–162. DOI: https://doi.org/10.1109/ICST.2014.28

[50] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (Aug. 2011), 32 pages. DOI: https://doi.org/10.1145/2000791.2000795

[51] Akira Ochiai. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish.* 22, 9 (1957), 526–530.

[52] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 1181–1192. DOI: https://doi.org/10.1145/2884781.2884845

[53] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate "Unknown" faults. In *Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation*. 691–700. DOI: https://doi.org/10.1109/ICST.2012.159

[54] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Softw. Test., Verif. Reliab.* 25, 5–7 (2015), 605–628. DOI: https://doi.org/10.1002/stvr.1509

[55] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, New York, NY, 199–209. DOI: https://doi.org/10.1145/2001420.2001445

[56] Karl Pearson. 1895. Note on regression and inheritance in the case of two parents. *Proc. Roy. Soc. Lond.* 58, 347–352 (1895), 240–242.

[57] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, Piscataway, NJ, 609–620. DOI: https://doi.org/10.1109/ICSE.2017.62

[58] Alexandre Perez, Rui Abreu, and Marcelo D'Amorim. 2017. Prevalence of single-fault fixes and its impact on fault localization. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*. 12–22. DOI: https://doi.org/10.1109/ICST.2017.9

[59] Alexandre Perez, Rui Abreu, and Arie van Deursen. 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, Piscataway, NJ, 654–664. DOI: https://doi.org/10.1109/ICSE.2017.66

[60] Loc Duy Phan, Yunho Kim, and Moonzoo Kim. 2018. MUSIC: Mutation analysis tool with high configurability and extensibility. In *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'18)*. 40–46. DOI: https://doi.org/10.1109/ICSTW.2018.00026

[61] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. 2013. Software fault prediction metrics: A systematic literature review. *Inf. Softw. Technol.* 55, 8 (2013), 1397–1418. DOI: https://doi.org/10.1016/j.infsof.2013.02.009

[62] Santosh S. Rathore and Sandeep Kumar. 2019. A study on software fault prediction techniques. *Artif. Intell. Rev.* 51, 2 (01 Feb. 2019), 255–327. DOI: https://doi.org/10.1007/s10462-017-9563-5

[63] Manos Renieris and Steven P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*. IEEE Press, Piscataway, NJ, 30–39. DOI: https://doi.org/10.1109/ASE.2003.1240292

[64] Paul F. Russell and T. Ramachandra Rao. 1940. On habitat and association of species of anopheline larvae in South-eastern Madras. *J. Malar. Inst. India* 3, 1 (1940), 153–178.

[65] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Washington, DC, 56–66. DOI: https://doi.org/10.1109/ICSE.2009.5070508

[66] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *J. Syst. Softw.* 138 (2018), 158–173. DOI: https://doi.org/10.1016/j.jss.2017.12.034

[67] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. 2012. Alternate and learn: Finding witnesses without looking all over. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, Berlin, 599–615.

[68] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*. ACM, New York, NY, 273–283. DOI: https://doi.org/10.1145/3092703.3092717

[69] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, New York, NY, 314–324. DOI: https://doi.org/10.1145/2483760.2483767

[70] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, New York, NY, 727–738. DOI: https://doi.org/10.1145/2950290.2950295

[71] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *Int. J. Man-Mach. Stud.* 23, 5 (1985), 459–494. DOI: https://doi.org/10.1016/S0020-7373(85)80054-7

[72] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometr. Bull.* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[73] W. Eric Wong, Vidroha Debroy, and Ruizhi Gao Yihao Li. 2014. The DStar method for effective software fault localization. *IEEE Trans. Reliabil.* 63, 1 (Mar. 2014), 290–308. DOI: https://doi.org/10.1109/TR.2013.2285319

[74] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (Aug. 2016), 707.

[75] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference, Vol. 01 (COMPSAC'07)*. IEEE Computer Society, Washington, DC, 449–456. DOI: https://doi.org/10.1109/COMPSAC.2007.109

[76] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. 267–278. DOI: https://doi.org/10.1109/ICSME.2016.67

[77] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 31 (Oct. 2013), 40 pages. DOI: https://doi.org/10.1145/2522920.2522924

[78] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. 2015. Directed test suite augmentation: An empirical investigation. *Softw. Test. Verif. Reliabil.* 25, 2 (2015), 77–114. DOI: https://doi.org/10.1002/stvr.1562 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1562

[79] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. 2011. A hybrid directed test suite augmentation technique. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE'11)*. IEEE Computer Society, Washington, DC, 150–159. DOI: https://doi.org/10.1109/ISSRE.2011.21

[80] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. 2010. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 257–266. DOI: https://doi.org/10.1145/1882291.1882330

[81] Jifeng Xuan and M. Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 191–200. DOI: https://doi.org/10.1109/ICSME.2014.41

[82] Shin Yoo. 2012. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering (SSBSE'12)*. Springer-Verlag, Berlin, 244–258. DOI: https://doi.org/10.1007/978-3-642-33119-0_18

[83] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 1, Article 4 (Jun. 2017), 30 pages. DOI: https://doi.org/10.1145/3078840

[84] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE-10)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/587051.587053

[85] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, New York, NY, 765–784. DOI: https://doi.org/10.1145/2509136.2509551

[86] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 272–281. DOI: https://doi.org/10.1145/1134285.1134324

Received July 2018; revised July 2019; accepted July 2019