Effective Isolation of Fault-Correlated Variables via Statistical and Mutation Analysis

Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen, Yibiao Yang, and Hai Jin, *Fellow, IEEE*,

Abstract—It is a widely-adopted strategy for developers to monitor the values of program variables when debugging in practice. In particular, developers often set breakpoints at specific locations or execute the program step by step in the debugging mode to inspect if abnormal values or statuses will be observed for concerned variables. Such a practical debugging strategy can facilitate developers in understanding and localizing the target fault. This study aims to identify suspicious program variables of a given fault (i.e., denoted as *fault-correlated variables*) automatically, thus facilitating the debugging activities for developers. To the best of our knowledge, this is the finest granularity in fault localization (FL) so far, which can address the limitations of being coarse-grained as faced by existing FL techniques. However, isolating fault-correlated variables precisely is challenging since there are usually substantially different variables used or defined in a program, and plenty of them are in the same basic block which cannot be well discriminated from each other since they will be either executed or not against the given test suite. To address such challenges, this study presents IsoVar, a two-phase model to isolate fault-correlated variables. Specifically, IsoVar first performs statistical analysis based on *variable execution matrices*, which is a novel concept proposed in this study, to identify a set of suspicious variables. It then observes the impacts of those variables on the program dynamically after applying subtle mutations at the bytecode level, to further isolate fault-correlated variables. Extensive experiments on Defects4J and Bears demonstrate that IsoVar can outperform state-of-the-art techniques significantly (13.0% for MAP and 19.3% for MRR). More importantly, we incorporated IsoVar into 11 existing FL techniques as well as 14 automated program repair techniques, and found that IsoVar can significantly boost their performance.

Index Terms—fault localization, program variables, debugging

1 Introduction

Modern software systems are inevitably shipped and launched with bugs (also known as faults or defects [1]), which might cause disastrous consequences. What is even worse is that *there can be far more software bugs in the world than we will likely ever know about* [2]. Therefore, debugging is widely adopted in practice aiming to expose and repair as many bugs as possible while it is always a time-consuming and labor-intensive task. Specifically, it has been reported that debugging software bugs can cost nearly 50% of the total budget of software development [3]. To facilitate developments in debugging, a wide range of automated techniques, such as *fault localization* (FL) [4], [5], [6], [7], [8], [9], [10] and *automated program repair* (APR) [11], [12], [13], [14], [15], [16], [17] have been proposed. The first and most pivotal step is to locate the buggy code entities, and the effectiveness of which can significantly affect the performance of other debugging activities [18], [19], [20].

Existing FL techniques can differentiate from each other in the granularity of the buggy code elements being located. Specifically, buggy code entities are usually identified at the source file level [4], method level [5], [6] or the statement level [7]. However, it has been pointed out that even identifying relevant statements is often insufficient to help developers fix bugs [21]. Locating buggy code elements in isolation can be of little usefulness in practice since developers still need assistance to identify and understand the root cause of buggy behaviors [21]. In particular, the contextual information [4] which can explain *why* the identified code elements are buggy [22] is much desired. Therefore, researchers have recently integrated other sources of information, such as software changes and bug-inducing commits in FL [4], [7], [23]; or adapted other methods to enhance FL's effectiveness and interpretability [22].

Specifically, Locus was recently proposed to locate software changes by mining software version histories with the aim to provide more contextual information for fault localization [4]. HSFL was proposed to enhance existing FL techniques via identifying the bug-inducing commits in the first place [7]. However, these two techniques rely on the complete version histories of a project, which might not always be available. Moreover, they provide the results at the statement or commit level, which is still not fine-grained enough as revealed by an existing study [23]. Program state-based techniques have also attracted considerable research attention over the years [24], which locate bugs via inspecting the variables and their values at a particular point during program execution. For instance, Zeller et al. proposed to identify variables and values of a program state that are relevant for a failure via isolating cause-effect chains based on delta debugging [25]. Gupta et al. proposed

- *Ming Wen, Zifan Xie, Kaixuan Luo, Xiao Chen are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail:mwenaa,xzff,u201814820, u201814850@hust.edu.cn. (Corresponding author: Ming Wen).*
- *Yibiao Yang is with the National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210023, China. E-mail:yangyibiao@nju.edu.cn.*
- *Hai Jin is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail:hjin@hust.edu.cn.*

to locate faulty code utilizing failure-inducing chops via performing forward and backward program slicing based on the minimized failure-inducing input [26]. However, these techniques heavily rely on the *delta debugging* technique [27], which is time-consuming. Delta debugging aims to identify the minimized inputs that can trigger the failure. However, thousands of states may exist in the program execution, and thus the cost is exceptionally high [24]. Such extensive overhead limits their practical usefulness to large-scale open-source projects. Moreover, the idea of simplifying failure-inducing inputs is orthogonal to conventional fault localization techniques [26], in which the identified locations may not be the locations where the bugs reside as pointed out by Gupta et al. [26]. Consequently, the results cannot be directly integrated into other debugging techniques such as automated program repair.

Driven by the limitations of existing techniques, we are motivated to revisit the debugging practices of developers and inspect what useful information they are looking for during debugging. Actually, in practice, it is widely adopted to monitor the values of program variables (e.g., primitive types or objects) when debugging with the aim to locate and understand buggy behaviors. In particular, developers often set breakpoints at specific locations or execute the program step by step in the debugging mode, and then monitor those concerned program variables. If unexpected values or status are observed for certain variables, it is likely that the fault resides at those statements containing those variables with abnormal values or status. The following shows two quotes from the discussions of bug reports tracked for large-scale open-source Apache projects. Developers are trying to debug and understand the faults via observing the abnormal values of concerned variables (i.e., `buffer` and `gTranscoder` for the two bug reports respectively).

"The NullPointerExceptions is seemingly caused by the ***variable*** *'buffer' being nulled too early."* [1]

"So gTranscoder is getting reset or overwritten at some point. Try setting a breakpoint on the ***variable*** *to see when it changed."*[2]

Such evidence reveals the importance and usefulness of identifying the variables that are correlated with (or even directly induced) the target fault for debugging in practice. We denote such variables as the ***fault-correlated variables*** in this study. Actually, it has already been pointed out that fault-correlated variables are crucial for developers to understand the fault, and also play a significant role in APR techniques [11], [23]. Equipped with such knowledge, developers can first check those variables with high suspicious values to investigate whether their values or status are expected during the debugging process, thus quickly identifying the root cause and implementing a patch subsequently. Therefore, we are inspired to perform fault localization to identify fault-correlated variables, which is the finest granularity in FL by far to our best knowledge.

However, locating fault-correlated variables precisely is challenging. First, there are usually substantially different variables *defined* or *used* in a program, and thus identifying suspicious ones among them is non-trivial. Second, variables in the same block can hardly be well discriminated. This is because code elements in one block are always executed by the same set of tests, and thus their suspicious values computed by conventional spectrum-based techniques are the same. Consequently, fault-correlated variables are often ranked in tie with other irrelevant ones. As a result, existing attempts which locate suspicious variables based on conventional spectrum-based techniques (e.g., VFL [23]) can hardly discriminate such cases. In addition, approaches which aim to generate more tests or minimize tests, as adopted by *causal testing* [22] and *delta debugging* [27], cannot handle such cases effectively neither (see Section 2 for more details).

To overcome the aforementioned challenges, we propose IsoVar, a novel approach aimed at effectively isolating fault-correlated variables. IsoVar first identifies a set of suspicious variables via statistical analysis. Specifically, it implements the concept of *variable execution matrix*, which tracks the execution traces among different basic blocks for each variable *w.r.t.* (i.e., *with respect to*) different executions. In contrast to the spectrum constructed by traditional spectrum-based FL [28], [29], the variable execution matrix is novel since it encodes the execution information from a sequence of basic blocks that contains either definitions or usages for each target variable. IsoVar then identifies a set of suspicious variables via statistically analyzing the execution matrix constructed for different variables. Program variables, even those in the same block, can therefore be better differentiated since a variable is usually involved in multiple basic blocks. Afterwards, IsoVar performs mutation analysis to further isolate fault-correlated variables. Specifically, inspired by real debugging practices, IsoVar tries to monitor the behavior changes of the program after slightly mutating those suspicious variables at the bytecode level. Different from causal testing [22], instead of mutating test inputs, IsoVar directly modifies suspicious variables and then executes tests to observe causal relationships and isolate fault-correlated variables. The intuition behind this mutation analysis is that *mutating the correlated variables will cast higher impact on failing executions while with less impact on passing ones, and vice versa for those variables that are not correlated.* Under such an intuition, IsoVar is able to isolate fault-correlated variables precisely in the second phase.

To evaluate the effectiveness of IsoVar, we applied it to real software bugs from the Defects4J [30] and Bears benchmark [31], which is widely adopted and evaluated in FL and APR. The results show that IsoVar can significantly outperform existing baselines with respect to different metrics. Specifically, the MAP and MRR can be improved by 13.0% and 19.3% on average, respectively. Moreover, IsoVar can rank the fault-correlated variables at Top-1 for 93 cases while such a number is only 68 for the existing baseline. With respect to Top-5 and Top-10, IsoVar can rank the fault-correlated variables for 198 and 243 cases while the performance achieved by the baseline is 171 and 229 respectively. To further demonstrate the usefulness of IsoVar, we performed experiments to integrate the results generated by IsoVar with 11 existing FL techniques at different granularities as well as 14 APR techniques to rank the generated patches. The evaluation results are promising. In particular, the performance of existing FL techniques can be significantly boosted (i.e., the improvements of MAP range

1. https://issues.apache.org/jira/browse/DERBY-1727
2. https://issues.apache.org/jira/browse/XERCESC-1222

```
462 // src/main/java/org/joda/time/Partial.java
463    System.arraycopy(iValues, i,newV, i+1, newV.length-i-1);
464 -  Partial newP = new Partial(iChronology, newTypes, newV);
465 +  //use public constructor to ensure full validation
466 +  Partial newP = new Partial(newTypes, newV, iChronology);
467    iChronology.validate(newP, newV);
468    return newP;
469
470 // failing test case
471    public void testWith3() {
472       Partial test = createHourMinPartial();
473       try {
474         test.with(DateTimeFieldType.clockhourOfDay(), 6);
475         fail();
476       } catch (IllegalArgumentException ex) {}
477       check(test, 10, 20);
478    }
```

Listing 1: A Motivating Example of Time 4

from 16.7% to 218.8% and the improvements of MRR range from 18.4% to 199.4%). For automated program repair, the rank of corrected patches can be significantly improved, and the precision of existing techniques can be improved from 69.6% to 79.7%.

To sum up, this paper makes the following contributions:

- **Originality:** This study attempts to locate buggy code elements at the *variable level*, which generates the results at the finest granularity in FL so far. The novelty of this study lies in proposing a new concept of variable execution matrix and a set of new mutation operators, specific to different variable types.
- **Approach:** We proposed IsoVar, which combines statistical and mutation analyses to isolate fault-correlated variables. Specifically, IsoVar first identifies a set of suspicious variables based on variable execution matrices, and then performs mutation analysis at the bytecode level to isolate fault-correlated variables.
- **Evaluation:** We implemented IsoVar as an open-source tool, and performed extensive experiments to demonstrate its effectiveness (e.g., IsoVar can improve MRR by 19.3% on average). The tool and all experimental data are available at: **https://github.com/justinwm/IsoVar**.
- **Application:** The results generated by IsoVar at the variable level can be incorporated into many applications (i.e., *automated program repair*) and enhance their results significantly, which reflects its usefulness.

2 Motivation and Background

2.1 A Motivating Example

Listing 1 shows a motivating example from the Defects4J benchmark [30]. If the original program works normally, it should throw `IllegalArgumentException` as expected. However, the program does not throw such an exception but fail at line 475. In this bug (Time 4), the developer has used the wrong constructor to create a `Partial` object since the original constructor does not check the value of the variables. Therefore, the constructed object `newP`, including its parameters `iChronology`, `newTypes` and `newV` are highly correlated with the bug. Both of the buggy and correct constructors are syntactically valid, and thus such a problem was imperceptible to developers until abnormal behaviors were witnessed during dynamic execution. We denote such variables involved in such constructors as the fault-correlated variables *w.r.t.* this issue. When using traditional FL techniques, such as Ochiai [29], to debug, the buggy statement (i.e., line 464 in `org/joda/time/Partial.java`) is ranked at a quite low position, which is 30, as shown in Table 1. Such a result can be of little use to developers in practice. Recently, more advanced FL techniques cannot enhance the results significantly. For instance, MCBFL [8] can only rank statement 464 at position 20 while HSFL [7] can only rank the buggy statement at position 33.

However, if we debug from another perspective, which is to inspect which variables are suspicious for introducing this fault, the results could be more useful. Delta debugging was proposed to identify suspicious variables that are relevant for a failure via isolating cause-effect chains [27]. Given several failing and passing test cases that are related to a bug, it compares the state of variables (i.e., variable value) via memory graphs (see [32] for more details). It suspends the execution of the passing test, replaces the value of a variable of the passing tests with the corresponding value from the identical point in the failing test, and then resumes the execution of the passing tests. Unless identical failure is observed, the variable is no longer considered as suspicious. As a result, delta debugging requires extensive additional test executions to compare the variable states at different positions. In our example, the failing test covers 1,671 lines of code and it does not cause the program to crash, thus making the debugging tool being unaware of the specific buggy position. Delta debugging would theoretically compare program states and repeat the above procedure at each position of the failing execution path. In addition, the execution of the program involves plenty of complex loop structures so that thousands of states may exist during program executions, making it particularly costly to finish the whole procedure. Gupta et al [26] proposes to solve this problem by introducing the concept of a failure-inducing chop. Specifically, it targets at identifying which part of the inputs in the failing tests are related to the original failure, thus minimizing the failing tests. However, in our example, all the inputs (i.e., `test`, `DateTimeFieldType.clockhourOfDay()` and the "6" at line 474) are related to the failure, and thus the failing test cannot be further minimized. Therefore, the overhead for delta debugging cannot be reduced for this example. Driven by the above limitations, we are seeking a more lightweight approach to identify suspicious variables.

In this paper, we propose IsoVar to isolate fault-correlated variables efficiently and precisely. Table 1 shows the results generated by IsoVar. As we can see, IsoVar can rank the concerned variable `iChronology` at the second place and other concerned variables at high positions. Such precise results offered by IsoVar can bring the following two benefits. First, it can provide developers with rich contextual information to debug. Since we have directly pointed out that variables `newP` and `iChronology` are significantly correlated with this fault, developers can quickly know that the usages toward these variables are buggy, thus determining where the problem is and how to fix it. Second, such results can boost the performance of existing debugging techniques in reverse. Since IsoVar has pointed out that the usages of variables `newP`, `iChronology`, and `newV` are correlated with

Table 1: The Fault Localization Results Obtained at the Statement Level and Variable Level

Ochiai				IsoVar			
Rank	Class	Line	Suspicious	Rank	Method	Variable Name	Suspicious
1	org.joda.time.field.ZeroIsMaxDateTimeField	111	1.0000	2	Partial with(DateTimeFieldType,int)	this.iChronology	0.6526
2	org.joda.time.field.ZeroIsMaxDateTimeField	138	1.0000	.....			
3	org.joda.time.field.ZeroIsMaxDateTimeField	178	1.0000	10	Partial with(DateTimeFieldType,int)	newV	0.6086
.....				14	Partial with(DateTimeFieldType,int)	newP	0.5941
30	**org.joda.time.Partial**	**464**	**0.2000**	15	Partial with(DateTimeFieldType,int)	newTypes	0.5940

this issue, and they are all either defined or used at line 464. Therefore, line 464 should be more suspicious than the others when performing fault localization *at the statement level*. Similarly for APR, patches generated via modifying those corrected variables should be more likely to be correct than the others. Therefore, we conjecture that the effectiveness of other debugging activities, such as FL and APR, can be potentially enhanced if we are able to locate the buggy variables precisely (see Section 5 for more evaluation).

However, isolating such fault-corrected variables is non-trivial since there are tens of different variables defined or used in the surrounding contexts of this fault. A single variable is usually involved in multiple *basic blocks* at different locations. While such a situation incurs great challenges, it brings new opportunities at the same time. We found that the execution information of a variable at different basic blocks forms a sequence of traces naturally, which can be leveraged to differentiate suspicious variables from the others through *statistical analysis*. Specifically, we can pinpoint those variables that are more frequently involved in failing execution traces while less frequently in passing ones. After applying this intuition based on our originally designed variable execution matrix (see Section 3.1), the four concerned variables (i.e., `ichronology`,`newV`,`newP`,`newTypes`) can be ranked at 2, 13, 14 and 15 respectively (i.e., the original rankings generated by Ochiai are 6, 15, 20 and 21 respectively). Furthermore, we also observe that different variables will cast diverse degrees of impact on the execution of the program. For instance, if we mutate the variable `newV`, an Integer array, it will cast more impact on the failing executions than the passing ones. After capturing this intuition, IsoVar can further improve its ranking from 13 to 10. In contrast, if we mutate other irrelevant variables, opposite results are observed. Consequently, such information can be leveraged to further differentiate fault-correlated variables (e.g., `newV` in this example) from the others. Therefore, it further motivates us to leverage *mutation analysis* in isolating fault-correlated variables.

2.2 Variables in Fixing Patches

Critical variables that induce or are correlated with bugs are essential for developers to debug. We aim to locate such variables in this study while it is a challenging task. Therefore, it motivates us to first understand the characteristics (e.g., their types and distributions) of those bug correlated variables, and the results of which can guide us to better design fault localization tools.

In ordinary *Object-Oriented Programming*, variables can be mainly categorized into several types, including *Primitive*, *String*, *Array*, *Objects* and so on. Note that, in this study, we use the **Java** programming language for illustration while our approach can be easily adapted to other languages. We choose Java since it remains to be one of the most widely adopted languages over the years [33]. Particularly, Objects in Java can be further categorized based on where the associated classes are defined. For instance, they can be project specific objects (i.e., classes defined in the target project), JDK objects (e.g., `HashMap`, `List`), and other objects from Third-Party Libraries.

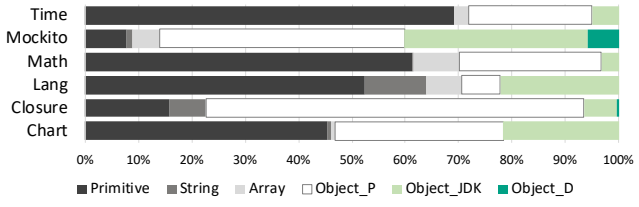
To demonstrate how frequently different types of variables are involved in the fixing patches, we conducted an empirical study based on two datasets. One is Defects4J [30] and the other is collected from large-scale open-source projects (e.g., Apache Flink, Lucene-Solr) by a recent study [34]. It contains over 1,000 fixing patches extracted from 7 open-source projects, and we denote it as the **Large** dataset in this study. First, we observe that many different variables are involved in fixing bugs. Specifically, it requires modifying 5.38 different variables on average with a medium number of 3.00 to fix a Defects4J bug; and 22.7 different ones on average with a medium number of 8.00 to fix a bug in the Large dataset. The number is larger for the Large dataset since its fixing patches are extracted from the commits directly, which are often more complex than those in Defects4J whose patches have been simplified [30].

Moreover, we observe that different types of variables are frequently involved in fixing bugs as shown in Figure 1. Specifically, the results show that variables defined by third-party libraries are rarely used to fix bugs (i.e., accounting for less than 1.0% in Defects4J and on average 7.6% in the Large dataset). On the contrary, four types of variables (i.e., primitive, String, Array, and project specific Objects) account for the majority of variables to repair bugs with an average ratio of 87.3% (varying from 59.9% to 96.8% over different projects) for Defects4J, and 68.2% for the Large dataset (varying from 54.0% to 90.8% over different projects). Such results indicate that different types of variables, especially primitive ones and project-specific objects are pervasively involved in bug fixes, and thus the designed fault localization tool should support all these different types of variables (see Table 3).

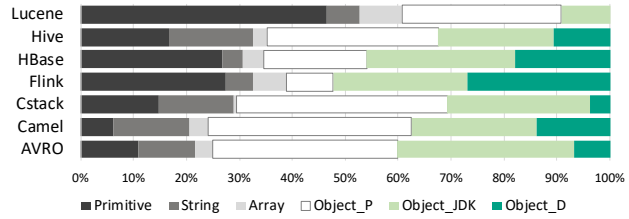
3 Approach

Figure 2 shows the overview of our proposed approach IsoVar, which combines two phases, a *statistical analysis* phase based on variable execution traces and a *mutation analysis* phase via analyzing the impacts of generated mutants. Specifically, given a buggy program with the associated test suite (i.e., the failure triggering ones and passing ones), IsoVar first identifies a set of suspicious variables based on the execution traces of different variables. The basic intuition of this step is that *fault-correlated variables should be more frequently involved in failing executions while less frequently involved in passing ones*.

Inspired by the practical debugging practices, IsoVar then tries to mutate the identified suspicious variables subtly



(a) Distribution of Variable Types in Defects4J



(b) Distribution of Variable Types in the Large Dataset

Figure 1: The Distribution of Variable Types. Object_P denotes project-specific Objects; Object_JDK denotes JDK Objects; while Object_D denotes Objects from dependencies

via applying effective mutation operators at the bytecode level. It observes the impacts of such subtle mutations on program executions and further isolates the fault-correlated variables precisely by comparing the execution traces of the generated mutants and those of the original program. The basic intuition behind is that *the correlated variables should cast more impact on failing executions while less on passing ones*. IsoVar finally isolates fault-correlated variables via integrating the statistical analysis and mutation analysis. We introduce the details of each analysis as follows.

3.1 Statistical Analysis: Identifying Suspicious Variables

IsoVar first aims to identify a set of suspicious variables for the target fault based on the execution traces of both failing and passing tests. Traditional spectrum-based FL techniques construct execution spectra at the statement level. Therefore, the coverage for a statement *w.r.t.* a specific test execution only contains a single value, which represents the number of times that the statement has been executed by the execution.

However, a variable could be involved in multiple statements across different basic blocks with either declarations or usages (i.e., *def* or *use*), and each of these basic blocks may be executed or not by a specific test. Therefore, the spectrum constructed by existing techniques at the statement level cannot fully reflect the execution traces for a single variable. For instance, Table 2 shows an example of the execution traces of a variable, where f_i denotes a failing execution, p_i denotes a passing execution and b_i denotes a basic block that contains either the *def* or *use* of the variable. In this example, the variable is involved in four basic blocks, and the failing execution f_1 has executed all these blocks twice as shown in the Table 2. On the other hand, the passing execution p_1 has only executed basic block b_4 six times. We denote such an execution trace as the ***variable execution matrix*** in this study, which can better reflect the extent to which the variable has been executed for different tests.

However, existing techniques (e.g., Tarantula [28], Ochiai [29]) cannot be directly adopted to such matrices to locate faults at the variable level. This is because each

Table 2: Execution Matrix for a Variable

		V_f		V_p		
		f_1	f_2	p_1	p_2	p_3
V_b	b_1	2	1	0	0	0
	b_2	2	0	0	1	0
	b_3	2	1	0	0	0
	b_4	2	0	6	2	4

variable contains a sequence of information (i.e., the coverage information of a sequence of basic blocks involving the target variable) *w.r.t.* each test execution. In the above example as shown in Table 2, *w.r.t.* failing test f_1, the sequence information is $V_{f_1} = \langle 2, 2, 2, 2 \rangle$ encoded in basic blocks b_1, b_2, b_3, b_4 respectively. Such sequence information in V_f cannot be ignored if we aim to locate fault-correlated variables precisely.

To fully utilize the sequence information encoded in variable execution matrices, we compute the suspicious value for each variable based on the following insights.

1) $Freq_f$*: the variable should be more frequently involved in failing execution traces w.r.t. different basic blocks.*
2) $Freq_p$*: the variable should be less frequently involved in passing execution traces w.r.t. different basic blocks.*
3) $Simi_{\langle f,p \rangle}$*: the execution trace concerning the variable w.r.t. failing tests should be dissimilar with that w.r.t. passing ones.*

To fulfill the first two insights, we investigate the frequency of a variable being executed by failing and passing tests. Different from the traditional spectrum constructed for statements by existing techniques, in which a statement has rather simple status (a single value denoting the number of times being executed), the situation for the variable matrix is more complex since a variable can be involved in multiple blocks as shown in Table 2. Therefore, we compute the frequency of coverage instead of simply inspecting whether the target variable has been executed by a test. Specifically, suppose a variable is involved in n basic blocks, we measure the ratio by counting how many of such basic blocks have been covered *w.r.t.* an execution. Formally, $Freq_{f_i}$ is computed as:

$$Freq_{f_i} = \frac{\sum_{v_i \in V_{f_i}} v_i > 0 \ ? \ 1 : 0}{|V_{f_i}|} \tag{1}$$

If there are multiple failing executions, the overall $Freq_f$ is measured by their average value. Similarly, the frequency of the variable participating in the passing executions can be computed accordingly, and we denote it as $Freq_p$.

To fulfill the third insight, we compute the similarities between the execution traces of failing tests and passing ones. For instance, between V_f and V_p as shown in Table 2. Specifically, $Simi_{\langle f,p \rangle}$ is computed as follows:

$$Simi_{\langle f,p \rangle} = \frac{\sum_{i=1 \to n}^{j=1 \to m} Cosine(V_{f_i}, V_{p_j})}{n * m} \tag{2}$$

where n denotes that there are n failing test executions while m denotes m passing ones. We measure the average similarity since it reflects how similar it is that the variable participates in the executions *w.r.t.* failing tests and passing ones. $V_{f_i} = \langle v_1, ..., v_i, ..., v_n \rangle$ denotes the execution trace *w.r.t.* failing test f_i, and v_i denotes that the ith *basic block* has been executed v_i times by f_i. Accordingly,

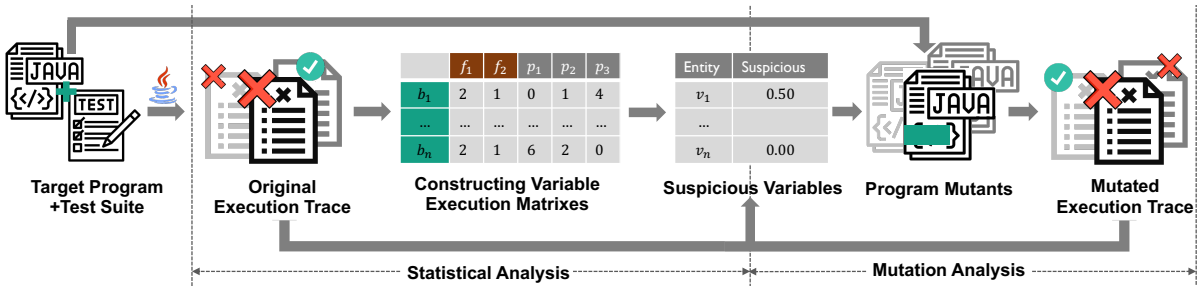


Figure 2: Overview of IsoVar

$V_{p_j} = \langle u_1, ..., u_i, ..., u_n \rangle$ denotes such trace *w.r.t.* passing test p_j. Their cosine similarity is computed as follows:

$$Cosine(V_{f_i}, V_{p_j}) = \frac{\sum_{i=1}^{n} v_i * u_i}{\sqrt{\sum_{i=1}^{n} v_i^2} * \sqrt{\sum_{i=1}^{n} u_i^2}} \quad (3)$$

The higher similarity, the less suspicious that the variable is correlated with the failure.

To integrate the above insights, we first combine $Freq_f$ and $Freq_p$ to investigate whether the variable is more frequently involved in failing executions, and then integrate $Simi_{\langle f,p \rangle}$ via weighted deduction since the larger similarity as measured in Equation 2, the less suspicious of the variable to be correlated. Consequently, the final ranking formula is designed as follows:

$$Susp(v) = \frac{Freq_f}{Freq_f + Freq_p} - \alpha * Simi_{\langle f,p \rangle} \quad (4)$$

in which α is the weight to integrate all these insights, and its impact on IsoVar's performance is investigated in Section 5.

3.2 Mutation Analysis: Isolating Fault-Correlated Variables

The above step might identify many different suspicious variables, and thus pinpointing fault-correlated variables precisely still remains a challenge. Therefore, IsoVar takes a step forward to identify correlated variables via observing the impact of each suspicious variable on the buggy program dynamically, in particular, *w.r.t.* different test executions. The intuition behind is that mutating the correlated variables will cast higher impact on failing executions while having less impact on passing ones, and vice versa for those irrelevant variables. To achieve such a goal, IsoVar first designs a set of mutation operators to apply simple transformation rules that affect a singular ***variable*** at a time. IsoVar then measures the impact of the mutant on both failing and passing tests. It finally isolates fault-correlated variables via comparing the trace similarities before and after applying the mutant. We next present how IsoVar mutates programs *w.r.t.* suspicious variables and how it observes and quantifies the impact of such mutations.

3.2.1 Effective Bytecode Mutation

IsoVar designs and applies mutation operators at the bytecode level for efficiency since it does not require recompiling programs [35]. However, the difficulty of mutating different types of program variables is diverse. In particular, mutating `int` variables is simple and straightforward since we can simply increase or decrease its values via utilizing instructions `iinc` at the bytecode level.

In contrast, mutating Objects is more complex since it is difficult to directly mutate its value. Instead, we need to iteratively mutate its fields. In the current implementation, IsoVar supports the mutation of ***four*** different types of variables, which are primitive, String, Array and project specific Objects. As shown in Figure 1a and Figure 1b, these four types of variables account for the majority of those involved in fixing patches *w.r.t.* different projects. Variables of Objects from JDK and other third-party libraries are not currently supported since mutating such variables requires analyzing the associated libraries, which brings significant extra overheads. Compared to conventional mutation-based FL techniques, the mutation analysis of IsoVar introduces the following enhancements. First, IsoVar designs a set of new mutation operators *w.r.t.* different types of variables. In particular, to mutate Object, it devises a novel algorithm (i.e., Algorithm 1) to iteratively search for mutable fields. Second, the proposed mutation operators are designed at the bytecode level, which further significantly enhances IsoVar's efficiency via avoiding repetitive compiling.

Table 3 presents the details of the operators, originally proposed in this study, with the aim of mutating variables, in rewriting rules. Each mutation operator $\mathcal{M}(e)$, where e is the target expression to be mutated, is associated with a rule, which is represented in the form of $p \vdash e \mapsto e'$. The rule denotes that when the premise p holds, the target program can be mutated via transforming a single instance of expression e to e'. In particular, we use notation $\tau(*)$ to denote the inference of the type information of a given expression or variable. For basic data type, the operators mainly contain two parts. First, we directly modify its value slightly (e.g., negate a `boolean` variable or add a small number to `byte, int`) via inserting extra instructions following the instructions of the target variable (e.g., `iinc` for int). Second, we mutate the operators in the assignment of a target variable (e.g., v=a+b $\rightarrow$ v=a-b if v is the target variable). For String, we concatenate a char randomly generated with the original variable. For a variable of type Array, we mutate one element of it randomly selected using the operators for the specific types. As for variable of Object, we iteratively mutate its fields as shown in Algorithm 1. IsoVar applies those defined mutation operators to suspicious variables identified in the statistical analysis, and then observes the impact of such suspicious variables.

Table 3: Mutation Operators for Different Variables

Name	Rule	Illustration
Basic Data Type $\mathcal{M}_v$	$\tau(v) = boolean \vdash v \rightarrow !v \mid v = a\ op_1\ b \rightarrow v = a\ op_2\ b, op \in \{\&\&, \|, ==, !=, >, <, \geq, \leq\}$	Negate boolean variables directly or change the operators in boolean expressions
	$\tau(v) = byte \mid char \mid short \mid int \vdash v \rightarrow v + c, c \in \{-2^7, 2^7\} \mid v = a\ op_1\ b \rightarrow v = a\ op_2\ b, op \in \{+, -, *, /\}$	Add a random constant number to the target variable from -2^7 to 2^7 or change the operators in expressions of target types.
	$\tau(v) = long \mid float \mid double \vdash v \rightarrow v + c \mid v \rightarrow v / c, c \in \{-2^8, 2^8\} \mid v = a\ op_1\ b \rightarrow v = a\ op_2\ b, op \in \{+, -, *, /\}$	Add or divide a random constant number to the target variable from -2^8 to 2^8 or change the operators in expressions of target types.
String $\mathcal{M}_s$	$\tau(s) = String \vdash s = s \oplus c, c \in \{a-z, A-Z\}$	Append a random character {a-z, A-Z} to the end of the String.
Object $\mathcal{M}_o$	$\tau(o) = Object \vdash o = mutate(o)$	Recursively mutate certain fields of the object based on Algorithm 1.
Array $\mathcal{M}_x$	$\tau(x) = Array \cap \vdash x[i] = \mathcal{M}(x[i]), i \in [0, x.length)$	If x is an array of String or Basic Data Type, IsoVar mutates one element randomly selected using $\mathcal{M}$ as mentioned above.

Algorithm 1: Mutating Project Specific Object

```
   input : o: an input variable
   output: o': a mutated variable
1  Q ← ⟨o, 0⟩ /* Using a queue to record the
       objects to be mutated. The key is the
       object while the value is the depth of the
       object to the target input object.      */
2  while !Q.isEmpty() do
3      entry ← Q.pop(); object ← entry.key
4      F ← RetrieveFields (object)
5      foreach f ∈ F do
6          if τ(f) = basic data type then
7              M_v(object.f); return object
8          end
9          if τ(f) = String then
10             M_s(object.f); return object
11         end
12         if τ(f) = Object && entry.value < 4 then
13             Q.add(⟨f, entry.value + 1⟩)
14         end
15     end
16 end
```

3.2.2 Impact Measurement

To isolate the most correlated variables *w.r.t.* a fault, IsoVar quantitatively measures the impact of variable mutations as follows:

$$fitness(v) = \overline{I_f}(m) - \beta * \overline{I_p}(m), m \in \mathcal{M}(v) \quad (5)$$

where $\bar{I}_f(m)$ denotes the average impact of mutant m concerning variable v *w.r.t.* the original failing tests, $\bar{I}_p(m)$ denotes the corresponding impact *w.r.t.* the original passing tests and β is the weight to integrate the above impacts.

To quantitatively measure the impact of mutant m *w.r.t.* a test case, we compute the trace similarity between the test execution before and after the mutant is applied as follows:

Execution Trace Similarity (S_{et}): Similar to our previous notations, let $V_b = \langle v_1, ..., v_i, ..., v_n \rangle$ denote the execution trace *before* the mutant is applied. Specifically, v_i denotes that the ith *basic block* has been executed by v_i times. Accordingly, $V_a = \langle u_1, ..., u_i, ..., u_n \rangle$ denotes such a trace *after* the mutant is applied. We then use the cosine metric as defined in Equation 3 to measure the similarity between V_b and V_a. A higher similarity indicates that the mutated variable cast less impact on the target execution, and thus the impact is quantified as one minus the cosine similarity. Consequently, following our intuition, a variable whose mutants preserve lower similarities *w.r.t.* failing-revealing tests while preserving higher similarities *w.r.t.* the original passing tests is more likely to be the fault-correlated variables.

IsoVar generates ten different mutants for each suspicious variable unless fewer than ten mutants can be generated for the variable. For instance, for a `boolean` variable, $\mathcal{M}_v$ might be unable to generate ten mutants as shown in Table 3. IsoVar takes the average of the impact measured by $fitness(v)$ over those mutants and denotes it as $\overline{fitness(v)}$. IsoVar then takes γ as the weight to incorporate such an average impact with the original suspicious value as shown in Equation 4 to make the final isolation of all suspicious variables as follows:

$$isolation(v) = Susp(v) + \gamma * \overline{fitness(v)} \quad (6)$$

4 Experimental Setup

4.1 Subjects

We evaluate the effectiveness of IsoVar on the benchmarks of Defects4J [30] and Bears [31]. These benchmarks contain substantial real bugs extracted from large open-source projects, and they were built to facilitate controlled experiments in software debugging and testing research [30]. Defects4J has been widely adopted by recent studies on fault localization and program repair (e.g., [8], [14], [13], [36]). Bears is a benchmark to facilitate research in fault localization and program repair in Java. These bugs were collected from open-source projects hosted on GitHub via scanning the build failures generated during the Travis Continuous Integration. Following existing studies [6], [37], we use all six projects in Defects4j of *version* 2.0.0 with a total of 395 real bugs and three projects in Bears of 125 real bugs as the subjects for our experiments. Following an existing study, we leverage the developer provided fixing patches to approximate a bug's root cause [38], and utilize those variables involved in the fixing patch as the correlated ones. That is, the variables involved in the fixing patches are extracted and served as the fault-correlated variables of the bug for evaluation. The demography of the dataset is shown in Table 4.

Table 4: Subjects for Evaluation

	Subject	#Bugs	#Crashes	KLOC	Test KLOC	Test Cases
Defects4J	Commons **Lang**	65	27	22	6	2,245
	JFree**Chart**	26	11	96	50	2,205
	Commons **Math**	106	33	85	19	3,602
	Joda-**Time**	27	10	28	53	4,130
	Closure Compiler	133	17	147	104	7,929
	Mockito	38	17	23	29	1,366
Bears	**FasterXML**	26	17	56	43	1,711
	INRIA **Spoon**	57	22	40	33	1,114
	Traccar	42	15	37	7	255
	Total	520	169	534	344	24,557

4.2 Measurements

To measure the effectiveness of IsoVar, we adopt the following three widely-used metrics in our study [4], [6], [37].

Top-N: This metric reports the number of bugs, whose buggy entities (i.e., variables in this study) can be identified

by inspecting the top N(N=1,2,3,...) of the returned suspicious list. The higher the value is, the less effort required for developers to locate the bugs, and thus the better performance.

MAP: *Mean Average Precision* (MAP) [39] and *Mean Reciprocal Rank* (MRR) [40] are the most widely adopted metrics to evaluate the effectiveness of FL [7], [4], [37], which were originally proposed to evaluate information retrieval queries. It takes all the buggy code elements into consideration with their ranks when computing the metric. MAP is computed by taking the mean of the average precision scores across all bug reports. The average precision (AP) of fault-correlated variables for one bug report is computed as Equation 7.

$$AP = \frac{\sum_{k=1}^{M} P(k) \times pos(k)}{\#fault\text{-}correlated\ variables} \tag{7}$$

where k is the rank of the returned ranked variables, M is the number of ranked variables and $pos(k)$ indicates whether the k^{th} variable is a fault-related one. $P(k)$ is the precision at a given top k variable and is computed as 8:

$$P(k) = \frac{\#fault\text{-}correlated\ variables\ in\ top\text{-}k\ variables}{k} \tag{8}$$

MRR: MRR measures the multiplicative inverse of the rank of the first buggy code elements identified while MAP takes all the buggy code elements into consideration with their ranks when computing the metric. In particular, MRR is the mean of the reciprocal ranks over a set of bug reports Q and it can be computed by 9.

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{9}$$

where $rank_i$ is the position of the first fault-correlated variable identified in the i^{th} bug report.

When multiple variables have the same suspicious score, we use the average rank to present their final rankings, following the strategy to handle the tie issues widely adopted by existing fault localization techniques [6], [8], [41], [42].

4.3 Research Questions

We investigate the following research questions to evaluate the effectiveness and usefulness of IsoVar.

- **[RQ-1]** *To what extent do parameters affect the overall effectiveness of IsoVar?* In this RQ, we investigate the impact of those parameters involved in the design of IsoVar (e.g., α in Equation 4 to combine the execution frequency and cosine similarities). This RQ focuses on how changes in parameters affect the effect of IsoVar and their generalizability when applied to a new project.
- **[RQ-2]** *How effective is IsoVar in locating fault-correlated variables?* In this RQ, we investigate IsoVar's effectiveness as well as the contributions of statistical analysis and mutation analysis. Specifically, we apply IsoVar on the benchmarks, and then evaluate its performance *w.r.t.* the above-mentioned three evaluation metrics. We also compare IsoVar with the state-of-the-art FL techniques that aim to locate bugs at the variable level. Then, we further dissect IsoVar's performance *w.r.t.* different types of variables aiming to understand whether the performance of IsoVar is significantly different on bugs with different characteristics.
- **[RQ-3]** *How useful is IsoVar?* In this RQ, we explore the usefulness of IsoVar. IsoVar is designed from a novel perspective (i.e., *w.r.t.* program variables) which is different from existing studies, and thus it is able to provide new knowledge and insights for better debugging. Therefore, we investigate whether the results provided by IsoVar can be integrated with existing debugging techniques, such as FL and APR, to enhance their performance.

4.4 Experiment Setting

4.4.1 RQ1

We inspect the impact of different parameters involved in the design of IsoVar and the generalizability of the tool to different benchmarks. We applied IsoVar to Defects4J and Bears, and observed the impact of different parameter values on IsoVar's performance. Specifically, for the parameters α, β and γ, we vary the value of such parameters from 0.0 to 1.0 with a step of 0.1, and then inspect and compare IsoVar's performance. Note that we first investigate the impacts of α via merely considering the part of statistical analysis. We then select the value of α that achieves the optimum performance to investigate β with γ set to 0.5. Finally, we choose the optimum values of both α and β to investigate γ. Finally, we identify the optimal parameters for Defects4J and Bears respectively and then explore and analyze the effect of such parameters on IsoVar with respect to these two different datasets.

4.4.2 RQ2

Baseline: To the best our knowledge, VFL [23] is the only FL technique that aims to locate faults at the variable level. The original VFL is designed based on Ochiai while its concept can be applied to all spectrum-based FL techniques that are designed at the statement level. Therefore, we constructed several variants of VFL based on different techniques (e.g., Tarantula, Barinel and so on), and then compared the performance of IsoVar with VFL and its variants.

We did not include program state-based techniques [24], [25], [26] as the baselines for comparison for the following reasons. First, these techniques heavily rely on the *delta debugging* technique [27], the aim of which is to identify the minimized inputs that can trigger the failure. Second, thousands of states may exist in program execution, and thus the cost is also high [24]. Such extensive overheads limit their practical usefulness to large-scale open-source projects. Third, these approaches are originally designed for C/C++ projects, which is non-trivial to adapt for Java programs. Actually, The authors have provided a delta debugging tool implemented in Java [43], but it has not been maintained for over 16 years. This tool relies on Java 1.5 to execute, which prevents us from applying it to projects using higher Java versions. Therefore, we exclude them from our baselines.

Note that to demonstrate the generalizability of our identified parameters across different projects while avoiding the overfitting problem, we select the optimum parameters obtained on Bears and apply them to Defects4J to investigate IsoVar's performance. Similarly, when evaluating IsoVar's

performance on Bears, we select and use the optimum parameters obtained on the Defects4J dataset.

Different Types of Bugs: We compare IsoVar's performance over bugs correlated with different types of variables. Specifically, we inspect the ratio of objects involved in the fixing patches of bugs, and then investigate whether the performance of IsoVar will be affected by the ratio of correlated objects.

4.4.3 RQ3

Enhancing FL: In addition to spectrum-based FL techniques, there are several other families of FL techniques as summarized by a recent study [9], which includes *mutation-based, slicing-based, stack trace-based, predicate switching-based*, and *hybrid-based* techniques. In this RQ, we select a wide range of such FL techniques to investigate whether IsoVar can be incorporated with them to enhance their performance. Specifically, as adopted by [9], we select *Ochiai [44], DStar [45], Metallaxis [46], MUSE [47] , Union [9], Intersection [9], Frequency [9], StackTrace [48], and PredicateSwitching [49]* since they are the most representative techniques for each category. We also select two recent state-of-the-art techniques, a hybrid-technique, *MCBFL* [8] and a bug-inducing commits driven technique, *HSFL* [7]. All the 11 techniques are designed to locate faults at the statement level. Therefore, for a target statement s, suppose the suspicious value computed for it by an existing technique is $susp(s)$, we use the following formula to refine such a suspicious value:

$$susp^b(s) = susp(s) + \Sigma_{v \in \mathcal{V}} IsoVar(v)/|\mathcal{V}| \qquad (10)$$

where $\mathcal{V}$ denotes all the variables used or defined in statement s and the $IsoVar(v)$ denotes the suspicious value returned by IsoVar for v. The idea is to enhance the original suspicious value for a statement with the average suspicious value of all the variables involved in the statement. Note that if a fixing patch is to change an expression from "`a>b`" to "`a==b`", variables "`a`" and "`b`" will also be regarded as the involved variables. We then evaluate the results of $susp^b(s)$ and compare them with those returned by $susp(s)$ for each technique.

Enhancing APR: The overfitting problem is a long-standing open challenge for automated program repair [50], in which case, the generated patches that pass the test suite are incorrect but merely plausible ones overfitting to the test suite. Plenty of approaches have been proposed aiming to address the overfitting problem, specifically, to prioritize correct patches over overfitting ones [11], [50], [51]. In this RQ, we investigate whether the results provided by IsoVar can help alleviate the overfitting problem. Our intuition is straightforward, which is that *those patches modifying more fault-correlated variables are more likely to be correct instead of plausible ones*. Therefore, we utilize the results returned by IsoVar to refine the ranking of all the patches generated by existing APR techniques as follows:

$$fitness^b(p) = fitness(p) + \Sigma_{v \in \mathcal{V}} IsoVar(v)/|\mathcal{V}| \qquad (11)$$

where $\mathcal{V}$ denotes the set of variables involved in the patch p and $IsoVar(v)$ denotes the suspicious value for variable v returned by IsoVar. Function $fitness(p)$ denotes the fitness value computed by existing APR techniques (if any) to rank the generated patches. Most APR techniques [52], [53], [35] directly utilized the suspicious value of the statement where the patch is generated to rank patches. For other techniques, they devise certain heuristics to prioritize and rank patches. For instance, CapGen [11] designed three heuristics measuring the contexts information of patches to rank all the generated patches. We perform this experiment based on a benchmark dataset *RepairThemAll* [54], which contains substantial patches together with their suspicious values and ranks generated by 11 diverse APR techniques. The correctness of such patches has been systematically labeled by another study [55], which produces 886 correct patches and 593 plausible but incorrect ones on Defects4J. Note that it takes one year for RepairThemAll to finish all the experiments and collect the patch generation and ranking results over the 11 APR tools, and thus it is infeasible for us to incorporate other patch ranking strategies to reproduce the results. Therefore, we directly utilize their ranking results for comparison.

We also tried to include other state-of-the-art APR techniques that utilize more advanced patch ranking techniques while most of them are not public or not working as revealed by an existing study [54]. Eventually, we further include PraPR [35], TBar [56] and SimFix [57] in our experiment. Specifically, for PraPR, we obtained all the patches via re-executing the open-sourced tool and labeled each of them manually via inquiring and confirming with the original authors. For TBar, since the original authors have already labeled whether the patches generated were correct or plausible but incorrect, we directly utilize their results for comparisons. As for SimFix, it fixes a bug until it finds a patch that passes all the test cases or reaches five hours in its original design. To verify whether IsoVar could help SimFix prioritize the ranking of the correct patches, we modified SimFix to allow it generate multiple patches for each bug within the time budget of five hours. We marked whether a patch is correct by checking its semantic consistency with the patches provided by the Defects4J benchmark.

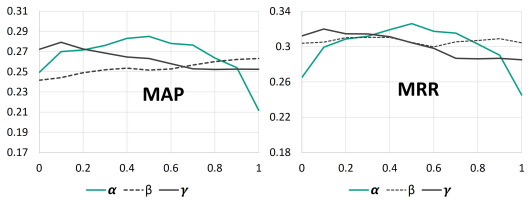
In addition to integrating our approach into existing APR tools, we also compare with other state-of-the-art approaches aiming at prioritizing patches to further evalaute IsoVar's performance. Specifically, we compare with ObjSim [58] and Patch-Sim [12]. ObjSim concerns the system state at the exit point on passing and failing test cases to prioritize correct patches, and has been shown to be able to improve the ranking of correct patches generated by PraPR. Therefore, we compare IsoVar with ObjSim with respect to the patches generated by PraPR. For Patch-Sim, it accepts a plausible patch that passes all test cases, and determines the correctness of the patch by heuristic. Specifically, Patch-Sim generates additional test cases to augment the original test suite, and then runs the tests on both the original program and the patched program. Finally, it determines patch correctness by comparing the behavioral similarity between the two executions. We apply Patch-Sim to prioritize the patches generated by TBar since it reports plausible patches for the largest number of bugs as shown in Table 6. We consider that Patch-Sim can help TBar correctly fix a bug if (1) it identifies all incorrect patches that rank before the correct patch as *incorrect*, and (2) it identifies the correct patch as indeed *correct*.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3209590

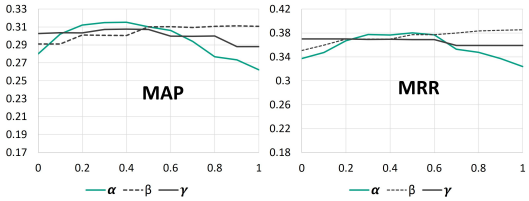
We evaluate *w.r.t.* the following two metrics.

- $Rank_{correct}$: the rank of the correct patches. A higher rank denotes that it is more efficient to generate the correct patches, and thus better results.
- $Precision$: the proportion of bugs correctly repaired over all the bugs with plausible patches generated. A bug is denoted as ***correctly repaired*** if there is *at least one correct patch that is ranked in prior to all the plausible but incorrect patches* [52], [11], [50].

Note that the RQ3 experiment was also performed with parameters set to the optimum values discussed in RQ2. When IsoVar performs on a new system for which it was not possible to optimize the parameters in advance, IsoVar will use the optimal parameters as discussed in RQ2 as the pre-configured parameters.



(a) Parameter Tuning on Defects4J



(b) Parameter Tuning on Bears

Figure 3: Impacts of Different Parameters

5 Evaluation

5.1 RQ1: Effect of Parameters

IsoVar combines statistical analysis and mutation analysis to locate fault-correlated variables. Figure 3 shows the impact of different parameters on the performance of IsoVar *w.r.t.* MAP and MRR, which reveal that different parameters will indeed cast certain impact on IsoVar's performance.

Specifically, for the results on Defects4J which are shown in Figure 3(a), if we vary the value of α, the MAP of IsoVar varies from 0.211 to 0.284 while the MRR varies from 0.245 to 0.326. For both of the two metrics, IsoVar achieves the optimum performance when α is set to around 0.5, β is set to around 0.9 and γ set to around 0.1. For parameters of Bears which are shown in Figure 3(b), when we change the value of α, the MAP of IsoVar varies from 0.262 to 0.315 while the MRR varies from 0.324 to 0.380. We observe a similar trend of α for Defects4J and Bears as shown in Figure 3(a) and Figure 3(b). Specifically, the curve is going upward when the α is set to 0, and it reaches the optimum when the value of alpha is set to around 0.4 to 0.5. Finally, the curve is going downward until α is set to 1, Similarly, for β, we can also observe a common trend, that is, the curve will slowly increase when it is set to the value from 0 to 0.9 or 1 on both the two datasets. As for γ, we do not observe a clear common trend between the curves. However, when its value is set between 0.1 and 0.5, IsoVar is able to achieve relatively good performance.

In general, we found that the two metrics MAP and MRR are very sensitive to α, and they can significantly affect the performance of IsoVar. For instance, when the tool is applied to Defects4J, the difference between the optimum and worst value of MAP can reach 0.073 ($0.284-0.211$). To summarize, IsoVar performs relatively well when α is set between 0.4 and 0.5, β is set between 0.9 and 1.0 and γ is set between 0.1 and 0.5. Such results reflect that the design of all the components in IsoVar is necessary, which significantly affects the final performance of IsoVar. However, their importance might be different as they contributed differently to the overall performance. For instance, the optimal value of β is close to 1, demonstrating that the effects of mutants on the original failing tests are more important to help locate fault-correlated variables than those on the passing tests. The optimum value of γ around 0.1 to 0.5 also reveals that the contribution of statistical analysis stage outweighs that of the mutation analysis stage.

5.2 RQ2: Effectiveness of IsoVar

To investigate IsoVar's performance while avoiding the potential overfitting problem, we select the optimum parameters achieved on Bears and apply them to Defects4J. Similarly, when evaluating the performance on Bears, we use the optimum parameters obtained on Defects4J. Table 5 shows the performance of IsoVar on different projects *w.r.t.* different metrics and the according comparisons with VFL. Specifically, we listed the results of IsoVar obtained via including only the statistical analysis and including both the statistical and mutation analyses (i.e., denoted as IsoVar$_s$ and IsoVar$_{sm}$ respectively). We also showed the results of three variants of VFL [23] based on Ochiai, Tarantula and Barinel (i.e., we selected such three variants since they achieved the optimum performance). The results show that IsoVar outperforms existing state-of-the-art *w.r.t.* different metrics for all cases.

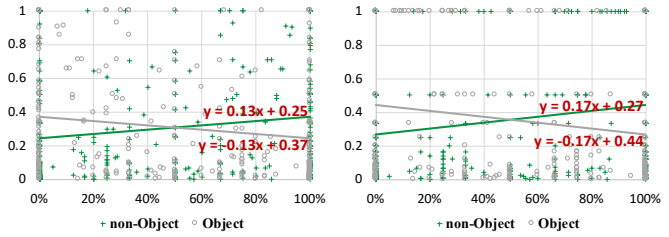
With respect to Defects4J, the MAP and MRR have been improved by 5.6% on average (varying from 1.5% to 18.2%) and 15.2% (varying from 1.8% to 31.2%) respectively compared with VFL$_{ochi}$. With respect to Top-N, IsoVar can identify the fault-correlated variables at Top-1 for 67 bugs, and the number are 149, and 177 for Top-5, and Top-10 respectively. In regard to the baseline, it can only locate the fault-correlated variables at Top-1, Top-5, and Top-10 for 43, 128, and 170 cases, respectively. For Bears, the MAP and MRR improved by 42.7% on average (varying from 23.8% to 87.9%) and 32.2% (varying from 4.6% to 90.1%) respectively compared with VFL$_{ochi}$, which achieved the optimum among the baseline variants. IsoVar can also identify the fault-correlated variables at Top-1, Top-5, and Top-10 for 26, 49, and 66 cases respectively. Such results demonstrate the superiority of IsoVar over existing baselines.

In Table 5, we also present the contributions of the two analyses separately. The results show that both analyses contribute significantly to the final performance of IsoVar. Specifically, if we only consider the statistical analysis, the MAP and MRR can be enhanced by 6.0% and 22.5% respectively. After integrating the mutation analysis, the im-

Table 5: The Performance of IsoVar and VFL with respect to Different Metrics.

	MAP						MRR						Top-1		Top-5		Top-10	
Projects	VFL_{ochi}	VFL_{tar}	VFL_{bar}	$IsoVar_s$	$IsoVar_{sm}$	Imp%	VFL_{ochi}	VFL_{tar}	VFL_{bar}	$IsoVar_s$	$IsoVar_{sm}$	Imp%	VFL_{och}	$IsoVar_{sm}$	VFL_{och}	$IsoVar_{sm}$	VFL_{och}	$IsoVar_{sm}$
Time	0.238	**0.268**	0.265	0.251	0.267	12.2%	0.252	0.252	0.251	0.321	**0.330**	31.0%	3	**4**	8	**13**	15	15
Chart	0.319	0.295	0.337	0.366	**0.377**	18.2%	0.317	0.254	0.295	0.380	**0.416**	31.2%	4	**6**	13	**14**	14	**15**
Lang	0.422	0.387	0.410	0.447	**0.451**	6.9%	0.415	0.359	0.375	0.485	**0.486**	17.1%	15	**18**	33	**38**	38	**43**
Math	0.265	0.254	0.272	0.270	**0.274**	3.4%	0.258	0.234	0.251	0.291	**0.295**	14.3%	12	**17**	39	**40**	**50**	48
Closure	0.154	0.121	0.152	**0.162**	0.160	3.9%	0.180	0.146	0.171	**0.210**	0.201	11.7%	12	**14**	25	**34**	39	**42**
Mockito	0.269	0.246	0.250	0.250	**0.273**	1.5%	0.271	0.239	0.242	0.275	**0.276**	1.8%	7	**8**	10	10	14	14
FasterXML	0.140	0.105	0.204	0.241	**0.263**	87.9%	0.181	0.162	0.252	0.321	**0.344**	90.1%	3	**5**	8	**11**	12	**15**
Spoon	0.151	0.103	0.148	0.184	**0.187**	23.8%	0.238	0.159	0.212	0.242	**0.249**	4.6%	7	**10**	**19**	18	22	**23**
Traccar	0.273	0.200	0.296	0.382	**0.384**	40.7%	0.316	0.271	0.298	0.385	**0.427**	35.1%	5	**11**	16	**20**	25	**28**
Summary	0.239	0.211	0.244	0.264	**0.270**	13.0%	0.259	0.220	0.247	0.302	**0.309**	19.3%	68	**93**	**171**	**198**	229	**243**

VFL_{ochi}, VFL_{tar} and VFL_{bar} denote the variants based on Ochiai, Tarantula and Barinel. $IsoVar_s$: results of statistical analyses; $IsoVar_{sm}$: results combining two analysis. The MAP and MRR in the Summary row indicates the weighted average, with the weight being the number of bugs per project.



(a) MAP (b) MRR

Figure 4: Performance *w.r.t.* Object and Non-Object Variables. The Trending Line is also Measured as Displayed

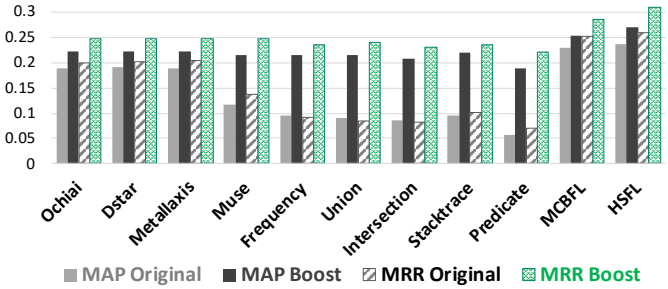


Figure 5: Boosting the Performance of FL techniques

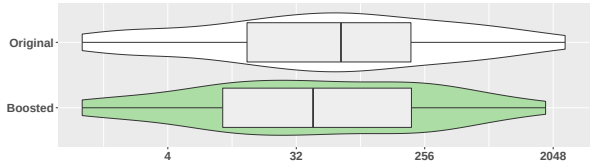


Figure 6: The Rank of Correct Patches

provements over baselines can be further enhanced to 10.8% and 27.2% respectively. Such results also reflect the generalizability of our approach since the tuned parameters are utilized in a cross project manner. Specifically, applying the parameters tuned on Defects4J can still lead IsoVar to achieve superior performance on the Bears dataset. Moreover, such results also indicate that the overhead of parameter tuning is limited since the involved parameters only need to be tuned in a one-shot manner, and can be generalized and utilized in other projects.

We further investigated the performance of IsoVar *w.r.t.* bugs with different ratios of Object fault-correlated variables and non-Object ones (i.e., basic type, String and Array). Figure 4 shows the results, which also depicts the trending lines measuring the relationship between the performance of IsoVar and different types of variables. The results reveal that with the increase of the ratio of non-Object correlated variables in the bug, IsoVar can also achieve better results *w.r.t.* MAP and MRR. However, the opposite trend is observed for Object correlated variables as shown in Figure 4. Such results indicate that IsoVar is more effective in locating fault-correlated variables of non-Objects. This might be caused by the fact that mutating Objects is more complex and tricky, and thus their impact on the whole program can hardly be precisely observed and quantified.

5.3 RQ3: Usefulness of IsoVar

5.3.1 Enhancing FL

We compare the original results of existing FL techniques (in total 11 different ones from 7 families) and the boosted results after incorporating the results at the variable level based on Equation 10. Figure 5 shows the average results over 357 bugs[3] in Defects4J *w.r.t.* MAP and MRR. The results show that IsoVar can boost the performance for all the existing techniques. Specifically, the improvement *w.r.t.* MAP ranges from 16.7% to 218.8% over different techniques, and that *w.r.t.* MRR ranges from 18.4% to 199.4%. Moreover, *w.r.t.* different subjects, the performance can all be boosted significantly (such results *w.r.t.* subjects are not displayed due to page limit). Take HSFL, which achieves the optimum results among existing techniques, as an example, the MAP has been improved by 17.5%, 21.3%, 18.8%, 15.7% and 4.6% *w.r.t.* subjects Time, Chart, Math, Lang and Closure respectively. Accordingly, the improvements *w.r.t.* MRR are 17.5%, 31.2%, 22.4%, 23.5% and 8.0% respectively. Such results demonstrate the usefulness of IsoVar, which can significantly boost the performance of existing FL.

5.3.2 Enhancing APR

We compare the original results of 14 existing APR techniques and the boosted results after incorporating the results at the variable level based on Equation 11. The key to overcoming the overfitting problem is to enhance the rank of correct patches, and prioritize them over all plausible but incorrect patches. Figure 6 shows the results *w.r.t.* the rank of corrected patches. Specifically, the rank of all correct patches generated by all the selected techniques can be significantly improved from an average rank of 791.0 to 522.9 (the median rank is improved from 71 to 43.5) with a $p\text{-}value$ of 0.043.

3. Bugs from Mockito are excluded since our experiments in this RQ were built based on an existing study [9], that excluded Mockito.

Table 6 shows the results *w.r.t.* the number of correct patches that are ranked prior to all incorrect patches, and the number of bugs that can be correctly repaired. We compare the results, for each technique, based on the original ranks generated by the technique, and the new ranking based on the boosted fitness value for each patch as shown in Equation 11. Actually, for all 14 APR tools, 7 of them have no room for improvement since they have already ranked the correct patches prior to all other plausible ones. For the other 7 tools, IsoVar can effectively improve their precision. Specifically, 49 (950-901) more correct patches can be ranked before all plausible but incorrect ones. Moreover, 10 (139-129) more bugs can be correctly repaired, thus enhancing the ***precision*** from 76.8% (=129/168) to 82.7% (=139/168) among those repairing results as shown in Table 6.

As aforementioned, we compare IsoVar with ObjSim to improve the ranking of correct patches generated by PraPR. In our experiments, PraPR generated plausible patches for 33 bugs, of which the ranking of correct patches for 6 bugs can be boosted by IsoVar or ObjSim. The ranking of the correct patches for the other 27 (33-6) bugs could not be boosted by either of the tools. Table 7 lists the detailed ranking results of the 6 bugs that can be improved while the full results are shown on our online website. In particular, IsoVar has improved the ranking of correct patches for four bugs (with the ✓ as shown in Table 7) and the correct patches of two bugs have been ranked at the first place. ObjSim has improved the ranking of correct patches for three bugs and two of them have been ranked 1st. In terms of the average ranking of correct patches for all the 33 bugs, ObjSim reduced the average ranking from 3.71 to 3.77 since it has degraded the ranking of correct patches for certain bugs. In contrast, IsoVar reduces the average ranking of correct patches from 3.71 to 3.42 averaged over the 33 bugs. Such results reflect the effectiveness of our proposed approach in prioritizing correct patches, which can benefit future APR researches.

We also compare IsoVar with Patch-Sim to prioritize the correct patches generated by TBar. Overall, Patch-Sim can help TBar correctly fix one more bug and IsoVar achieves the same results (see the TBar row in Table 6). However, Patch-Sim is more costly than IsoVar since it usually takes 5 to 10 minutes to identify the correctness of a single plausible patch. For example, TBar generated 296 plausible patches for Math-50, and the correct patch was originally ranked at the 215th place. Consequently, it will take plenty of time for Patch-Sim to prioritize the correct patches. In contrast, IsoVar spends 7 minutes in identifying the fault-correlated variables, and the results can be utilized to prioritize the correct patches with limited overhead.

Typically, an APR tool will first locate the fault, which identifies a set of suspicious statements, and then generates patches via applying predefined repair templates. The suspicious values of these patches generated in the same location are often the same since they often share the same contexts. Therefore, even if a correct patch can be generated, it is difficult to efficiently differentiate it from the other plausible patches. In contrast, IsoVar can identify correct patches more effectively via measuring how many fault-correlated variables are involved in the patches, thus improving the precision of APR techniques. Such results indicate that IsoVar reveals a promising direction to address the overfitting problem in APR via inspecting the divergence of the concerned variables. Such an idea is new to the APR community, which is orthogonal to other more advanced patch ranking strategies. Therefore, our idea can be easily integrated with existing studies to further boost their performance.

Table 6: Patch Prioritization for Existing APR Techniques

APR Tool	#CP	#CPBP	#CPBP^b	#Bugs	#Bugs$_c$	#Bugs$_c^b$
Cardumen [15]	3	3	3	3	3	3
jMutRepair [14]	4	4	4	4	4	4
NPEFix [59]	4	0	**1**	1	0	1
Nopol [13]	3	3	3	3	3	3
Arja [60]	774	764	**767**	7	6	**7**
DynaMoth [61]	2	2	2	2	2	2
GenProg [53]	45	3	**36**	3	1	3
JGenProg [14]	3	3	3	3	3	3
jKali [14]	2	2	2	2	2	2
RSRepair [60], [16]	43	13	**20**	6	5	**6**
Kali [60], [17]	3	3	3	3	3	3
PraPR [35]	39	17	**19**	33	17	**19**
TBar [56]	80	55	**56**	68	54	**55**
SimFix [57]	33	29	**31**	30	26	**28**
Total	1,034	901	950	168	129	139

#CP denotes the total number of correct patches. #CPBP denotes the number of correct patches that are prioritized before all plausible but incorrect patches while #CPBP^b denotes such a number boosted by IsoVar. #Bugs denotes the total number of bugs with plausible patches generated. #Bugs$_c$ denotes the total number of bugs that are correctly repaired while #Bugs$_c^b$ denotes such a number using the boosted fitness value in Equation 11.

Table 7: Comparison of Prioritizing Correct Patches Generated by PraPR between IsoVar and ObjSim

Bugs	#PP	#CP	IsoVar		ObjSim	
			Before	After	Before	After
Time-11	39	1	4	**3** ✓	4	4
Lang-59	2	1	2	**1** ✓	2	2
Math-58	2	1	2	2	2	**1** ✓
Math-82	9	1	8	**1** ✓	8	2 ✓
Closure-31	6	1	6	**3** ✓	6	Timeout
Mockito-29	2	1	2	2	2	**1** ✓

#PP denotes the total number of plausible patches that pass all test cases. #CP denotes the total number of correct patches. Before reports the best rank of correct patch among other plausible patches while After reports the such a rank boosted by IsoVar or ObjSim.

6 Discussion

6.1 Threats to Validity

In this study, the oracle information (i.e.,the fault-correlated variables) used to evaluate the effectiveness of IsoVar is extracted from the fixing patches following the strategy adopted by existing studies [38]. However, such a strategy might be biased since the variables such extracted might not represent the real fault-correlated variables of the bug. Nevertheless, it is the best practice we can take to perform large-scale evaluations. In the future, we plan to perform in-depth case studies to refine such oracle information and then inspect the performance of IsoVar.

6.2 Limitations and Future Work

IsoVar can only be applied to those bugs that are related to program variables (i.e., 92.0%=478/520 in our experiments).

There are also other software issues that are caused by configurations, documentation, and so on. For such cases, IsoVar is inapplicable. In the future, we will explore how to adapt our idea to a wider range of applications. Our evaluation results reveal that IsoVar can boost the performance of existing debugging techniques. This is because IsoVar works from a novel perspective, which is program variables, to facilitate debugging.

Although promising results have been achieved, many enhancements can be made. First, the strategies adopted to incorporate IsoVar and existing techniques are simple. We will explore how to better incorporate IsoVar with existing techniques in the future. Second, different program variables are actually not independent. Instead, they are highly correlated by different relations (e.g., control-flow relations, data-flow relations and call/callee relations), and whether such relations can guide us to better prioritize and isolate fault-correlated variables is worth exploring. Third, different types of variables are likely to trigger different types of faults. For example, `Objects` are often correlated with `NullPointerExceptions` while `Arrays` often trigger `OutOfBoundsException`. Therefore, it is important to consider the types of bug symptoms to isolate fault-correlated variables precisely. We left such explorations as our important future work.

Table 8: Average Overheads of IsoVar for Each Project.

	Time-overhead (mins)		
Projects	Statistical Analysis	Mutation Analysis	IsoVar
Lang	1.06 ± 0.23	1.25 ± 0.83	2.15 ± 0.55
Chart	1.19 ± 0.12	1.01 ± 0.15	1.91 ± 0.25
Math	4.26 ± 0.42	2.95 ± 1.50	6.51 ± 2.31
Time	0.91 ± 0.22	0.86 ± 0.16	1.97 ± 0.81
Closure	3.97 ± 1.12	5.15 ± 3.33	8.67 ± 4.18
Mockito	1.23 ± 0.21	1.21 ± 0.95	2.14 ± 1.01
jackson-databind	2.01 ± 1.18	2.21 ± 1.95	4.72 ± 1.58
Spoon	3.12 ± 2.11	4.84 ± 1.45	7.96 ± 3.01
Traccar	3.23 ± 1.56	1.93 ± 0.56	5.12 ± 2.18

6.3 Overheads of IsoVar

Overhead Analysis: Table 8 shows the average overheads of IsoVar for each project. In general, depending on the complexity of the target project and the number of associated test cases, IsoVar is able to complete the statistical analysis within a few minutes (varying from 0.91 to 4.12 minutes on average), and additional a few minutes for the mutation analysis (varying from 0.86 to 5.15 minutes). For the additional memory overhead required by the tool, depending on the size of the project under test and the complexity of the test cases, IsoVar requires additional memories of around 100M to 500M to store the matrix as mentioned in Section 3.1, which is roughly equivalent to the memory consumption of spectrum-based fault location techniques such as Ochiai.

Among all these projects, Closure takes the longest time for IsoVar to analyze since the size of Closure is the largest. Specifically, it contains 147k lines of code and 7,929 test cases, which is far more than that of the other projects. The complexity of the test cases in Closure is also another important reason that degrades IsoVar' efficiency. In particular, a test of Closure can include more than one thousand functions on average, while each test case of the other projects only involved dozens of functions. Such results indicate that IsoVar, with its high efficiency, is practical to be applied into open-source projects, even for large-scale ones. Note that we perform the mutation analysis in parallel when measuring its efficiency. In particular, the process to apply mutations (up to 10) for each variable and then execute the test suite is independent for each variable, and thus we can perform the mutation analysis in parallel to reduce the overheads.

Scalability Constraint: We also observe that IsoVar's scalability can be significantly affected by certain tests, which may consume a large amount of memory or time. For example, some tests are designed to evaluate whether the program will throw an exception as expected after a timeout while IsoVar needs to run these tests multiple times in the mutation phase. Such abnormal behavior may cause IsoVar to consume excessive resources. To alleviate the limitations of IsoVar's scalability, we monitor which test is resource-consuming, and will skip it in the mutation phase (about 15 bugs have encountered this problem in our dataset). Note that all the other mutation-based FL techniques also face the same problem, and we will continue to investigate how to better address such scalability constraints in the future.

6.4 Parameter Auto-tuning

To simplify the process of automatic parameters tuning, we implemented a program to automatically identify the optimal parameter setting (i.e.,α, β and γ). We utilize the intermediate results associated with each variable to perform the automated tuning of parameters (i.e.,$Freq_f$, $Freq_p$, $Simi_{\langle f,p \rangle}$ in Equation 4 and $\overline{I_f}(m)$, $\overline{I_p}(m)$ in Equation 5). We only need to run IsoVar once since we logged the variable suspicious values along with the corresponding intermediate results. The detailed process is described as follows. (1) As aforementioned, the three parameters take values from 0 to 1, and we iterate through all possible values with the step of 0.01 (e.g., $\alpha = 0.50$, $\beta = 0.95$, $\gamma = 0.20$). (2) Then we utilize the intermediate results associated with each variable to re-calculate the suspicious values and evaluate the MAP and MRR under the current parameters setting. (3) Finally, the optimal setting of parameters that maximizes MAP+MRR will be outputted. According to our experiments, this process can be completed within 30 seconds.

7 Related Work

7.1 Automated Fault Localization

Automated Fault Localization is a white-box debugging technique that aims to identify code elements (e.g., methods, statements) that are likely to cause the target fault. Various FL techniques have been proposed such as spectrum-based techniques (e.g., [62], [63], [6], [64]), mutation-based techniques (e.g., [65], [47], [66]), slice-based techniques (e.g., [67], [68]), machine-learning based techniques (e.g., [69]), program-state based techniques (e.g., [49]), data-augmented (e.g., [70]), feedback-based (e.g., [71]), and qualitative reasoning-based techniques (e.g., [72]) to facilitate developers in locating faults.

Spectrum-based and mutation-based ones are the two types of techniques that are most similar to this work. However, existing techniques mainly focus on locating buggy code elements at the method or statement level. In contrast, we explore the fault localization at the granularity of program variables in this study. To the best of our knowledge, this is by far the finest granularity in FL. VFL is the most related work to this study [23], which also tries to locate bugs at the variable level. However, it leverages the traditional spectrum constructed for statements while ignoring the sequence information of variables proposed by this study. Experiments show that our approach IsoVar outperforms VFL significantly, which reflects the effectiveness of our proposed *variable execution matrices*, and the mutation operators aiming to mutate various types of variables.

7.2 Automated Program Repair

Automated Program Repair (APR) is another important debugging activity, which has recently been extensively researched [11], [15], [14], [13], [53], [60], [35]. Diverse mutation operators have been proposed, either manually based on heuristics [53], [35] or automatically via mining substantial fixing patches [11], [14], with the aim to generate more correct patches. However, the overfitting problem is a long-standing open challenge for APR [50], in which case, the generated patches that pass the test suite are incorrect but merely plausible ones overfitting to the test suite.

Plenty of approaches have been proposed to address the overfitting problem, specifically, to prioritize correct patches over overfitting ones [50]. Ghanbari et al. proposed ObjSim [58] to prioritize the correct patches generated by PraPR [35]. As aforementioned, it concerns the system state at the exit point of passing and failing test cases to prioritize correct patches and ObjSim can be integrated into PraPR to boost its performance. Xiong et at. [12] proposed Patch-Sim to determine the correctness of plausible patches that pass all test cases. The authors observe that the failing tests on the original and patched programs are likely to behave differently while the opposite behavior is observed in the executions of passing tests. Based on the observation, they utilize the behavior similarity of test case executions to exclude incorrect patches.

Our proposed approach differs from the above techniques. First, we consider the *program variables* since they are essential to program ingredients in the process of patch generation (e.g., the adoption of mutation operators and synthesis of program entities). Moreover, IsoVar identifies fault-correlated variables within 10 minutes on average and the results can be utilized to better prioritize all the generated patches efficiently. In contrast, Patch-Sim usually takes 5 to 10 minutes to identify the correctness of a patch, making it costly to process multiple plausible patches generated by APR tools. Such an idea is similar to the recently proposed *unified debugging* which has pointed out that FL and APR can boost the performance of each other [73]. However, we are the first to propose leveraging the information of fault-correlated variables to enhance APR, and our experiments as shown in Section 5.3 have shown promising results. Therefore, we believe it is a promising and worth-exploring research direction.

8 Conclusion

Program variables are important information for developers to debug. Developers often set breakpoints at specific locations or execute the program step by step, and then monitor concerned program variables to see if abnormal values or status will be witnessed. Therefore, isolating fault-correlated variables is important. Motivated by this, we propose IsoVar, which combines statistical analysis and mutation analysis with the aim to isolate fault-correlated variables. Extensive experiments on Defects4J show that IsoVar can outperform existing techniques significantly. More importantly, we further made attempts to incorporate IsoVar with other existing debugging techniques, including 11 FL techniques and 14 APR techniques, and found that IsoVar can significantly boost the performance of existing techniques.

Acknowledgement

We sincerely thank all anonymous reviewers for their valuable comments. This work was supported by the National Natural Science Foundation of China (Grant No.62002125 and No.62072194) as well as the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2021QNRC001).

References

[1] "Software bug - wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Software_bug, Wikipedia contributors. 2021.
[2] "Tricentis report," https://www.tricentis.com, 2022-07-02.
[3] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.
[4] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.
[5] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.
[6] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.
[7] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, 2019.
[8] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.
[9] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.
[10] J. Yang, Y. Yang, M. Sun, M. Wen, Y. Zhou, and H. Jin, "Isolating compiler optimization faults via differentiating finer-grained options," in *2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2012.
[11] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180233
[12] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 789–799.

[13] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[14] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[15] ——, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *International Symposium on Search Based Software Engineering*. Springer, 2018, pp. 65–86.

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

[17] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[18] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *arXiv preprint arXiv:1707.05172*, 2017.

[19] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, vol. 25, no. 1, pp. 171–199, 2017.

[20] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[21] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[22] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: Understanding defects' root causes," in *Proceedings of the 2020 International Conference on Software Engineering*. ACM, 2020, pp. 87–99.

[23] J. Kim, J. Kim, and E. Lee, "Vfl: Variable-based fault localization," *Information and Software Technology*, vol. 107, pp. 179–191, 2019.

[24] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[25] A. Zeller, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[26] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 263–272.

[27] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

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

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

[30] R. Just, D. Jalali, and M. D. Ernst, "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*. ACM, 2014, pp. 437–440.

[31] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019. [Online]. Available: https://arxiv.org/abs/1901.06024

[32] T. Zimmermann and A. Zeller, "Visualizing memory graphs," in *Software Visualization*. Springer, 2002, pp. 191–204.

[33] "Github language stats," https://madnight.github.io/githut, 2022-07-02.

[34] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[35] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[36] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 201–211.

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

[38] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "Aurora: Statistical crash analysis for automated root cause explanation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[39] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999.

[40] E. M. Voorhees *et al.*, "The trec-8 question answering track report." in *Trec*, vol. 99, 1999, pp. 77–82.

[41] F. Steimann, M. Frenkel, and R. Abreu, "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*. ACM, 2013, pp. 314–324.

[42] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th international conference on Software engineering*. ACM, 1995, pp. 41–50.

[43] "Delta debugging plug-in," https://www.st.cs.uni-saarland.de/eclipse/, 2022-07-02.

[44] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[45] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[46] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[47] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 153–162.

[48] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[49] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006, pp. 272–281.

[50] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 968–980.

[51] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

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

[53] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[54] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 302–313.

[55] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 981–992.

[56] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[57] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," ser. ISSTA, 2018.

[58] A. Ghanbari, "Objsim: lightweight automatic patch prioritization via object similarity," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 541–544.

[59] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 349–358.

[60] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[61] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016, pp. 85–91.

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

[63] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[64] D. Lo, X. Xia *et al.*, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 127–138.

[65] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 464–475.

[66] M. Delahaye, L. C. Briand, A. Gotlieb, and M. Petit, "μtil: Mutation-based statistical test inputs generation for automatic fault localization," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 197–206.

[67] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[68] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.

[69] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *ISSRE*, 2007, pp. 137–146.

[70] A. Elmishali, R. Stern, and M. Kalech, "Data-augmented software diagnosis," in *Twenty-Eighth IAAI Conference*, 2016.

[71] N. Cardoso and R. Abreu, "A kernel density estimate-based approach to component goodness modeling," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[72] A. Perez, R. Abreu, and I. HASLab, "Leveraging qualitative reasoning to improve sfl."

[73] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 75–87.