A Low-Cost Feature Interaction Fault Localization Approach for Software Product Lines

HAINING WANG, South China University of Technology, China
YI XIANG, South China University of Technology, China
HAN HUANG, South China University of Technology, China
JIE CAO, iSOFT INFRASTRUCTURE SOFTWARE CO., LTD., China
KAICHEN CHEN, South China University of Technology, China
XIAOWEI YANG, South China University of Technology, China

In Software Product Lines (SPLs), localizing buggy feature interactions helps developers identify the root cause of test failures, thereby reducing their workload. This task is challenging because the number of potential interactions grows exponentially with the number of features, resulting in a vast search space, especially for large SPLs. Previous approaches have partially addressed this issue by constructing and examining potential feature interactions based on suspicious feature selections (e.g., those present in failed configurations but not in passed ones). However, these approaches often overlook the causal relationship between buggy feature interaction and test failures, resulting in an excessive search space and high-cost fault localization. To address this, we propose a low-cost Counterfactual Reasoning-Based Fault Localization (CRFL) approach for SPLs, which enhances fault localization efficiency by reducing both the search space and redundant computations. Specifically, CRFL employs counterfactual reasoning to infer suspicious feature selections and utilizes symmetric uncertainty to filter out irrelevant feature interactions. Additionally, CRFL incorporates two findings to prevent the repeated generation and examination of the same feature interactions. We evaluate the performance of our approach using eight publicly available SPL systems. To enable comparisons on larger real-world SPLs, we generate multiple buggy mutants for both `BerkeleyDB` and `TankWar`. Experimental results show that our approach reduces the search space by 51%∼73% for small SPLs (with 6∼9 features) and by 71%∼88% for larger SPLs (with 13∼99 features). The average runtime of our approach is approximately 15.6 times faster than that of a state-of-the-art method. Furthermore, when combined with statement-level localization techniques, CRFL can efficiently localize buggy statements, demonstrating its ability to accurately identify buggy feature interactions.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Theory of computation** → **Program reasoning**.

Additional Key Words and Phrases: Software product lines, fault localization, counterfactual reasoning, feature interaction

ACM Reference Format:
Haining Wang, Yi Xiang, Han Huang, Jie Cao, Kaichen Chen, and Xiaowei Yang. 2025. A Low-Cost Feature Interaction Fault Localization Approach for Software Product Lines. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA042 (July 2025), 23 pages. https://doi.org/10.1145/3728917

1 Introduction

The Software Product Line (SPL) is a highly effective technique for modern software development. Numerous widely used software systems, including `Linux` and `BerkeleyDB`, have been developed using this methodology [5, 47]. SPLs can be flexibly customized to meet user requirements through

Corresponding authors: Yi Xiang (xiangyi@scut.edu.cn), Han Huang (hhan@scut.edu.cn).

ACM 2994-970X/2025/7-ARTISSTA042
https://doi.org/10.1145/3728917

the adoption of the concept of *features*, where each feature represents a specific system functionality [18]. A *configuration* (or *product*) is defined as a set of selected or deselected features. Generally, there exist complicated constraints among features, which are usually captured by *Feature Models* (FMs) [53, 60].

Features in SPLs can influence one another, leading to the so-called *feature interaction* problem [14, 23]. Faults resulting from feature interactions can cause the system to exhibit unpredictable behaviors [8], making it essential for developers to analyze and understand all feature interactions within an SPL [7, 15]. Feature interaction faults can be categorized as either functional or non-functional [12, 31, 56]. Functional feature interaction faults result in the software system failing to provide the correct output specified by the intended functionality [12], typically due to buggy statements influenced by feature interactions [34]. In contrast, non-functional faults do not cause system failures but lead to issues such as increased response time, reduced throughput, and other performance degradations. These non-functional faults are often caused by incorrect configurations of features [41]. Regardless of the fault type, localizing buggy feature interactions is a critical step in debugging SPLs [1, 20, 35, 39]. However, localizing potential feature interactions remains a significant challenge, primarily because the number of possible interactions to consider increases exponentially with the number of features [7, 50]. For instance, a system comprising 25 features may have a total of $\sum_{i=1}^{25} 2^i \times C_{25}^i \approx 1.3 \times 10^7$ different feature interactions.

There have been numerous studies on non-functional feature interaction fault localization for SPLs (e.g., [4, 20, 25, 26, 28, 38]), whereas studies on functional fault localization remain scarce. To address functional fault localization for SPLs, Arrieta *et al.* [9] employed Spectrum-Based Fault Localization (SBFL) to calculate and rank the suspiciousness of each feature interaction. However, their approach overlooks the "curse of dimensionality" associated with the number of features. In practice, faults are typically caused by only a few feature interactions, making it inefficient to consider all interactions [45]. To more precisely identify buggy feature interactions, Nguyen *et al.* [39] proposed a state-of-the-art (SOTA) method called VarCop. They observed that buggy feature interactions could be identified by analyzing the feature selection differences between passing and failing configurations, referred to as *suspicious feature selections* in this paper. In this manner, potential feature interactions, which constitute the search space, can be generated from suspicious feature selections. However, their approach neglects the causal effect of feature interaction faults on test failures, potentially increasing the search space (an example of this is provided in Section 3.1). Furthermore, as demonstrated by our experiments in Section 6.1, many feature interactions in VarCop are redundantly generated and examined, leading to increased costs. Therefore, efficiently and accurately localizing buggy feature interactions in a vast search space remains a significant challenge.

This paper focuses on functional feature interaction faults and proposes a low-cost feature interaction fault localization approach, named CRFL. The proposed approach enhances the efficiency of localizing buggy feature interactions in two key aspects. First, counterfactual reasoning and symmetric uncertainty are employed to achieve more accurate and fewer potential feature interactions, resulting in a smaller search space. Counterfactual reasoning [13, 20, 28] has demonstrated significant advantages in root-cause localization, particularly due to its minimal data requirements, which makes it well-suited for the problem addressed in this paper. In addition, symmetric uncertainty, a widely adopted technique in high-dimensional feature selection problems [51, 52], further filters out irrelevant feature interactions. Second, two key findings are utilized to avoid duplicated suspiciousness examinations of potential feature interactions. The first finding is that there exists an inclusion relationship among suspicious feature selections, which leads to repeated generation of potential feature interactions. The second finding is that many identical potential feature interactions are examined multiple times, resulting in repetitive computations. Based on these findings, the efficiency of the proposed approach is further improved by avoiding redundant generations and computations. We evaluate our approach using eight real-world SPL systems. The experimental results show that CRFL reduces the search space by 51% to 73% for SPLs with fewer than ten features and by 71% to 88% for larger SPLs. In terms of efficiency, CRFL is, on average, 15.6 times faster than VarCop. Moreover, we demonstrate the accuracy of the suspicious feature interactions identified by CRFL through statement-level localization. Compared to the four state-of-the-art approaches, CRFL ranks buggy statements in the top 1 in 30% more mutants. The main contributions of this paper are summarized as follows:

- A low-cost feature interaction fault localization approach [1] is proposed for SPLs.
- Counterfactual reasoning and symmetric uncertainty are used to effectively reduce the search space for localizing buggy feature interactions.
- Two findings regarding the repetitive generation and examination of feature interactions have been observed, and further utilized to reduce the cost of the approach.
- We propose a publicly available benchmark containing multiple buggy feature interaction mutants for `BerkeleyDB` and `TankWar` to demonstrate the advantages of CRFL in large SPLs.

2 Preliminaries

In this section, we provide necessary preliminaries on SPLs, followed by a brief introduction to counterfactual reasoning.

2.1 Software product lines (SPLs)

A software product line delineates a product family characterized by a shared foundational code base, wherein a set of distinct products is systematically derived [18, 46]. *Features* are abstract representations of functional modules in SPLs.

Generally, an SPL system can be seen as a 2-tuple $\Gamma = \langle \mathcal{F}, \varphi \rangle$, where $\mathcal{F}$ is a set of features in Γ, and φ denotes all the constraints among features. For a feature set $\mathcal{F} = \{f_1, f_2, ..., f_{|\mathcal{F}|}\}$, each feature $f_i \in \mathcal{F}(i = 1, 2, ..., |\mathcal{F}|)$ has two states, selected (on) or deselected (off). The presence of features in SPLs can be considered as `if-then` statements, such as the preprocessor directive `#ifdef`. Different selections of all features define a *configuration* (or *product*), which can be denoted as $\{\pm f_1, \pm f_2, ..., \pm f_{|\mathcal{F}|}\}$. Notably, each feature can be represented only as $-f$ or $+f$, where $+f$ is simplified as f. Apparently, each configuration can be represented by a binary set consisting of '0' and '1'. For example, $\{f_1, -f_2, -f_3, f_4, f_5\}$ represents a configuration where features f_1, f_4 and f_5 are selected, and f_2 and f_3 are deselected. Its binary set is {1, 0, 0, 1, 1}. A *partial configuration* is any non-empty subset of a configuration, which is a manifestation of *feature interactions.* A configuration that meets all the constraints in φ is called a *valid configuration.*

2.2 Causal inference and counterfactuals

Causal inference techniques have been demonstrated to trace root causes better than other techniques (e.g., deep learning) [43]. Counterfactual reasoning is a widely used causal inference technique to analyze causal relationships by exploring hypothetical scenarios that diverge from actual events. The key idea is to explore what the outcome would have been had a different action or decision been made [24]. It's like asking "What if?" questions to explore different possibilities and understand cause-and-effect relationships [57]. For example, when evaluating the effectiveness of a new teaching method, educational researchers compare the performance of students who use the method with those who do not. This kind of thinking is counterfactual reasoning.

[1] https://github.com/Songluhaining/CRFL.git

Counterfactual reasoning is particularly valuable for understanding cause-and-effect dynamics, as it enables researchers to model and evaluate alternative outcomes by varying factors of interest. For non-functional faults in feature interactions, counterfactual reasoning can be employed to assess their observability (i.e., whether a feature interaction affects the validity of configurations) [58]. Specifically, determining whether a feature interaction *FI* is observable in invalid configurations involves checking the presence of both a witness and a counter witness [2]. Inspired by their study, we apply counterfactual reasoning to localize functional feature interaction faults.

3 A motivating example

In this section, we explain the challenge of detecting buggy feature interactions and our motivation via an example.

3.1 An example of faults in SPLs

Fig. 1 illustrates an example of a functional feature interaction fault in a partial code snippet from the `Elevator` SPL system [39]. The sampled products (configurations) and corresponding test results are given in Fig. 2. Configurations c_6 and c_7 in Fig. 2 are failed tests because of the buggy statement in line 30 of Fig. 1.

```
1  int maxWeight = 1000, maxPersons = 10,
     weight = 0;
2  // #ifdef Empty
3  void empty () { persons.clear();}
4  // #endif
5  void enterElevator (Person p) {
6    persons.add(p);
7    // #ifdef Weight
8    weight += p.getWeight();
9    // #endif
10 }
11 void leaveElevator (Person p) {
12   persons.remove(p);
13   // #ifdef Weight
14   weight -= p.getWeight();
15   // #endif
16 }
17 ElevatorState stopAtAFloor (int floorID) {
18   ElevatorState state = Elev.openDoors;
19   boolean block = false;
20   for (Person p : new ArrayList<Person> (persons))
21     if (p.getDestination() == floorID)
22       leaveElevator(p);
23   for (Person p : waiting) enterElevator(p);
24   // #ifdef TwoThirdsFull
25   if ((weight == 0 && persons.size() >=
         maxPersons*2/3 || weight >= maxWeight*2/3)
26     block = true;
27   // #endif
28 // #ifdef Overloaded
29   if (block == false) {
30     if ((weight == 0 && persons.size() >=s
           maxPerson || weight == maxWeight)
           //Patch: weight >= maxWeight
31         block = true;
32   }
33   // #endif
34   if (block == true)
35     return Elev.blockDoors;
36   return Elev.closeDoors;
37 }
```

Fig. 1. An illustrative example of a variability bug in `Elevator` System.

P	*C*	*Base*	*Empty*	*Weight*	*TwoThirdsFull*	*Overloaded*
p_1	c_1	T	F	T	F	F
p_2	c_2	T	T	T	F	F
p_3	c_3	T	T	F	F	F
p_4	c_4	T	F	T	T	F
p_5	c_5	T	F	T	T	T
p_6	c_6	T	T	T	F	T
p_7	c_7	T	F	T	F	T

T: selected F:deselected

Fig. 2. Sampled configurations and corresponding test results.

[2]A counter witness $\bar{v}$ represents a valid configuration in which all features selected in the feature interaction *FI* are switched relative to configuration v.

In Fig. 1, the `#ifdef` directive represents a functionality within the system by controlling the selection/deselection of features. There are five features: *Base*, *Empty*, *Weight*, *TwoThirdsFull*, and *Overloaded*. The *Base* feature controls basic functions, *Empty* supports elevator operation when it is empty, *Weight* controls elevator operation when it is loaded, and *TwoThirdsFull* and *Overloaded* control elevator operation when the loaded weight reaches two thirds of the maximum limit or exceeds it. However, the statement `weight==maxWeight` in line 30 does not adhere to the logic of the system design and should be replaced with `weight>=maxWeight`. This fault does not cause failures in all configurations, as it is a variability bug. In other words, configurations may fail only when *Overloaded* is selected. The variable influencing the normal execution of line 30 is `weight`, which is governed by both *Weight* and *TwoThirdsFull*, indicating that they interact with *Overloaded*. Specifically, configurations will fail only when *Weight*=T, *TwoThirdsFull*=F, and *Overloaded*=T. It is important to note that *Overloaded* can be selected only if *Weight* is also selected. Therefore, the actual buggy feature interaction is (-*TwoThirdsFull*, *Overloaded*), where '-' indicates that the feature is deselected.

In VarCop [37], potential feature interactions can be obtained by comparing the feature selection differences (its definition is given in Section 3.2) between failed and passed configurations. However, we further observe that, for each failed configuration, the feature selection differences generated with dissimilar passed configurations may increase the size of search spaces. For example, the feature selection difference between c_3 and c_7 is -*Empty*, *Weight*, *Overloaded*. The generated feature interactions are (-*Empty*), (*Weight*), (*Overloaded*), (-*Empty*, *Weight*), (-*Empty*, *Overloaded*), (*Weight*, *Overloaded*), and (-*Empty*, *Weight*, *Overloaded*). It should be noted that feature interactions containing $-$*Empty* are not the root cause of the failure of c_6 and c_7. Thus, the search space for potential feature interactions generated remains large, especially when the number of features and configurations is numerous.

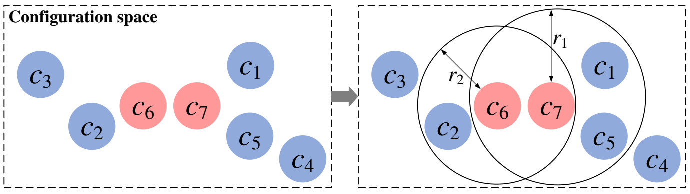


Fig. 3. An illustrative example of our motivation.

Based on the key idea of counterfactual reasoning, a feature interaction may cause a configuration c to fail, depending on whether there is a counter witness for c. However, modifying the feature selections of failed configurations and retesting them is inefficient. Therefore, for a set of configurations, the similarity in feature selection between passed and failed configurations partially reflects the counterfactual nature of feature interactions, particularly their differences in feature selection. In other words, the more similar the feature selections between two configurations—one passing and the other failing—the higher the likelihood that their feature selection differences indicate potential buggy feature interactions. For instance, the most similar passed configuration to c_6 is c_2, where the *Overloaded* feature is deselected, leading to configuration failure. Conversely, for c_7, the most similar passed configurations are c_1 and c_5, where the selection or deselection of *Overloaded* or *TwoThirdsFull* directly impacts the test outcome. Similarly, the suspicious feature selection is *TwoThirdsFull*, *Overloaded*, resulting in potential feature interactions of (*TwoThirdsFull*), (*Overloaded*), and (*TwoThirdsFull*, *Overloaded*). Thus, counterfactual reasoning yields more suspicious and fewer feature interactions, thereby reducing the search space.

As illustrated in Fig. 3, these configurations can be represented in a configuration space where the similarity between two configurations can be measured using distances such as the Hamming distance. Consequently, a search radius can be established for each failed configuration, allowing for the identification of similar passed configurations.

Furthermore, we have observed in practice that the selection of each feature correlates with the test results. For instance, *Base*, *Weight*, *TwoThirdsFull*, and *Overloaded* are more relevant to the test results, whereas *Empty* is not. In our study, we utilize symmetric uncertainty [44] to quantify the correlation between each feature and the test results. This correlation is determined based on the entropy and conditional entropy of the features with respect to the test results. The value of symmetric uncertainty ranges from 0 to 1, with values closer to 1 indicating a stronger correlation. Specifically, the symmetric uncertainties of *Base*, *Empty*, *Weight*, *TwoThirdsFull*, and *Overloaded* with respect to the test results are 0.67, 0.00, 0.50, 0.50, and 0.50, respectively. Consequently, irrelevant low-order feature interactions, such as (*Empty*), can be filtered to further reduce the search space.

3.2 Key definitions

We give the following key definitions used throughout the paper.

Definition 1. (Feature selection difference). For a failed configuration e and a passed configuration z, the feature selection difference refers to the features selected differently by e relative to z. It can be expressed as follows:

$$\mathcal{D}(e,z) = \{e_q | e_q \neq z_q, q = 1, 2, ..., |\mathcal{F}|\}, \tag{1}$$

where e_q and z_q are the qth values in the binary set of e and z, respectively; and $\mathcal{D}(e,z)$ represents the feature selection difference between e and z.

Example 1. *In Fig. 2, Empty and Overloaded are selected while TwoThirdsFull is deselected for c_6; Empty and Overloaded are deselected while TwoThirdsFull is selected for c_4. Therefore, $\mathcal{D}(c_6, c_4)$ is {Empty, -TwoThirdsFull, Overloaded}.*

Definition 2. (Suspicious feature selection). The suspicious feature selection is defined as the union of the feature selection differences generated based on a failed configuration e with respect to a Passed Configuration Set (*PCS*),

$$\mathcal{U}(e) = \bigcup_{z \in PCS} \mathcal{D}(e,z), \tag{2}$$

where $\mathcal{U}(e)$ indicates a suspicious feature selection for e.

Example 2. *For c_7 in Fig. 2, and a passed configuration set $PCS = \{c_1, c_5\}$, $\mathcal{D}(c_7, c_1)$ and $\mathcal{D}(c_7, c_5)$ are {Overloaded} and {-TwoThirdsFull}, respectively. Therefore, the suspicious feature selection for c_7 is {Overloaded, -TwoThirdsFull}.*

Definition 3. (n-way feature interaction). An n-way feature interaction is defined as any subset of size n within a suspicious feature selection.

Example 3. *For {Overloaded, −TwoThirdsFull}, 1-way feature interactions are (Overloaded) and (−TwoThirdsFull), and 2-way feature interaction is (Overloaded, −TwoThirdsFull).*

It is worth noting that feature interaction faults typically involve no more than six features [23, 32], therefore we only focus on 1∼7-way feature interactions, following the practice in [39].

Definition 4. (Suspicious feature interactions). A potential feature interaction (*FI*) being examined is considered a suspicious feature interaction if it satisfies both of the following properties [37]:

- Bug-Revelation: Any configuration containing *FI* corresponds to a failed product.
- Minimality: There are no strict subsets of *FI* that satisfy the Bug-Revelation property.

4 Method

In this section, we provide a detailed description of the proposed CRFL. Specifically, Section 4.1 outlines the basic framework of the approach, while Sections 4.2 to 4.4 present detailed implementations of the main components.

4.1 Basic framework

Fig. 4 presents the framework of CRFL, which consists of the following five parts: ① Obtaining suspicious feature selections based on counterfactual reasoning; ② Filtering irrelevant feature interactions based on symmetric uncertainty; ③ Removing included suspicious feature selections using inclusion relationships; ④ Reducing duplicated feature interactions by utilizing a caching mechanism; and ⑤ Examining feature interactions for suspiciousness.

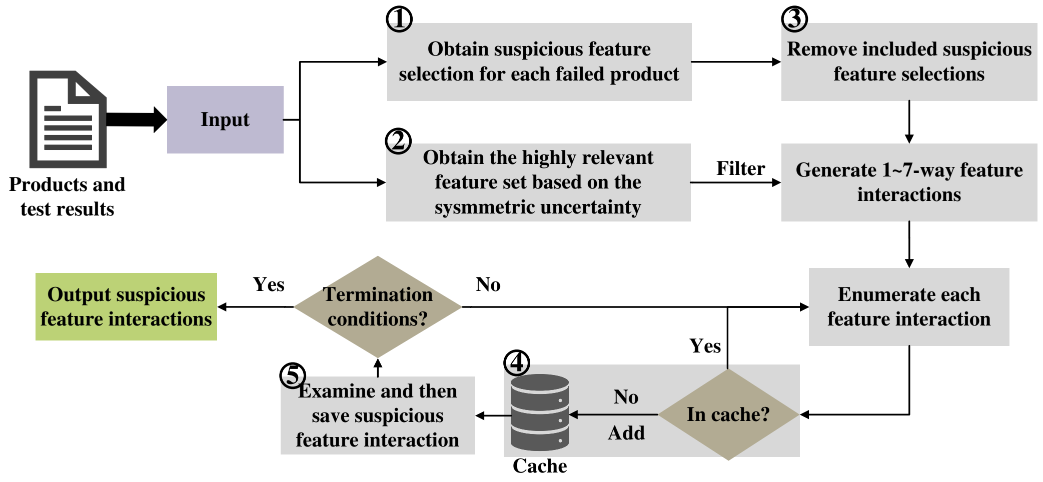

Products and test results
Input
①
Obtain suspicious feature selection for each failed product
②
Obtain the highly relevant feature set based on the sysmmetric uncertainty
Filter
③
Remove included suspicious feature selections
Generate 1~7-way feature interactions
Enumerate each feature interaction
Output suspicious feature interactions
Yes
Termination conditions?
No
Yes
④
No
Add
In cache?
Cache
⑤
Examine and then save suspicious feature interaction

Fig. 4. Framework of the proposed CRFL, which consists of five parts.

We provide the pseudo-code of the proposed CRFL in Algorithm 1. The input of CRFL is the configuration-level test suite ($\mathcal{TS}$) which includes configurations and the corresponding test results. After examination using CRFL, the set of suspicious feature interactions (*SuspiciousFIsets*) is returned. Line 5 computes the radius of each failed configuration, while lines 6-13 aim to identify potential feature interactions. Next, line 14 calculates the symmetric uncertainty of each feature with respect to the test results, and obtains the set of features with high correlation. Finally, lines 17-28 remove duplicated feature interactions and perform a suspiciousness examination to obtain the set of suspicious feature interactions.

4.2 Generate suspicious feature selection based on counterfactual reasoning

Lines 1-13 in Algorithm 1 generate suspicious feature selections with the following two steps, i.e., finding similar passed configurations for each failed configuration, and generating corresponding suspicious feature selection.

Let all failed configurations be set to $\mathcal{FC}$, and all passed configurations be set to $\mathcal{PC}$. The first step is to determine the set of passed configurations that are similar to each failed configuration. This

Algorithm 1: CRFL

```
Input: Configuration-level test suite $\mathcal{TS}$=$\mathcal{PC} \cup \mathcal{FC}$ //$\mathcal{PC}$, $\mathcal{FC}$ denote the sets of passed
       and failed configurations, respectively
Output: $SuspiciousFIsets$ //Suspicious feature interaction sets
1  $SuspiciousFIsets \leftarrow \emptyset$;
2  $SuspiciousFSsets \leftarrow \emptyset$ //Suspicious feature selection set;
3  for each $c_i \in \mathcal{FC}$ do
4      $SusFS_i \leftarrow \emptyset$ //The suspicious feature selection of $c_i$;
5      Calculate the radius $r_i$ of $c_i$ by Eq. 4;
6      for each $c_j \in \mathcal{PC}$ do
7          Calculate the distance $d_j$ between $c_i$ and $c_j$ by Eq. (3);
8          if $d_j < r_i$ then
9              $SusFS_i \leftarrow SusFS_i \cup \mathcal{D}(c_i, c_j)$;
10         end
11     end
12     Add $SusFS_i$ into $SuspiciousFSsets$;
13 end
14 Calculate symmetric uncertainty of each feature with the test result set, and obtain the
     highly relevant feature set $HRF$ (see Section 4.3);
15 Remove included suspicious feature selections;
16 $Cache \leftarrow \emptyset$;
17 for $SusFS_i \in SuspiciousFSsets$ do
18     Generate 1~7-way feature interactions of $SusFS_i$, denoted as $FIS$;
19     for $FI_j \in FIS$ do
20         if $FI_j \notin Cache$ then
21             Add $FI_j$ into $Cache$;
22             if $FI_j \cap HRF \neq \emptyset$ then
23                 Perform suspiciousness examination for $FI_j$ and save it to $SuspiciousFIsets$
                     if $FI_j$ is suspicious;
24             end
25         end
26     end
27 end
28 return $SuspiciousFIsets$
```

goal can be achieved by using any distance metric to measure the similarity between configurations. Considering that configurations can be represented by binary sets, we choose Hamming distance in this paper. For a failed configuration $e \in \mathcal{FC}$ and a passed configuration $z \in \mathcal{PC}$, their distance can be calculated as:

$$Dis(e, z) = \sum_{q=1}^{|\mathcal{F}|} |e_q - z_q|. \tag{3}$$

Example 4. *As shown in Fig. 2, the distance from c_6 to c_2 is (0+0+0+0+0+1)=1, while the distances to c_1, p_3, c_4 and c_5 are all 2.*

Two configurations are closer if their feature selections are more similar. Consequently, passed configurations that are closer to e are identified to calculate their feature selection differences. Popular strategies for determining the closest passed configurations include the *TopK* strategy and the *Threshold* strategy [16, 19]. The *TopK* strategy selects the K nearest passed configurations to the failed configuration. In contrast, the *Threshold* strategy selects a variable number of passed configurations by setting a threshold, choosing those with distances less than the specified threshold. In practice, determining the appropriate value for K can be challenging, as the number of passed configurations can vary widely—from over 20 in some mutants to just one in others. Therefore, instead of adopting the *TopK* strategy, a radius is set for each failed configuration to identify similar passed configurations. In our approach, this radius is defined as the average distance from e to all passed configurations. The radius for e can be calculated as follows:

$$r = \frac{\sum_{z \in \mathcal{P}C} Dis(e, z)}{|\mathcal{P}C|}, \tag{4}$$

where r is the radius of e. When $Dis(e, z) < r$, the feature selection difference between e and z is calculated.

For a failed configuration e, let PCS denote the identified passed configurations. The suspicious feature selection of e is obtained based on Eq. (2), which calculates the union of the feature selection differences between e and each passed configuration in PCS.

4.3 Correlation-based filtering for potential feature interactions

In our work, symmetric uncertainty is used to evaluate the correlation between the selection of each feature and the test results. First, the selection vectors for each feature and the test result vector are required to be constructed. Let the set of selection vectors be VFs, and the vector of the test results be R. For a feature f_i, the jth dimension of its vector $VF_i \in VFs$ is '1' if f_i is selected in the jth configuration, and '0' otherwise. Similarly, the jth dimension of R is '1' if the jth configuration passes the test and 0 otherwise.

Example 5. *Considering the feature Empty in Fig. 2, its selection vector, based on its selection state in each configuration, is represented as (0, 1, 1, 0, 0, 0, 1, 0). Similarly, the test result vector is (1, 1, 1, 1, 1, 0, 0).*

Next, the selection vector for each feature can be calculated with the test results vector for symmetric uncertainty, as follows.

$$SU(VF_i, R) = 2\frac{H(VF_i) - H(VF_i|R)}{H(VF_i) + H(R)}, i = 1, 2, ..., k, \tag{5}$$

where $H(VF_i)$ and $H(R)$ are the entropies of VF_i and R; $H(VF_i|R)$ is the conditional entropy of VF_i when R is known. The sets $p(v)$ and $p(r)$ are the prior probabilities of Vb_i and R, respectively. The $H(Vb_i)$, $H(R)$ and $H(Vb_i|R)$ are calculated as follows.

$$H(VF_i) = -\sum_{v \in VF_i} p(v) log_2 p(v), i = 1, 2, ..., k, \tag{6}$$

$$H(R) = -\sum_{r \in R} p(r) log_2 p(r). \tag{7}$$

$$H(VF_i|R) = -\sum_{r \in R} p(r) \sum_{v \in Vb_i} p(v|r) log_2 p(v|r). \tag{8}$$

Finally, the selection of each feature can be calculated to correlate with the set of test results. For a feature f_i, its selection state is added to the high relevance feature set (denoted as HRF) if its symmetric uncertainty is greater than 0. It is worth mentioning that the added feature states should include both selected and deselected state. In our work, *irrelevant feature interactions are filtered based on the rule that buggy feature interactions much contain at least one highly relevant feature.* Therefore, as shown in Line 22 of Algorithm 1, if the intersection of a feature interaction and *HRF* is an empty set, it will be filtered out.

4.4 Minimize examination of potential feature interactions

Lines 16-27 in Algorithm 1 aim at avoiding duplicated potential feature interaction examinations in two ways. First, for two suspicious feature selections generated based on any two failed configurations, one of them may be a subset of the other, namely, $\exists e_i, e_j \in \mathcal{FP} (i \neq j), \mathcal{U}(e_i) \subset \mathcal{U}(e_j)$ or $\mathcal{U}(e_j) \subset \mathcal{U}(e_i)$. This relationship among suspicious feature selections is called the *inclusion relationship*, and removing included suspicious feature selections can reduce the potential feature interactions that are generated repeatedly. Second, any two suspicious feature selections that do not have the inclusion relationship could produce the same potential feature interactions. That is to say, for e_i and e_j, $\exists FI_k \in \mathcal{U}(e_i), \exists FI_l \in \mathcal{U}(e_j)$, such that $FI_k = FI_l$, where FI_k and FI_l are two potential feature interactions. This means that duplicated suspiciousness examinations are performed for the same potential feature interactions. However, these two findings are ignored in VarCop, and therefore its search efficiency is affected.

If there is an inclusion relation between any two different suspicious feature selections, the included one should be removed. For example, there are three suspicious feature selections, $\mathcal{D}_1 = \{f_1, f_2, f_3, -f_4\}$, $\mathcal{D}_2 = \{-f_1, -f_2, f_3, -f_4\}$, and $\mathcal{D}_3$=$\{-f_1, -f_2, f_3, -f_4, -f_5\}$. Apparently, $\mathcal{D}_3$ includes $\mathcal{D}_2$. As a result, $\mathcal{D}_2$ is removed to avoid duplicated feature interactions being generated. As will be shown in Section 6.1, this phenomenon is common in our context.

Next, the 1~7-way feature interactions of each suspicious feature selection are generated and examined for suspiciousnes. However, we further observe that identical feature interactions are generated based on different suspicious feature selections. For instance, the potential feature interactions generated by $\mathcal{D}_1$ and $\mathcal{D}_3$ are {..., (f_2, f_3), (f_3, $-f_4$), ...} and {..., (f_3, -f_4), (-f_4, -f_5), ...}, respectively. It is worth noting that we only use 2-way feature interactions as an example. $\mathcal{D}_1$ and $\mathcal{D}_3$ have common feature interactions $\{(f_3, -f_4)\}$, and it should be examined only once for suspiciousness. Therefore, a caching mechanism which is implemented by the hash map is used to avoid duplicated suspiciousness examination for feature interactions.

5 Experiment setup

This section details the experimental setup, encompassing research questions, datasets, baselines, and evaluation metrics. The hardware specifications utilized for the experiments include a Linux-based server equipped with an Intel Core i5-12450H CPU running at 2.00 GHz and 16.00 GB of RAM.

5.1 Research Questions

We aim to answer the following research questions (RQs).

RQ1: *How much is the search space reduced in CRFL?*

RQ2: *How efficient is CRFL in comparison with the state-of-the-art approach?*

RQ3: *How effective is CRFL when extended for statement-level fault localization?*

RQ4: *Which component of CRFL contributes the most to efficiency gains?*

To address RQ1, we count the number of the included suspicious feature selections and duplicated feature interactions, to demonstrate how many redundant suspiciousness examinations can be eliminated. In addition, we assess the size of the search space by quantifying the size of feature interactions examined by CRFL, and then calculate the reduction rate (*Rate*) relative to VarCop. To answer RQ2, we compare CRFL with the state-of-the-art VarCop in terms of examination time. To address RQ3, we extend our approach for statement-level fault localization by using the program slicing and ranking techniques as in VarCop. To answer RQ4, we develop three variants of CRFL by removing key components: CRFL without the symmetric uncertainty technique (CRFL-wSU), CRFL without the two findings (CRFL-wTF), and CRFL without the counterfactual reasoning technique (CRFL-wCR). We then compare these variants in terms of their runtime to determine which component contributes the most to efficiency gains.

5.2 Dataset

CRFL is evaluated using eight real Java SPL systems, which are widely used in SPL studies [6, 37]. For each system, each sampled configuration has multiple buggy mutants, each of which has a corresponding unit-level test suite. A configuration is failed whenever there is a failed test suite for that configuration. The details of the dataset are presented in Table 1, where $\mathcal{N}$ indicates the number of single-bug mutants; $\mathcal{M}$ represents the number of multiple-bug mutants; $|\mathcal{SC}|$ is the size of the sampled configurations; *Cov* denotes the average statement coverage of unit-level test suites; and *LoC* is the lines of code. This dataset encompasses 343 single-bug mutants and 1006 multiple-bug mutants across eight systems. Note that mutants for `BerkeleyDB` and `TankWar` were generated by ourselves following [37]. First, we sample a set of valid configurations using the `SamplingCA` [3] tool and then compose the corresponding product systems using the `FeatureHouse` [4] framework. Subsequently, we seed some random bugs into these systems and generate the unit-level test suite for each configuration using the `Evosuite` [5] tool. The unit-level test suites are then executed to collect feature-level data, i.e., the configuration-level test suite. The remaining six systems are sourced from [37].

Table 1. The statistics of the used eight Java SPL systems

System	$\vert\mathcal{F}\vert$	$\mathcal{N}$	$\mathcal{M}$	$\vert\mathcal{SC}\vert$	*Cov*	*LoC*
BankAccountTP	8	73	298	34	99.9	143
BerkeleyDB	99	0	2	17	73.6	58030
Email	9	36	55	27	97.7	439
ExamDB	8	49	214	8	99.5	513
Elevator	6	20	26	18	92.9	854
GPL	27	105	267	99	99.4	1944
TankWar	31	0	5	26	63.1	4845
ZipMe	13	55	139	25	42.9	3460

It is important to note that RQ1, RQ2, and RQ4 focus on feature-level validation, which is independent of whether a mutant involves a single bug or multiple bugs. Therefore, we utilize the multiple-bug mutants of `BerkeleyDB` and `TankWar` along with the single-bug mutants of the other systems to validate these research questions. Furthermore, considering the phenomenon of false-passing configurations, we only use the single-bug and multiple-bug mutants from `BankAccountTP`, `Email`, `ExamDB`, `Elevator`, `GPL`, and `ZipMe` to address RQ3, as they are reported to include false-passing configurations [40].

[3]https://github.com/chuanluocs/SamplingCA.git
[4]http://www.fosd.de/fh
[5]https://www.evosuite.org/

5.3 Baselines

In our work, SBFL [2, 3, 30, 36, 42], S-SBFL [11, 33], FB [9], and VarCop [39] are selected as baselines. They represent the state-of-the-art approaches in fault localization for SPLs, and a brief description of them is provided below:

- SBFL [2, 3, 30, 36, 42], a classic fault localization technique that directly ranks code statements based on the program spectrum.
- S-SBFL [11, 33], an improved SBFL approach based on the slicing technique. Therefore, it has a more robust performance for fault localization.
- FB [9], a feature-based fault localization approach that utilizes SBFL to compute and rank the suspiciousness of each feature interaction.
- VarCop [39], a novel variability fault localization technique leveraging feature interactions and spectral information, achieving state-of-the-art performance.

5.4 Metrics

In our work, we evaluate the performance of fault localization in two aspects, i.e., efficiency and accuracy. Metrics of efficiency are used to address RQ1 , RQ2, and RQ4, while metrics of accuracy are used to address RQ3.

To evaluate efficiency, the runtime ($Time$) is employed as a critical metric. Given that each system comprises multiple mutants, the stability of the approaches is assessed by calculating the standard deviation (STD) of runtime across different mutants. Furthermore, the size of generated potential feature interaction sets, which affects the size of the search space, serves as another key metric for evaluating efficiency. In addition, to evaluate the validity of the two findings, two metrics are designed based on the given configuration set: the *inclusion rate*, which quantifies the proportion of suspicious feature selections from failed configurations that are incorporated into other configurations, and the *duplication rate*, which measures the number of identical potential feature interactions examined across different suspicious feature selections. The inclusion rate (IR) is calculated as:

$$IR = \frac{|\Upsilon|}{|\mathcal{FC}|}, \tag{9}$$

where $|\Upsilon|$ indicates the number of included suspicious feature selections, and $\mathcal{FC}$ is the set of failed configurations. The duplication rate (DR) can be calculated as:

$$DR = \frac{|\varepsilon|}{|\Theta|}. \tag{10}$$

where $|\varepsilon|$ denotes the number of duplicated potential feature interactions, and $|\Theta|$ is the total number of 1~7-way feature interactions.

To evaluate accuracy, we employ four widely-used metrics for statement-level fault localization: *Rank*, $EXAM$, $Hit@X$, and PBL. Each of these metrics provides distinct insights into the accuracy of the fault localization process. These are a brief description of these metrics.

- *Rank* is the rank of the buggy statement among all suspicious statements. A lower *Rank* indicates more accurate fault localization.
- $EXAM$ denotes the percentage of the number of examined statements before detecting the buggy statements [59].
- $Hit@X$ indicates the number of mutants in which the first X examinations are able to detect the buggy statement. A higher number of mutants with a smaller X means better fault localization accuracy. In our work, we only focus on $X \in [1, 5]$.

- Proportion of Bugs Localized (*PBL*) is a commonly used metric for evaluating multiple-bug mutants, representing the proportion of buggy statements identified when a certain number of statements are detected. A higher *PBL* value indicates a more effective approach.

6 Results

This section presents a series of experimental results on the eight systems to address the research questions.

6.1 RQ1: The reduced search space in CRFL

Fig. 5 shows, in the form of boxplots, the inclusion rate for all the buggy mutants on each of the eight systems. In order to enhance the interpretation of the data in Fig. 5, we take *BankAccountTP* as an example. From Table 1, this system has 73 single-bug mutants, for each of which the inclusion rate can be calculated, resulting in 73 data points (shown in a boxplot). According to Fig. 5, most systems have an average inclusion rate greater than zero, meaning that the inclusion relationship indeed exists on these systems. Moreover, the average rates are relatively high (i.e., between 16.2% and 57.3%) on `BankAccountTP`, `Elevator`, `Email`, `GPL`, and `ZipMe`. For the other three systems, the inclusion rate is zero in most mutants. One possible explanation is that the inclusion rate is sensitive to the size of sampled configurations. For `BerkeleyDB`, `ExamDB`, and `TankWar`, fewer configurations are sampled than other systems. When the number of sampled configurations is small, the number of suspicious feature selections also decreases, significantly reducing the likelihood of an inclusion relationship among them.

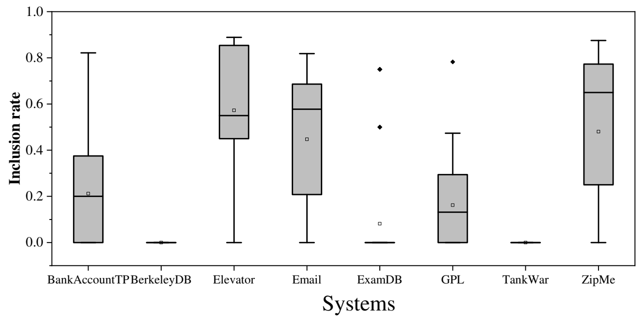


Fig. 5. Ratio of included suspicious feature selections across the eight systems.

Furthermore, the duplication rate all the buggy mutants across the eight systems is presented in Fig. 6. The average duplication rate for most systems is greater than zero, indicating the presence of a substantial number of duplicated potential feature interactions in suspicious feature selections. Notably, a high duplication rate is observed in `BankAccountTP`, `Elevator`, `Email`, `GPL`, and `ZipMe`, with average duplication rates of 47.9%, 54.9%, 44.5%, 43.6%, and 47.8%, respectively. In contrast, the average duplication rate of `BerkeleyDB` and `TankWar` is less than 20%, because the excessive number of features generates a huge potential feature interaction space. Additionally, most mutants in `ExamDB` exhibit a duplication rate of zero. This is likely due to the sparse suspicious feature selections generated from an extremely limited number of configurations, leading to a significantly small number of potential feature interactions.

Finally, the average number of potential feature interactions (i.e., the size of search space) generated by VarCop and CRFL is presented in Table 2. It includes 1~7-way feature interactions for each system, along with the reduction rate (*Rate*). Due to the large number of potential feature interactions in `BerkeleyDB`, only 1~4-way feature interactions are compared. From Table 2, it

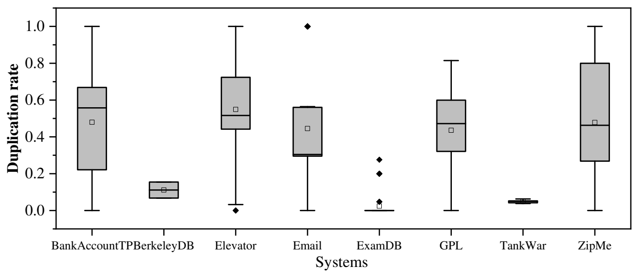


Fig. 6. Duplication rate of feature interactions across the eight systems.

is evident that CRFL examines fewer 1~7-way feature interactions on each system compared to VarCop. In addition, the advantage of CRFL is more noticeable for larger systems. For systems with fewer than ten features, such as `BankAccountTP`, `Email`, and `ExamDB`, CRFL reduces examinations by 33.4%~58.8% compared to VarCop. For larger systems, such as `ZipMe`, `GPL`, `TankWar`, and `BerkeleyDB`, CRFL can reduce the search space by about 84%, 88%, 73% and 71% compared to VarCop, respectively.

Table 2. The average number of 1~7-way feature interactions examined by CRFL and VarCop

Systems	*n*-way	VarCop	CRFL	*Rate* (%)	Systems	*n*-way	VarCop	CRFL	*Rate* (%)
	1-way	55.97	2.42	95.7		1-way	7.61	4.69	38.4
	2-way	141.42	17.41	87.7		2-way	20.18	10.90	46.0
	3-way	208.68	52.25	75.0		3-way	32.02	15.31	52.2
BankAccountTP	4-way	187.66	75.36	59.8	ExamDB	4-way	31.41	14.29	57.8
	5-way	102.26	52.88	48.3		5-way	18.67	8.39	54.5
	6-way	31.18	19.03	39.0		6-way	6.18	2.76	55.3
	7-way	4.10	2.73	33.4		7-way	0.88	0.39	55.7
	1-way	856.00	183.00	78.6		1-way	432.50	9.10	97.9
	2-way	40310.00	12364.00	69.3		2-way	4492.66	241.52	94.6
	3-way	1252994.50	410817.00	67.2		3-way	30989.59	2897.80	90.6
BerkeleyDB	4-way	28919106.00	8833702.00	69.5	GPL	4-way	156563.35	21091.91	86.5
	5-way	-	-	-		5-way	612209.61	101700.66	83.4
	6-way	-	-	-		6-way	1917635.11	368773.49	80.8
	7-way	-	-	-		7-way	4923805.55	1027440.15	79.1
	1-way	18.60	1.40	92.5		1-way	186.00	34.00	81.7
	2-way	25.20	4.60	81.7		2-way	1692.40	461.60	72.7
	3-way	19.75	5.90	70.1		3-way	9786.80	3136.40	68.0
Elevator	4-way	8.05	3.40	57.8	TankWar	4-way	40254.40	12796.00	68.2
	5-way	1.35	0.75	44.4		5-way	125001.40	36024.40	71.2
	6-way	0.00	0.00	-		6-way	303782.00	78203.00	74.3
	7-way	0.00	0.00	-		7-way	591282.40	135145.80	77.1
	1-way	64.86	3.67	94.3		1-way	66.20	3.71	94.4
	2-way	165.28	20.64	87.5		2-way	195.04	23.29	88.1
	3-way	266.61	57.33	78.5		3-way	403.49	69.36	82.8
Email	4-way	285.31	86.39	69.7	ZipMe	4-way	622.65	121.45	80.5
	5-way	203.19	74.86	63.2		5-way	727.91	143.36	80.3
	6-way	92.94	38.25	58.8		6-way	642.78	122.29	81.0
	7-way	24.75	10.81	56.3		7-way	421.20	76.75	81.8

It is essential to analyze the reasons behind the significant reduction in the search space observed in CRFL. For low-way feature interactions (e.g., 1~2-way), the average number of suspiciousness examinations performed by CRFL is less than VarCop, with the average *Rate* exceeding 71% across the eight systems. This substantial reduction can be attributed to the highly relevant feature set identified through symmetric uncertainty. Low-way feature interactions that are not in this set are filtered directly. Consequently, irrelevant 1∼2-way feature interactions are effectively filtered, reducing computational overhead. For high-order feature interactions (e.g., 3∼7-way interactions), CRFL examines fewer potential feature selections while achieving higher accuracy. This improvement stems from CRFL's ability to precisely identify suspicious feature selections through counterfactual reasoning, enhancing its effectiveness in reducing unnecessary computations while maintaining accuracy.

Therefore, we have the following answers to RQ1. Compared to the state-of-the-art VarCop, the search space in CRFL is reduced above 71% on average. This reduction is achieved because CRFL examines more suspicious and fewer potential feature interactions, avoiding the repeated generation and examination of the same feature interactions.

6.2 RQ2: Efficiency of CRFL

Table 3 presents the average runtime (in seconds) of CRFL and VarCop across the eight systems, along with the runtime ratio of VarCop to CRFL. In addition, the standard deviations (STD) are also provided. A lower STD indicates a more stable approach. As shown in Table 3, CRFL outperforms VarCop in both average runtime and STD. Furthermore, CRFL exhibits significantly lower STD values compared to VarCop, further highlighting its stability. It is noteworthy that CRFL runs considerably faster than VarCop on all systems, with the speedup ratio ranging from 5.5 to 31.4. Specifically, for smaller systems with fewer than ten features, such as `BankAccountTP`, `Email`, `ExamDB`, and `Elevator`, CRFL is at least 17.6 times faster than VarCop. For larger systems, such as `GPL` and `BerkeleyDB`, both approaches require more time due to the exponentially growing search space. However, CRFL remains faster than VarCop. On average, CRFL is 22.2 times faster on smaller systems and 8.9 times faster on larger ones. These results demonstrate the efficiency of CRFL as a fault localization approach.

Table 3. Comparisons between CRFL and VarCop in terms of the average runtime (in seconds) and standard deviation (STD)

Systems	VarCop		CRFL		*Ratio*
	Average	*STD*	Average	*STD*	
BankAccountTP	3.17	1.61	**0.18**	**0.07**	17.6
BerkeleyDB	179425.64	**8095.63**	**21551.35**	16359.10	8.3
Email	16.34	8.79	**0.89**	**0.42**	18.4
ExamDB	2.83	0.97	**0.09**	**0.03**	31.4
Elevator	40.28	18.74	**1.88**	**0.76**	21.4
GPL	3590.49	3127.89	**649.20**	**433.52**	5.5
TankWar	96.95	63.81	**15.91**	**10.35**	6.1
ZipMe	1150.00	706.35	**72.94**	**41.89**	15.8

Therefore, the answer to RQ2 is clear. The proposed CRFL, which runs at least 5.5 times faster than the state-of-the-art VarCop, is efficient and stable as a feature interaction fault localization approach for SPLs.

6.3 RQ3: Localization accuracy of CRFL

In our work, $Rank$, $EXAM$, and $Hit@X$ are used for single-bug mutant evaluation, while PBL is used for multi-bug mutants. For the $Hit@X$ and PBL metrics, the most suitable ranking criteria from the available $Rank$ metrics are selected for evaluation.

The results of average *Rank* and *EXAM* for CRFL and four baselines are presented in Table 4, where the column "Metric" represents three popular ranking metrics used in SBFL. As shown in Table 4, CRFL outperforms the baselines in terms of average *Rank* and *EXAM* regarding the three ranking metrics. Compared to SBFL and S-SBFL, CRFL demonstrates a clear advantage, particularly on the Elevator and GPL systems. The localization accuracy of FB is significantly lower than that of CRFL for both Rank and EXAM, as it does not extend to statement-level fault localization. Compared to VarCop, CRFL shows a substantial improvement on GPL, while achieving similar or slightly better performance on the other five systems. This can be attributed to the fact that larger systems generally contain a greater number of suspicious statements, and CRFL, leveraging counterfactual reasoning techniques, effectively isolates a smaller but more precise set of suspicious statements, thereby enhancing localization performance.

Table 4. Comparison of average *Rank* and *EXAM* between CRFL and baselines on the six systems

Metric	Systems	*Rank*					*EXAM*				
		CRFL	VarCop	S-SBFL	SBFL	FB	CRFL	VarCop	S-SBFL	SBFL	FB
Dstar	BankAccountTP	**3.73**	3.81	4.03	3.92	20.52	**4.86**	4.98	5.26	5.11	26.84
	Email	**3.42**	3.58	3.75	4.61	60.58	**1.38**	1.45	1.52	1.87	24.53
	ExamDB	**3.04**	3.29	3.29	3.29	26.57	**1.21**	1.31	1.31	1.31	10.59
	Elevator	**4.25**	4.25	4.00	8.30	92.05	**0.95**	0.95	0.89	1.85	20.55
	GPL	**5.50**	6.72	9.80	9.09	47.97	**0.57**	0.70	1.01	0.94	4.97
	ZipMe	**12.82**	12.82	14.71	18.20	452.87	**0.55**	0.55	0.63	0.78	19.52
Op2	BankAccountTP	**3.47**	3.51	3.71	3.58	19.77	**4.52**	4.58	4.84	1.80	25.84
	Email	**3.72**	3.75	4.06	4.03	54.89	**1.51**	1.52	1.64	1.63	22.22
	ExamDB	**3.00**	3.24	3.24	3.24	26.57	**1.20**	1.29	1.29	1.29	10.59
	Elevator	**3.70**	**3.70**	4.15	4.25	91.10	**0.83**	**0.83**	0.93	0.95	20.33
	GPL	**5.52**	6.74	8.90	11.48	136.27	**0.57**	0.70	0.92	3.15	21.79
	ZipMe	**10.96**	**10.96**	12.38	12.67	447.44	**0.47**	**0.47**	0.53	0.55	19.29
Tarantula	BankAccountTP	**3.74**	3.84	4.25	5.49	28.64	**4.88**	5.01	5.54	7.17	37.52
	Email	**3.92**	4.31	4.31	13.61	99.64	**1.59**	1.74	1.74	5.51	40.34
	ExamDB	**5.10**	5.31	5.38	4.65	38.06	**2.03**	2.11	2.15	1.85	15.16
	Elevator	**6.10**	6.15	6.75	18.40	102.25	**1.36**	1.37	1.51	4.11	22.82
	GPL	**5.79**	7.20	9.61	10.36	63.12	**0.60**	0.75	0.99	1.07	6.53
	ZipMe	**13.82**	**13.82**	15.62	23.73	542.62	**0.60**	**0.60**	0.67	1.02	23.39

Furthermore, regardless of the ranking metric used, CRFL consistently achieves high localization accuracy across most mutants. The choice of ranking metric influences the localization accuracy of both CRFL and the baseline methods. For instance, Dstar performs better on `Email` and `GPL`, whereas Op2 is more effective for `ZipMe`. Overall, CRFL and all baseline approaches exhibit improved localization performance when using the Op2 metric, suggesting that it is better suited for the evaluated systems. This adaptability enhances the ability to accurately identify buggy statements. Therefore, we further analyze *Hit@X* and *PBL* using the Op2 metric.

Moreover, comparisons of $Hit@1 \sim Hit@5$ among CRFL, SBFL, S-SBFL, and VarCop are presented in Fig. 7. Our approach is able to detect buggy statements on the first investigation in 35.5% of all buggy mutants. In contrast, SBFL, S-SBFL, and VarCop rank buggy statements in the top 1 in fewer than 30% of mutants. Furthermore, CRFL ranks buggy statements within the top 2 in over half of the mutants, demonstrating a significant improvement over the other three approaches. CRFL ranks buggy statements in the top 2 in more than half of the mutants, which is significantly better than the other three approaches. The advantage of CRFL is particularly evident at *Hit*@4, where developers using CRFL have a 76.6% probability of identifying buggy statements by examining

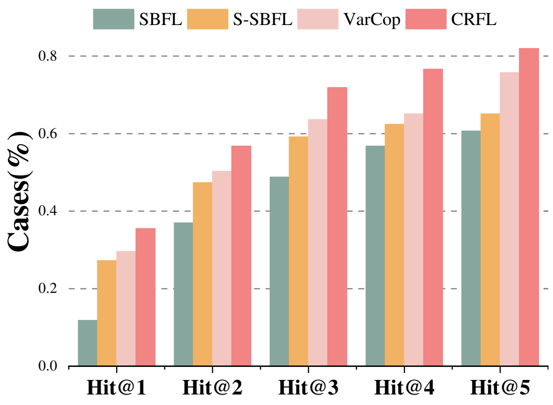


Fig. 7. Comparison of $Hit@1 \sim Hit@5$ for CRFL, SBFL, S-SBFL and VarCop.

only the top four suspicious statements. The results for $Hit@1 \sim Hit@5$ further confirm that CRFL provides more precise feature-level fault localization compared to the baselines.

Finally, for multiple-bug mutants, Fig. 8 presents the comparison results of *PBL*. Among the evaluated approaches, SBFL exhibits the lowest fault localization accuracy, followed by S-SBFL. VarCop and CRFL achieve comparable accuracy, as both employ the same program slicing and statement ranking techniques. However, as shown in Section 6.1, CRFL demonstrates significantly higher efficiency than VarCop. Therefore, CRFL not only preserves localization accuracy but also substantially improves fault localization efficiency, making it a key advantage of the proposed approach.

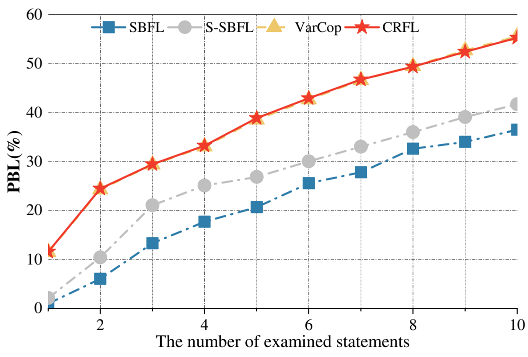
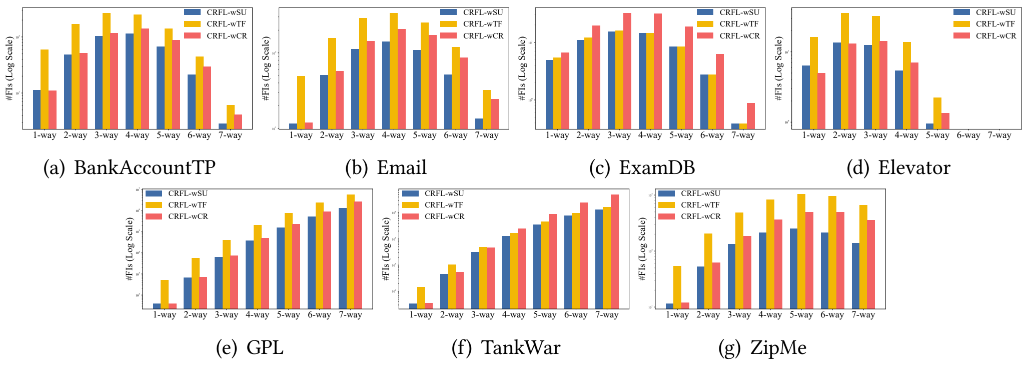


Fig. 8. Comparison of *PBL* among CRFL, SBFL, S-SBFL, and VarCop.

We can reach the following answers to RQ3. When extended for statement-level fault localization, CRFL can examine at least 80% and 35% of buggy statements in single-bug and multi-bug mutants during the first five examinations, respectively. Therefore, CRFL ensures effectiveness for statement-level localization.

6.4 RQ4: Ablation analysis

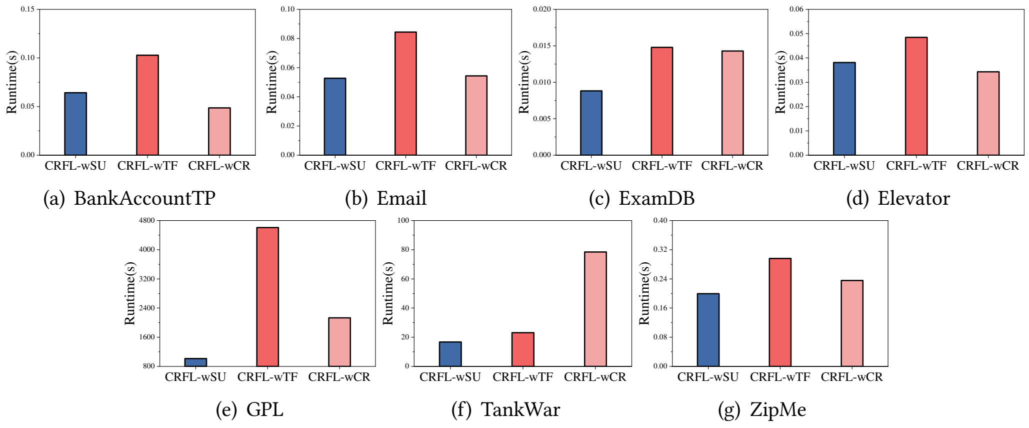
Fig. 9 presents a comparison of the average number of 1∼7-way feature interactions examined by the three variants of CRFL. Notably, CRFL-wSU and CRFL-wCR are excluded from the analysis for `BerkeleyDB` due to the caching technique, which causes server memory overflow. The results indicate that the two findings have the most significant impact on CRFL's efficiency, as their removal (CRFL-wTF) results in the largest number of examined feature interactions across most systems. However, ExamDB and TankWar deviate from this trend. In ExamDB, the counterfactual reasoning technique contributes the most to efficiency gains, likely because ExamDB has a lower inclusion rate and duplication rate, reducing the effectiveness of the two findings. In TankWar, the two findings are more effective for 1~3-way feature interactions, while counterfactual reasoning technique is more beneficial for 4~7-way interactions, as duplicate feature interactions in TankWar mostly occur at lower interaction levels. Generally, the same trend is observed in the average runtime results shown in Fig. 10, where the two findings significantly enhance efficiency in most systems, while the counterfactual reasoning technique plays a more critical role in TankWar.



(a) BankAccountTP (b) Email (c) ExamDB (d) Elevator

(e) GPL (f) TankWar (g) ZipMe

Fig. 9. The average number of 1~7way feature interactions examined by three variants of CRFL.



(a) BankAccountTP (b) Email (c) ExamDB (d) Elevator

(e) GPL (f) TankWar (g) ZipMe

Fig. 10. Comparison of runtime for three variants of CRFL on seven systems.

In summary, regarding RQ4, the two findings contribute most significantly to CRFL, particularly in systems with high inclusion and duplication rates. Notably, both the counterfactual reasoning and symmetric uncertainty techniques play equally essential roles in enhancing CRFL's efficiency. The synergy among these three components enables CRFL to identify buggy feature interactions more effectively while reducing computational cost.

6.5 Threats to Validity

In this section, we present a brief discussion on internal and external validity, and provide simple views on how to mitigate these threats.

Internal validity. This type of threat may arise from potential errors in the implementation of CRFL and the baselines used for comparison. To address this, we conducted extensive testing on our code to identify and correct any errors. This included a thorough examination of the results through a systematic analysis of small SPL systems. For the baselines used in the experimental comparisons, we directly utilized the code provided by their authors.

External validity. A noteworthy limitation arises from our underlying assumption that buggy configurations will invariably manifest detectable failures through test suites. While our empirical evaluation demonstrates strong correlation between feature-level fault and test failures in the studied scenarios, we acknowledge the theoretical possibility of silent configuration faults—scenarios where defective configurations do not trigger immediate test failures. This limitation is inherent to any testing-based validation approach, as documented in prior work on configuration error detection [40]. This threat is related to the dataset used for the experiments, i.e., to verify that the accuracy of our approach is comprehensive. To mitigate this threat, we selected six SPL systems containing false-passing configurations to evaluate our approach, and these systems are publicly available [39].

6.6 Related work

In this section, related techniques are briefly described, including fault localization and counterfactual reasoning.

Feature-level fault localization: Feature-level fault localization aims to examine buggy feature interactions, which can help enhance fault comprehension and identification at an abstract level. As a fundamental component of statement-level fault localization, it serves a complementary role, together enabling a more comprehensive fault localization approach in configurable software systems.

For non-functional feature-level faults, common solution techniques include search-based algorithms [54], machine learning [48, 49, 55], and causal inference algorithms [20, 28, 35, 61]. For example, Valle *et al.* [54] proposed a search-based method for automatically correcting incorrect parameters. This method is designed to alleviate the tedious and time-consuming task of manually configuring cyber-physical system parameters. Another way is to use machine learning techniques to calculate the impact weight of each feature on the metrics, with higher weighted features resulting in a higher probability of failure [48, 49, 55]. In particular, several recent studies have concluded that causal techniques have a significant advantage in detecting the root cause of buggy configurations [20, 28, 35, 61].

In contrast, there has been limited research on functional feature-level fault localization in SPLs. Arrieta *et al.* [9] analyzed each failed configuration and calculated the suspiciousness of each feature using the program spectrum technique. Their study confirms that single-system fault localization techniques can be applied in SPL systems. Building on this idea, Nguyen *et al.* [39] proposed an approach to examine suspicious feature interactions based on the relationship between partial configurations and test results, along with a tool that combines program spectrum and program slicing techniques to locate buggy statements.

It is worth noting that our approach is built upon VarCop, where feature-level fault localization is achieved by enumeratively mining suspicious feature selections. In contrast, CRFL employs counterfactual reasoning techniques to obtain more precise suspicious feature selections, thereby reducing the search space. Furthermore, CRFL significantly enhances the efficiency of examining suspicious feature interactions by leveraging the two findings and symmetric uncertainty techniques. Therefore, our approach improves both the efficiency and accuracy of statement-level localization by narrowing the range of suspicious statements.

Counterfactual reasoning: Counterfactual reasoning has been used successfully in software engineering, including single-system software [10, 21, 29], neural models [17, 22], and configurable software [20, 27]. Counterfactual reasoning can be used in single-system software for program analysis and fault localization. For example, Baah *et al.* [10] applied the probabilistic graphical model to program dependency graphs and used counterfactual reasoning techniques to estimate the effect of statements on test results. Compared to traditional single-system software, neural models are a specific type of software within the deep learning category, and their faults are more specialized, such as when the LOSS fails to converge. Gao *et al.* [22] proposed a counterfactual reasoning-based framework to model and identify the impact of neural models. Their work explicitly captures the misleading information of identifiers and reduces its impact. Similarly, some tasks in configurable systems can be addressed using counterfactual reasoning. For instance, Clemens *et al.* [20] employed counterfactual reasoning to achieve root cause localization of non-functional faults in feature interactions within configurable systems.

Different from existing researches, we use counterfactual reasoning to efficiently detect functional buggy feature interactions for SPLs.

6.7 Data availability

The source code of our tool CRFL and the benchmarks used in evaluations are available at https://github.com/Songluhaining/CRFL.git.

7 Conclusions

This paper proposes a high-performing and low-cost approach for functional feature interaction fault localization in SPLs by reducing the search space with counterfactual reasoning and eliminating redundant feature interaction examinations based on two key empirical findings. Extensive experiments demonstrate that CRFL outperforms the state-of-the-art VarCop, achieving a smaller and more precise search space while localizing suspicious feature interactions more efficiently. Furthermore, when extended to statement-level localization, our approach accurately identifies buggy statements in both single and multiple buggy mutants. These results highlight the effectiveness of our approach in enhancing fault localization for functional feature interactions in SPLs.

Since CRFL requires only configurations and corresponding test results as input, it is independent of specific programming languages and can be considered a general feature interaction fault localization method. However, our current experiments focus on Java systems due to the availability of Java-based datasets in the SPL fault localization domain. Additionally, when extending CRFL to statement-level fault localization, we employ a Java-specific slicing technique.

To further enhance CRFL's applicability, we plan to extend it to systems using other programming languages, such as C. Moreover, statement-level fault localization in SPLs remains an important research direction. We will further evaluate CRFL's effectiveness at the statement level, especially in systems with different programming languages. Finally, extending CRFL to identify non-functional feature interaction faults in configurable systems is another promising avenue for future research.

Acknowledgments

This work is supported by the Guangdong Basic and Applied Basic Research Foundation (No. 2024A1-515030022); the Fundamental Research Funds for the Central Universities (No. 2024ZYGXZR097); the Research and Development Project on Key Technologies for Intelligent Sensing and Analysis of Urban Events Based on Low-Altitude Drones(No. 2024BQ010011).

References

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, et al. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *ASE'18*. 143–154. doi:10.1145/3238147.3238192

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION 2007*. IEEE, 89–98. doi:10.1109/TAIC.PART.2007.13

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *ASE'09*. IEEE, 88–99. doi:10.1109/ASE.2009.25

[4] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, et al. 2016. Tool demo: testing configurable systems with featureIDE. In *GPCE'16*. 173–177. doi:10.1145/2993236.2993254

[5] Vander Alves, Nan Niu, Carina Alves, et al. 2010. Requirements engineering for software product lines: A systematic literature review. *Inf. Softw. Technol.* 52, 8 (2010), 806–820. doi:10.1016/j.infsof.2010.03.014

[6] Sven Apel, Christian Kastner, and Christian Lengauer. 2009. Featurehouse: Language-independent, automated software composition. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 221–231. doi:10.1109/ICSE.2009.5070523

[7] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, et al. 2013. Exploring feature interactions in the wild: the new feature-interaction challenge. In *FOSD'13*. 1–8. doi:10.1145/2528265.2528267

[8] Sven Apel, Hendrik Speidel, Philipp Wendler, et al. 2011. Detection of feature interactions using feature-aware verification. In *ASE'11*. IEEE, 372–375. doi:10.1109/ASE.2011.6100075

[9] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, et al. 2018. Spectrum-based fault localization in software product lines. *Inf. Softw. Technol.* 100 (2018), 18–31.

[10] George K Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *ISSTA'10*. 73–84. doi:10.1145/1831708.1831717

[11] Nazanin Bayati Chaleshtari and Saeed Parsa. 2020. SMBFL: slice-based cost reduction of mutation-based fault localization. *Empir. Softw. Eng.* 25 (2020), 4282–4314.

[12] Thorsten Berger, Daniela Lettner, Julia Rubin, et al. 2015. What is a feature? a qualitative study of features in industrial software product lines. In *SPLC'15*. 16–25.

[13] Léon Bottou, Jonas Peters, Joaquin Quiñonero-Candela, Denis X Charles, et al. 2013. Counterfactual Reasoning and Learning Systems: The Example of Computational Advertising. *J. Mach. Learn. Res.* 14, 11 (2013).

[14] Muffy Calder, Mario Kolberg, Evan H Magill, et al. 2003. Feature interaction: a critical review and considered forecast. *Comput. Netw.* 41, 1 (2003), 115–141.

[15] E Jane Cameron and Hugo Velthuijsen. 1993. Feature interactions in telecommunications systems. *IEEE Commun. Mag.* 31, 8 (1993), 18–23.

[16] Pei Cao and Zhe Wang. 2004. Efficient top-k query calculation in distributed networks. In *PODC'04*. 206–215.

[17] Jürgen Cito, Isil Dillig, Vijayaraghavan Murali, et al. 2022. Counterfactual explanations for models of code. In *ICSE-SEIP'22*. 125–134. doi:10.1145/3510457.3513081

[18] Paul Clements and Linda Northrop. 2002. *Software product lines.* Addison-Wesley Boston.

[19] Manoranjan Dash and Huan Liu. 1997. Feature selection for classification. *Intelligent data analysis* 1, 1-4 (1997), 131–156.

[20] Clemens Dubslaff, Kallistos Weis, Christel Baier, et al. 2022. Causality in configurable software systems. In *ICSE'22*. 325–337. doi:10.1145/3510003.3510200

[21] Carlo A Furia, Richard Torkar, and Robert Feldt. 2023. Towards causal analysis of empirical software engineering data: The impact of programming languages on coding competitions. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2023), 1–35. doi:10.1145/3611667

[22] Shuzheng Gao, Cuiyun Gao, Chaozheng Wang, et al. 2023. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *ICSE'23*. IEEE, 1933–1945. doi:10.1109/ICSE48619.2023.00164

[23] Brady J Garvin and Myra B Cohen. 2011. Feature interaction faults revisited: An exploratory study. In *ISSRE'11*. IEEE, 90–99.

[24] Joseph Y Halpern and Judea Pearl. 2005. Causes and explanations: A structural-model approach. Part I: Causes. *Br. J. Philos. Sci.* (2005).

[25] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *ESEM'16*. 1–10. doi:10.1145/2961111.2962602

[26] Md Abir Hossen, Sonam Kharade, Bradley Schmerl, et al. 2023. CaRE: Finding Root Causes of Configuration Issues in Highly-Configurable Robots. *IEEE Robot. Autom. Lett.* (2023). doi:10.1109/LRA.2023.3280810

[27] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, et al. 2021. CADET: Debugging and fixing misconfigurations using counterfactual reasoning. *arXiv preprint arXiv:2010.06061* (2021).

[28] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, et al. 2022. Unicorn: reasoning about configurable system performance through the lens of causality. In *EuroSys'22*. 199–217.

[29] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal testing: understanding defects' root causes. In *ICSE'20*. 87–99. doi:10.1145/3377811.3380377

[30] Fabian Keller, Lars Grunske, Simon Heiden, et al. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *QRS'17*. IEEE, 114–125. doi:10.1109/QRS.2017.22

[31] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, et al. 2017. On the relation of external and internal feature interactions: A case study. *arXiv preprint arXiv:1712.07440* (2017).

[32] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.* 30, 6 (2004), 418–421.

[33] Xiangyu Li and Alessandro Orso. 2020. More accurate dynamic slicing for better supporting software debugging. In *ICST'20*. IEEE, 28–38.

[34] Sonia Montagud, Silvia Abrahão, and Emilio Insfran. 2012. A systematic review of quality attributes and measures for software product lines. *Softw. Qual. J.* 20 (2012), 425–486.

[35] Bryan J Muscedere, Robert Hackman, Davood Anbarnam, et al. 2019. Detecting feature-interaction symptoms in automotive software using lightweight analysis. In *SANER'19*. IEEE, 175–185. doi:10.1109/SANER.2019.8668042

[36] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 1–32.

[37] Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, et al. 2021. Variability fault localization: a benchmark. In *SPLC'21*. 120–125. doi:10.1145/3461001.3473058

[38] Son Nguyen. 2019. Configuration-dependent fault localization. In *ICSE-Companion'19*. IEEE, 156–158. doi:10.1109/ICSE-Companion.2019.00065

[39] Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen, et al. 2022. A Variability Fault Localization Approach for Software Product Lines. *IEEE Trans. Softw. Eng.* 48, 10 (2022), 4100–4118. doi:10.1109/TSE.2021.3113859

[40] Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. 2023. Detecting false-passing products and mitigating their impact on variability fault localization in software product lines. *Information and Software Technology* 153 (2023), 107080. doi:10.1016/j.infsof.2022.107080

[41] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *MSR'13*. IEEE, 237–246. doi:10.1109/MSR.2013.6624035

[42] Spencer Pearson, José Campos, René Just, et al. 2017. Evaluating and improving fault localization. In *ICSE'17*. IEEE, 609–620. doi:10.1109/ICSE.2017.62

[43] Mattia Prosperi, Yi Guo, Matt Sperrin, et al. 2020. Causal inference and counterfactual prediction in machine learning for actionable healthcare. *Nat. Mach. Intell.* 2, 7 (2020), 369–375. doi:10.1038/s42256-020-0197-y

[44] Guangzhi Qu, Salim Hariri, and Mazin Yousif. 2005. A new dependency and correlation analysis for features. *IEEE Trans. Knowl. Data Eng.* 17, 9 (2005), 1199–1207. doi:10.1109/TKDE.2005.136

[45] Silva Robak and Bogdan Franczyk. 2001. Feature interaction and composition problems in software product lines. In *ECOOP 2001*. Citeseer.

[46] SK Golam Saroar, Waseefa Ahmed, Elmira Onagh, and Maleknaz Nayebi. 2024. Github marketplace for automation and innovation in software production. *Information and Software Technology* 175 (2024), 107522. doi:10.1016/j.infsof.2024.107522

[47] Ramy Shahin, Murad Akhundov, and Marsha Chechik. 2022. Annotative Software Product Line Analysis Using Variability-Aware Datalog. *IEEE Trans. Softw. Eng.* 49, 3 (2022), 1323–1341.

[48] Tanuja Shailesh, Ashalatha Nayak, and Devi Prasad. 2018. Performance Prediction of Configurable softwares using Machine learning approach. In *iCATccT'18*. IEEE, 7–10.

[49] Norbert Siegmund, Alexander Grebhahn, Sven Apel, et al. 2015. Performance-influence models for highly configurable systems. In *FSE'15*. 284–294. doi:10.1109/iCATccT44854.2018.9001957

[50] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, et al. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Inf. Softw. Technol.* 98 (2018), 44–58. doi:10.1016/j.infsof.2018.01.016

[51] Xianfang Song, Yong Zhang, Dunwei Gong, et al. 2022. Surrogate sample-assisted particle swarm optimization for feature selection on high-dimensional data. *IEEE Trans. Evol. Comput.* (2022).

[52] Xian-Fang Song, Yong Zhang, Dun-Wei Gong, et al. 2021. A fast hybrid feature selection based on correlation-guided clustering and particle swarm optimization for high-dimensional data. *IEEE T. Cybern.* 52, 9 (2021), 9573–9586. doi:10.1109/TCYB.2021.3061152

[53] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference*. 54–65. doi:10.1145/3646548.3672590

[54] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Automated Misconfiguration Repair of Configurable Cyber-Physical Systems with Search: an Industrial Case Study on Elevator Dispatching Algorithms. *arXiv preprint*

arXiv:2301.01487 (2023). doi:10.1109/ICSE-SEIP58684.2023.00042

[55] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, et al. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *ICSE'21*. IEEE, 1072–1084. doi:10.1109/ICSE43902.2021.00100

[56] Alexander Von Rhein, Alexander Grebhahn, Sven Apel, et al. 2015. Presence-condition simplification in highly configurable systems. In *ICSE'15*, Vol. 1. IEEE, 178–188. doi:10.1109/ICSE.2015.39

[57] Tianxin Wei, Fuli Feng, Jiawei Chen, et al. 2021. Model-agnostic counterfactual reasoning for eliminating popularity bias in recommender system. 1791–1800. doi:10.1145/3447548.3467289

[58] Kallistos Weis, Leopoldo Teixeira, Clemens Dubslaff, and Sven Apel. 2024. Blackbox Observability of Features and Feature Interactions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1120–1132. doi:10.1145/3691620.3695490

[59] Eric Wong, Tingting Wei, Yu Qi, et al. 2008. A crosstab-based statistical method for effective fault localization. In *ICST'08*. IEEE, 42–51. doi:10.1109/ICST.2008.65

[60] Yi Xiang, Han Huang, Yuren Zhou, et al. 2022. Search-based diverse sampling from real-world software product lines. In *ICSE'22*. 1945–1957. doi:10.1145/3510003.3510053

[61] Yingnan Zhou, Xue Hu, Sihan Xu, et al. 2023. Multi-misconfiguration Diagnosis via Identifying Correlated Configuration Parameters. *IEEE Trans. Softw. Eng.* (2023). doi:10.1109/TSE.2023.3308755

Received 2024-10-30; accepted 2025-03-31