# Detecting False-passing Products and Mitigating their Impact on Variability Fault Localization in Software Product Lines

Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen and Hieu Dinh Vo*

*Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Vietnam*

ARTICLE INFO

*Keywords*:
False-passing products, variability bugs, coincidental correctness, fault localization, software product line

ABSTRACT

In a Software Product Line (SPL) system, variability bugs can cause failures in certain products (buggy products), not in the others. In practice, variability bugs are not always exposed, and buggy products can still pass all the tests due to their ineffective test suites (so-called *false-passing* products). The misleading indications caused by those *false-passing* products' test results can negatively impact variability fault localization performance. In this paper, we introduce CLAP, a novel approach to detect *false-passing* products in SPL systems failed by variability bugs. Our key idea is that given a set of tested products of an SPL system, we collect failure indications in failing products based on their implementation and test quality. For a passing product, we evaluate these indications, and the stronger indications, the more likely the product is *false-passing*. Specifically, the possibility of the product to be false-passing is evaluated based on if it has a large number of the statements which are highly suspicious in the failing products, and if its test suite is in lower quality compared to the failing products' test suites. We conducted several experiments to evaluate our *false-passing* product detection approach on a large benchmark of 14,191 *false-passing* products and 22,555 *true-passing* products in 823 buggy versions of the existing SPL systems. The experimental results show that CLAP can effectively detect *false-passing* and *true-passing* products with the average accuracy of more than 90%. Especially, the precision of *false-passing* product detection by CLAP is up to 96%. This means, among 10 products predicted as *false-passing* products, more than 9 products are precisely detected. Furthermore, we propose two simple and effective methods to mitigate the negative impact of *false-passing* products on variability fault localization. These methods can improve the performance of the state-of-the-art variability fault localization techniques by up to 34%.

## 1. Introduction

Thorough testing is generally required to guarantee the quality of programs. However, it is often hard, tedious, and time-consuming to conduct thorough testing in practice. Various bugs could be neglected by the test suites since it is extremely difficult to cover all the programs' behaviors. Moreover, there are kinds of bugs which are challenging to be detected due to their difficulties in infecting the program states and propagating their incorrectness to the outputs [1]. Consequently, even when they reached the defects, there are test cases that still obtain correct outputs. Such test cases are called *coincidentally correct/passed* tests. Indeed, coincidental correctness is a prevalent problem in software testing [2], and this phenomenon causes a severely negative impact on fault localization performance [2, 3, 4].

Similar to testing in non-configurable code, the coincidental correctness phenomenon also happens in software product lines (SPL) and causes difficulties in finding faults in these systems. Specifically, for an SPL system, a set of products is often sampled for testing. Each sampled product is composed of a set of features of the system and tested individually by its test suite as a singleton program. For a buggy SPL system, the bugs could be in one or more products. Ideally, if a product contains bugs (*buggy products*), the bugs should be revealed by its test suite. In other words, there should be at least a failed test after testing. However, if the test suite of a buggy product is ineffective in detecting the bugs, the product's overall test result will be passing. For instance, the suite does not cover the product's buggy statements or those test cases could reach the buggy statements but could not propagate the incorrectness to the outputs, the product still passes all the tests. Such a passing product is indeed a buggy product, yet incorrectly considered as passing. That passing product is namely a *false-passing* product. Due to the unreliability of test results, these *false-passing* products might negatively impact the fault localization (FL) performance. In particular, the performance of two main spectrum-based FL strategies in SPL systems, *product-based* and *test case-based*, is affected.

First, the *product-based* FL techniques [5, 6] evaluate the suspiciousness of a statement in a buggy SPL system based on the appearance of the statement in failing and/or passing products. Specially, the key idea to find bugs in an SPL system is that a statement which is included in more failing products and fewer passing products is more likely to be buggy than the other statements of the system. Misleadingly counting a buggy product as a passing product incorrectly decreases the number of failing products and increases the number of passing products containing the buggy statement. Consequently, the buggy statement is considered less suspicious than it should be.

Second, the *test case-based* FL techniques [6, 7] measure the suspicious scores of the statements based on the numbers of failed and passed tests executed by them. Indeed, *false-passing* products could lead to under-counting the number

*Corresponding author
trang.nguyen@vnu.edu.vn (T. Nguyen); tuanngokien@vnu.edu.vn (K. Ngo); sonnguyen@vnu.edu.vn (S. Nguyen); hieuvd@vnu.edu.vn (H.D. Vo)
ORCID(s): 0000-0002-3596-2352 (T. Nguyen); 0000-0001-7136-7529 (K. Ngo); 0000-0002-8970-9870 (S. Nguyen); 0000-0002-9407-1971 (H.D. Vo)

of failed tests and over-counting the number of passed tests executed by the buggy statements. The reason is that *false-passing* products contain bugs, but there is no failed test. In these *false-passing* products, the buggy statements are not executed by any test, or they are reached by several tests, yet those tests *coincidentally* passed. Both low coverage test suite and coincidentally passed tests can cause inaccurate suspiciousness evaluation for the buggy statements.

In this paper, we introduce CLAP, a novel *false-passing* product detection approach for SPL systems that failed because of variability bugs (*buggy SPL systems* for short). The intuition of our approach is that for a buggy SPL system, the sampled products can share some common functionalities. If the unexpected behaviors of the functionalities are revealed by the tests in some (failing) products, the other products having similar functionalities are likely to be caused failures by those unexpected behaviors. In CLAP, *false-passing* products can be detected based on the *failure indications* which are collected by reviewing the implementation and test quality of the failing products. To evaluate the possibility that a passing product is a *false-passing* one, we propose several measurable attributes to assess the strength of these failure indications in the product. The stronger indications, the more likely the product is *false-passing*.

The proposed attributes are belonged to two aspects: *product implementation* (products' source code) and *test quality* (the adequacy and the effectiveness of test suites). The attributes regarding *product implementation* reflect the possibility that the passing product contains bugs. Intuitively, if the product has more (suspicious) statements executing the tests failed in the failing products of the system, the product is more likely to contain bugs. For the *test quality* of the product, the *test adequacy* reflects how its suite covers the product's code elements such as statements, branches, or paths [8]. A low-coverage test suite could be unable to cover the incorrect elements in the buggy product. Hence, the product with a lower-coverage test suite is more likely to be *false-passing*. Meanwhile, the *test effectiveness* reflects how intensively the test suite verifies the product's behaviors and its ability to explore the product's (in)correctness [9, 10]. The intuition is that if the product is checked by a test suite which is less effective, its overall test result is less reliable. Then, the product is more likely to be a *false-passing* one.

Furthermore, we discuss several strategies to mitigate the impact of *false-passing* products on FL approaches. Since the negative impact is mainly caused by the unreliability of the test results, our goal is to improve the *reliability* of the test results by enhancing the test quality based on the failure indications. Moreover, the reliability of test results could also be improved by disregarding the unreliable test results at either product-level or test case-level.

We conducted several experiments on a large dataset of variability bugs which contains 823 buggy versions of six widely-used SPL systems [11]. Totally, there are 14,191 *false-passing* products and 22,555 *true-passing* products. Our results show that CLAP achieves more than 90% *Accuracy* in detecting *false-passing* and *true-passing* products. Especially, the *Precision* of CLAP in *false-passing* product detection is up to 96%. This means, among 10 products predicted as *false-passing* products by CLAP, there are more than 9 products which are indeed *false-passing* ones.

We also evaluate the capability of CLAP in mitigating the negative impact of *false-passing* products on the FL performance. We conducted experiments on two state-of-the-art variability fault localization approaches with the five most popular SBFL ranking metrics [7, 12]. Interestingly, CLAP can significantly improve their performance in ranking buggy statements by up to 30%. This shows that CLAP can greatly mitigate the negative impact of *false-passing* products on localizing variability bugs and help developers find bugs much faster.

In brief, this paper makes the following contributions:

1. The formulation of the *false-passing* product detection problem in SPL systems and a large benchmark for evaluating *false-passing* product detection techniques.

2. CLAP: an effective approach to detect *false-passing* products in SPL systems and mitigate their negative impact on variability fault localization performance.

3. An extensive empirical evaluation to show the effectiveness of CLAP in both detecting *false-passing* products and mitigating their negative impact on variability fault localization performance in SPL systems.

The benchmark and implementation of CLAP could be found in details in: https://ttrangnguyen.github.io/CLAP/.

## 2. Motivation and Problem Formulation

### 2.1. Motivation

To empirically investigate the impact of *false-passing* products on FL, we conduct a preliminary study on 600 buggy versions of six SPL systems in a dataset of variability bugs [11]. For each buggy version, we simulate the existence of *false-passing* products by modifying the test suites of a random number of failing products. Specially, all the failed tests in the test suite $T$ of each selected product $p$ are removed. Once all the failed tests in $T$ are removed to create test suite $T'$, the bugs in $p$ revealed by $T$ would not be revealed by $T'$. As a result, $p$ becomes a *false-passing* product with test suite $T'$. After the simulation, each buggy version contains three groups of products: (1) failing products which contain both failed and passed tests; (2) *false-passing* products which were originally failing products, yet their failed tests were removed; and (3) passing products which originally passed all the tests.

We apply two state-of-the-art FL approaches, spectrum-based FL (SBFL) [7] and VARCOP [6] to localize the variability bugs in each buggy system with and without the existence of the (simulated) *false-passing* products. With the existence of *false-passing* products (*With FPs*), testing information of all the three groups of products, i.e., failing products, *false-passing* products, and passing products, are used to measure the suspiciousness of the statements in the

**Table 1**
Empirical study about the impact of *false-passing* products on variability fault localization performance (in *Rank*)

| Ranking Metrics | VARCOP | | SBFL | |
|---|---|---|---|---|
| | With *FPs* | Without *FPs* | With *FPs* | Without *FPs* |
| **Tarantula** | 11.47 | 6.25 | 9.92 | 8.22 |
| **Ochiai** | 7.93 | 5.05 | 7.38 | 5.91 |
| **Op2** | 6.40 | 5.71 | 7.31 | 7.15 |
| **Barinel** | 11.89 | 5.66 | 9.91 | 8.22 |
| **Dstar** | 7.13 | 4.87 | 7.36 | 5.91 |

systems. Meanwhile, without the existence of *false-passing* products (*Without FPs*), testing information of only failing products and passing products are used for localizing faults.

Table 1 shows the average *Rank* of the buggy statements which are localized by VARCOP and SBFL using the five most popular ranking metrics. As seen, *the presence of false-passing products after testing could significantly reduce the performance of the FL techniques*. On average, with the presence of *false-passing* products, the results of both VARCOP and SBFL are significantly decreased by 60% and 20%, respectively. For instance, without the existence of the *false-passing* products, the buggy statements can be found after investigating 6 and 8 statements ranked by VARCOP and SBFL using Tarantula as the ranking metric. However, due to the presence of the *false-passing* products, by SBFL and VARCOP, to hit the bugs, developers need to examine up to 10 and 11 statements, respectively.

## 2.2. Problem Formulation

After testing, for an SPL system $\mathfrak{S}$, let $P = \{p_1, ..., p_n\}$ be the set of the sampled products. Each product $p_i \in P$ is tested by a corresponding test suite $T_i$. In general, the system $\mathfrak{S}$ contains variability bugs if and only if $P$ is categorized into two separate non-empty sets based on their overall test results: the *passing products* $P_P$ and the *failing products* $P_F$, $P_P \cup P_F = P$ [6, 13, 14]. Each product in $P_F$ fails at least one test, while every product in $P_P$ passes all the test cases in its test suite. Among the passing products, a *false-passing* product contains bugs and should be a failing product yet has passed all the tests because of its ineffective test suite.

**Definition 1.** ***False-passing*** **product**. Given a tested product $p \in P$ and its test suite $T$, product $p$ is a *false-passing* product if the following conditions are satisfied:

1. There exists a statement $s$ in $p$, such that $s$ can cause failures for $p$
2. Product $p$ passed all the test cases in $T$

In other words, for *false-passing* product $p$, the current test suite $T$ of $p$ is ineffective in detecting bugs in $p$, thus $p$ has not failed any test in $T$. Additionally, there exists a test suite $T' \neq T$ such that $p$ could fail at least a test in $T'$. In this case, $T'$ is more *bug-detecting effective* than $T$, and the test results of $T'$ is more *reliable* than that of $T$. On the opposite side, *true-passing* products in a tested buggy SPL system can be formally defined as follows.

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ |
|---|---|---|---|---|
| $s_1$ | 1 | 0 | 1 | 1 |
| $s_2$ | 1 | 1 | 0 | 0 |
| $s_3$ | 0 | 1 | 0 | 1 |
| $e$ | 1 | 1 | 1 | 0 |

**Figure 1:** Spectrum of a product with 3 statements and 4 tests

**Definition 2.** ***True-passing*** **product**. Given a tested product $p$ whose test suite is $T$, $p$ is a *true-passing* product if:

1. There does **not** exist a statement $s$ in $p$, such that $s$ can cause the failures for $p$
2. As a result, product $p$ passed all the test cases in $T$

The testing information of a product is recorded in its *program spectrum* [7]. A program spectrum is a collection of execution information of the program. It provides a specific view on the dynamic behaviors of the software. For a product which has $n$ statements and $m$ test cases, its spectrum constitutes a $n \times m$ matrix. For each test, the product's execution is recorded by the column vector in the matrix.

Figure 1 shows the spectrum of a product with 3 statements and 4 tests. Specifically, vector $e = \langle 1, 1, 1, 0 \rangle$ indicates the outcome of the test cases, where $e[1] = 1$ means that test $t_1$ passed, and $e[4] = 0$ indicates that test $t_4$ failed. Vector $v_4 = \langle 1, 0, 1 \rangle$ represents the dynamic behavior of the product with the failed test $t_4$. As $s_1$ and $s_3$ are executed by $t_4$, $v_4[s_1] = 1$ and $v_4[s_3] = 1$. Meanwhile, $s_2$ is not covered by this test, and $v_4[s_2] = 0$.

**Definition 3.** ***False-passing*** **product detection**. Given 4-tuple $\langle \mathfrak{S}, P, \mathcal{T}, \mathcal{V} \rangle$, where:

- $\mathfrak{S}$ is a tested SPL system containing variability bugs.
- $P = \{p_1, ..., p_n\}$ is the set of $n$ sampled products, $P = P_P \cup P_F$, where $P_P$ and $P_F$ are the sets of *passing* and *failing* products of $\mathfrak{S}$.
- $\mathcal{T} = \{T_1, ..., T_n\}$ is a set of test suites, where $\forall i \in [1, n]$, $T_i \in \mathcal{T}$ is the test suite of $p_i \in P$.
- $\mathcal{V} = \{V_1, ..., V_n\}$ is the set of the program spectra, where $\forall i \in [1, n]$, $V_i \in \mathcal{V}$ is the program spectrum of $p_i$ with the test suite $T_i \in \mathcal{T}$.

*False-passing* product detection is to output the set of the *false-passing* products in $P_P$.

As shown in Sec. 2.1, detecting *false-passing* products could help significantly improve the performance of FL techniques. However, *false-passing* products could be very challenging to be detected. Indeed, *false-passing* phenomenon is caused by the ineffectiveness of testing, thus the bugs in the (*false-passing*) products are not revealed. To identify

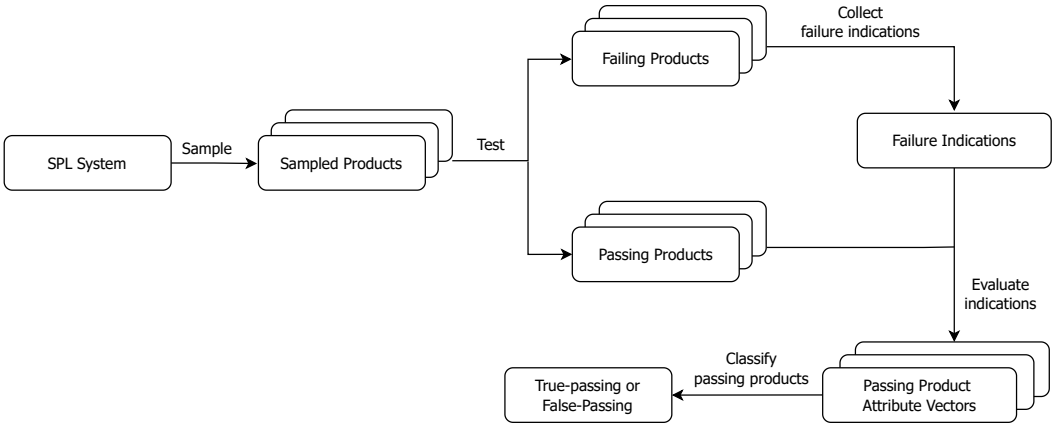


**Figure 2:** CLAP's overview

whether or not a passing product is *false-passing*, new test cases can be generated to further test the product. In practice, it could be prohibitively expensive to test all the product's behaviors. Furthermore, although a large number of tests are added, if the product still passes all the newly generated test cases, it still is not able to confirm the product is *false-passing* or *true-passing*. Moreover, another approach in detecting *false-passing* products is to figure out whether the passing products contain the bugs. However, the bugs, which cause failures in the system, have not been identified yet at that point. This means identifying the presence of bugs in the passing products is also problematic. Hence, *false-passing product detection is necessary but challenging.*

## 3. *False-passing* Product Detection

As a *false-passing* product is a product containing bugs but still passed all its test cases, its overall test result, i.e., being a passing product, is unreliable. Hence, determining if a passing product is *false-passing* can be done by examining: (1) *whether or not the product contains any bug* and (2) *the reliability of the state of passing all its test cases*.

In general, a product is considered to *contain a bug* if it contains a buggy statement and the statement's bugginess can be propagated to product's incorrect output(s). To expose the incorrectness, the buggy statement rarely executes solo, it often executes together with the other statements. Such statements are called *bug-involving statements*. Thus, to determine whether the bug is contained in the product, *we examine whether the product contains both the buggy statement and the corresponding set of bug-involving statements.*

If a product contains bugs, yet *passes all its tests*, its test suite could be inadequate [8] and ineffective [9, 10] in exploring the bug(s) in the product. Its test suite is in low coverage of the product's behaviors and could not cover the the unexpected ones. Consequently, the product still passed all its test cases and is misleadingly considered as a passing product. Intuitively, the more inadequate and ineffective the test suite, the less reliable the product's overall test result. Hence, to verify the reliability of the product's overall test result, *it is essential to examine the adequacy and effectiveness of the product's test suite.*

In practice, a buggy system could contain multiple bugs, and not all of the bugs are revealed after testing the sampled products. In this work, we focus on detecting *false-passing* products regarding the bugs which have been revealed by the failed tests in the failing products of the system. The other passing products, which can not be failed by any of these revealed bugs, are considered as *true-passing* products. Intuitively, for a set of sampled products of the system, the program and the tests of the failing products can provide the indications to examine the passing ones. The *failure indications* in the failing products are investigated in terms of product implementation (i.e., the existence of *buggy statements* and *bug-involving statements*) and test quality (i.e., *test adequacy* and *test effectiveness*). Next, determining if a passing product is *false-passing* can be done by measuring the strength of these indications in the product.

To evaluate the strength of the failure indications in a passing product, we propose a set of measurable attributes. For *product implementation*, we measure the possibility that the product contains buggy statements and corresponding sets of bug-involving statements (Sec. 3.1). The test quality is examined in terms of test adequacy and test effectiveness. For *test adequacy*, we measure to what extent the suite covers product's elements (Sec 3.2). For *test effectiveness*, we examine how each test case verifies the product's behaviors (Sec. 3.3). Overall, if a passing product has a high possibility of containing bug(s) and has a low-quality test suite, the product is more likely to be *false-passing*. The overview of our approach is shown in Figure 2.

To verify our selection of the proposed attributes in detecting *false-passing* products, we conduct experiments on 159 buggy versions of BankAccountTP (so-called *verification dataset*). Overall, there are 1,626 failing products, 1,763 *true-passing* products, and 2,017 *false-passing* products. The construction of this dataset is described in detail in Sec. 5.2. We measure each attribute's value in each product to confirm that the attribute can distinguish the *true-passing* and *false-passing* products.

### 3.1. Suspiciousness of Product Implementation

To evaluate the failure indications in a passing product $p$ of the buggy system $\mathfrak{S}$ regarding the product's implementation, we investigate the possibility that $p$ contains the *buggy statements* which caused the failures in the failing products of $\mathfrak{S}$. Additionally, the likelihood that $p$ has the statements which involve contributing and propagating the incorrectness of the buggy statements (*bug-involving statements*) is also evaluated while examining the implementation of $p$.

#### *3.1.1. How possibly does a product contain buggy statements?*

For a passing product $p$, we estimate the possibility that $p$ contains buggy statements by preliminarily measuring the suspiciousness of the statements in $p$. Intuitively, *if statements in p are highly suspicious, p will be more likely to contain the buggy statements.* The possibility that $p$ contains buggy statements, $bscp(p)$, can be estimated by the total suspicious scores of the statements in that product:

$$bscp(p) = normalize\left(\sum_{s \in S_p} \phi(s, P_F, M)\right)$$

where $S_p = \{s_1, ..., s_n\}$ is the set of statements of product $p$, and $\phi(s, P_F, M)$ is the function measuring the suspiciousness score of statement $s$ by using the testing information of the failing products in $P_F$ and the FL technique $M$. Also, $bscp(p)$ is normalized into the range of $[0, 1]$.

In $bscp(p)$, any FL technique, which can calculate the suspicious score of a statement in an SPL system, can be applied as $M$ in function $\phi$. In this work, we use the testing information of only failing products $P_F$, whose overall test results are reliable at this point, to measure the suspiciousness of the statements. However, to avoid missing the useful information provided by test results of the passing products, one can use all the sampled products $P$ in the statement suspiciousness evaluation function, $\phi$.

Figure 3 shows the possibility that passing products contain buggy statements in our *verification dataset*. For each buggy version, we preliminarily measure the suspiciousness of the statements by using SBFL with Op2 on the program spectra of the failing products (function $\phi$). As seen, 80% of the *true-passing* products have the bug-containing possibility less than 0.2. Meanwhile, more than 50% of the *false-passing* products have this possibility greater than 0.8. This illustrates that the *false-passing* products often contain a large number of highly suspicious statements. Thus, *the higher bug-containing possibility of a passing product is, the more likely it is a false-passing product*.

#### *3.1.2. How possibly does a product have bug-involving statements?*

*Bug-involving* statements of a buggy statement $s$ are the statements, which impact/be impacted by $s$. These statements must be executed together with $s$ to expose the incorrectness of $s$ to the outputs. To compute the possibility that a passing product $p$ contains bug-involving statements of $s$, we measure how similarly $s$ impacting/being impacted in $p$ and a failing product $p'$ containing $s$. Intuitively, the more similarly $s$ impacting/being impacted in $p$ and $p'$, the more similarly $s$ behaves in these two products. As a result, the higher possibility that $s$ can cause failures in $p$ in a similar way $s$ has caused the failures in $p'$. This means that $p$ is more likely to be a *false-passing* product.

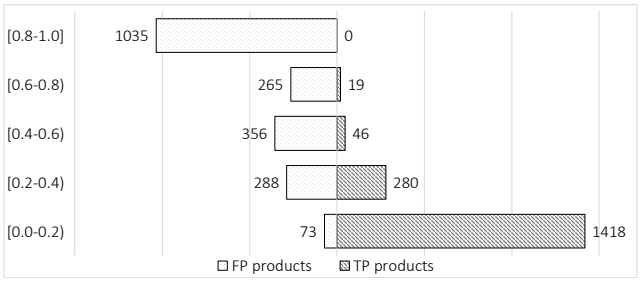


**Figure 3:** The presence of the incorrect statements in the passing products

Meanwhile, the buggy statements have not been found yet at this point. To estimate the possibility that a passing product $p$ contains bug-involving statements, we use suspicious statements and their impacting/being impacted statements. The suspicious statements are the statements executed by at least a failed test in a failing product. The statements impacting/being impacted by a suspicious statement are suspiciously to be bug-involving statements, so called suspiciously-involving statements.

Let $\mathcal{S} = \{s_1, ..., s_k\}$ be the set of suspicious statements of the given SPL system. For a statement $s \in \mathcal{S}$, let $B$ and $B'$ be the sets of impacting statements which impact $s$ in $p$ and in a failing product $p'$, respectively. Also, $F$ and $F'$ are the sets of being-impacted statements which are impacted by $s$ in $p$ and $p'$. The similarity of the sets of suspiciously-involving statements of $s$ in $p$ and $p'$ is measured as the similarities of its impacting statements($B$ and $B'$) and being impacted statements ($F$ and $F'$) in these two products:

$$invol\_sim(s, p, p') = \frac{|\{B \cap B'\} \cup \{F \cap F'\}|}{|B \cup F \cup B' \cup F'|}$$

The *suspicious-involvement score* of statement $s$ in $p$ is the *maximum* of the similarity of the suspiciously-involving statement set of $s$ in $p$ to these sets of $s$ in the failing products:

$$sis(s, p) = \max_{p' \in P_F} invol\_sim(s, p, p')$$

The reason for the use of the max function is that we would like to consider the most similarly $s$ behaves in $p$ compared to the other failing products of the system. Intuitively, if $p$ contains more suspiciously-involving statement sets which are more similar to such sets in the failing products, $p$ is more likely to be *false-passing*. The estimated possibility that $p$ contains bug-involving statements is aggregated from the suspicious-involvement scores of all suspicious statements:

$$invol(p) = \sum_{s_i \in S} sis(s_i, p)$$

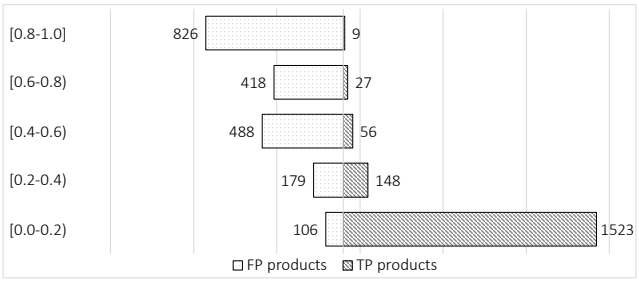


**Figure 4:** The presence of bug-infected statements in the passing products

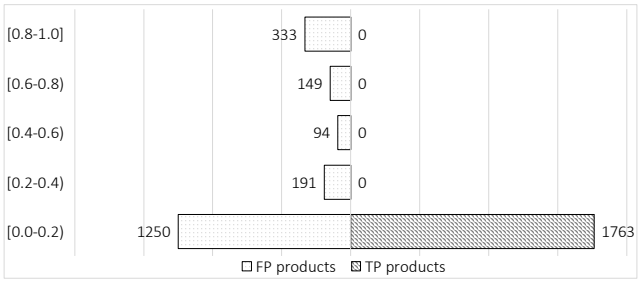


**Figure 5:** The portion of suspicious statements in the passing products which are not covered by their test suites

Figure 4 shows the suspicious-involvement scores of the passing products in the *verification dataset*. As seen, 85% of the *true-passing* products have scores less than 0.2. Meanwhile, about 90% of the *false-passing* products have scores greater than 0.2. Specially, the scores of about 40% of *false-passing* products are in the top range, i.e., [0.8-1.0]. It shows that in the *false-passing* products, suspicious statements frequently impact/be impacted by the other statements in the similar way they do in the failing products.

## 3.2. Test Adequacy

In general, assuring the quality of a program requires an adequate test suite which can cover a large number of the program's elements such as statements, branches, or paths [8]. If a program is tested by an inadequate test suite, the state of passing all the cases in the test suite could not be reliable, since a large portion of the program's elements is not tested (thoroughly). In practice, there are various adequacy criteria, and they are all imperfect. However, adequacy criteria are useful indicators for determining the inadequacy of test suites [8]. For simplicity, we measure the adequacy of a test suite in statement coverage. Additionally, to evaluate the fault diagnosability of test suites, we also apply DDU [1] [15], which is a simple and effective criterion, as an adequacy attribute of the test suite.

### *3.2.1. Code coverage*

The prerequisite condition for a bug to be revealed in a product is that the buggy statement is reached (executed) by a test. Indeed, an adequate test suite should widely cover the suspicious statements contained in the product. *The more suspicious statements which are* ***not covered*** *by the test suite of the product, the less reliable the overall passing state of the product is.*

In general, for the given SPL system, any statement executed by any failed test in a failing product is suspicious to be a buggy one. Let $S = \{s_1, ..., s_k\}$ be the set of suspicious statements, and $S_p$ be the statements of a passing product $p \in P_P$. The set of suspicious statements in $p$ is $K = S \cap S_p$. Specially, $K$ is categorized into two sets, $K_e$ and $K_{ne}$, such that $K = K_e \cup K_{ne}$ where $K_e$ and $K_{ne}$ are the sets of statements which are *covered* and *not covered* by the test suite $T$ of $p$. The *noncoverage rate* of $p$ reflects the portion of suspicious statements in $p$, yet not covered by $T$:

$$noncov(p, T) = \frac{|K_{ne}|}{|K|}$$

Figure 5 shows this *code coverage* attribute of the passing products in the *verification dataset*. In fact, the general statement coverage of the test suites of our experimental data is quite high. This means that almost the statements in each product are covered by the test suite, and very few number statements in the product are not covered. That is the reason why in Figure 5 most of the products (both the *true-passing* and *false-passing* products) have small portions of non-coverage. However, this attribute is still useful to detect *false-passing* products, since various *false-passing* products have higher noncoverage rates than the others. As seen, 100% of the *true-passing* products have the noncoverage rates lower than 0.2, while the noncoverage rates of 40% *false-passing* products are greater than this number. This result shows that the test suites of the *false-passing* products usually do not cover their suspicious statements. Meanwhile, the test suites of the *true-passing* products often have better coverage of the suspicious statements.

### *3.2.2. Density-Diversity-Uniqueness*

Besides code coverage information, to thoroughly evaluate the test suite, we take into account *DDU* metric proposed by Perez et al. [15] to evaluate the adequacy of the test suite regarding *fault diagnosability*. While the *coverage attribute* abstracts the execution information of test executions to favor an overall assessment of the suite, *DDU* takes into account per-test execution information, so it provides further insight about each test case of the suite.

The main idea of *DDU* is that a high-quality test suite must contain the test cases such that program elements are frequently tested (*density*) in diverse combinations (*diversity*), as well as the corresponding execution vectors of the elements in the program spectrum are distinguishable (*uniqueness*) [15]. The *DDU* value is from 0.0 to 1.0, and the *DDU* of an ideal test suite of a product is 1.0. Thus, the product whose test suite with a lower *fault diagnosability* in *DDU* is more likely to be *false-passing*. Specially, for a product $p$, the "undiagnosablity" of its test suite is $DDU'(p, T) = 1 - DDU(p, T)$ where $DDU(p, T)$ is the DDU value of the test suite $T$ of product $p$. As a result,

[1] *DDU* is an acronym for *Density-Diversity-Uniqueness*.

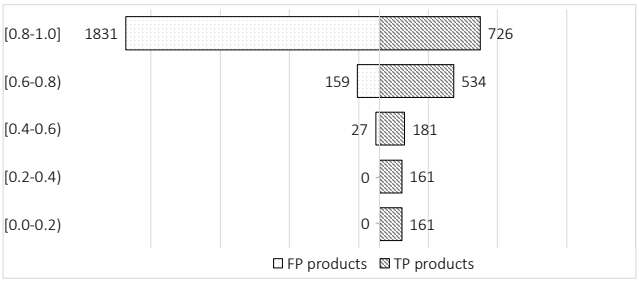


**Figure 6:** The undiagnosability (DDU') of the passing products' test suites

the higher $DDU'(p, T)$, the lower-quality test suite, and $p$ is more likely to be a *false-passing* product.

Figure 6 shows $DDU'$ of the passing products in the *verification dataset*. There are about 90% of the *false-passing* products have *DDU'* in the range of [0.8, 1], while 60% of the *true-passing* products have *DDU'* in the lower range. This shows that the *true-passing* products have more diagnosable test suites, so their states of passing all the tests are more reliable. With the less diagnosable test suites, the overall test results of the *false-passing* products are less reliable.

In practice, there are multiple criteria to evaluate the adequacy of the test suites, such as branch coverage, path coverage, mutation score, etc. These criteria could be applied using the same principle. Although sophisticated criteria could provide a more comprehensive adequacy evaluation, the computation could be expensive. For example, mutation score is a popular and powerful metric to evaluate the test suite. However, to calculate the mutation score, a large number of mutants need to be tested against the original test suite. This could be costly and time-consuming, especially for the large and complex systems containing many sampled products with large test suites [16, 17, 18]. Thus, to ensure the efficiency of the proposed attributes measuring the test suite's adequacy, we employ statement coverage and DDU. Their effectiveness has been demonstrated by our evaluation by the verification dataset and the overall performance of CLAP shown in Sec. 6.

## 3.3. Test Effectiveness

For a product $p$, the fault-detecting effectiveness of its test suite $T$ shows how intensively it tests the product's behaviors and its ability to explore the product's incorrectness [9, 10]. To evaluate the effectiveness of the test suite $T$, we aim to investigate $T$ by two attributes: *incorrectness verifiability* and *correctness reflectability*. For a passing product $p$, *incorrectness verifiability* attribute measures how $p$'s test suite, $T$, covers the product's suspicious behaviors. Meanwhile, *correctness reflectability* indicates that the passed tests of the product really reflect its correct behaviors (i.e., these tests are not just coincidentally passed).

Note that, the test adequacy attributes focus on how and to what extent the product's elements are covered by the test suite. These attributes do not take into account the results of the test cases. Meanwhile, the test effectiveness attributes focus on how the suite verifies the product's behaviors. Not only the product's elements but also the results of the tests are considered in the evaluation process.

In practice, the behaviors of the products are dynamically considered regarding the execution vectors in the product's spectrum. *The vector for a failed test represents an incorrect behavior of the product*, so-called *incorrect behavior vector*. In a buggy system, the incorrect behaviors are represented by the incorrect behavior vectors in the failing products. These vectors record the executions of the failing products in the failed tests. Meanwhile, the *correct behaviors* are represented by executions of truly passed tests which are not just coincidentally passed. Thus, in this work, we leverage execution vectors in spectra to measure *incorrectness verifiability* and *correctness reflectability*.

### 3.3.1. Incorrectness verifiability

From the incorrect behaviors of the failing products $P_F$ of the given system $\mathfrak{S}$, we identify the behaviors which are *suspiciously* contained in a passing product $p \in P_p$ and then evaluate whether these behaviors are covered by the test cases of $p$. Intuitively, if $p$ contains suspicious behaviors, these behaviors should be verified by test cases in its test suite, $T$, to sufficiently ensure the correctness of the product. To evaluate the reliability of the state of being a passing product of $p$ with its test suite $T$, *we measure the number of incorrect behaviors are (1) potentially contained in p and (2) not covered by the test cases in T*. Intuitively, the correctness of $p$ is less reliable if $p$ potentially contains more such incorrect behaviors.

For (1), an incorrect behavior expressed by the execution vector $v$ is *potentially contained* in $p$ if $p$ contains a large portion of the executed statements in $v$. This portion should be larger than a threshold $\mathcal{R}_{I1}$. The reason is that the products of an SPL system are composed of different sets of features, a product rarely contains a set of statements which is exactly the same as an execution of a test in another.

For (2), the behavior expressed by $v$ is *not covered* by the test in $p$ if $v$ is not similar to any execution vector in $p$'s spectrum. Similar to (1), two execution vectors are similar if they share a large portion executed statements and this portion should be greater than a threshold $\mathcal{R}_{I2}$.

For (1), let $\mathbf{V}_{IB} = \{v_{f1}, ..., v_{fn}\}$ be the set of all the incorrect behavior vectors of the failing products $P_F$. Also, let $S_p$ be the set of statements of the passing product $p$, and $V_p = \{v_1, ..., v_m\}$ be the set of the execution vectors in $p$'s spectrum. *p potentially contains* a behavior presented by an incorrect behavior vector, $v_{fi} \in \mathbf{V}_{IB}$, if $S_p$ contains a large portion of the statements executed in that vector:

$$\frac{|\{s \in v_{fi} | v_{fi}[s] = 1 \wedge s \in S_p\}|}{|\{s \in v_{fi} | v_{fi}[s] = 1\}|} \geq \mathcal{R}_{I1}$$

For (2), the behavior expressed by $v_{fi}$ is covered by the test cases in $T$ if there exists an execution vector recorded in $p$'s spectrum similar to $v_{fi}$. To calculate the similarity of two vectors, we adapt Jaccard similarity formulation. An

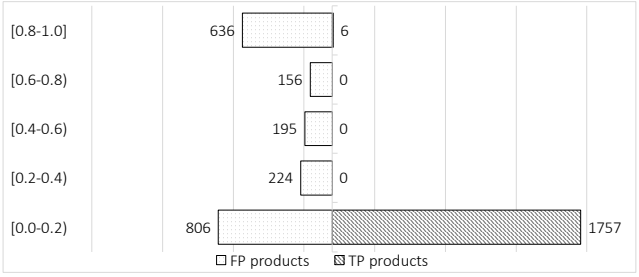


**Figure 7:** The incorrectness verification capability of the passing products' test suites

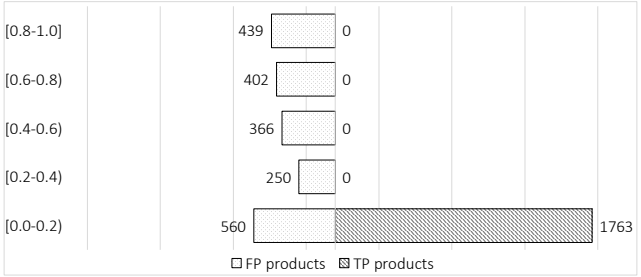


**Figure 8:** The correctness presentation capability of the passing products' test suites

execution vector $v_j \in V_p$ is similar to $v_{fi}$ if:

$$sim(v_j, v_{fi}) = \frac{a}{a+b+c} \geq \mathcal{R}_{I2}$$

where $a$ is the number of statements that executed in both $v_j$ and $v_{fi}$, $a = |\{s \in v_j | v_j[s] = 1 \wedge v_{fi}[s] = 1\}|$. $b$ is the number of statements executed in $v_j$ but not in $v_{fi}$, $b = |\{s \in v_j | v_j[s] = 1 \wedge v_{fi}[s] = 0\}|$. Meanwhile, $c$ is the number of statements which are not executed in $v_j$ but executed in $v_{fi}$, $c = |\{s \in v_j | v_j[s] = 0 \wedge v_{fi}[s] = 1\}|$.

The incorrectness verifiability score of the product $p$ is calculated as the portion of the incorrect behaviors potentially contained by $p$ but not covered by any test in the product's test suite $T$. Let $I_1$ be the set of incorrect behaviors vectors whose executed statements are contained in $p$. Also, let $I_2$ be the set of incorrect behaviors vectors in $I_1$ and not similar to any execution vector in the spectrum of $p$. The incorrectness verifiability attribute is measured as:

$$iv(p, T) = \frac{|I_2|}{|I_1|}$$

Figure 7 shows the portion of the incorrect behaviors suspiciously contained in the passing products in the *verification dataset*, yet not tested by the products' test suites. As seen, in most of the *true-passing* products, the suspicious behaviors are covered by at least a test in their suites. However, two-thirds of the *false-passing* products contain suspicious behaviors, and these behaviors are not tested by the products' tests. This shows that the *false-passing* products' test suites are often ineffective in verifying their suspicious behaviors.

### *3.3.2. Correctness reflectability*

In practice, not all of the passed tests can reliably confirm the success of the program since they could be coincidentally correct. Also, coincidentally passed tests are unbefitting for verifying the correctness of the product [2]. To determine if the correctness of a product is reliably reflected by its passed tests, we measure the portion of passed tests of $p$ which are likely to be *truly* correct. The more truly passed tests, the more effective $p$'s test suite. Intuitively, with fewer truly passed tests, $p$ has a higher possibility to be *false-passing*.

For a passing product $p$, *a passed test in p can truly represent a correct behavior if its execution vector is similar to any truly passed test's execution vector of any failing product* of the system. In a failing product, a test whose execution vector *is not similar* to the execution vector of a failed test can be considered as a truly passed test. This is because these tests are less likely to execute the faults revealed in the failed tests. Meanwhile, a passed test whose execution vectors *are similar* to the execution vector of a failed test, could be coincidentally passed. The reason is that with the similar execution vector but only one of them can reveal the bugs, then the other could reach the buggy statements but is ineffective in exposing the bugginess.

Given an SPL system $\mathfrak{S}$, let $\mathbf{V}_{IB} = \{v_{f1}, ..., v_{fn}\}$ and $\mathbf{V}_{CB} = \{v_{p1}, ..., v_{pm}\}$ be the sets of incorrect and correct behaviors of the failing products. In other words, $\mathbf{V}_{IB}$ and $\mathbf{V}_{CB}$ are sets of execution vectors of the failed tests and passed tests in the failing products of system $\mathfrak{S}$. A passed test in a failing product whose execution vector $v_{pi} \in \mathbf{V}_{CB}$, is truly passed if its execution vector is not similar to any failed test's execution vector, formally, $\nexists v_{fj} \in \mathbf{V}_{IB}, sim(v_{pi}, v_{fj}) \geq \mathcal{R}_C$, where $\mathcal{R}_C$ is a similarity threshold.

Let $V_p = \{v_1, ..., v_k\}$ be the set of execution vectors of of passing product $p$ with its test suite $T$. The correctness reflected by a passed test in $T$, whose execution vector $v_i \in V_p$, is confirmed if $v_i$ is similar to a truly correct test's execution vector in the failing products. Specifically, $\exists v_{pj} \in \mathbf{V}_{CB}$, $\nexists v_{ft} \in \mathbf{V}_{IB}, sim(v_{pj}, v_{ft}) \geq \mathcal{R}_C$, and $sim(v_{pj}, v_i) \geq \mathcal{R}_C$.

Let $C$ be the number of truly correct tests of $p$, the correctness reflectability value of a passing product $p$ is:

$$cr(p, T) = 1 - \frac{C}{|V_p|}$$

The correctness reflectability value is 0 when all the passed tests of the passing product $p$ are confirmed by truly correct tests of the failing products. This value is 1 when none of the tests reflecting the correctness of $p$ is confirmed.

Figure 8 shows the *correctness reflectability* of the passing products in the *verification dataset*. The figures of this attribute of all the *true-passing* products are in the lowest range, i.e., [0, 0.2). This means that the correctness reflected by almost passed tests in the *true-passing* products is confirmed by at least a truly passed test of a failing product. Meanwhile, more than 70% of the *false-passing* products contain passed tests, but the correctness reflected by their

passed tests could not be confirmed. This indicates the *false-passing* products often contain passed tests, yet many of them could not represent the products' true correctness.

### 3.4. Detecting *False-passing* Products

In this work, we consider the problem of *false-passing* product detection as a binary classification problem. Specifically, the possibility that a product $p$ is *false-passing* is measured by the proposed attributes regarding product's implementation and its test quality. Product $p$ is represented by a six-dimension vector $x = \langle a_1, a_2, a_3, a_4, a_5, a_6 \rangle$, where $a_1 = bscp(p)$ and $a_2 = involv(p)$ reflect whether the product's implementation contains buggy statements and the corresponding bug-involving statements. The remaining attributes reflect its test quality, $a_3 = noncov(p, T)$ and $a_4 = DDU'(p, T)$ are about test adequacy, while $a_5 = iv(p, T)$, and $a_6 = cr(p, T)$ are about test effectiveness. For each attribute, the higher score, the more likely $p$ is *false-passing*.

A classifier $h(x)$ could be applied to predict the possibility that product presented by vector $x$ is a *false-passing* product. The label $y$ of the product is *false-passing* if the result of function $h(x)$ is greater than a threshold $\mathcal{R}_{fp}$, otherwise, it is a *true-passing* product. In this work, we use the default threshold, $\mathcal{R}_{fp} = 0.5$, however, one can further consider Precision-Recall curve to trade-off between precision and recall and select the optimal decision threshold.

$$y = \begin{cases} false\text{-}passing & \text{if } h(x) \geq \mathcal{R}_{fp} \\ true\text{-}passing & otherwise \end{cases}$$

## 4. Mitigation of Negative Impact of *False-passing* Products on Variability FL

As the negative impact on variability fault localization (FL) is caused by *false-passing* products' unreliable test results, there are two main directions to mitigate their negative impacts. Specifically, we can either improve the reliability or eliminate the unreliability in the test results of these products. Firstly, the reliability of test results can be improved by enhancing the quality of the test suites of the *false-passing* products. Secondly, the unreliability of test results can be eliminated by removing products whose unreliable test results and/or removing low-quality test cases.

First, *the FL performance can be boosted by improving the reliability of the false-passing products' test results*. Specifically, the *false-passing* products can be more thoroughly tested by a better test suite to explore their bugginess. Once their bugs are revealed by test cases, we not only have more information about the faults but also have better assessments of the (overall) test results of the products. This could help improve variability FL performance.

In particular, the failure indications (e.g., suspicious statements and behaviors) can be used to guide test generators to produce better test suites for these *false-passing* products. Specially, the newly generated test cases should focus on these failure indications. For instance, to have a more adequate test suite, new test cases can be added to cover the suspicious statements which have not been covered yet (Sec. 3.2.1). To improve the test effectiveness, new test cases could be added to verify whether the suspicious behaviors can cause failures for the *false-passing* products (Sec. 3.3.1).

Second, *the performance of FL techniques can also be enhanced by eliminating the unreliability in the test results of the false-passing products*. In particular, the unreliable test results can be at either the product-level or the test case-level. For the product-level, the test result is the overall test result (i.e., being failing or passing) of a product. Meanwhile, the test result at the test case-level is the result (i.e., failed or passed) of every single test case.

For the *product-level*, we can remove all the detected *false-passing* products before localizing variability faults. This strategy reduces the number of buggy products, yet incorrectly considered as passing ones. Thus, it can benefit the *product-base assessment* FL techniques [6, 5] which evaluate the suspiciousness of the statements according to the number of failing and/or passing products.

For the *test case-level*, the coincidentally passed tests should be removed. The reason is that the coincidentally passed tests improperly increase the number of passed tests executed by the buggy statements [2]. As a result, these test cases can negatively impact the performance of *test case-based assessment* FL approaches [6, 7] which evaluate the statements' suspiciousness based on the number of failed and/or passed tests. To clean such tests, each passed test of a product should be carefully investigated.

In summary, this section discusses the directions to mitigate *false-passing* products' impact on FL techniques. In this work, we evaluate the effectiveness of two methods: adding new tests to improve the product's test quality and eliminating the unreliability at the product-level. The mitigation direction eliminating coincidentally passed tests required thorough investigation to carefully review and eliminate unreliable tests. The techniques in this direction are beyond the scope of our work and left for future work.

## 5. Empirical Methodology

### 5.1. Research Questions

To evaluate CLAP in detecting *false-passing* products and mitigating their negative impacts on fault localization, we seek to answer the following research questions:

**RQ1: Accuracy Analysis**: How accurate is CLAP in detecting *false-passing* products?

**RQ2: Mitigating Impact of *False-passing* Products on Fault Localization**: How does CLAP mitigate the negative impact of *false-passing* products on the performance of state-of-the-art FL techniques including VARCOP [6] and SBFL [19, 20]?

**RQ3. Sensitivity Analysis**: How does CLAP perform on different evaluation scenarios? And how do different training data sizes impact CLAP's performance?

**RQ4. Intrinsic Analysis**: How do the different attributes of CLAP contribute to the *false-passing* product detection performance?

**Table 2**
Products' test suites before and after being transformed

| System | #Transformed products | Original test suite T | | New test suite T' | |
|---|---|---|---|---|---|
| | | #Tests | Cov (%) | #Tests | Cov (%) |
| BankAccountTP | 1,833 | 22 | 96.7 | 19 | 92.8 |
| Elevator | 161 | 165 | 76.5 | 120 | 69.4 |
| Email | 420 | 94 | 97.2 | 89 | 96.0 |
| ExamDB | 126 | 136 | 96.6 | 128 | 96.4 |
| GPL | 6,091 | 90 | 98.7 | 88 | 98.2 |
| ZipMe | 541 | 253 | 42.9 | 249 | 42.7 |

#Tests and Cov stand for the number of tests and statement coverage in a product.

**RQ5. Time Complexity**: What is CLAP's running time?

## 5.2. Dataset

To evaluate CLAP, we systematically collect *failing*, *true-passing*, and *false-passing* products in existing buggy SPL systems as follows.

First, to practically collect *false-passing* products, we transform a random number of failing products in a buggy SPL system into passing products by removing all their failed tests. Indeed, for a failing product $p$ with the original test suite $T$, removing all the failed tests in $T$ creates a new test suite $T'$. Product $p$ with the test suite $T'$ is *false-passing* since $p$ fails with $T$, but not with $T'$. The average number of tests in the original test suite $T$ and the new test suite $T'$, as well as the corresponding statement coverage of the products in each system are shown in Table 2.

Although removing failed tests in $T$ slightly affects the statement coverage of $p$ (i.e., about 2% on average), *this data collecting procedure is independent of our approach*. By this procedure, only failed tests in $T$ are removed, and all the passed tests in the original test suite of $p$ are kept unchanged. Meanwhile, the proposed attributes of CLAP measure how likely $p$ is a *false-passing* product by investigating its implementations and its passed tests.

Second, to collect *true-passing* products, we verify the overall test results of the passing products and label them. In practice, this process could be very tedious and time-consuming even for experts. To practically verify the overall test result of a passing product $p'$, we propose a semi-automated procedure generating the tests and making the product fail in three steps. If $p'$ fails any test in each step, then $p'$ with its original test suite is a *false-passing* one. Otherwise, it can be considered as a *true-passing* product.

*Step 1. Automatically generating new test cases for $p'$.* We used multiple test generation tools including Evosuite [21] and Randoop [22], to generate new tests for $p'$. If the product is failed by any added test, the product with the original test suite is a *false-passing* product. If the product still passes all the added tests, we move to the next step.

*Step 2. Applying a hybrid test generator.* The bugs of the system have been explored by failed tests of the failing products. Thus, these failed tests can be used as the guidance to verify the existence of the bugs in $p'$. Specifically, we tried to adapt failed tests of the failing products to test $p'$. In addition, for an SPL system, each product is composed from different set of features, thus tests need to be adapted appropriately according to each product, and not all of the failed tests can be adapted to test another products. If any adapted test can be executed by $p'$ and $p'$ creates incorrect output, $p'$ with the original test suite is a *false-passing* product. Otherwise, we move to the next step.

*Step 3. Manually investigating the product.* $p'$ is manually investigated and generated tests to carefully confirm its correctness or bugginess. If $p'$ fails any newly generated test, $p'$ with the original test suite is *false-passing*. Otherwise, if the product still passes all the manually generated tests, $p'$ with the original test suite is a *true-passing* product.

In this work, we applied the procedure on the sampled products of the buggy systems in the dataset of vulnerability bugs collected by Ngo et al. [11]. This dataset includes 1,570 buggy versions with their sampled products and the corresponding test suites of six Java SPL systems which are widely used in SPL studies. After labeling, there are 823 buggy versions which contains large numbers of products in all three kinds: failing, *true-passing*, and *false-passing* products. All the other buggy versions which are not satisfied are removed. Table 3 shows the overview of the dataset.

## 5.3. Empirical Procedure

**Accuracy Analysis**. We split 823 buggy versions into five folds (5-fold cross-validation). Specifically, four folds are picked for training and one remaining fold is used for testing. We adapted cross-validation to aggregate average results for the final assessment.

**Mitigating Impact of *False-passing* Products on Fault Localization**. We used the buggy versions in five systems to train the *false-passing* product detection model, and then the trained model was used to detect *false-passing* products in the buggy versions of the remaining system. Next, we evaluated the effectiveness of our mitigation techniques on FL performance. The *original* FL performance is measured using all the sampled products and their corresponding testing information. The performance after applying our mitigation techniques: (1) removing all the detected *false-passing* products, only using the sampled products which are not predicted to be *false-passing* and their testing information (*removing FPs*); (2) generating new tests for detected *false-passing* products for further testing, then if the faults are revealed (i.e., at least one added tests is failed), these products are used with the other products which are not predicted to be *false-passing* to localize faults (*adding tests*).

**Sensitivity Analysis.** We conducted two experiments to measure the impact of different evaluation scenarios and training data sizes.

First, we posit that *different systems, buggy versions of a system, and products in a version have different degrees of relevance*. Hence, to evaluate the impact of the specialities of systems, buggy versions, and products on CLAP, we conducted the following experiment scenarios.

- **System-based edition**. CLAP is trained with the products in buggy versions of five systems, and the products in the buggy versions of the remaining system are used for testing. In practice, this setting reflects

**Table 3**
Dataset overview

| System | Details | | Test info | | | Buggy versions | | | Products | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | #LOC | #F | #SP | #Tests | Cov (%) | 1-Bug | 2-Bug | 3-Bug | #Fs | #FPs | #TPs |
| BankAccountTP | 143 | 8 | 34 | 20 | 97.8 | 41 | 117 | 29 | 2,055 | 2,328 | 1,975 |
| Elevator | 854 | 6 | 18 | 166 | 85.4 | 14 | 17 | 10 | 217 | 326 | 195 |
| Email | 439 | 9 | 27 | 86 | 97.3 | 14 | 21 | 34 | 553 | 587 | 723 |
| ExamDB | 513 | 8 | 8 | 133 | 98.0 | 10 | 44 | 23 | 201 | 127 | 288 |
| GPL | 1,944 | 27 | 99 | 87 | 99.1 | 97 | 188 | 70 | 6,612 | 9,995 | 18,538 |
| ZipMe | 3,460 | 13 | 25 | 255 | 44.4 | 17 | 46 | 31 | 686 | 828 | 836 |
| Total | | | | | | | | | 10,433 | 14,191 | 22,555 |

#F and #SP stand for the number of features and the sample size.
#Tests and Cov stand for the number of tests and statement coverage in a product.
N-Bug represents the number of bugs (i.e., N) in the buggy version.
#Fs, #FPs, and #TPs stand for failing, *false-passing*, and *true-passing* products.

the case when the development history of the system is not very long, and the data about the current system is not available/sufficient. Thus, the data of the other systems are leveraged for detecting *false-passing* products in the developing system.

- **Version-based edition**. We shuffled all the buggy versions of the six systems and then split these buggy versions into a training set and a testing set by the ratio 8:2. This scenario comes from the idea when the system is developed for a while, and the information from other systems, as well as the other buggy versions of the developing system are available for detecting new *false-passing* products.
- **Product-based edition**. We shuffled all the products in all the buggy versions of the six systems and then split them into a training set and a testing set by the ratio 8:2. The idea of this scenario is that during assuring systems' quality, the sampled products are progressively tested and determined if they are *false-passing*. Thus, the detected and confirmed *false-passing* products could be used for training and detecting other *false-passing* products.
- **Within-system edition**. We conducted an experiment on the buggy versions of each system. The buggy versions of a system are split into a training set and a testing set by the ratio 8:2. This is also another real-world setting. When the data from other systems is not available, then only the information from testing system is used to detect *false-passing* in the system.

Second, as CLAP is data-driven, we study the **impact of training data sizes** on CLAP's performance. We randomly picked a system for testing. The training data is gradually increased by adding data from the remaining systems.

**Intrinsic Analysis.** To better understand how our approach works, we study the impact of the proposed attributes on CLAP's performance: product implementation, test adequacy, and test effectiveness. We built different variants of CLAP, which use attributes only in one of these aspects to detect *false-passing* products, and measure their performance.

## 5.4. Metrics

We adopt *Accuracy*, *Precision*, *Recall*, and *F1-score* which are widely used to evaluate classification model [23]. Let $P_{TP}$ and $P_{FP}$ be the sets of predicted *true-passing* and *false-passing* products, and $A_{TP}$ and $A_{FP}$ be the sets of actual *true-passing* and *false-passing* products, respectively. These metrics are measured as the following formulas.

**Acc.** Accuracy measures the general performance of the classification model. Accuracy indicates the ratio of correctly predicted *true-passing* and *false-passing* products out of the total passing products:

$$Acc = \frac{|\{P_{TP} \cap A_{TP}\} \cup \{P_{FP} \cap A_{FP}\}|}{|A_{TP} \cup A_{FP}|}$$

**Prec.** Precision indicates the prediction correctness for *true-passing*, $Prec(TP)$, or *false-passing*, $Prec(FP)$:

$$Prec(TP) = \frac{|P_{TP} \cap A_{TP}|}{|P_{TP}|}$$

$$Prec(FP) = \frac{|P_{FP} \cap A_{FP}|}{|P_{FP}|}$$

**Recall.** Recall indicates the effectiveness of prediction, the portion of correctly predicted *true-passing* (*false-passing*) products out of the total number of actual *true-passing* (*false-passing*) products.

$$Recall(TP) = \frac{|P_{TP} \cap A_{TP}|}{|A_{TP}|}$$

$$Recall(FP) = \frac{|P_{FP} \cap A_{FP}|}{|A_{FP}|}$$

**F1-score.** F1-score is defined as the harmonic mean of precision and recall, it indicates balance between those.

$$F1(TP) = 2 * \frac{Prec(TP) \times Recall(TP)}{Prec(TP) + Recall(TP)}$$

$$F1(FP) = 2 * \frac{Prec(FP) \times Recall(FP)}{Prec(FP) + Recall(FP)}$$

To measure FL performance, we employed *Rank* and *EXAM*, which is widely used in the existing FL studies [7, 24].

**Table 4**
Accuracy of FP product detection model

| Classifier | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|
| SVM | True-passing | 88.16% | **97.09%** | 92.41% | 90.04% |
| | False-passing | **94.19%** | 78.36% | 85.55% | |
| KNN | True-passing | 90.41% | 93.97% | 92.16% | 90.02% |
| | False-passing | 89.30% | 83.46% | 86.28% | |
| Naïve Bayes | True-passing | 88.36% | 95.25% | 91.68% | 89.21% |
| | False-passing | 90.95% | 79.18% | 84.66% | |
| Logistic Regression | True-passing | 88.75% | 95.99% | 92.23% | 89.91% |
| | False-passing | 92.30% | 79.81% | 85.60% | |
| Decision Tree | True-passing | 90.03% | 96.26% | 93.04% | 91.01% |
| | False-passing | 92.99% | 82.30% | 87.32% | |
| Random Forest | True-passing | 89.81% | 95.00% | 92.33% | 90.16% |
| | False-passing | 90.83% | 82.12% | 86.26% | |

**Rank.** Rank indicates the position of the buggy statements in the resulted lists of the FL techniques. The lower rank of buggy statements, the more effective approach. If there are multiple statements having the same score, buggy statements are ranked last among them. For the cases of multiple bugs, we measured *Rank* of the first buggy statement (*best rank*) in the resulted lists.

**EXAM.** EXAM [25] is the proportion of the statements needs to be examined until the first faulty one is reached:

$$EXAM = \frac{r}{N} \times 100\%$$

where $r$ is the rank of the buggy statement and $N$ is the total number of statements in the list. The lower *EXAM*, the better FL technique.

### 5.5. Experimental Setup

**Classifier selection.** We selected six popular classifiers based on their use in previous and related studies [26, 27]. The classifiers include Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Naïve Bayes, Logistic Regression, Decision Tree, and Random Forest.

**Experimental setup.** We implemented classification models using *Sklearn* library. For each classification algorithm, we train the model with the respective standard settings. Each item in the dataset is a passing product represented by a 6-dimensional vector whose values are computed based on six attributes proposed in Sec. 3. The training and the testing sets in each experiment are separated by different scenarios as described in Sec. 5.3. All the experiments are conducted on a desktop with Intel Core i5 2.7GHz, 8GB RAM.

## 6. Experimental Results

### 6.1. Accuracy Analysis (RQ1)

As shown in Table 4, CLAP *is highly effective in detecting false-passing products for all the studied classifiers*. The average accuracy for all six classifiers is about 90%. This figure indicates that CLAP can correctly detect 9 out of 10 products to be *true-passing* or *false-passing*. The average F1-scores of *true-passing* and *false-passing* product detection are also high, about 92% and 86%, respectively. Furthermore, the average *Recall* for *false-passing* product detection is about 81%. In other words, if there are 10 *false-passing* products, 8 of them are correctly detected. Meanwhile, this figure for *true-passing* product classification is approximately 96%, which demonstrates that almost *true-passing* products can be accurately detected.

Among the studied classifiers, Decision Tree has the best *Accuracy* (91.01%). This indicates that Decision Tree most effectively separates *true-passing* and *false-passing* products in a set of passing products. Meanwhile, SVM obtains the highest *false-passing* product detection *Precision* and *true-passing* product classification *Recall*. This demonstrates that compared to the other classifiers, although SVM can detect fewer numbers of *false-passing* products (the SVM's *Recall* on *false-passing* product detection is slightly lower than the other classifiers' results), it more precisely predicted *false-passing* products (its *Precision* on *false-passing* product detection is higher than the others classifiers). At the same time, it also less erroneously detected *true-passing* products (higher *Recall* on *true-passing* product detection). In fact, misleadingly detecting *true-passing* products can cause missing useful information in FL procedure and it could negatively affect FL performance. Thus, for our dataset, SVM is the safest and most suitable for our approach, and we use SVM for the following experiments.

### 6.2. Mitigating Impact of *False-passing* Products on Fault Localization (RQ2)

Table 5 shows the performance of the state-of-the-art variability fault localization techniques in three settings, the original performance and the FL performance after applying our mitigation methods: removing *false-passing* products and adding tests for *false-passing* products. In this experiment, CLAP used SVM to detect *false-passing* products.

As shown in Table 5, *removing "noises" caused by false-passing products helps both* VARCOP *and SBFL obtain better performance compared to when they are applied on the original testing information*. Specifically, when *false-passing* products are detected and removed, the performance of the VARCOP improved up to 25% in Rank and 19% in EXAM, also these improvements of SBFL are 8% and 3%, respectively. Indeed, the presence of *false-passing* products could lower the suspicious scores of the incorrect statements. The reason is that these products not only decrease the ratio of failing and passing products containing the buggy statements but also decrease the ratio of failed tests and passed tests executed by these statements. This causes confusion for the FL techniques which often distinguish the incorrect statements from the others based on the number of failing/-passing products, as well as the number of failed/passed test cases. Therefore, eliminating *false-passing* products can help to improve the performance of FL techniques.

Figure 9 shows a variability bug (line 4) in the BankAccountTP system. The system is sampled into 34 products for testing. After testing, the bug is revealed in two products, i.e., there are two failing products and 32 passing products. By using the program spectra of all of these 34 products

**Table 5**
Mitigating the *false-passing* products' negative impact on FL performance

| Metric | Ranking formula | VarCop | | | SBFL | | |
|---|---|---|---|---|---|---|---|
| | | Original | Removing FPs | Adding tests for FPs | Original | Removing FPs | Adding tests for FPs |
| Rank | Tarantula | 3.35 | 2.52 | 2.22 | 5.10 | 4.75 | 4.53 |
| | Ochiai | 2.39 | 2.23 | 2.28 | 3.00 | 2.77 | 2.86 |
| | Op2 | 4.31 | 4.18 | 4.33 | 7.03 | 6.84 | 6.96 |
| | Barinel | 3.69 | 2.83 | 2.91 | 5.10 | 4.74 | 4.53 |
| | Dstar | 2.55 | 2.14 | 2.19 | 3.06 | 2.91 | 2.98 |
| EXAM | Tarantula | 1.35 | 1.10 | 1.00 | 1.89 | 1.86 | 1.82 |
| | Ochiai | 1.02 | 1.01 | 0.97 | 1.12 | 1.09 | 1.11 |
| | Op2 | 1.40 | 1.38 | 1.40 | 2.29 | 2.24 | 2.27 |
| | Barinel | 1.46 | 1.21 | 1.22 | 1.89 | 1.86 | 1.82 |
| | Dstar | 1.01 | 0.94 | 0.90 | 1.14 | 1.09 | 1.12 |

```
1  boolean update(int x){
2      int newWithdraw = withdraw;
3      if (x < 0){
4          newWithdraw += x--;
5          //Patch: x-- => x
6          if (newWithdraw < DAILY_LIMIT) {
7              return false;
8          }
9      }
10     //...
11 }
```

**Figure 9:** A variability bug in the feature DailyLimit of system BankAccountTP

for localizing fault, VARCOP ranks the buggy statement at $7^{th}$ and SBFL ranks it at $5^{th}$. However, among the passing products, there are 14 *false-passing* products. *After detecting and removing false-passing products, FL performance of* VARCOP *and SBFL is improved 30% and 40%, respectively*. Specifically, VARCOP ranks the bug at $5^{th}$ and SBFL ranks it at $3^{rd}$. In this case, CLAP correctly predicted 13 products as *false-passing*. As a result, a large number of products, which contain the bug and are incorrectly considered as passing products, have been removed. Moreover, we found that by removing *false-passing* products, 52 coincidentally passed tests in these products are also removed. Therefore, the buggy statement becomes more distinguishable from the other statements of the system in terms of both product-based and test case-based assessment.

Additionally, *when the products predicted as false-passing are further tested by better quality test suites, FL techniques have more useful information to effectively detect the faults*. Specifically, when more tests are added for further testing *false-passing* products, the performance of VARCOP is improved up to 34% in Rank and 26% in EXAM, also these figures of SBFL are 11% and 4% compared to that they are applied on the original testing information. In Figure 9, after adding new tests for detected *false-passing* products, the bug is revealed in 2 more products, increasing the number of failing products from 2 to 4. By using the spectra of these 4 failing products and 18 predicted *true-passing* products of the systems, both VARCOP and SBFL could rank the bug at $3^{rd}$, which is also better than directly removing all 13 detected *false-passing* products.

However, *in some cases, the FL performance after adding new (failed) tests for false-passing products could be worse than that when removing all of the detected false-passing products*. The reason is that besides the added tests which are failed, the original test suites of the *false-passing* products already contain test cases which coincidentally passed (unreliable passed tests). Such tests also produce noises and negatively affect FL performance. In Figure 10, the bug (line 2) is ranked $9^{th}$, $3^{rd}$, and $5^{th}$ by SBFL in the original setting and the two mitigation settings. By adding tests to the detected *false-passing* products, the buggy statement's rank is worse than that of removing all of these products ($5^{th}$ vs. $3^{rd}$). In this case, two *false-passing* products are detected and via added test cases, the bug is revealed in these products. However, in their test suites, there are 15 coincidentally passed tests which executed the faults but cannot reveal the failure. Consequently, using the program spectra of these products with the original test suites and added tests decreased the suspicious score of the buggy statement 0.01 and this causes its rank becomes worse. For these *false-passing* products, an analysis to remove the unreliable passed tests and add new effective test cases should be designed to help FL techniques improve their performance, and we leave this for future work.

## 6.3. Sensitivity Analysis (RQ3)

### *6.3.1. Impact of evaluation scenarios*

Table 6 shows the *false-passing* product detection performance of CLAP with SVM in four scenarios by different degrees of relevance of the training and testing data: System-based, Version-based, Product-based, and Within-system editions. Overall, *the more relevant training and*

```
public boolean consistent(){
   for (int i = 0; i < students.length; i--) {
   //Patch: i-- => i++
      if (students[i] != null && !students[i].backedOut &&
           students[i].points < 0) {
         return false;
      }
   }
   return true;
}
```

**Figure 10:** A variability bug in the feature ExamDataBaseImpl of system ExamDB

**Table 6**
Impact of different experimental scenarios

| Edition | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|
| System-based | True-passing | 85.51% | 92.16% | 89.15% | 88.44% |
| | False-passing | 89.42% | 85.83% | 86.83% | |
| Version-based | True-passing | 88.16% | 97.09% | 92.41% | 90.04% |
| | False-passing | 94.19% | 78.36% | 85.55% | |
| Product-based | True-passing | 87.53% | 96.97% | 92.01% | 89.70% |
| | False-passing | 94.27% | 78.26% | 85.52% | |
| Within-system | True-passing | 88.73% | 96.29% | 92.21% | 92.29% |
| | False-passing | 96.12% | 87.02% | 91.16% | |

**Table 7**
CLAP's performance on each system in system-based edition

| System | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|
| BankAccountTP | True-passing | 89.43% | 99.14% | 94.04% | 94.13% |
| | False-passing | 99.17% | 89.73% | 94.21% | |
| Elevator | True-passing | 60.25% | 91.83% | 72.76% | 74.23% |
| | False-passing | 92.86% | 63.69% | 75.56% | |
| Email | True-passing | 91.19% | 84.51% | 87.72% | 86.95% |
| | False-passing | 82.50% | 89.95% | 86.06% | |
| ExamDB | True-passing | 100.00% | 89.58% | 94.50% | 92.77% |
| | False-passing | 80.89% | 100.00% | 89.44% | |
| GPL | True-passing | 87.47% | 94.66% | 90.92% | 87.71% |
| | False-passing | 88.29% | 74.82% | 81.00% | |
| ZipMe | True-passing | 96.84% | 91.51% | 94.10% | 94.23% |
| | False-passing | 91.88% | 96.98% | 94.36% | |

*testing data, the better performance of the false-passing product detection model*. Specifically, in the *System-based edition*, the training and testing data are *the least relevant* since the testing set contains the systems which are totally different from the training set. In this edition, the average classification accuracy is 88.44%. Meanwhile, in the *Within-system edition*, the training and testing data are *the most relevant* since both the training and testing data contain products in the buggy versions of the same system. Thus, these products could share some similar characteristics about the programs and tests. Intuitively, CLAP can better capture these attributes and better detect *false-passing* products. Specially, the performance of CLAP in the *Within-system edition* is 92.29% in *Accuracy* and 96.12% in *false-passing* product prediction *Precision*, which are higher than those of *System-based edition* about 4% and 7%, respectively.

The detail performance of CLAP in each system in the *System-based edition* and *Within-system edition* are shown in Table 7 and Table 8, respectively. For the System-based edition, CLAP obtained the best performance in BankAccountTP and ZipMe systems. For the Within-system edition, CLAP correctly detected most of *true-passing* and *false-passing* products in the buggy versions of BankAccountTP and ExamDB systems. Meanwhile, in both editions, the accuracy of CLAP in the buggy versions of Elevator is the lowest. One of the reasons is the existence of flaky tests in their test suites, which negatively affect the model in both the training and testing phases. The impact of such tests on the training phase is discussed in Sec. 6.3.2.

**Table 8**
CLAP's performance on each system in within-system edition

| System | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|
| BankAccountTP | True-passing | 96.28% | 98.23% | 97.25% | 97.39% |
| | False-passing | 98.40% | 96.34% | 97.36% | |
| Elevator | True-passing | 72.22% | 92.86% | 81.25% | 85.71% |
| | False-passing | 95.83% | 82.14% | 88.46% | |
| Email | True-passing | 90.58% | 97.66% | 93.99% | 92.98% |
| | False-passing | 96.67% | 87.00% | 91.58% | |
| ExamDB | True-passing | 98.04% | 100.00% | 99.01% | 98.70% |
| | False-passing | 100.00% | 96.30% | 98.12% | |
| GPL | True-passing | 86.74% | 97.52% | 91.81% | 88.48% |
| | False-passing | 94.00% | 70.71% | 80.54% | |
| ZipMe | True-passing | 88.54% | 91.45% | 89.97% | 90.46% |
| | False-passing | 92.26% | 89.60% | 90.91% | |

In addition, CLAP obtained better *false-passing* product detection *Recall* in the System-based edition (86%) compared to the Version-based edition and Product-based edition (about 78%), although these two later settings have better accuracy. This demonstrates that for the System-based edition, a product has a higher probability to be predicted as a *false-passing* product. This is because for the *false-passing* products of some systems, their values in several attributes are significantly smaller than those figures in the products of the other systems. Consequently, once CLAP learns from these small values, and then predicts new passing products, more products will be predicted as *false-passing* products. For example, the fault diagnosability (DDU') attribute, the average value of the *false-passing* products of system Email is 0.81, while this figure of system ExamDB is only 0.6. If the model learns from products of system ExamDB and then predicts products of system Email, many products of this system, including *true-passing* products, could be predicted as *false-passing* products. As a result, the System-based edition obtained higher *false-passing* product detection *Recall*, yet lower *true-passing* product classification *Precision* compared to the Version-based edition and the Product-based edition.

Our approach performed stably in the Version-based edition and Product-based edition. The reason is that the data separation methods in these two scenarios are quite similar. The only difference is that in the Product-based edition, the training set could contain the products in the same buggy versions as the testing set. Thus, the detection performance of CLAP in the Version-based edition and Product-based edition is quite similar.

**Table 9**
Impact of different training data sizes (the number of systems)

| #System | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|
| 1 | True-passing | 92.02% | 81.88% | 86.65% | 82.60% |
| | False-passing | **74.90%** | 93.19% | 83.05% | |
| 2 | True-passing | 96.82% | 63.07% | 76.38% | 78.47% |
| | False-passing | **68.18%** | 97.44% | 80.23% | |
| 3 | True-passing | 95.37% | 76.90% | 85.14% | 85.19% |
| | False-passing | **77.03%** | 95.40% | 85.24% | |
| 4 | True-passing | 90.18% | 81.33% | 85.53% | 84.81% |
| | False-passing | **79.48%** | 89.10% | 84.02% | |
| 5 | True-passing | 91.19% | 84.51% | 87.72% | 86.95% |
| | False-passing | **82.50%** | 89.95% | 86.06% | |

**Table 10**
Impact of attributes on CLAP's performance

| System | Attributes | Product | Precision | Recall | F1-Score | Accuracy |
|---|---|---|---|---|---|---|
| GPL | Product implementation | True-passing | 84.69% | 87.71% | 86.17% | 81.52% |
| | | False-passing | 74.80% | **69.69%** | 72.15% | |
| | Test adequacy | True-passing | 80.45% | 99.74% | 89.06% | 83.92% |
| | | False-passing | 99.07% | 53.69% | 69.64% | |
| | Test effectiveness | True-passing | 78.74% | 96.59% | 86.76% | 80.64% |
| | | False-passing | 88.50% | 50.18% | 64.05% | |
| | All | True-passing | 87.47% | 94.66% | 90.92% | 87.71% |
| | | False-passing | 88.29% | 74.82% | 81.00% | |
| ZipMe | Product implementation | True-passing | 86.09% | 99.16% | 92.16% | 91.53% |
| | | False-passing | 99.00% | **83.82%** | 90.78% | |
| | Test adequacy | True-passing | 76.82% | 93.54% | 84.36% | 82.57% |
| | | False-passing | 91.61% | 71.50% | 80.32% | |
| | Test effectiveness | True-passing | 87.39% | 48.92% | 62.73% | 70.79% |
| | | False-passing | 96.84% | 91.51% | 94.10% | |
| | All | True-passing | 96.84% | 91.51% | 94.10% | 94.23% |
| | | False-passing | 91.88% | 96.98% | 94.36% | |

#### *6.3.2. Impact of training data sizes*

Table 9 shows the performance of CLAP by the training data size with SVM. In general, *the larger the training data set, the better performance of* CLAP. If the training set contains the buggy versions of only one system, CLAP's *Accuracy* is 82.60% and its *false-passing* product classification *Precision* is 74.9%. When the training data increases to five systems, the figures of CLAP are improved by about 5% and 7%, respectively. This is reasonable because by learning from more data, CLAP can observe and recognize better *false-passing* and *true-passing* products.

However, *if added training data contains noises, it could decrease the performance of* CLAP. For example, the *Accuracy* of CLAP dropped from 82.60% to 78.47% when the training set is increased from one to two systems. This figure also slightly declined, about 0.4%, when the training set increased from three to four systems. The reason is that in the added systems, there are several "flaky" tests in their buggy versions. The results of these tests are inconsistent, sometimes they are passed and sometimes failed without any code changes [28]. For example, some tests invoked `random()` function to get a random number, and then the tests failed due to the numbers are generated differently in each run, not because of the bugs in the system. Due to their inconsistent test results, these tests might create noises for *false-passing* product detection tools. Consequently, the performance of CLAP in these cases are decreased.

### 6.4. Intrinsic Analysis (RQ4)

To study the impact of each attribute set on CLAP's performance, we built several variants of CLAP, each of them enables a single attribute set: *product implementation*, *test adequacy*, and *test effectiveness*. In this experiment, we apply these variants of CLAP in setting cross-system edition of the two largest systems, GPL and ZipMe. Note in this experiment, we used SVM in CLAP, the impact of different classifiers and different setting editions on CLAP's performance has been shown in Sec. 6.1 and Sec. 6.3.1.

*The product implementation attributes help* CLAP *achieve better performance compared to the test adequacy and test effectiveness attributes.* The reason is that the *product implementation* attributes directly provide information about buggy statements in the products. The model using these attributes can capture the information about the bugginess of products, and have better performance. As seen in Table 10, for GPL, by using the *product implementation* attributes, CLAP obtains higher *false-passing* product detection *Recall* than the others. Specifically, only using these attributes, CLAP obtained 69.69% in *false-passing* product detection *Recall*, while these figures of CLAP with the *test adequacy* attributes and *test effectiveness* attributes are just 53.69% and 50.18%, respectively. For ZipMe, by using the *product implementation* attributes, CLAP can correctly detect more than 8 *false-passing* products, while this figure when using the *test adequacy* attributes is only 7. Although, using *test effectiveness* attributes, CLAP's *false-passing* product detection *Recall* is better than using the *product implementation* attributes, CLAP's *true-passing* product detection Recall is much lower, only 48.92%.

Meanwhile, based on only test quality attributes (the *test adequacy* or *test effectiveness*), the model faces difficulties on distinguishing *true-passing* and *false-passing* products. As a result, the CLAP's variants with these attribute have a very high *true-passing* product detection *Recall* but low *false-passing* products detection *Recall* or vice versa. Indeed, the information about test quality is an important factor for CLAP to detect *true-passing* and *false-passing* products. However, the low quality of the test suite is just a sign showing that the test result is less reliable but it cannot confirm the bugginess of the product. Thus, in some cases, using only test quality attributes cannot help to detect *false-passing* products.

As expected, CLAP *obtained the best results when the failure indications are measured based on all three aspects: product implementation, test adequacy, and test effectiveness*. By using all of these attributes, CLAP has more comprehensive information to evaluate the bugginess in the product's source code, as well as the reliability of the product's overall test result. Thus, *all of these attributes should be used together in* CLAP *to achieve the best performance*.

### 6.5. Time Complexity (RQ5)

On average, CLAP took 53 seconds to measure attributes of the passing products of a buggy version (about 2.5 seconds/product). Specifically, each buggy version of ExamDB

took only 3 seconds, meanwhile this figure for each version of ZipMe is 192 seconds. Indeed, running time of CLAP depends on the number of sampled products of each system, the number of test cases of each product, and the system's size. The reason is that our attributes are calculated on each passing product of the system. Also, they are measured based on the failure indications investigated from all the failed tests of the failing products of the buggy version. Thus, ExamDB, which contains the least number of sampled products, has the smallest running time. Meanwhile, ZipMe has the largest running time since it is sampled into a large number of products, each product is tested by a large number of test cases, and this is also the largest system.

### 6.6. Threats to Validity

The main threats to the validity of our work are consisted of internal, external, and construct validity threats.

**Threats to internal validity** mainly lie in the correctness of the ground truth which we labeled for the passing products. To mitigate this threat, we applied a systematic process to label these products. In the step which requires manual investigation and test generation, we made every effort to carefully investigate the products. Another threat is that the ratio of *false-passing* and *true-passing* in our benchmark might be different from the ratio in practice. To address this threat, we are planning to collect real-world *false-passing* and *true-passing* products in larger SPL systems to evaluate our technique.

**Threats to external validity** mainly lie in the benchmark used in our experiments. Although the dataset uses the systems which are widely used in the existing work, this dataset only contains artificial bugs of Java SPL systems, so we cannot conclude the similar results for real-world faults and SPL systems in other programming languages. In addition, another threat may come from the quality of the products' test suites. For instance, the product could contain low quality test cases (e.g., flaky tests). To mitigate this threat, we chose the dataset containing a large number of buggy products with a diversity of artificial faults and each products are tested by a large number of test cases. Also, the dataset contains both single-bug and multiple-bug buggy systems. To address these threats, we are planning to collect more real-world variability bugs in larger SPL systems to evaluate our technique.

**Threats to construct validity** mainly lie in the rationality of the assessment metrics. To reduce this threat, we chose the metrics which are widely used in the related studies [23]. For evaluating our approach of detecting *false-passing* products, we use common metrics in classification problem: Precision, Recall, F1-Score, and Accuracy. For evaluating how CLAP helps to mitigate the negative impact of *false-passing* products on FL performance, we used the most popular metrics in FL studies, Rank and EXAM. For assessing the performance of the classifiers, we following the instructions from the related studies [23, 29] for choosing the value of $k = 5$ for $k$-fold cross-validation. Another threat may come from our selected SBFL metrics. To reduce this threat, we chose five most popular SBFL metrics, which are widely used in FL studies [7, 24].

## 7. Related Work

**Coincidental Correctness Detection and Impact Mitigation.** Coincidental correctness has been proven as a prevalent problem in software testing [2]. Also, practical experiments have been conducted to demonstrate that this problem adversely affects FL performance [2, 30, 31]. Many techniques have been proposed to detect coincidentally passed tests [2, 32, 33, 34], which execute the faults, yet produce correct outputs. After that, they cleansed the test suites from these detected unreliable tests to enhance FL performance. Bandyopadhyay et al. [35] proposed an approach to predict and weight coincidentally passed tests to calculate the suspiciousness scores. In general, these approaches investigate each individual passed test case to detect coincidental correctness and boost FL performance in non-configurable code. These studies are different from our work which is designed for configurable code. Instead of verifying test cases, we verify the overall test result of a product, i.e., the state of passing all its tests. In addition, CLAP focuses on improving variability FL performance in SPL systems.

**Test Suite Effectiveness Measurement.** Various metrics have been proposed to measure the quality of the test suites. Specially, code coverage is one of the most popular metrics, which measures the percentage of code elements (e.g., statements, branches, decisions, etc.) covered by test suites [36]. Also, Perez et al. [15] proposed DDU which aims at measuring the effectiveness of the suites in term of applying SBFL to detect faults in the corresponding source code. Gonzalez-Sanchez et al. [37] employed information gain algorithm to predict the efficiency of the test suites based on the system's size, the coverage density, and the uniform of coverage distribution. Baudry et al. [38] proposed a test criterion based on the Dynamic Basic Block, the test data (traces), and the software control structure to evaluate the fault localizing capacity of the test cases. Besides, mutation testing techniques [39, 40] are also proposed to evaluate the effectiveness of the test suites in detecting mutated faults.

**Testing SPL Systems.** Testing an SPL system is a complex and costly task since the variety of the interactions of system features and a large number of derived products. A large number of studies have been conducted, and various testing strategies have been proposed. To efficiently assure software quality, various *sampling algorithms* have been introduced [41, 42, 43, 44, 13, 12, 45]. In addition, to improve the efficiency of the testing process, several approaches about *configuration selection* [46, 47] and *configuration prioritization* [48, 49, 50] have also been proposed. These studies' objective is different from ours. They target the size of sample sets and the fault-detection capability, i.e., whether the faults of the system are explored via testing sampled products. Meanwhile, our focus is the consistency of the overall testing results of the sampled products. Specifically, we verify the sampled products and their test suites to

guarantee that if a fault cause failure for a product, any other product containing the same fault should be failed by at least a test. As shown in Sec. 4, the failure indications given by CLAP could be utilized to guide the existing test generators to improve the quality of test suites.

**Fault Localization.** There are various approaches proposed to identify the locations of faults in programs [7, 51, 52, 53]. Program slicing [54, 55] improves the efficiency of finding faults by detecting and removing irrelevant elements in source code. In addition, SBFL leverages the execution information (i.e., program spectra) of a program to localize bugs which cause program's failures. Moreover, several studies [56, 57] have shown that the FL performance is improved by combining SBFL technique with slicing methods. Also, Arrieta et al. [5] use SBFL metrics to localize bugs in SPL system at the feature-level. For localizing variability bugs in SPL systems at the statement-level, VARCOP [6, 58] analyzes the overall test results of the sampled products and their source code to isolate suspicious code statements. After that, each of isolated suspicious statements is measured the suspiciousness according to both the overall test results of the sampled products and the individual result of each test case. As shown in Sec. 6.2, our approach could be applied before localizing variability faults to improve the fault localization performance of existing techniques.

## 8. Conclusion

In an SPL system, variability bugs can cause failures in certain products (buggy products). However, these buggy products could be incorrectly considered as passing products due to the ineffectiveness of their test suites in detecting the bugs. The buggy products which still passed all the tests in their test suits are called *false-passing* products. Our empirical study has shown that *false-passing* products can produce negative impacts on FL performance. This paper introduces CLAP, a novel approach to detect *false-passing* products in SPL systems. Our key idea is that the stronger failure indications in the passing product, the more likely the product is *false-passing*. The failure indications are derived from the failing products of the system based on products' implementation and products' test quality. Then a passing product is measured according to these indications to determine its possibility to be *false-passing*. Our experimental results on a large dataset of 823 buggy SPL systems with 14,191 *false-passing* products and 22,555 *true-passing* products show that CLAP can effectively classify *false-passing* and *true-passing* products of the SPL systems, with the average Accuracy of more than 90%. In different experimental scenarios, the precision of *false-passing* product detection by CLAP is up to 96%. Furthermore, we propose simple and effective methods to mitigate the negative impact of *false-passing* products on fault localization performance. These methods can effectively help the state-of-the-art FL techniques improve their performance by up to 34%.

## Acknowledgments

Nguyen Thu Trang, ID VNU.2021.NCS.09, thanks The Development Foundation of Vietnam National University, Hanoi for sponsoring this research.

## References

[1] T. T. Chekam, M. Papadakis, M. Cordy, Y. L. Traon, Killing stubborn mutants with symbolic execution, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (2) (2021) 1–23.

[2] W. Masri, R. A. Assi, Prevalence of coincidental correctness and mitigation of its impact on fault localization, ACM transactions on software engineering and methodology (TOSEM) 23 (1) (2014) 1–28.

[3] Y. Lei, C. Sun, X. Mao, Z. Su, How test suites impact fault localisation starting from the size, IET software 12 (3) (2018) 190–205.

[4] W. Wang, Y. Wu, Y. Liu, A passed test case cluster method to improve fault localization, Journal of Circuits, Systems and Computers 30 (03) (2021) 2150053.

[5] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, L. Etxeberria, Spectrum-based fault localization in software product lines, Information and Software Technology 100 (2018) 18–31.

[6] T. T. Nguyen, K.-T. Ngo, S. Nguyen, H. Vo, A variability fault localization approach for software product lines, IEEE Transactions on Software Engineering.

[7] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (8) (2016) 707–740.

[8] H. Zhu, P. A. Hall, J. H. May, Software unit test coverage and adequacy, Acm computing surveys (csur) 29 (4) (1997) 366–427.

[9] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria, in: Proceedings of 16th International conference on Software engineering, IEEE, 1994, pp. 191–200.

[10] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur, Effect of test set minimization on fault detection effectiveness, Software: Practice and Experience 28 (4) (1998) 347–369.

[11] K.-T. Ngo, T.-T. Nguyen, S. Nguyen, H. D. Vo, Variability fault localization: a benchmark, in: Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A, 2021, pp. 120–125.

[12] C. Nie, H. Leung, A survey of combinatorial testing, ACM Computing Surveys (CSUR) 43 (2) (2011) 1–29.

[13] I. Abal, C. Brabrand, A. Wasowski, 42 variability bugs in the linux kernel: a qualitative analysis, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 421–432.

[14] B. J. Garvin, M. B. Cohen, Feature interaction faults revisited: An exploratory study, in: 2011 IEEE 22nd International Symposium on Software Reliability Engineering, IEEE, 2011, pp. 90–99.

[15] A. Perez, R. Abreu, A. Van Deursen, A theoretical and empirical analysis of program spectra diagnosability, IEEE Transactions on Software Engineering.

[16] G. Guizzo, F. Sarro, M. Harman, Cost measures matter for mutation testing study validity, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1127–1139.

[17] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, M. Ribeiro, A systematic literature review of techniques and metrics to reduce the cost of mutation testing, Journal of Systems and Software 157 (2019) 110388.

[18] W. E. Wong, A. P. Mathur, Reducing the cost of mutation testing: An empirical study, Journal of Systems and Software 31 (3) (1995) 185–196.

[19] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on software engineering and methodology (TOSEM) 20 (3) (2011) 1–32.

[20] R. Abreu, P. Zoeteweij, A. J. Van Gemund, Spectrum-based multiple fault localization, in: 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2009, pp. 88–99.

[21] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 416–419.

[22] C. Pacheco, M. D. Ernst, Randoop: feedback-directed random testing for java, in: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, 2007, pp. 815–816.

[23] G. James, D. Witten, T. Hastie, R. Tibshirani, An introduction to statistical learning, Vol. 112, Springer, 2013.

[24] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, B. Keller, Evaluating and improving fault localization, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 609–620.

[25] E. Wong, T. Wei, Y. Qi, L. Zhao, A crosstab-based statistical method for effective fault localization, in: 2008 1st international conference on software testing, verification, and validation, IEEE, 2008, pp. 42–51.

[26] S. Strüder, M. Mukelabai, D. Strüber, T. Berger, Feature-oriented defect prediction, in: Proceedings of the 24th ACM conference on systems and software product line: Volume A-Volume A, 2020, pp. 1–12.

[27] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. T. M. Phuong, P. H. Thong, Empirical study of software defect prediction: a systematic mapping, Symmetry 11 (2) (2019) 212.

[28] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, D. Marinov, Deflaker: Automatically detecting flaky tests, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 433–444.

[29] M. Kuhn, K. Johnson, et al., Applied predictive modeling, Vol. 26, Springer, 2013.

[30] X. Wang, S.-C. Cheung, W. K. Chan, Z. Zhang, Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 45–55.

[31] F. Steimann, M. Frenkel, 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, 2013, pp. 314–324.

[32] Z. Li, M. Li, Y. Liu, J. Geng, Identify coincidental correct test cases based on fuzzy classification, in: 2016 International Conference on Software Analysis, Testing and Evolution (SATE), IEEE, 2016, pp. 72–77.

[33] X. Xue, Y. Pang, A. S. Namin, Trimming test suites with coincidentally correct test cases for enhancing fault localizations, in: 2014 IEEE 38th Annual Computer Software and Applications Conference, IEEE, 2014, pp. 239–244.

[34] W. Masri, R. Abou Assi, Cleansing test suites from coincidental correctness to enhance fault-localization, in: 2010 third international conference on software testing, verification and validation, IEEE, 2010, pp. 165–174.

[35] A. Bandyopadhyay, Mitigating the effect of coincidental correctness in spectrum based fault localization, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, 2012, pp. 479–482.

[36] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72–82.

[37] A. Gonzalez-Sanchez, H.-G. Gross, A. J. van Gemund, Modeling the diagnostic efficiency of regression test suites, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2011, pp. 634–643.

[38] B. Baudry, F. Fleurey, Y. Le Traon, Improving test suites for efficient fault localization, in: Proceedings of the 28th international conference on Software engineering, 2006, pp. 82–91.

[39] E. S. Mresa, L. Bottaci, Efficiency of mutation operators and selective mutation strategies: An empirical study, Software Testing, Verification and Reliability 9 (4) (1999) 205–232.

[40] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE transactions on software engineering 37 (5) (2010) 649–678.

[41] M. F. Johansen, Ø. Haugen, F. Fleurey, An algorithm for generating t-wise covering arrays from large feature models, in: Proceedings of the 16th International Software Product Line Conference-Volume 1, 2012, pp. 46–55.

[42] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, J. Lawrence, Ipog/ipog-d: efficient test generation for multi-way combinatorial testing, Software Testing, Verification and Reliability 18 (3) (2008) 125–148.

[43] D. Marijan, A. Gotlieb, S. Sen, A. Hervieu, Practical pairwise testing for software product lines, in: Proceedings of the 17th international software product line conference, 2013, pp. 227–235.

[44] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, J. Sincero, Configuration coverage in the analysis of large-scale system software, in: Proceedings of the 6th Workshop on Programming Languages and Operating Systems, 2011, pp. 1–5.

[45] S. Oster, F. Markert, P. Ritter, Automated incremental pairwise testing of software product lines, in: International Conference on Software Product Lines, Springer, 2010, pp. 196–210.

[46] M. Greiler, A. van Deursen, M.-A. Storey, Test confessions: A study of testing practices for plug-in systems, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 244–254.

[47] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, E. S. De Almeida, On strategies for testing software product lines: A systematic literature review, Information and Software Technology 56 (10) (2014) 1183–1199.

[48] S. Nguyen, H. Nguyen, N. Tran, H. Tran, T. Nguyen, Feature-interaction aware configuration prioritization for configurable code, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 489–501.

[49] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, G. Saake, Similarity-based prioritization in software product-line testing, in: Proceedings of the 18th International Software Product Line Conference-Volume 1, 2014, pp. 197–206.

[50] S. Nguyen, Feature-interaction aware configuration prioritization, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 974–976.

[51] S. Kusumoto, A. Nishimatsu, K. Nishie, K. Inoue, Experimental evaluation of program slicing for fault localization, Empirical Software Engineering 7 (1) (2002) 49–76.

[52] F. Tip, T. Dinesh, A slicing-based approach for locating type errors, ACM Transactions on Software Engineering and Methodology (TOSEM) 10 (1) (2001) 5–55.

[53] R. A. DeMillo, H. Pan, E. H. Spafford, Critical slicing for software fault localization, ACM SIGSOFT Software Engineering Notes 21 (3) (1996) 121–134.

[54] M. Weiser, Program slicing, IEEE Transactions on software engineering (4) (1984) 352–357.

[55] H. Agrawal, J. R. Horgan, Dynamic program slicing, ACM SIGPlan Notices 25 (6) (1990) 246–256.

[56] N. B. Chaleshtari, S. Parsa, Smbfl: slice-based cost reduction of mutation-based fault localization, Empirical Software Engineering 25 (5) (2020) 4282–4314.

[57] X. Li, A. Orso, More accurate dynamic slicing for better supporting software debugging, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, 2020, pp. 28–38.

[58] S. Nguyen, Configuration-dependent fault localization, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE Computer Society, 2019, pp. 156–158.