# A Combinatorial Testing-Based Approach to Fault Localization

Laleh Sh. Ghandehari, Yu Lei, Raghu Kacker, Richard Kuhn, Tao Xie, David Kung

**Abstract**— Combinatorial testing has been shown to be a very effective strategy for software testing. After a failure is detected, the next task is to identify one or more faulty statements in the source code that have caused the failure. In this paper, we present a fault localization approach, called BEN, which produces a ranking of statements in terms of their likelihood of being faulty by leveraging the result of combinatorial testing.

BEN consists of two major phases. In the first phase, BEN identifies a combination that is very likely to be failure-inducing. A combination is failure-inducing if it causes any test in which it appears to fail. In the second phase, BEN takes as input a failure-inducing combination identified in the first phase and produces a ranking of statements in terms of their likelihood to be faulty. We conducted an experiment in which our approach was applied to the *Siemens suite* and four real-world programs, *flex*, *grep*, *gzip* and *sed*, from Software Infrastructure Repository (SIR). The experimental results show that our approach can effectively and efficiently localize the faulty statements in these programs.

**Index Terms— Combinatorial Testing, Fault Localization, Debugging**

## 1. INTRODUCTION

Combinatorial testing is based on the observation that a large number of software failures are caused by interactions of only a few input parameters [26]. A t-way combinatorial test set, or simply a t-way test set, is designed to cover all the t-way combinations, i.e., combinations involving any t parameters [8][9][29]. Typically, t is a small number and is referred to as the strength of a combinatorial test set [25][26]. When the input parameters are properly modeled, a t-way test set could trigger any failure caused by interaction of at most t parameters. Empirical studies have shown that combinatorial testing is very effective in practice [6][16][25].

After a failure is detected during combinatorial testing, the next task is locating the fault that caused the failure. In this paper, we present a fault localization approach called BEN that leverages the result of combinatorial testing. BEN takes as input a combinatorial test set and the execution status, i.e., pass or fail, of each test, and produces as output a ranking of statements in terms of their likelihood to be faulty.

Most research in combinatorial testing has focused on developing efficient combinatorial test generation algorithms [8][29][33], or demonstrating the effectiveness of combinatorial testing in different application domains [6][15][44][48]. Several approaches have been developed to identify failure-inducing combinations in a combinatorial test set [49][57]. A failure-inducing combination, or simply an inducing combination, is a combination that causes all tests containing this combination to fail [34][57]. These approaches, however, are not designed to locate faulty statements in the source code.

A significant amount of research has been reported on spectrum-based approaches to fault localization [1][23][40][50]. A program spectrum records information about certain aspects of a test execution [50], such as function call counts, program paths, program slices and use-def chains [40]. Examples of spectrum-based methods include Tarantula [24], set union, set intersection, and nearest neighbor [40]. These approaches identify faulty statements by analyzing the spectra of passing and failing test executions [24][40][31]. These approaches are not designed to work with combinatorial testing. However, they can be applied to analyze test executions obtained from combinatorial testing, provided that the test executions were traced. In case that a combinatorial test set is already executed without being traced, which is often the case in practice considering that testing and debugging are fundamentally different activities and are often performed separately, the test set must be re-executed before these approaches could be applied. In contrast, our approach does not require every test execution to be traced and is designed to be applied after normal testing is performed where test executions are not traced. We will compare our approach, i.e., BEN, to these approaches both analytically (Section 6.2) and experimentally (Section 5.2.3).

Our approach consists of two major phases, *inducing combination identification* and *faulty statement localization*. In the first phase, BEN takes as input a t-way combinatorial test set. It adopts an iterative framework to identify an inducing combination of size t or larger. During each iteration, a set F of tests is analyzed. Initially F is the t-way combinatorial test set taken as input by BEN. BEN first identifies the set π of


- *Laleh Sh. Ghandehari, Yu Lei and David Kung are with the Department of Computer Science and Engineering, University of Texas at Arlington, Texas, USA. E-mail: laleh.shikhgholamhosseing@mavs.uta.edu, ylei@uta.edu and kung@uta.edu.*
- *Raghu Kacker, and Richard Kuhn are with the Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, Maryland, USA. E-mail: raghu.kacker@nist.gov and kuhn@nist.gov.*
- *Tao Xie is with the Department of Computer Science, North Carolina State University, E-mail: taoxie@illinois.edu*

all t-way suspicious combinations in F, and ranks them based on their likelihood to be inducing. Suspicious combinations are candidates of inducing combinations.

Next, our approach generates a set F′ of new tests. If all the tests containing a suspicious combination c in F′ fail, c is marked as an inducing combination, and the process stops. Otherwise, all the tests in F′ are added to F and the process is repeated until a t-way combination is marked as an inducing combination or a stopping condition is satisfied. In the latter case, no t-way inducing combination is identified and we increase the size of inducing combination. That is, we try to identify a (t+1)-way inducing combination. This process is repeated until an inducing combination is found. Note that this process must terminate, as a failing test is by definition an inducing combination.

The novelty of our approach in this phase lies in the fact that we rank suspicious combinations based on two notions, namely the *combination suspiciousness* and *environment suspiciousness* of a combination. Informally, the environment of a combination consists of parameter values that appear in the same test case but do not appear in the combination. The higher the combination suspiciousness of a combination, the lower its environment suspiciousness, the higher this combination is ranked. Moreover, new tests are generated for the most suspicious combinations. Let f be a new test generated for a suspicious combination c. Test f is generated such that it contains c and the environment suspiciousness for c is minimized. If f fails, it is more likely to be caused by c instead of other values in f.

In the second phase of our approach, i.e., faulty statement localization, BEN systematically generates a small group of tests from an inducing combination such that the execution traces of these tests can be analyzed to quickly locate the faults. One of the tests in the group is referred to as the core member, which contains the inducing combination and produces a failing test execution. The other tests in the group are referred to as the derived members, which are derived from the core member in a way such that they are likely to execute a trace that is very similar to the trace of the core member but produce a different outcome, i.e., a passing execution. The spectrum of the core member is then compared to the spectrum of each derived member to produce a ranking of statements in terms of their likelihood to be faulty.

Our approach differs from existing spectrum-based approaches, which do not deal with the problem of test generation. Instead, they assume that an existing test set is generated randomly and/or using other techniques [24][40][50].

The second phase of BEN is inspired by the notion of nearest neighbor [40]. The key idea of nearest neighbor is that faulty statements are likely to appear in the execution trace of a failing test but not in the execution trace of a passing test that is as similar to this failing test as possible. If two tests are significantly different, they are likely to represent different application scenarios. Thus, the differences in the execution traces of these two tests are likely due to program logic, instead of faults. The novelty of our approach lies in the fact that we generate, in a systematic manner, a failing test, i.e., the core member, and then derive its nearest neighbors from this failing test, i.e., the derived members. This is in contrast with the approach in [40], which executes a large number of tests from which a failing test and its nearest neighbors are selected.

We report an experiment in which we applied our approach to the *Siemens suite* and four real-world programs, *flex*, *grep, gzip* and *sed*, in the Software Infrastructure Repository (SIR) [46]. The *Siemens suite* has been used in several studies to evaluate fault localization methods [23][40][50]. It contains seven relatively small programs, each of which has a number of faulty versions. Similarly, the real-world programs, i.e., *flex*, *grep, gzip* and *sed*, have a number of faulty versions and have been used in other studies such as [22][35][36][38][39]. Each of the faulty versions in SIR contains a single-fault. In order to evaluate the performance of BEN with multiple faults, we created several faulty versions that contain multiple faults.

The results show that our approach is effective in localizing faulty statements and also efficient in that only a small number of tests need to be executed and traced. For example, one of the *grep* programs called *grep3* has 18 faulty versions. Among these faulty versions, four versions were detected by a 2-way test set consisting of 121 tests. On average, BEN generated and executed 17.5 additional tests and traced 6.75 tests for these four versions. One needs to examine only 0.64% (on average) of the code to locate the faulty statement.

Moreover, we compared the results of BEN and two other spectrum based approaches, Tarantula [24] and Ochiai [31]. Since Tarantula and Ochiai do not deal with test generation, they were applied to the initial combinatorial test set. Our experimental results show that BEN performed better than or as well as Tarantula and Ochiai for all the programs, but BEN requires a significantly smaller number of test executions to be traced and analyzed.

The approach presented in this paper is the extension of our previous work, which has been presented in [19] and [18]. To the best of our knowledge, our work is the first to deal with code-based fault localization based on combinatorial testing. Existing work in this area, i.e., fault localization based on combinatorial testing, has mainly dealt with the problem of how to identify inducing combinations [34][49][45][57].

The remainder of this paper is organized as follows. Section 2 explains basic concepts and assumptions of our approach. Section 3 presents the BEN approach. Section 4 gives an example to illustrate the approach. Section 5 reports the experimental results of applying our approach to the subject programs. Section 6 discusses existing work on fault localization. Section 7 provides the concluding remarks and plans for future work.

## 2. PRELIMINARIES

In this section, we introduce the basic concepts and assumptions needed in our approach.

### 2.1. Basic Concepts

Assume that the system under test (SUT) has a set P of k

input parameters, denoted by $P = \{p_1, p_2, \ldots, p_k\}$. Let $d_i$ be the domain of parameter $p_i$. That is, $d_i$ contains all possible values that $p_i$ could take. Let $D = \{d_1 \cup d_2 \cup \ldots \cup d_k\}$. Let $\Pi = d_1 \times d_2 \times \ldots \times d_k$. Let $S$ be the set of program statements.

**Definition 1**. (*Test Case*) A test case or simply a test is a function that assigns a value to each parameter. Formally, a test is a function $f: P \rightarrow D$.

**Definition 2**. (*Constraint*) A constraint $\psi$ is a function that maps a test case to a Boolean value *true* or *false*, formally, $\psi: \Pi \longrightarrow \{true, false\}$.

The SUT includes a set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_{|\Psi|}\}$ of constraints. We use $\Gamma \subseteq \Pi$ to represent all valid tests for the SUT. A test $f \in \Gamma$ is valid if and only if $\forall \psi \in \Psi, \psi(f) = \text{true}$. In the rest of the paper, we refer to a valid test simply as a test unless otherwise specified.

**Definition 3**. (*Test Oracle*) A test oracle determines whether the execution of a test is *pass* or *fail*. Formally, a test oracle is a function $r: \Gamma \rightarrow \{\text{pass}, \text{fail}\}$.

**Definition 4.** (*Combination*) A combination $c$ is a test $f$ restricted to a non-empty, subset $M$ of parameters in $P$. Formally, $c = f|_M$ where $M \subseteq P$, and $|M| > 0$.

In the preceding definition, $M$ is a subset of $P$. Thus, a test is a combination where $M = P$. We use $\text{dom}(c)$ to denote the domain of $c$, which is a set of parameters involved in $c$. Note that $\text{dom}(c)$ is the domain of a function, which is different from the domain of a parameter.

A combination of size one is a special combination, which we refer to as a component. Since there is only one parameter involved, we denote a component $o$ as an assignment, i.e., $o = p \leftarrow v$, where $o(p) = v$.

**Definition 5**. (*Component Containment*) A component $o = p \leftarrow v$ is contained in a combination $c$ denoted by $o \in c$, if and only if $p \in \text{dom}(c)$ and $c(p) = v$.

**Definition 6**. (*Combination Containment*) A combination $c$ is contained in a test $f$, denoted by $c \subseteq f$, if and only if $\forall p \in \text{dom}(c), f(p) = c(p)$.

**Definition 7**. (*Inducing Combination*) A combination $c$ is failure-inducing, or simply inducing, if any test $f$ in which $c$ is contained fails. Formally, $\forall f \in \Gamma: c \subseteq f \Longrightarrow r(f) = \text{fail}$.

Definition 7 is consistent with the definition of inducing combinations in previous work [57][45][34][49].

**Definition 8**. (*Inducing Probability*) The inducing probability of a combination $c$ is the ratio of the number of all failing tests containing $c$ to the number of all tests containing $c$. The inducing probability is computed by

$$\frac{|\{f \in \Gamma | r(f) = \text{fail} \wedge c \subseteq f\}|}{|\{f \in \Gamma | c \subseteq f\}|}$$

The computation of inducing probability requires all tests containing a combination, which is often not possible in practice. This notion is, however, useful to evaluate our experimental results. By Definition 7, an inducing combination is a combination whose inducing probability is one.

**Definition 9**. (*Suspicious Combination*) A combination $c$ is a suspicious combination in a test set $F \subseteq \Gamma$ if $c$ is contained only in failing tests in $F$. Formally, $\forall f \in F: c \subseteq f \Rightarrow r(f) = \text{fail}$.

Inducing combinations must be suspicious combinations, but suspicious combinations may or may not be inducing combinations.

**Definition 10**. (*Test Spectrum*) A test spectrum is a membership function $\gamma$ that determines whether a statement is exercised by a test (or precisely the execution of a test). Formally, $\gamma: S \times \Gamma \rightarrow \{\text{true}, \text{false}\}$, where $\gamma(s, f) = \text{true}$ if $s \in S$ is executed by $f \in \Gamma$, and $\gamma(s, f) = \text{false}$ otherwise.

In the rest of the paper, we also use $\gamma(f)$ to represent all the statements that are executed by $f$. Formally, $\gamma(f) = \{s \in S \mid \gamma(s, f) = true\}$.

### 2.2. Assumptions

In this section, we present several assumptions that must hold to apply BEN.

**Assumption 1.** There exists an input parameter model of the SUT.

This assumption is also required by combinatorial testing. Our approach is designed to be applied after combinatorial testing has been performed, and our approach uses the same input parameter model used to perform combinatorial testing. In cases that the SUT has multiple points of entry, an input parameter model could be created for each entry point, and our approach could be applied to one entry point at a time.

**Assumption 2**. The output of the SUT is deterministic. In other words, the SUT always produces the same output for a given test.

**Assumption 3**. There exists a test oracle that determines the status of a test execution, i.e., pass or fail.

Assumption 3 is made to simplify the presentation of our approach. The construction of a test oracle is an independent research problem. Test oracles can be derived automatically or semi-automatically from formal or informal specifications [10][13][59]. When a test oracle exists, our approach can be fully automated. When a test oracle does not exist, our approach can still be applied, but the user needs to assist in determining the execution status of a test case. We note that the test oracle problem is common for many testing and fault localization approaches, including spectrum-based fault localization approaches such as Tarantula. We refer the reader to existing literature for more detailed discussion on the test oracle problem [4].

**Assumption 4**. There is at least one failing and one passing test in the initial test set.

If there is no failing test, no fault is detected. Fault localization is typically performed when at least one fault is detected. If there is no passing test, the fault is likely easy to locate.

## 3. Approach

In this section, we present the BEN approach. BEN consists of two major phases, inducing combination identification and faulty statement localization. BEN assumes that a combinatorial test set has been executed on the subject program. Thus, the execution status of each test is known. Also, it assumes that the input parameter model used to generate the combinatorial test set is known. An input parameter model includes a set of parameters, each of which has a set of values, and a set of constraints that must be

satisfied for a test to be valid.

The output of our approach is the ranking of statements such that the higher a statement is ranked, the more likely it is faulty. In the rest of this section, we explain the details of BEN.

### 3.1. Phase 1: Inducing Combination Identification

This phase takes three inputs, including an input parameter model Ω, a combinatorial test set $F_0$ created based on Ω, and the strength t of $F_0$. It produces as output an inducing combination, or more precisely the top ranked suspicious combination.

#### 3.1.1. Framework

As shown in Fig 1, our approach adopts an iterative framework in this phase. During each iteration, the identify algorithm is used to analyze a set F of test cases and identify an ℓ-way inducing combination.

Initially, F is the initial combinatorial test set $F_0$, and ℓ is the strength t of the initial test set. If the *identify* algorithm identifies an ℓ-way inducing combination, c (line 5), the *while* loop stops and reports c as an inducing combination. If no ℓ-way inducing combination is found, i.e. the *identify* algorithm returns null (line 2), ℓ will be incremented. In the next iteration, the framework searches for inducing combinations of size ℓ+1. As shown in Fig 2, new tests may be added into F by the *identify* algorithm each time it is called.

Based on assumption 4, there is at least one failing test in the initial test set. Recall that a failing test is an inducing combination by definition. Therefore, there is at least one inducing combination in the initial test set. Thus, the framework must terminate.

**The Phase 1 Framework**

```
1   l ← t and F ← F0
2   while ((c ← identify(Ω, l, F)) = null) {
3       l ← l + 1
4   }
5   return c
```

Fig 1. The Framework for Identifying Inducing Combination

#### 3.1.2. Algorithm Identify

Algorithm *identify* is shown in Fig 2, and is designed to find an ℓ-way inducing combination in the test set F. It takes as input the input parameter model, Ω, test set F and ℓ. The algorithm consists of two main steps:

(1) Rank generation: In this step, we first identify all the ℓ-way suspicious combinations in F (line 3). Then, the component suspiciousness of each component, combination suspiciousness, $\rho_c$, and environment suspiciousness, $\rho_e$, of each suspicious combination are computed (line 7 and line 10). The different types of suspiciousness will be defined in Section 3.1.3. Finally, a ranking of the suspicious combinations is produced (line 12).

(2) Test generation: In this step, for a user-specified number of top-ranked suspicious combinations, a set of new tests is generated (line 16). Note that the user could specify the number of top-ranked suspicious combinations and the number of tests generated for each top-ranked combination. If an inducing combination is not found, all the new tests in F′ are added to the test set F to refine the ranking of

**Algorithm identify**

```
1  while ( true ) {
2    // Step 1. rank suspicious combinations
3    π ← l-way suspicious combinations in F
4    if (π = empty) then return null //No ℓ-way inducing combination is found
5    let Θ be the set of suspicious components that appear in π
6    for each component o ∈ Θ {
7        compute ρ(o) based on formula 1
8    }
9    for each combination τ ∈ π {
10       compute ρc(τ) and ρe(τ) based on formulas 2 and 3, respectively
11   }
12   produce a ranking of ℓ-way combinations in π based on ρc and ρe
13   // Step 2. generate new tests
14   let T be the set containing a user-specified number of top-ranked combinations
15   for every combination τ ∈ T {
16     generate a set F′of a user-specified number of new tests that contain τ
17     if (|F′| == 0 || (∀f ∈ F′, r(f) = fail) ) {
18       c ← τ    // an ℓ-way inducing combination is found
19       return c
20     }
21     else {
22       F ← F ⋃ F′
23     }
24   }
25 }
```

Fig 2. The Identify Algorithm

suspicious combinations in the next iteration (line 22).

The two steps, rank generation and test generation, are performed iteratively until one of the following two stopping conditions is satisfied:

(1) The set $\pi$ of $\ell$-way suspicious combinations becomes empty (line 4); or

(2) An $\ell$-way inducing combination is found (line 18). An $\ell$-way suspicious combination $\tau$ is considered to be an inducing combination if no new test containing $\tau$ can be generated, or all newly generated tests containing $\tau$ fail (line 17). In the former case, it is very likely that all tests containing $\tau$ have been executed, and all of them must have failed (otherwise, $\tau$ is not suspicious). Thus, $\tau$ is the inducing combination. In the latter case, $\tau$ is likely to be inducing due to the way the new tests are generated as explained in Section 3.1.4. Later, we will discuss how BEN works when a non-inducing combination is reported as an inducing combination.

In the following subsections, we will explain the two major steps, rank generation and test generation.

### 3.1.3. Rank Generation

In this step, we first identify the set $\pi$ of all $\ell$-way suspicious combinations in $F$. Initially, $\pi$ contains all the $\ell$-way combinations covered by $F$. We then check each $\ell$-way combination $\tau$ in $\pi$. If $\tau$ appears in at least one passing test, $\tau$ is removed from $\pi$, since it is not suspicious anymore. In the subsequent iterations, we do not re-compute $\pi$ from the scratch. Instead, we only remove from $\pi$ all the combinations contained by newly added tests that passed.

If there is no $\ell$-way suspicious combination, there is no $\ell$-way inducing combination. In this case, the *identify* algorithm returns null. The main framework, as shown in Fig 1, then increases the size of inducing combination by one, and calls the *identify* algorithm again.

In the first iteration, where $F = F_0$ and $\ell = t$, all the t-way combinations are covered by F, as $F_0$ is a t-way test set. But, when $\ell > t$, F does not contain all the $\ell$-way combinations. Therefore, our approach focuses on $\ell$-way combinations that appear in F.

We next discuss how to rank the suspicious combinations in $\pi$. First, we introduce three important notions of suspiciousness, including *component suspiciousness*, *combination suspiciousness*, and *environment suspiciousness*.

*Component suspiciousness* ($\rho$): This notion is defined such that the higher $\rho$ a component $o$ has, the more likely $o$ contributes to a failure, and the more likely $o$ appears in an inducing combination. Let F be the test set that is analyzed in the current iteration. In our approach, $\rho$ is computed by the following formula:

$$\rho(o) = \frac{1}{3}\big(u(o) + v(o) + w(o)\big) \qquad (1)$$

Where

$$u(o) = \frac{|\{f \in F | r(f) = \text{fail} \wedge o \in f\}|}{|\{f \in F | r(f) = \text{fail}\}|}$$

$$v(o) = \frac{|\{f \in F | r(f) = \text{fail} \wedge o \in f\}|}{|\{f \in F | o \in f\}|}$$

$$w(o) = \frac{|\{\tau | o \in \tau \wedge \tau \in \pi\}|}{|\pi|}$$

The first factor, $u(o)$, shows the ratio of the number of failing test cases in which component $o$ appears over the total number of failing test cases. The second factor, $v(o)$, shows the ratio of the number of failing test cases in which component $o$ appears over the total number of test cases in which component $o$ appears. The third factor shows the ratio of the number of suspicious combinations in which component $o$ appears over the total number of suspicious combinations. The three factors are averaged to produce a value between 0 and 1.

The motivation behind the first two factors is that the more frequently a component appears in failing test cases, this component is more likely to contribute to a failure.

There is an important difference between the first two factors. Since the greater the domain size is, the less frequently each individual value of this parameter appears in a test set and consequently in failing test cases, the first factor, $u(o)$, has a bias towards smaller domain size parameters. The second factor, $v(o)$, is used to reduce this bias.

The motivation for the third factor is that components of inducing combinations tend to appear more frequently in suspicious combinations. For example, assume that combination $c = (a \leftarrow 0, b \leftarrow 0)$ is inducing. Let $f = (a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 0)$ be a test case. Test case f fails as it contains c. Let $f' = (a \leftarrow 1, b \leftarrow 1, c \leftarrow 0, d \leftarrow 0)$ be another test case, which passes since it does not contain c. The set of suspicious combinations derived from these two test cases is

$$\pi = \{(a \leftarrow 0, b \leftarrow 0), (a \leftarrow 0, c \leftarrow 0), (a \leftarrow 0, d \leftarrow 0), (b \leftarrow 0, c \leftarrow 0), (b \leftarrow 0, d \leftarrow 0)\}$$

In this set, the frequencies of $a \leftarrow 0$ and $b \leftarrow 0$ are greater than others. The reason is that $(c \leftarrow 0, d \leftarrow 0)$ appears in $f'$, which is a passing test case.

*Combination suspiciousness* ($\rho_c$): Combination suspiciousness of a combination $\tau$ is defined to be the average component suspiciousness of the components that appear in $\tau$.

Formally combination suspiciousness of $\tau$, $\rho_c(\tau)$ is computed by

$$\rho_c(\tau) = \frac{1}{|\tau|}\sum_{\forall\, o \in \tau} \rho(o) \qquad (2)$$

*Environment suspiciousness* ($\rho_e$): The environment of a combination $\tau$ in a test f includes all the components that appear in f but do not appear in $\tau$. The environment suspiciousness of a combination $\tau$ in a test f is the average component suspiciousness of the components in the environment of $\tau$. If there is more than one (failing) test containing $\tau$ in a test set, the environment suspiciousness of $\tau$ in this test set is the minimum environment suspiciousness of $\tau$ in all the tests containing $\tau$. Formally, environment suspiciousness $\rho_e$ is computed by

$$\rho_e(\tau) = \min_{\substack{f\in F \wedge \tau\subseteq f \\ \wedge\, r(f)=fail}} \left( \frac{1}{|f| - |\tau|} \sum_{o\in f \wedge o\notin\tau} \rho(o) \right) \qquad (3)$$

Now we discuss how to actually rank the suspicious combinations based on $\rho_c$ and $\rho_e$. Intuitively, the higher the value of $\rho_c$, the lower the value of $\rho_e$, the higher a combination is ranked.

To produce the final ranking, we first produce two rankings $R_c$ and $R_e$ of suspicious combinations, where $R_c$ is in the non-ascending order of $\rho_c$ and $R_e$ is in the non-descending order of $\rho_e$. The final ranking R is produced by combining $R_c$ and $R_e$ as follows. Let $\tau$ and $\tau'$ be two suspicious combinations. Assume that $\tau$ has ranks $r_c$ and $r_e$ in $R_c$ and $R_e$, respectively, and $\tau'$ has ranks $r'_c$ and $r'_e$ in $R_c$ and $R_e$, respectively. In the final ranking R, $\tau$ is ranked before $\tau'$ if and only if $r_c + r_e < r'_c + r'_e$.

### 3.1.4. Test Generation

This step is responsible for generating new test cases for a user-specified number of top-ranked suspicious combinations. Let $\tau$ be a top-ranked suspicious combination. A new test f is generated for $\tau$ such that f contains $\tau$ and the environment suspiciousness for $\tau$ is minimized in f. When such a test passes, this combination is removed from the suspicious set. When such a test fails, the failure is more likely due to this combination since its environment suspiciousness is minimized. Therefore, the suspicious combination should be marked as an inducing combination. To increase the confidence, a user-specified number of tests can be generated for a top-ranked suspicious combination.

One approach to generating a given number n of new tests with minimum $\rho_e$ for a suspicious combination is to generate all possible tests containing this combination, remove tests which already exist in F, and then select n tests that have the lowest $\rho_e$. This algorithm is very expensive. We next describe a more efficient, heuristic algorithm.

First, we generate a base test f as follows. For each parameter involved in $\tau$, we give the same value in f as in $\tau$. Doing so makes sure that f contains $\tau$. For each parameter in the environment of $\tau$, i.e., each parameter that is not involved in $\tau$, we choose a value (or component) whose suspiciousness $\rho$ is the minimum. If there is more than one value with minimum $\rho$, one of them is selected randomly.

Next, we check whether the base test f is valid and new, i.e., making sure that f satisfies all constraints if there is any, and has not been executed before. If so, f is returned as the new test that contains $\tau$ and has minimum $\rho_e$. If not, we pick one parameter randomly and change its value to a value with the next minimum $\rho$. Again, this test is checked to see whether it is a valid and new test. These steps are repeated until a new, valid test is found, or the number of attempts for finding a new test reaches a predefined number. The process is repeated until a desired number of new tests are generated.

If BEN does not find any new, valid test, the combination is marked as an inducing combination, because it is likely that all the test cases containing this combination have been executed (and all of them must have failed).

The newly generated tests, i.e., those in set F′, are executed. If all the tests fail, the suspicious combination, $\tau$, is marked as an inducing combination (line 18 - Fig 2). If not, F′ is added to the test set (line 22 - Fig 2) to refine the suspicious combinations set in the next iteration. By adding F′ to the test set the suspicious combination $\tau$ and all other suspicious combinations appear in passing tests of F′ are removed from the suspicious combinations set. Therefore, the number of suspicious combinations could be reduced by the new tests added into the test set.

### 3.1.5. Discussion

To successfully identify an inducing combination, BEN must first identify the combination to be a suspicious combination. Assume that c is an inducing combination. Let t be the strength of the initial test set. We consider the following three cases.

Case (1): c is a t-way combination. As the initial test set is a t-way test set, there is at least one test that contains c, and all test cases containing c must fail, since c is inducing. Therefore, c is identified to be a suspicious combination.

Case (2): The size of c is less than t. All t-way combinations containing c are inducing combinations, and are identified to be suspicious combinations.

Case (3): The size of c is more than t. The initial t-way test set is not guaranteed to cover every combination whose size is larger than t. If c appears in the initial t-way test set or the newly generated tests, thus causing a test containing it to fail, it is identified to be a suspicious combination when $\ell$ is equal to the size of c.

Let c be an inducing combination that has been identified as a suspicious combination. If it is in the top-ranked set, i.e., the set of a user-specified number of top-ranked combinations, all the tests generated for c fail since they contain c. Therefore, c is identified to be an inducing combination.

Now consider the case that c is not in the top-ranked set. Without loss of generality, assume that every combination c′ in the top-ranked set is not inducing. If any new test generated for c′ passes, c′ is no longer suspicious and is thus removed from $\pi$. This will cause c to move up in the ranking. With a sufficient number of iterations, c will be moved into the top-ranked set and will be identified to be an inducing combination.

If all the tests generated for c′ fail, c′ will be reported as an inducing combination. As discussed earlier, a new test for c′ is generated such that if it fails, it is likely due to c′. Thus, if all the tests generated for c′ fail, c′ is likely to have a high inducing probability even if it is not truly inducing.

BEN provides the user with several options to control the cost and effectiveness of the process. First, BEN allows the user to specify the number of new tests generated for each top-ranked suspicious combination. The more tests generated, the more effort it takes to execute them, but the more confidence we have about the identified inducing combinations.

Second, BEN allows the user to specify the size of the top-ranked set for which new tests will be generated. The bigger the top-ranked set, the more effort to generate and execute the new tests, but the faster an inducing combination may be identified. This is because if an inducing combination c is included in the top-ranked set, c

is identified to be an inducing combination in the first iteration. Otherwise, it may take multiple iterations for c to move up into the top-ranked set.

Finally, BEN allows the user to stop the first phase (and move to the second phase) in the following three ways if there is limited resource:

1- The user could define the maximum number of iterations for the *identify* algorithm. That is, if none of the two stopping conditions is satisfied after a specified number of iterations, the *identify* algorithm stops and returns null. Returning null shows that there is no inducing combination of the current size; therefore, the main framework increments the size in the next iteration.
2- The user could decide to stop at the end of each iteration of the framework. In this case, the top ranked suspicious combination would be reported as an inducing combination.
3- The user could define the maximum size of inducing combination. If the maximum size is reached but BEN still does not find any inducing combination, the top ranked suspicious combination in the last iteration is reported as an inducing combination. Recall that in the worst case, the size of inducing combination is equal to the number of parameters.

### 3.2. Phase 2: Faulty Statement Localization

Fig 3 shows the algorithm used by BEN to localize faulty statements. It consists of two major steps: (1) Test Generation: In this step, we generate a small group of tests. The group contains one failing test, which is referred to as the core member, and at most $\ell$ passing tests, where $\ell$ is the size of the inducing combination. The passing tests are referred to as the derived members. Each derived member is expected to produce a similar execution trace as the core member. (2) Rank Generation: In this step, we compare the spectrum of the core member to the spectrum of each derived member, and then produce a ranking of statements in terms of their likelihood of being faulty. More details of these two steps are explained in the following sections.

#### 3.2.1. Test Generation

In this step, as shown in Fig 3 (lines 3-9), a group of tests, M, which includes the core member f and at most $\ell$ derived members, are generated. Let c be the $\ell$-way inducing combination identified in Phase 1. The core member f is created such that it contains c and the environment suspiciousness of c in f is minimized (line 4). To generate such a test, the same algorithm used for test generation in Phase 1 is applied: For each parameter p involved in c, f has the same value for p as c, i.e. $c \subset f$, and for each parameter p that does not appear in c, f takes a value that has the minimum suspiciousness value among all the values of p. As discussed later, the reason why we want to minimize the environment suspiciousness of c is to maximize the likelihood of a derived member to be a passing test. If such a test does not satisfy system constraints, we randomly pick one parameter that does not appear in c and change its value to a parameter value that has the next minimum suspiciousness value. We repeat these steps until a valid test is found, or the number of attempts for finding a test reaches a predefined number. In the latter case, a test that contains c from the initial test set is picked as the core member.

The core member f is likely to fail, since it contains the inducing combination c identified in the first phase. Next, for each component $o \in c$, a set of derived member candidates, $M_o$, is generated. A derived member candidate $m_i \in M_o$ is generated such that it has the same values as f

**Algorithm localize**

1 *// Step 1. Generate core and derived members*
2 let c be the inducing combination identified in Phase 1
3 let M be an empty set
4 generate core member $f \in \Gamma$ such that $c \subset f$ and for all $o \in f$ and $o \notin c$, $\rho(o) = \min_{v_i \in d}\{\rho(p \leftarrow v_i)\}$
5 for (each component $o \in c$) {
6 generate the derived member candidate set $M_o$ for component o based on $\Theta$ and $\Omega$
7 select derived member $m_o \in M_o$ where $r(m_o) = \text{pass}$ and $|\gamma(f) - \gamma(m_o)| > 0$ and $|\gamma(f) - \gamma(m_o)| = \min_{m \in M_o}\{|\gamma(f) - \gamma(m)|\}$
8 $M = M \cup \{m_o\}$
9 }
10 *// Step 2. Rank statements*
11 for each statement $s \in S$ {
12 for all derived members in $m \in M$)
13 compute $\rho(s, m)$ with respect of core member f, based on formula (5)
14 $\rho(s) = \sum_{m \in M} \rho(s, m) / |M|$
15 }
16 Let R be the ranking of statement in the non-increasing order of $\rho(s)$
17 return R

Fig 3. The Localize Algorithm

for all parameters except for one component $o \in c$. The component o is replaced with another component, $o'$, of the same parameter with the minimum suspiciousness value. Note that a parameter may have multiple least suspicious components, i.e., multiple components with the minimum suspiciousness value. So, all the tests in $M_o$ are different from the core member and from each other in one component, o. Moreover, invalid tests are discarded from the set.

Fig 4 shows how the derived member candidate set, or simply candidate set $M_{o_1}$ is generated from the core member f. (In the remainder of this paper, we will refer to a derived member candidate set as a candidate set if there is no ambiguity.) The core member f contains k components, $o_1, o_2 \dots, o_k$, where k is the number of parameters. Without loss of generality, assume that the first $\ell$ components in f, i.e., $o_1, o_2 \dots, o_l$, are in the inducing combination c. As shown in Fig 4, each test in candidate set $M_{o_1}$ is different from the core member f in component $o_1 \in c$. The $o_1$ component is replaced with $o_1^j = p_1 \leftarrow v_j$ where $o_1^j$ is a least suspicious component of $p_1$. For each least suspicious component $p_1$, one derived candidate test is generated. Formally:

$$\rho(o_1^1 = p_1 \leftarrow v_1) = \rho(o_1^2 = p_1 \leftarrow v_2) \dots = \min_{\forall j \in d_1} \rho(p_1 \leftarrow v_j)$$

The number of tests in $M_{o_1}$ depends on the number of least suspicious components of parameter $p_1$ and constraints, as all tests in $M_{o_1}$ must be valid, i.e., they must satisfy all the system constraints. Candidate tests are likely to pass. First, the replacement effectively removes inducing combination c from tests. Second, the use of a least suspicious component for the replacement and having the suspiciousness environment minimized reduce the chance of introducing another inducing combination to the test.

Next, a derived member $m_o$ is selected from each candidate set $M_o$ (line 7). There are two criteria for derived member $m_o$. First, it must pass. Second, it has the minimum positive spectrum difference with the core member f among all the passing tests in $M_o$. Formally, $|\gamma(f) - \gamma(m_o)| = \min_{\substack{m \in M_o \wedge \\ r(m)=pass}} \{|\gamma(f) - \gamma(m)|\}$ and $|\gamma(f) - \gamma(m_o)| > 0$.

If there is more than one test that satisfies the two criteria, one of them is selected randomly. All the derived members are stored in a set called M (line 8). Fig 5 shows the core member f and the set M of derived members.

The execution trace of a derived member $m_i \in M$ is likely to be very similar to the execution trace of the core member, because these two tests only differ in one value, and they have the minimum spectrum differences among other similar tests. Since all the derived members $m_i$ pass whereas the core member f fail, the faulty statement is very likely to be one of the statements that appear in the execution trace of f but do not appear in the execution trace of $m_1, m_2 \dots$, and $m_l$.

### 3.2.2. Rank Generation

In this step, BEN computes the suspiciousness of each statement and then ranks them in terms of their likelihood to be faulty by analyzing the spectrums of the core member and derived members. The suspiciousness of statement s is denoted by $\rho(s)$ and computed by analyzing the spectrums of the core member and derived members. The suspiciousness of statement s is the average suspiciousness of s with respect to every derived member. Formally:

$$\rho(s) = \sum_{m_i \in M} \rho(s, m_i)/(|M|) \quad (4)$$

where $\rho(s, m_i)$ is the suspiciousness of s with respect to a derived member $m_i$ and is computed by the following formula:

$$\rho(s, m_i) = \begin{cases} 1 & \text{if } \gamma(s, f) = \text{true and } \gamma(s, m_i) = \text{false} \\ 0.5 & \text{if } \gamma(s, f) = \gamma(s, m_i) = \text{true} \\ 0 & \text{if } \gamma(s, f) = \text{false} \end{cases} \quad (5)$$

The idea behind formula (5) is the following. Statements that are only executed by the core member f are most suspicious and are given 1 as their suspiciousness value. Statements that are executed by both the core member and a derived member are less suspicious, and are given 0.5 as their suspiciousness value. Note that the execution of a faulty statement by a test does not necessarily make the test fail. For example, if there exists a fault in a conditional expression, this fault can be executed by all the tests but only cause some to fail. Finally, statements that are not executed by f are not suspicious.

For example, if there are two derived members in M, $m_1$ and $m_2$, and the core member is f. Assume that a statement s is executed by f and $m_2$, but not by $m_1$ The suspiciousnes $\rho(s)$ of s would be 0.75. This is because $\rho(s, m_1) = 1$ and $\rho(s, m_2) = 0.5$, and the average of $\rho(s, m_1)$ and $\rho(s, m_2)$

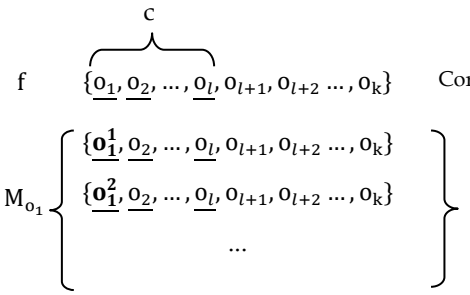


Fig 4. Generation of the candidate set $\mathbf{M_{o_1}}$

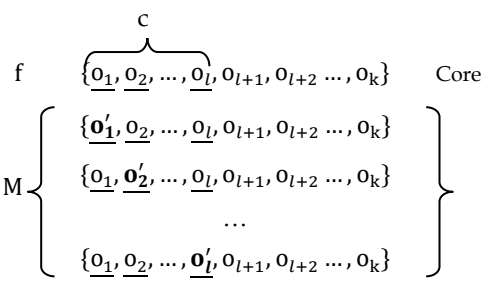


Fig 5. The core and derived members

would be 0.75.

The higher the suspiciousness value of a statement, the more likely this statement is faulty. We rank statements by a non-ascending order of their suspiciousness value. To locate the faulty statement, the statements in the top rank are examined first, and then the statements in the next rank, until the faulty statement is found.

### 3.2.3. Discussion

The effectiveness of our approach in this phase depends to some extent on the quality of the inducing combination c identified in the first phase. If combination c is truly inducing, the core member generated by our approach, i.e., the one that contains this combination and minimizes its environment suspiciousness, must fail. However, if c is not truly inducing, but with a high inducing probability, the core member still has a high probably to fail. The experimental results in Section 5.2.1.1 and 5.2.2.1 show that Phase 1 of our approach can identify truly inducing combinations or combinations that have a high inducing probability.

If the core member generated in the second phase does not fail, we pick a test from the initial t-way test set that contains c as the core member. Since c is identified as an inducing combination, there must exist at least one failing test that contains c in the initial test set. (Otherwise, c would not even be a suspicious combination.) In this case, the environment suspiciousness of c in this test may not be minimized. This may reduce the probability for the derived members to pass.

If BEN could not find any passing test in a candidate set $M_o$ for a component o (in the inducing combination), it ignores the candidate set and thus no derived member is generated for component o. We note that the existence of constraints could reduce the number of possible tests in the candidate set and thus increase the chance of being unable to find a passing test. In case that no derived member is generated for all the components in the inducing combination, BEN picks a passing test from the test set such that the number of components that differ between the passing test and the core member is minimized. In this case, the difference between the core member and this derived member may not be minimal, which might affect the efficiency of our approach. We believe the chance for this case, i.e., all the tests in all the candidate sets for all the components fail, to occur is small, which is consistent with our experiments in which it occurred in 16 out of 171 single-fault versions of our subject programs.

### 3.2.4. Complexity Analysis

In our analysis, we do not consider the complexity of constraint solving and the cost of test execution. Our approach uses a third-party solver for constraint solving. The cost of test execution depends on the subject program.

Let k be the number of parameters, t the strength of the initial test set and d the largest domain size of the parameters. Let N be the number of tests in the current iteration, which includes the tests in the initial test set and the tests generated in the previous iterations. Note that the number of tests generated at each iteration depends on two user-specified numbers, i.e., the size of the top-ranked set consisting of suspicious combinations for which tests are to be generated, and the number of tests to be generated for each suspicious combination in the top-ranked set. Assume that the inducing combination is of size $l$, which is greater than or equal to t. The maximum number of $\ell$-way combinations contained in the test set is $\eta = \binom{k}{l}N$.

To determine whether a combination is suspicious, the *identify* algorithm needs to check if the combination appears in any passing test, which takes $O(N \times l)$. Therefore, building the suspicious combination set takes $\eta \times O(N \times l)$. Next, the *identify* algorithm computes the suspiciousness values for all the components, which includes computing the frequency of each component in the suspicious combination set, test set and failing tests. Computing the frequency in the suspicious combination set dominates the other two, which takes $O(\eta)$ for each component. The maximum number of components is $k \times d$. Thus, computing suspiciousness values for all the components takes $k \times d \times O(\eta)$.

After having suspiciousness values of all the components, computing combination suspiciousness of each combination ($\rho_c$) takes $l$, and thus $l \times O(\eta)$ for all the combinations. To compute $\rho_e$ of a combination, BEN first searches in the test set to find all the failing tests that contain this combination, which takes $l \times O(N)$. Next, for each of these failing tests, it computes the average suspiciousness value of $k - l$ components in the environment. Therefore, it takes in total $l \times (k - l) \times O(\eta) \times O(N)$. Finally, BEN finds the minimum environment suspiciousness among all these failing tests, which takes $O(N)$. Therefore, the complexity of computing $\rho_e$ for all the combination is $l \times (k - l) \times O(\eta) \times O(N)$.

The *identify* algorithm sorts the set of suspicious combinations three times, once for each ranking $R_c$, $R_e$, and $R$, taking $O(\eta \times \log(\eta))$. This dominates the complexity of the rank generation step, if the number of tests N is far less than the number of combinations, $\eta$.

The test generation step needs to select $(k - l)$ values with minimum $\rho$ first, which takes $(k - l) \times O(d)$. Then it needs to check whether it is new, which is $O(N)$. Since k, $\ell$ and d are smaller than $\eta$, $O(\eta \times \log(\eta))$ dominates the complexity of the rank generation and test generation steps. Therefore the complexity of the *identify* algorithm is $O(\eta \times \log(\eta))$. In the worst case, the *identify* algorithm is called $(k - t)$ times. Thus, the complexity of this phase is $(k - t) \times O(\eta \times \log(\eta))$.

In Phase 2, in order to generate the core member, we need to select values with minimum suspiciousness for $(k - l)$ components, which takes $(k - l) \times O(d)$. There are $l$ candidate sets, and for each it takes $O(d)$ to find components with minimum $\rho$. Therefore, generating all candidate sets takes $l \times O(d)$.

Each candidate set at most contains $d - 1$ derived members. Selecting a test with minimum difference in the spectrum with the core member takes $[l \times (d - 1)] \times |S|$, where $|S|$ is the number of statements of the program. The complexity of selecting a test, $[l \times (d - 1)] \times |S|$, dominates the complexity of this step.

In the rank generation step, the complexity of assigning a suspiciousness value to each statement with respect to the $l$ derived members is $O(l)$. So for all the statements S of the program, it takes $|S| \times O(l)$. Then all the statements need to be sorted to rank the statements, which is $O(|S| \times \log(|S|))$. Since $l$ is typically much smaller than the program size $|S|$, this sorting operation dominates the complexity of this part. The complexity of the rank generation step, $O(|S| \times \log(|S|))$, dominates the complexity of this phase.

Depending on the programs size, $|S|$ and the number of suspicious combinations, $\eta$, the complexity of Phase 1 or Phase 2 may dominate the complexity of BEN.

## 4. EXAMPLE

In this section, we illustrate our approach using an example program shown in Fig 6. Method *foo* has a fault in line 9. The correct statement should be $r \mathrel{+}= (b - d)/(a + 2)$, but operator "+" is missing. The input parameter model consists of $P = \{a, b, c, d\}$, and $d_a = \{0,1\}$, $d_b = \{0,1\}$, $d_c = \{0,1,2\}$, and $d_d = \{0,1,2,3\}$. The faulty statement is reached when a is 0 and c is 0 or d is 3. So there are two inducing combinations $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

Assume that the program is tested by a two-way test set. The test result is shown in Table 1, where three out of twelve tests fail. Test cases #1 and #7 fail because they contain combination $(a \leftarrow 0, c \leftarrow 0)$. Test case #10 fails because it contains $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

### 4.1. Phase 1: Inducing Combination Identification

Table 1 shows a t-way test set with test execution statuses for the example program. In the first iteration, the *identify* algorithm identifies nine suspicious combinations (Fig 2, line 3) which are listed in the first column of Table 2. Then the algorithm computes the suspiciousness values of all the (seven) components that appear in one or more of these suspicious combinations.

For example, component $c \leftarrow 0$ appears in all of the three failing test cases, so $u(c \leftarrow 0) = 1$. Also, it appears in a total of four tests, three of which are failing tests, so $v(c \leftarrow 0) = 3/4$; five out of nine members of suspicious combinations set contain $c \leftarrow 0$, so $w(c \leftarrow 0) = 5/9$. The computations for all the seven components are as follows:

$$\rho(c \leftarrow 0) = \frac{1}{3} \times \left(1 + \frac{3}{4} + \frac{5}{9}\right) = 0.7685$$

$$\rho(d \leftarrow 0) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 2) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 3) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{3} + \frac{3}{9}\right) = 0.3333$$

$$\rho(b \leftarrow 0) = \frac{1}{3} \times \left(\frac{1}{3} + \frac{1}{7} + \frac{1}{9}\right) = 0.1958$$

$$\rho(b \leftarrow 1) = \frac{1}{3} \times \left(\frac{2}{3} + \frac{2}{5} + \frac{3}{9}\right) = 0.4667$$

$$\rho(a \leftarrow 0) = \frac{1}{3} \times \left(1 + \frac{3}{6} + \frac{2}{9}\right) = 0.5741$$

Table 3 illustrates the suspiciousness values of all the components. The suspiciousness values for the components that do not appear in any suspicious combination are zero.

According to formula (2), $\rho_c$ for a suspicious combination $\tau$ is the average component suspiciousness of components that $\tau$ contains. For example, in combination $(a \leftarrow 0, c \leftarrow 0)$, $\rho_c$ is $(0.5741 + 0.7685)/2 = 0.6713$. After computing $\rho_c$ for all suspicious combinations, we rank them based on the non-ascending order of $\rho_c$. The values of $\rho_c$ and $R_c$ for each suspicious combination are shown in the second and third columns of Table 2.

Next, we compute $\rho_e$ for each suspicious combination using formula (3). For example, there are three test cases, test #1, test #7, and test #10, that contain $(a \leftarrow 0, c \leftarrow 0)$. Therefore,

$$\rho_e(a \leftarrow 0, c \leftarrow 0) = \min\left(\left(\frac{\rho(b \leftarrow 0) + \rho(d \leftarrow 0)}{2}\right) = 0.2460,\ \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 2)}{2}\right) = 0.3815, \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 3)}{2}\right) = 0.4000\right) = 0.2460$$

```
1  public static int foo(int a,int b,int c,int d){
2    int r = 1;
3    b += a + c;
4    switch (a){
5      case 0 :
6        if (c<1 || d>2)
7          //r += (b-d)/(a+2);
8          //fault : + is missing;
9          r = (b-d)/(a+2);
10       else
11         r = b/(c+2);
12       break;
13     case 1 :
14       r = c*(a-d);
15       break;
16   }
17   return r;
18 }
```

Fig 6. An example faulty program

TABLE 1
TWO-WAY TEST SET AND STATUS

| Test # | a | b | c | d | Status |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | Fail |
| 2 | 1 | 1 | 1 | 0 | Pass |
| 3 | 0 | 1 | 2 | 0 | Pass |
| 4 | 1 | 0 | 0 | 1 | Pass |
| 5 | 0 | 0 | 1 | 1 | Pass |
| 6 | 1 | 1 | 2 | 1 | Pass |
| 7 | 0 | 1 | 0 | 2 | Fail |
| 8 | 1 | 0 | 1 | 2 | Pass |
| 9 | 0 | 0 | 2 | 2 | Pass |
| 10 | 0 | 1 | 0 | 3 | Fail |
| 11 | 1 | 0 | 1 | 3 | Pass |
| 12 | 1 | 0 | 2 | 3 | Pass |

TABLE 2
SUSPICIOUS COMBINATIONS AND THEIR CORRESPONDING VALUES

| Suspicious combination | $\rho_c$ | $R_c$ | $\rho_e$ | $R_e$ | $R_c + R_e$ | $R$ |
|---|---|---|---|---|---|---|
| $a \leftarrow 0, c \leftarrow 0$ | 0.6713 | 1 | 0.2460 | 1 | 2 | 1 |
| $b \leftarrow 1, c \leftarrow 0$ | 0.6176 | 2 | 0.4352 | 3 | 5 | 2 |
| $c \leftarrow 0, d \leftarrow 0$ | 0.5324 | 4 | 0.3849 | 2 | 6 | 3 |
| $c \leftarrow 0, d \leftarrow 3$ | 0.5509 | 3 | 0.5204 | 4 | 7 | 4 |
| $c \leftarrow 0, d \leftarrow 2$ | 0.5324 | 4 | 0.5204 | 4 | 8 | 5 |
| $a \leftarrow 0, d \leftarrow 3$ | 0.4537 | 5 | 0.6176 | 5 | 10 | 6 |
| $b \leftarrow 1, d \leftarrow 3$ | 0.4000 | 6 | 0.6713 | 6 | 12 | 7 |
| $b \leftarrow 1, d \leftarrow 2$ | 0.3815 | 7 | 0.6713 | 6 | 13 | 8 |
| $b \leftarrow 1, d \leftarrow 0$ | 0.2460 | 8 | 0.6713 | 6 | 14 | 9 |

TABLE 3
COMPONENT SUSPICIOUSNESS

| Parameter | Value | $\rho_c$ | Parameter | Value | $\rho_c$ |
|---|---|---|---|---|---|
| a | 0 | 0.5741 | b | 0 | 0.1958 |
| | 1 | 0 | | 1 | 0.4667 |
| c | 0 | 0.7685 | d | 0 | 0.2963 |
| | 1 | 0 | | 1 | 0 |
| | 2 | 0 | | 2 | 0.2963 |
| | | | | 3 | 0.3333 |

Next we rank suspicious combinations by a non-descending order of $\rho_e$, as shown in column $R_e$ of Table 2.

Finally, the two rankings in columns $R_c$ and $R_e$ are combined to produce a final ranking of the suspicious components (column R). In this final ranking, inducing combination $(a \leftarrow 0, c \leftarrow 0)$ is ranked on the top, and the other inducing combination $(a \leftarrow 0, d \leftarrow 3)$ is ranked 6th.

Then, a new test is generated for the top ranked suspicious combination $(a \leftarrow 0, c \leftarrow 0)$. We assign values to parameters in its environment, i.e., b and d, such that the suspiciousness of each value is minimum. For b, 0 is selected, as $\min(\rho(b \leftarrow 0) = 0.1958, \rho(b \leftarrow 1) = 0.4667) = 0.1958$. For d, 1 is selected as $\min(\rho(d \leftarrow 0) = 0.2963, \rho(d \leftarrow 1) = 0, \rho(d \leftarrow 2) = 0.2963, \rho(d \leftarrow 3) = 0.3333) = 0$. So a new test $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ is generated.

The newly generated test, $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$, fails. For simplicity of presentation, assume that only one test is generated for this combination. (If more tests are generated, all of them would fail too in this example.) Therefore, suspicious combination $(a \leftarrow 0, c \leftarrow 0)$ is marked as an inducing combination and returned by the *identify* algorithm. The main framework of the first phase stops at the end of the first iteration and reports $(a \leftarrow 0, c \leftarrow 0)$ as the inducing combination.

### 4.2. Phase 2: Faulty Statement Localization

In the test generation step of the second phase, the core member $f = (a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ is generated. It contains the inducing combination $(a \leftarrow 0, c \leftarrow 0)$, and two components $b \leftarrow 0$ and $d \leftarrow 1$ which have the minimum suspiciousness value (among components of the same parameter) as shown in Table 3. The core member fails.

$$\begin{array}{ll} f & \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{c \leftarrow 0}, d \leftarrow 1\right) \text{ Fail} \\ M_{a\leftarrow 0}\left\{ m_{a\leftarrow 0} = \left(\underline{\mathbf{a \leftarrow 1}}, b \leftarrow 0, \underline{c \leftarrow 0}, d \leftarrow 1\right) \right. & \left. \text{Pass} \right\} \end{array}$$

Fig 7. Candidate set of $\mathbf{M_{a\leftarrow 0}}$

$$\begin{array}{ll} f & \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{c \leftarrow 0}, d \leftarrow 1\right) \text{ Fail} \\ M_{c\leftarrow 0}\left\{ m^1_{c\leftarrow 0} = \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{\mathbf{c \leftarrow 1}}, d \leftarrow 1\right) \right. & \text{Pass} \\ \left. m^2_{c\leftarrow 0} = \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{\mathbf{c \leftarrow 2}}, d \leftarrow 1\right) \right\} & \text{Pass} \end{array}$$

Fig 8. Candidate set of $\mathbf{M_{c\leftarrow 0}}$

As shown in Fig 7 the candidate set $M_{a\leftarrow 0}$ of component $a \leftarrow 0$ contains only one test, $(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$, since $a \leftarrow 1$ is the only component with minimum suspiciousness. The test passes and therefore is selected as a derived member, $m_{a\leftarrow 0}$.

The second candidate set $M_{c\leftarrow 0}$, shown in Fig 8, has two tests, where component $c \leftarrow 0$ from the core member is replaced with $c \leftarrow 1$ and $c \leftarrow 2$, since $\min(\rho(c \leftarrow 0) = 0.7685, \rho(c \leftarrow 1) = 0, \rho(c \leftarrow 2) = 0) = 0$ and both components $c \leftarrow 1$ and $c \leftarrow 2$ have the minimum suspiciousness value, 0.

To select a derived member $m_{c\leftarrow 0}$ from candidate set $M_{c\leftarrow 0}$, both tests $m^1_{c\leftarrow 0}$ and $m^2_{c\leftarrow 0}$ are executed and their execution traces are recorded. A test is selected as a derived member if it passes and it has minimum spectrum difference with the core member.

Both tests $m^1_{c\leftarrow 0}$ and $m^2_{c\leftarrow 0}$ pass. The spectra of the core member, f, and two members of candidate set $M_{c\leftarrow 0}$ are shown in Table 4. The second column of Table 4 shows the program statements. The third column shows the spectrum of the core member f. The fourth column shows the program spectrum of $m^1_{c\leftarrow 0}$. The fifth column contains 1 if a statement is executed by the core member but not by $m^1_{c\leftarrow 0}$. Otherwise it contains 0. The sixth column shows the program spectrum of $m^2_{c\leftarrow 0}$. The last column is assigned to 1 iff the corresponding statement is executed by the core member and not by $m^2_{c\leftarrow 0}$. The fifth and seventh columns are used to compute the spectrum differences of the core and $m^1_{c\leftarrow 0}$ or $m^2_{c\leftarrow 0}$. The last row of Table 4 shows the spectrum difference of the core and each member of $M_{c\leftarrow 0}$, which are computed by the summation of fifth and last columns.

Since both tests $m^1_{c\leftarrow 0}$ and $m^2_{c\leftarrow 0}$ pass and have the same spectrum difference with the core member, test $m^1_{c\leftarrow 0}$ is selected randomly as the derived member $m_{c\leftarrow 0}$. Fig 9 shows the output of the test generation step, the core member, f, in the first row and the derived members set M, which contains two tests.

$$\begin{array}{ll} f & \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{c \leftarrow 0}, d \leftarrow 1\right) \text{ Fail} \\ M\left\{ m_{a\leftarrow 0} = \left(\underline{\mathbf{a \leftarrow 1}}, b \leftarrow 0, \underline{c \leftarrow 0}, d \leftarrow 1\right) \right. & \text{Pass} \\ \left. m_{c\leftarrow 0} = \left(\underline{a \leftarrow 0}, b \leftarrow 0, \underline{\mathbf{c \leftarrow 1}}, d \leftarrow 1\right) \right\} & \text{Pass} \end{array}$$

Fig 9. Core and derived members of the example program

TABLE 4
PROGRAM SPECTRA OF CORE AND CANDIDATE SET $\mathbf{M_{c\leftarrow 0}}$

| | Subject Program | $\gamma(s,f)$ | $\gamma(s,m^1_{c\leftarrow 0})$ | $\gamma(f)-\gamma(m^1_{c\leftarrow 0})$ | $\gamma(s,m^2_{c\leftarrow 0})$ | $\gamma(f)-\gamma(m^2_{c\leftarrow 0})$ |
|---|---|---|---|---|---|---|
| 1 | `public static int foo(int a,int b, int c,int d){` | True | True | 0 | True | 0 |
| 2 | `int r = 1;` | True | True | 0 | True | 0 |
| 3 | `b += a + c;` | True | True | 0 | True | 0 |
| 4 | `switch (a){` | True | True | 0 | True | 0 |
| 5 | `case 0 :` | True | True | 0 | True | 0 |
| 6 | `if (c<1 \|\| d>2)` | True | True | 0 | True | 0 |
| 7 | `//r += (b-d)/(a+2);` | - | - | 0 | - | 0 |
| 8 | `//fault:+is missing;` | - | - | 0 | - | 0 |
| 9 | `r = (b-d)/(a+2);` | True | False | 1 | False | 1 |
| 10 | `else` | False | True | 0 | True | 0 |
| 11 | `r = b/(c+2);` | False | True | 0 | True | 0 |
| 12 | `break;` | True | True | 0 | True | 0 |
| 13 | `case 1 :` | False | False | 0 | False | 0 |
| 14 | `r = c*(a-d);` | False | False | 0 | False | 0 |
| 15 | `break;` | False | False | 0 | False | 0 |
| 16 | `}` | True | True | 0 | True | 0 |
| 17 | `return r;` | True | True | 0 | True | 0 |
| 18 | `}` | True | True | 0 | True | 0 |
| | $\lvert\gamma(f)-\gamma(m_{c\leftarrow 0})\rvert$ | - | - | 1 | - | 1 |

TABLE 5
PROGRAM SPECTRA AND STATEMENTS SUSPICIOUSNESS VALUES

| | Subject Program | $\gamma(s,f)$ | $\gamma(s,m_{a\leftarrow 0})$ | $\gamma(s,m_{c\leftarrow 0})$ | $\rho(s,m_{a\leftarrow 0})$ | $\rho(s,m_{c\leftarrow 0})$ | $\rho(s)$ | Rank |
|---|---|---|---|---|---|---|---|---|
| 1 | `public static int foo(int a,int b, int c,int d){` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 2 | `int r = 1;` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 3 | `b += a + c;` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 4 | `switch (a){` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 5 | `case 0 :` | True | False | True | 1 | 0.5 | 0.75 | 2 |
| 6 | `if (c<1 \|\| d>2)` | True | False | True | 1 | 0.5 | 0.75 | 2 |
| 7 | `//r += (b-d)/(a+2);` | - | - | - | - | - | - | - |
| 8 | `//fault:+is missing;` | - | - | - | - | - | - | - |
| 9 | `r = (b-d)/(a+2);` | True | False | False | 1 | 1 | 1 | 1 |
| 10 | `else` | False | False | True | 0 | 0 | 0 | 4 |
| 11 | `r = b/(c+2);` | False | False | True | 0 | 0 | 0 | 4 |
| 12 | `break;` | True | False | True | 1 | 0.5 | 0.75 | 2 |
| 13 | `case 1 :` | False | True | False | 0 | 0 | 0 | 4 |
| 14 | `r = c*(a-d);` | False | True | False | 0 | 0 | 0 | 4 |
| 15 | `break;` | False | True | False | 0 | 0 | 0 | 4 |
| 16 | `}` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 17 | `return r;` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |
| 18 | `}` | True | True | True | 0.5 | 0.5 | 0.5 | 3 |

In the rank generation step, the spectrum of the core member is compared to that of each derived member m ∈ M and the statement suspiciousness with respect to m is computed. Table 5 shows the program spectra for the core member and two derived members in columns three to five. The suspiciousness values for each statement with

respect to derived tests $m_{a\leftarrow 0}$ and $m_{c\leftarrow 0}$ are shown in columns six and seven ($\rho(s, m_{a\leftarrow 0})$ and $\rho(s, m_{c\leftarrow 0})$ ) of Table 5, respectively. The last two columns of Table 5 show the statement suspiciousness and ranks. The faulty statement in line 9 is ranked to be the first.

Note that this example represents a best-case scenario of our approach. In the next section, we provide an experimental evaluation of our approach.

## 5. Experiment

We built a tool called BEN [17] that implements our approach. (BEN is a Chinese word that means "root cause".) BEN is available for public download [5]. For our experiment, we used the command line version of BEN.

The subject programs are selected from SIR [46], including seven small programs in the *Siemens suite* and four large real-world programs *flex*, *grep*, *gzip* and *sed*. Furthermore, we conducted an experimental comparison between our approach and two well-known spectrum based approaches, Tarantula [24] and Ochiai [31].

### 5.1. Experimental Design

#### 5.1.1. Subject Programs

The *Siemens Suite* has been used to evaluate several fault localization techniques [23][40][50]. The four real-world programs, *flex*, *grep, gzip* and *sed*, are significantly larger programs than the Siemens programs and are included to evaluate how our approach works on larger programs. These programs are also used in other studies such as [22][35][36][38][39]. The *Siemens suite* and the four real-world programs are among the most widely used subject programs for fault localization studies [50]. Note that the details of the subject programs as well as the faults in these programs can be found in the SIR [44].

**The Siemens suite -** The *Siemens suite* contains seven programs and each of these programs contains a number of faulty versions. The *Siemens suite* also provides an error-free version and a test set for each program. Table 6 represents characteristics of the Siemens programs. The second column shows the size of executable code computed by Gcov 4.1.2 [14], and the third column indicates the number of faulty versions provided for each program in SIR. Note that the number of lines of executable code is different from the number of lines of code reported in [46]. This is because the number of lines of executable code does not include commented lines, declaration lines, or code in header files.

TABLE 6
CHARACTERISTICS OF SIEMENS SUITE

| Program | # of lines of executable code | # of faulty versions |
|---|---|---|
| printtokens | 188 | 7 |
| printtokens2 | 201 | 10 |
| replace | 242 | 32 |
| schedule | 154 | 9 |
| schedule2 | 127 | 10 |
| tcas | 65 | 41 |
| totinfo | 123 | 23 |

Both of the two programs, *printtokens* and *printtokens*2, are used to tokenize the input file and determine the type of each token. A token could be one of the following types: *identifier*, *special*, *keyword*, *number*, *comment*, *character constant* or *string constant*.

The *replace* program has three inputs, *pattern*, *substitute* and *input text*. The program finds every match of *pattern* in the *input text* and replaces it with *substitute*. The *pattern* is a restricted form of regular expression. The *substitute* is a string that allows three meta-characters to be used. These include "@t", which matches a tab, @n, which matches the end of a line, and &, which represents the string that matches the *pattern*. For example, if the string that matches *pattern* is *ab* and *substitute* is *a&c*, all the occurrences of *ab* in the input file are replaced with *aabc*.

Two programs, *schedule* and *schedule*2, take the same input and produce the same output, but use different scheduling algorithms. The input includes: (1) three non-negative integers representing the number of processes in three different priority queues, *low*, *medium* and *high*; and (2) a list of commands that must be executed on queues. There are seven commands, *new job*, *upgrade_prio*, *block*, *unblock*, *quantum_expire*, *finish* and *flush*. The output of these two programs is a list of numbers indicating the order in which the processes exit (from the scheduling system).

The *tcas* program is an aircraft collision avoidance system. It takes as input twelve numbers that represent different flight parameters of two aircrafts and generates as output a resolution advisory, which can be *unresolved*, *upward* and *downward*.

The *totinfo* program takes as input a file containing one or more tables. The program uses the notions of chi-square and degree of freedom to calculate whether the distribution of the numbers in these tables is logarithm gamma distribution. The output is the total degree of freedom of rows and columns and chi-square.

**The Flex program -** The *flex* program is a fast lexical analyzer or scanner generator. The *flex* program reads the given input file (or files) and generates a C source file, called scanner. The input file includes pairs of regular expression and C code, called rules. There are several options to control the behavior of the *flex* program. For example, option "-d" is to enable debugging mode in the scanner.

There are five versions of the *flex* in the benchmark, and each has a number of seeded faults. All versions are written in C and have four header files and one C file. Table 7 shows the size of executable code computed by Gcov 4.1.2 and the number of faulty versions for each release of *flex*. The third column, i.e. the number of lines of executable code, shows the number for the error-free version. Note that all the faults in a given version of the *flex* program are different from the faults of the other versions, and reside in the code that has been modified from the previous version.

**THE GREP PROGRAM -** The *grep* program has two input parameters, *patterns* and *files*. It prints lines in each file that contain a match of any of the patterns. While the *grep* program can take multiple patterns and files, we only used a single pattern and file in this experiment. In addition, different options can be used to control the behavior of the *grep* program. For example, option "–w" causes the program to print only lines containing whole-word matches.

The *grep* program can take four different types of patterns: (1) basic-regexp: a basic regular expression; (2) extended-regexp: an extended regular expression; (3) fixed-strings: a list of fixed strings; (4) perl-regexp: a Perl regular expression. In this experiment, we only used extended-regexp.

There are five versions of *grep* in the benchmark, each of which has a number of seeded faults. All the versions are written in C consisting of ten header files and one C file. Table 8 shows the release number of each version, the size of executable code computed by Gcov 4.1.2 and the number of faulty versions for each version.

Note that all the faults in a given version are different from the faults of the other versions, and reside in the code that has been modified from the previous version. For example, for *grep*2, all the faults reside in the code modified from *grep* 2.2 to *grep* 2.3.

**THE GZIP PROGRAM -** The *gzip* program is used for file compression and decompression. The input of *gzip* includes 13 options and a list of files. For example, "-S" option is used to define the suffix of the result file, where the default is ".gz".

There are five versions of *gzip*, each of which has a number of seeded faults. All the versions are written in C, consisting of six header files and one C file. Table 9 shows the number of lines of executable code computed by Gcov 4.1.2 and the number of faulty versions for each error-free version, in the third and fourth columns, respectively. The release number for each program is shown in the second column of Table 9. The base version is *gzip* 1.0.7. The faults for different *gzip* versions are different from each other except for one case where the first fault of *gzip*5 is the same as the first fault of *gzip*2. In addition, all the faults reside in the code that has been modified from the previous version, except the fault mentioned above. For example, for *gzip*2, all the faults reside in the code modified from *gzip* 1.1.2 to *gzip* 1.2.2.

TABLE 7
CHARACTERISTICS OF FLEX PROGRAMS

| Program | Release number | # of lines of executable code | # of faulty versions |
|---|---|---|---|
| flex1 | 2.4.6 | 3393 | 19 |
| flex2 | 2.4.7 | 3934 | 20 |
| flex3 | 2.5.1 | 3939 | 17 |
| flex4 | 2.5.2 | 3965 | 16 |
| flex5 | 2.5.3 | 3967 | 9 |

TABLE 8
CHARACTERISTICS OF GREP PROGRAMS

| Program | Release number | # of lines of executable code | # of faulty versions |
|---|---|---|---|
| grep1 | 2.2 | 3078 | 18 |
| grep2 | 2.3 | 3224 | 8 |
| grep3 | 2.4 | 3294 | 18 |
| grep4 | 2.4.1 | 3313 | 12 |
| grep5 | 2.4.2 | 3314 | 1 |

TABLE 9
CHARACTERISTICS OF GZIP PROGRAMS

| Program | Release number | # of lines of executable code | # of faulty versions |
|---|---|---|---|
| gzip1 | 1.1.2 | 1705 | 16 |
| gzip2 | 1.2.2 | 2006 | 7 |
| gzip3 | 1.2.3 | 1866 | 10 |
| gzip4 | 1.2.4 | 1892 | 12 |
| gzip5 | 1.3 | 1993 | 14 |

**THE SED PROGRAM -** The *sed* program reads and performs basic transformations on an input stream. The *sed* program takes as input a *sed* script and one or more text files. The script file includes some *sed* commands, such as *append*, *replace*, *delete* and *insert*. In addition, a number of options are available to control the behavior of the *sed* program. For example, the "-r" option is used to have extended regular expressions in the script rather than basic regular expressions.

There are seven versions of the *sed* program, and each has a number of seeded and/or real faults. All the versions are written in C. Table 10 shows the number of header files, the number of C files, the lines of executable code of computed by Gcov 4.1.2, and the number of faulty versions. Note that the number of lines of executable code in Table 10 is the total number of lines of executable code

TABLE 10
CHARACTERISTICS OF SED PROGRAMS

| Program | Release number | # of header files | # of C files | # of lines of executable code | # of faulty versions |
|---|---|---|---|---|---|
| sed1 | 1.08 | 2 | 1 | 1923 | 3 |
| sed2 | 2.04 | 2 | 1 | 3391 | 5 |
| sed3 | 3.01 | 7 | 1 | 2171 | 6 |
| sed4 | 3.02 | 7 | 1 | 2172 | 4 |
| sed5 | 4.0.6 | 10 | 5 | 4540 | 4 |
| sed6 | 4.0.7 | 8 | 5 | 4544 | 6 |
| sed7 | 4.1.5 | 8 | 5 | 4919 | 4 |

of all the C files in each version.

### 5.1.2. Initial Test Set

The input parameter model of each program is shown in Table 11. In [15], we explained how we modeled the input parameters of the Siemens programs to apply combinatorial testing. The model of the *grep* program is explained in [18]. The detailed models for the Siemens programs and the *grep* program are available in [12]. The models for three programs, *flex*, *gzip* and *sed*, are taken from [39]. We note that the models presented in [39] are also used in other studies, e.g., [22].

The model column of Table 11 shows the number of parameters and their domain sizes. We represent it by $(d_1^{k_1} \times d_2^{k_2} \times ...)$, where $d_i^{k_i}$ indicates that there are $k_i$ number of parameters with domain size as $d_i$. Note that $k_1 + k_2 + \cdots = k$, where k is the total number of parameters. For example, *totinfo* has six parameters, among which three parameters have a domain size of 3, two parameters have a domain size of 5, and one parameter has a domain size of 6.

The constraint column shows the number of constraints in each model. Constraints exclude invalid combinations from the resulting test set. Consider the input model of the *printtokens* program, which contains different positions for different tokens. For example, *keyword* and *identifier* are two types of tokens that could appear at the beginning, middle or end of the input stream. A constraint is needed to prevent having more than one type of token appear at the same position.

Note that programs *printtokens* and *printtokens*2 share the same model, and so do programs *schedule* and *schedule*2. The model of *tcas* is the same as [25]. Also note that the models are built based on the specification of the programs, i.e., independent from their implementations.

We assume that boundary testing is done before combinatorial testing is applied. Combinatorial testing focuses on failures caused by interactions between parameters, while boundary testing focuses on failure caused by boundary values of individual parameters. We used the ACTS tool [2] to generate t-way test sets. For each (faulty) program, we first test it with a 2-way test set. If a faulty program is not detected by a 2-way test set, we increase the test strength and then test the program with a 3-way test set. This process is repeated until we reach strength 4.

TABLE 11
PROGRAMS MODEL

| Program | | Model | # of constraints |
|---|---|---|---|
| Siemens Suite | printtokens | $(2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2)$ | 8 |
| | replace | $(2^4 \times 4^{16})$ | 36 |
| | schedule | $(2^1 \times 3^8 \times 8^2)$ | 0 |
| | tcas | $(2^7 \times 3^2 \times 4^1 \times 10^2)$ | 0 |
| | totinfo | $(3^3 \times 5^2 \times 6^1)$ | 0 |
| flex | | $(2^6 \times 3^2 \times 5^1)$ | 12 |
| grep | | $(2^7 \times 4^1 \times 5^1 \times 6^3 \times 8^1 \times 9^1 \times 13^1)$ | 1 |
| gzip | | $(2^{13} \times 3^1)$ | 61 |
| sed | | $(2^7 \times 3^1 \times 4^1 \times 6^1 \times 10^1)$ | 50 |

To determine whether a test fails or passes for each faulty version, we run the test on the error-free version of the same program and the faulty version. The test fails if the faulty version produces a different output than the error-free version. Otherwise, it passes.

Table 12 shows the number of faulty versions detected by our test sets of different strengths for the *Siemens suite*. Note that the number of detected faulty versions by a t-way test set indicates all the faulty versions that are detected by the t-way test set but not by $(t-1)$-way test set. For example, 17, 12 and 7 faulty versions of *tcas* are detected by the 2-way, 3-way and 4-way test sets, respectively. The 12 faulty versions that are detected by the 3-way test set are different from the 17 and 7 faulty versions that are detected by the 2-way and 4-way test set, respectively. Therefore, in total, 36 faulty versions of *tcas* are detected. The same information for the *flex*, *grep*, *gzip* and *sed* programs is shown in Tables 13 to 16.

We also executed all the tests in the test pool that come with each program in SIR. (We will refer to the test pools in SIR as the SIR test pools.) These test pools are created initially in a black box manner based on the tester's understanding of the program's functionality and knowledge of special and boundary values. Then, white-box tests are created and added into the pools to ensure that each executable statement, branch, and definition-use pair in the error-free version was exercised [46]. All the faulty versions of the Siemens programs are detected by the test pools, except version 9 of *schedule2*. Combinatorial testing does not detect this version either. The results of executing the test pools on the *flex*, *grep*, *gzip* and *sed* programs are shown in the last column of Tables 13 to 16.

For the *grep*1 program, both test sets, i.e., our combinatorial test set and the SIR test pool, detected four faulty versions. Three out of these four versions are the same, and one is different. The combinatorial test set detected version 8 while the SIR test pool detected version 7. The combinatorial test set did not detect version 7 because the particular value that triggers the fault was not modeled in our model.

Moreover, version 2 of *grep*4 was detected by the combinatorial test set but not by the test pool. However, the test pool detected version 10, which is due to a boundary value that is not handled correctly.

Note that four faulty versions out of eighteen versions of *grep*1 were detected by the 2-way test set. However, in one of the detected faulty versions, i.e., version 11, all the tests failed. Based on Assumption 4, BEN was not applied

TABLE 12
TEST RESULTS FOR SIEMENS SUITE

| Program | # of faulty versions | # of detected versions | | | |
|---|---|---|---|---|---|
| | | *2-way* | *3-way* | *4-way* | *All* |
| printtokens | 7 | 3 | 0 | 0 | 3 |
| printtokens2 | 10 | 9 | 0 | 0 | 9 |
| replace | 32 | 32 | 0 | 0 | 32 |
| schedule | 9 | 7 | 0 | 0 | 7 |
| schedule2 | 10 | 3 | 0 | 0 | 3 |
| tcas | 41 | 17 | 12 | 7 | 36 |
| totinfo | 23 | 5 | 7 | 0 | 12 |

TABLE 13
TEST RESULTS FOR FLEX

| Program | # of faulty versions | # of detected versions | | | | # of detected versions by SIR test pool |
|---|---|---|---|---|---|---|
| | | *2-way* | *3-way* | *4-way* | *All* | |
| flex1 | 19 | 16 | 0 | 0 | 16 | 16 |
| flex2 | 20 | 10 | 0 | 0 | 10 | 14 |
| flex3 | 17 | 5 | 1 | 0 | 6 | 9 |
| flex4 | 16 | 11 | 0 | 0 | 11 | 11 |
| flex5 | 6 | 5 | 0 | 0 | 5 | 5 |

TABLE 14
TEST RESULTS FOR GREP

| Program | # of faulty versions | # of detected versions | | | | # of detected versions by SIR test pool |
|---|---|---|---|---|---|---|
| | | *2-way* | *3-way* | *4-way* | *All* | |
| grep1 | 18 | 4 | 0 | 0 | 4 | 4 |
| grep2 | 8 | 0 | 0 | 0 | 0 | 4 |
| grep3 | 18 | 4 | 0 | 0 | 4 | 7 |
| grep4 | 12 | 2 | 0 | 0 | 2 | 2 |
| grep5 | 1 | 0 | 0 | 0 | 0 | 0 |

TABLE 15
TEST RESULTS FOR GZIP

| Program | # of faulty versions | # of detected versions | | | | # of detected versions by SIR test pool |
|---|---|---|---|---|---|---|
| | | *2-way* | *3-way* | *4-way* | *All* | |
| gzip1 | 16 | 5 | 0 | 0 | 5 | 7 |
| gzip2 | 7 | 3 | 0 | 0 | 3 | 3 |
| gzip3 | 10 | 0 | 0 | 0 | 0 | 0 |
| gzip4 | 12 | 0 | 0 | 0 | 0 | 3 |
| gzip5 | 14 | 2 | 0 | 0 | 2 | 5 |

TABLE 16
TEST RESULTS FOR SED

| Program | # of faulty versions | # of detected versions | | | | # of detected versions by SIR test pool |
|---|---|---|---|---|---|---|
| | | *2-way* | *3-way* | *4-way* | *All* | |
| sed1 | 3 | 0 | 0 | 0 | 0 | 0 |
| sed2 | 5 | 4 | 0 | 0 | 4 | 5 |
| sed3 | 6 | 3 | 0 | 0 | 3 | 6 |
| sed4 | 4 | 0 | 0 | 0 | 0 | 1 |
| sed5 | 4 | 3 | 0 | 0 | 3 | 4 |
| sed6 | 6 | 6 | 0 | 0 | 6 | 6 |
| sed7 | 4 | 3 | 0 | 0 | 3 | 4 |

to this version.

In addition, BEN was not applied to four versions out of sixteen detected faulty versions of the *flex1*, four versions out of ten detected faulty versions of *flex2*, two versions out of eleven detected faulty versions of flex4 and four versions out of five detected faulty versions of *flex5*, as these versions did not have any passing test.

Moreover, in two versions out of five detected faulty versions of *gzip1* and one version out of three detected faulty versions of *gzip2* all the tests failed.

### 5.1.3. Multiple-Fault Versions

To evaluate the effectiveness of our approach when the program under test has more than one fault, we create several multiple-fault versions for each program. To increase the diversity, different multiple-fault versions have different numbers of faults. Table 17 shows the number of faulty versions with the number of faults created for each program. For example, we created three versions with two faults and one version with three faults for *printtokens*.

To create multiple-fault versions, we randomly pick faults from faults that are detected by our combinatorial test sets. Consider the *schedule* program. There are nine faulty versions and each version has one fault. The combinatorial test set detects seven of them (Table 12), versions 1 to 7, and the other two versions, versions 9 and 10, are not detected. To create multiple-fault versions with two faults, two faulty versions from 1 to 7 are selected randomly.

For each program, we generate one or more multiple-fault versions for a given number of faults. The maximum number of multiple-fault versions for each program depends on the number of detected faulty versions. When the total number of detected faulty versions is large, e.g., *replace* and *tcas*, we create multiple-fault versions with a maximum number of 10 faults. When the total number of detected faulty versions is small, e.g., *printtokens* and *schedule*2, more than one multiple-fault version is created for the same number of faults.

Since some faults may conflict with each other, combining them in one version is not possible. For example, the *schedule*2 program has three detected faulty versions, versions 2, 3 and 7. Two faulty versions, versions 3 and 7, conflict with each other. In version 7, the condition of an *if* statement is changed, while in version 3, the whole block that contains the same *if* statement is removed. Therefore, having these two versions in one multiple-fault version is not possible. For the *schedule*2 program, two multiple-fault versions with two faults are created. One of them contains the faults in versions 2 and 3, and the other contains the faults in versions 2 and 7.

Table 18 shows the result of combinatorial testing on multiple-fault versions. All of them are detected by a 2-way test set except one faulty version of program *printtokens*2 and one faulty version of program *tcas*, which are detected by a 3-way test set. In addition, one version of the *replace* program (the version with 8 faults), five versions of the *flex1* program (the versions with 5, 7, 8, 9, and 10 faults) and six versions of the *flex4* program (the versions with 4, 5, 6, 7, 8 and 9 faults) are ignored because all the tests in the 2-way test set failed for these versions.

### 5.1.4. Trace Collection

We used Gcov 4.1.2 [14] to collect execution traces. Gcov reports the number of times a statement is executed by a given test. A statement is included in the execution trace of a given test if and only if it is executed by the test for one or more times.

Gcov distinguishes between statements that are executable but are not executed and statements that are not executable. We used this information to compute the percentage of executable code that must be inspected to find the faulty statement. If a program crashes, Gcov does not report any coverage. To deal with this problem, we add a statement to call function *gcov_flush* before every

TABLE 17
MULTIPLE-FAULT VERSIONS

| Program | | # of multiple-fault versions | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 2 faults | 3 faults | 4 faults | 5 faults | 6 faults | 7 faults | 8 faults | 9 faults | 10 faults | ALL |
| Siemens Suite | printtokens | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | printtokens2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 7 |
| | replace | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 |
| | schedule | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 5 |
| | schedule2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| | tcas | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 |
| | totinfo | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 |
| flex | flex1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 |
| | flex2 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 5 |
| | flex3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 5 |
| | flex4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 9 |
| grep | grep1 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | grep3 | 3 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| | grep4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| gzip | gzip1 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | gzip2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | gzip5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| sed | sed2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| | sed3 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | sed5 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| | sed6 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| | sed7 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |

TABLE 18
TEST RESULTS FOR MULTIPLE-FAULT VERSIONS

| Program | | # of faulty versions | # of detected versions | | |
|---|---|---|---|---|---|
| | | | 2-way | 3-way | All |
| Siemens Suite | printtokens | 4 | 4 | 0 | 4 |
| | printtokens2 | 7 | 6 | 1 | 7 |
| | replace | 9 | 9 | 0 | 9 |
| | schedule | 5 | 5 | 0 | 5 |
| | schedule2 | 2 | 2 | 0 | 2 |
| | tcas | 9 | 8 | 1 | 9 |
| | totinfo | 9 | 9 | 0 | 9 |
| flex | flex1 | 9 | 9 | 0 | 9 |
| | flex2 | 5 | 5 | 0 | 5 |
| | flex3 | 5 | 5 | 0 | 5 |
| | flex4 | 9 | 9 | 0 | 9 |
| grep | grep1 | 4 | 4 | 0 | 4 |
| | grep3 | 7 | 7 | 0 | 7 |
| | grep4 | 1 | 1 | 0 | 1 |
| gzip | gzip1 | 4 | 4 | 0 | 4 |
| | gzip2 | 1 | 1 | 0 | 1 |
| | gzip5 | 1 | 1 | 0 | 1 |
| sed | sed2 | 3 | 3 | 0 | 3 |
| | sed3 | 4 | 4 | 0 | 4 |
| | sed5 | 4 | 4 | 0 | 4 |
| | sed6 | 3 | 3 | 0 | 3 |
| | sed7 | 4 | 4 | 0 | 4 |

statement. Note that this is only done after a program crashes.

### 5.1.5. BEN Configuration

For our experiments, we configure BEN to generate two tests for each of the five top ranked suspicious combinations at each iteration. In addition, because of resource limitation, the size of an inducing combination is limited to 6 for the *Siemens suite*, and it is limited to 4 for the four real-world programs, *flex*, *grep*, *gzip* and *sed*.

### 5.1.6. Metrics

Recall that the output of BEN is a ranking of statements in terms of their likelihood to be faulty. In order to find the faulty statement, we inspect statements in the first rank, and then statements in the second rank, and continue to do so until we find the actual faulty statement. Statements in the same rank are inspected in the order that they appear in the program. We record the number of statements that must be inspected to find the actual faulty statement in each program to measure the effectiveness of our approach.

Moreover, the effectiveness of the first phase, i.e., identifying inducing combination, is measured by the inducing probability (Definition 8) of the identified combination. The higher inducing probability the identified inducing combination has, the more effective our approach is.

The efficiency of our approach is measured by two factors: the number of tests that are executed and the

number of test runs that are traced for coverage collection. We show the number of tests executed in different stages of our approach, i.e., number of tests of the initial combinatorial test set, number of tests needed to identify inducing combinations (Phase 1), and number of tests needed to produce the ranking of faulty statements (Phase 2).

We also compare our approach to two approaches, Tarantula and Ochiai, in terms of effectiveness, i.e., the number of statements that must be inspected to find the actual faulty statement, and efficiency, i.e., the number of tests executed and the number of tests whose execution traces must be collected.

### 5.2. Results and Discussion

In this section, we discuss the results of applying BEN to the subject programs. We first report the results of BEN on the single-fault programs, followed by the results on the multiple-fault programs. Next, we compare the results of BEN to two approaches, Tarantula and Ochiai. Finally, the threats to validity are discussed.

#### 5.2.1. Results on Single-Fault Versions

This section is divided into two subsections. The first subsection reports the results of the first phase, identifying inducing combination. The second subsection discusses the results of the second phase, faulty statement localization.

##### 5.2.1.1. Phase 1: Identifying Inducing Combination

Table 19 shows the inducing probabilities of inducing combinations identified in the first phase. To compute the inducing probability for combination c, we generated and executed all the tests containing c. Then, inducing probability is computed using the formula explained in Section 2.1.

Depending on the input parameter model of the program, number of parameters, their domain size and constraints, generating all the tests containing a combination can be a very expensive task. This is the case for the inducing combinations identified for the two programs, *replace* and *grep*. Thus, inducing probabilities are not computed for these two programs. Note that this computation is only needed for the evaluation purpose. That is, it is not needed when our approach is applied in practice.

In Table 19 the "Test strength (t)" column shows the strength of the initial test set, and the next column, i.e., "# of detected versions", indicates the number of faulty versions detected using the corresponding test set.

Column "Avg size of inducing combinations" indicates the average size of inducing combinations for faulty versions that are detected by the t-way test set. For example, the sizes of the inducing combinations for three faulty versions, 3, 5 and 6, of *printtokens* that are detected by the 2-way test set, are 2, 4 and 3, respectively. Therefore, the average size of inducing combinations is 3. As explained in Section 3.1, the size of an inducing combination could be greater than the strength of the initial test set. The last column of Table 19 shows the average of inducing probabilities of inducing combinations.

As shown in Table 19, in most cases, the inducing probability is one, which means that the identified inducing combination is truly inducing. For *printtokens*2, *schedule*, *sed5* and *sed7*, the inducing probability is close to one. However, the inducing probability is very low in the *tcas* program and in one faulty version of the *flex*3 program, which is detected by a 3-way test set.

Recall that for our experiments, we limit the size of inducing combination to six for the Siemens programs and four for the four large programs. BEN reports the top ranked suspicious combination of size six (or four), if the inducing combination of a smaller size was not identified. For *tcas*, the average sizes of inducing combinations reported in Table 19 for test sets of different strengths are 6, 5.82 and 5.92. It is likely that BEN does not find the truly inducing combination in many cases. Thus, the average inducing probabilities are low. Similarly, for *flex3*, the average sizes of inducing combinations is 4. It is likely that BEN does not find the truly inducing combination in this case, as BEN limits the size of inducing combination to 4.

TABLE 19
INDUCING PROBABILITIES FOR SINGLE-FAULT VERSIONS

| Program | | Test strength (t) | # of detected versions | Avg size of inducing combinations | Avg inducing probability of the identified inducing combinations |
|---|---|---|---|---|---|
| Siemens Suite | printtokens | 2 | 3 | 3 | 1 |
| | printtokens2 | 2 | 9 | 2.56 | 0.93 |
| | schedule | 2 | 7 | 2.86 | 0.86 |
| | schedule2 | 2 | 3 | 2 | 1 |
| | tcas | 2 | 17 | 5.82 | 0.09 |
| | | 3 | 12 | 5.92 | 0.11 |
| | | 4 | 7 | 6 | 0.06 |
| | totinfo | 2 | 5 | 4.8 | 1 |
| | | 3 | 7 | 4.86 | 1 |
| flex | flex1 | 2 | 12 | 2 | 1 |
| | flex2 | 2 | 6 | 2 | 1 |
| | flex3 | 2 | 5 | 2.2 | 1 |
| | | 3 | 1 | 4 | 0.25 |
| | flex4 | 2 | 9 | 2.11 | 1 |
| | flex5 | 2 | 1 | 2 | 1 |
| gzip | gzip1 | 2 | 3 | 2 | 1 |
| | gzip2 | 2 | 2 | 2 | 1 |
| | gzip5 | 2 | 2 | 2 | 1 |
| sed | sed2 | 2 | 4 | 2.25 | 1 |
| | sed3 | 2 | 3 | 2 | 1 |
| | sed5 | 2 | 3 | 2 | 0.92 |
| | sed6 | 2 | 6 | 2.83 | 1 |
| | sed7 | 2 | 3 | 3 | 0.96 |

TABLE 20
RESULTS FOR SINGLE-FAULT VERSIONS

| Program | | Test strength (t) | # of tests in t-way test set | # of detected versions | Avg size of inducing combinations | Avg # of tests for identifying inducing combination | Avg # of times the core member does not fail | Avg # of tests executed for generating derived members | Avg # of times derived members selected from initial test set | Avg # of test runs traced for coverage collection | Avg # of statements inspected to find actual fault | Avg percentage of statements inspected to locate actual fault |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Siemens Suite | printtokens | 2 | 170 | 3 | 3 | 20 | 0 | 10 | 0 | 11 | 25.66 | 13.65 |
| | printtokens2 | 2 | 170 | 9 | 2.56 | 16.67 | 0 | 10.89 | 0 | 11.89 | 13.55 | 6.74 |
| | replace | 2 | 193 | 32 | 3.66 | 19.37 | 0.41 | 4.16 | 0 | 5.57 | 30.91 | 12.77 |
| | schedule | 2 | 64 | 7 | 2.86 | 17.14 | 0.14 | 6.43 | 0 | 7.57 | 18.71 | 12.15 |
| | schedule2 | 2 | 64 | 3 | 2 | 10 | 0 | 4.33 | 0 | 5.33 | 59.67 | 46.98 |
| | tcas | 2 | 100 | 17 | 5.82 | 32.23 | 0.94 | 21.35 | 0 | 23.29 | 14 | 21.54 |
| | | 3 | 405 | 12 | 5.91 | 25 | 0.92 | 20.83 | 0 | 22.75 | 14.67 | 22.57 |
| | | 4 | 1434 | 7 | 6 | 20 | 1 | 18.57 | 0 | 20.57 | 11.14 | 17.14 |
| | totinfo | 2 | 30 | 5 | 4.8 | 40 | 0 | 11.5 | 0 | 12.5 | 20.8 | 16.91 |
| | | 3 | 156 | 7 | 4.86 | 27.43 | 0 | 13.5 | 0 | 14.5 | 11.71 | 9.52 |
| flex | flex1 | 2 | 26 | 12 | 2 | 10 | 0 | 2.25 | 0.25 | 3.5 | 161.58 | 4.76 |
| | flex2 | 2 | 26 | 6 | 2 | 9.5 | 0 | 3.17 | 0 | 4.17 | 36.67 | 0.93 |
| | flex3 | 2 | 26 | 5 | 2.2 | 12 | 0 | 3.2 | 0 | 4.2 | 316.6 | 8.04 |
| | | 3 | 66 | 1 | 4 | 4 | 1 | 7 | 0 | 9 | 88 | 2.23 |
| | flex4 | 2 | 26 | 9 | 2.11 | 11.55 | 0 | 2.22 | 0.11 | 3.33 | 240.67 | 6.07 |
| | flex5 | 2 | 26 | 1 | 2 | 10 | 0 | 5 | 0 | 6 | 262 | 6.60 |
| grep | grep1 | 2 | 121 | 3 | 2.67 | 13.33 | 0.33 | 10.67 | 0 | 12 | 347 | 11.27 |
| | grep3 | 2 | 121 | 4 | 3 | 17.5 | 0.5 | 5.25 | 0 | 6.75 | 21.25 | 0.64 |
| | grep4 | 2 | 121 | 2 | 2 | 10 | 0 | 5 | 0 | 6 | 172.5 | 5.21 |
| gzip | gzip1 | 2 | 21 | 3 | 2 | 8.67 | 0 | 2 | 0.67 | 3.67 | 339.33 | 19.90 |
| | gzip2 | 2 | 21 | 2 | 2 | 9 | 0 | 2 | 1 | 4 | 76.5 | 3.81 |
| | gzip5 | 2 | 21 | 2 | 2 | 8 | 0 | 2 | 1 | 4 | 86 | 4.31 |
| sed | sed2 | 2 | 58 | 4 | 2.25 | 9 | 0 | 5 | 0.25 | 6.25 | 85.5 | 2.52 |
| | sed3 | 2 | 58 | 3 | 2 | 6.33 | 0 | 10 | 0 | 11 | 4 | 0.18 |
| | sed5 | 2 | 58 | 3 | 2 | 3.67 | 0 | 2 | 0.67 | 3.67 | 4 | 0.09 |
| | sed6 | 2 | 58 | 6 | 2.83 | 6.17 | 0 | 7.17 | 0.5 | 8.67 | 160.67 | 3.54 |
| | sed7 | 2 | 58 | 3 | 3 | 13.67 | 0 | 7.67 | 0 | 8.67 | 11.67 | 0.24 |

#### 5.2.1.2. Phase 2: Faulty Statement Localization

Table 20 shows the results of our approach on each program. We will not explain the column headers one by one, as they are self-explanatory. Note that in the last eight columns, average values are used, since the data could be different in different versions.

Column, "Avg # of tests for identifying inducing combination", shows the average number of tests generated in the first phase, i.e., inducing combination identification.

If a combination c identified in the first phase is not inducing, there is a probability that the core member does not fail. The higher the inducing probability, the more likely the core member fails. If the inducing probability is 1, the core member will definitely fail. However, our approach can still apply if the core member does not fail. We select as the core member a failing test that contains the inducing combination from the initial test set. Column "Avg # of times the core member does not fail" shows the average number of such cases. For all the seven versions of *tcas*, when the initial test set is 4-way, the core member is selected from the initial test set. This is consistent with the fact that the inducing probabilities of the identified inducing combinations were very small (Table 19).

For each version, we compute the total number of tests in all the derived member sets, i.e., all the tests executed for generating the derived members. The average of this number for all versions is shown in the ninth column, "Avg # of tests executed for generating derived members". The number includes all the tests, although later some of them are discarded since they do not pass. The maximum value of this column, 21.35, is for the *tcas* program with a 2-way test set. The minimum value, 2, happens for *gzip1, gzip2, gzip5* and *sed5*. Note that the number of tests executed for generating derived members depends on the size of inducing combination, the domain size of inducing components, and also system constraints.

The column, "Avg # of times derived members are selected from initial test set", shows the number of cases that all the derived member candidates failed, and a derived member is selected from the initial test set.

The column, "Avg # of test runs traced for coverage", shows the average number of derived members whose traces are collected. Recall from Section 3.2, the tests of a candidate set are traced for coverage collection. Note that BEN also needs the execution trace of the core member. Therefore the total number of test runs traced by the coverage tool is the summation of the following four numbers: 1) one representing the core member; 2) number of times the core member selected from initial test set (column eight of Table 20); 3) number of tests executed for generating derived members (column nine of Table 20); and 4) number of derived members selected from initial test set (column ten of Table 20).

The last two columns show the average number and percentage of statements that must be inspected to locate a fault. To compute this number, we include statements that are ranked higher and statements that are ranked at the same rank but appear before the faulty statement. We did not perform any dependency analysis, which could reduce the number of statements that must be inspected.

We note that the number of executable statements in *tcas* is 65, less than 100. In this program, when only one statement is needed to inspect, it is 1.54% of executable code. Therefore, for the *tcas* program the number of statements gives better insight than the percentage of code.

As shown in Table 20 our approach works better for the *flex*, *grep*, *gzip* and *sed* programs than the Siemens programs, i.e. small programs. The best case happens with *sed5* where only 0.09% of code must be inspected to locate the fault. The worst case happens with *gzip*1 where 19.90% of the code must be inspected. For the Siemens programs, the best and worst cases happen with *printtokens* and *schedule*2, where 6.74% and 46.98% of the code must be inspected, respectively.

### 5.2.2. Results on Multiple-Fault Versions

In this section, we discuss the results of our experiments on the subject programs that have multiple faults.

#### 5.2.2.1. Phase 1: Identifying Inducing Combination

Table 21 shows the inducing probabilities for the inducing combinations identified in the first phase. To compute inducing probability, the same procedure used in Section 5.2.1.1 for single-fault versions is performed. Again, two programs, *grep* and *replace*, are ignored as it is very expensive to compute inducing probabilities for these programs.

As shown in Table 21, the inducing probabilities for all programs are one or close to one, except for the *tcas* program. In the five faulty versions (four versions detected by 2-way test set and one detected by 3-way) of the *tcas* program, BEN does not find any inducing combination of size of five or less. Therefore, the most suspicious combination whose size is six is reported as an inducing combination.

#### 5.2.2.2. Phase 2: Faulty Statement Localization

The results are summarized in Table 22, where the columns are the same as in Table 20. The last two columns, "Avg # of statements inspected to find actual faults" and "Avg percentage of statements inspected to locate actual faults", show respectively the number of statements and percentage of statements that should be inspected to locate the first faulty statement.

Similar to the single-fault versions, BEN works better for *flex*, *grep*, *gzip* and *sed*, than for the Siemens programs. For the four large programs, the worst case happens in *flex4*, where 10.82% of executable code must be inspected to locate the fault. However, the worst case for the Siemens programs happens with *schedule*2, where 25.83% of the executable code must be inspected.

The results in Tables 20 and 22 suggest that BEN works better when there are multiple faults. For all the programs, BEN is more effective for multiple-fault versions than single-fault versions, except *flex4*, *grep*3, *sed3* and *sed7*, in terms of percentage of code that needs to be inspected. Moreover, BEN is more efficient for multiple-fault versions than single-fault versions, in terms of the total number of tests generated in phases 1 and 2 and the number of test runs traced by the coverage tool for multiple-fault versions. This can be explained as follows.

The more faults a program has, the more likely that a test fails. When there are more failing tests in the initial test set, it is likely to have more inducing combinations or the

TABLE 21
INDUCING PROBABILITIES FOR MULTIPLE-FAULT VERSIONS

| Programs | | Test strength (t) | # of detected versions | Avg size of inducing combinations | Avg inducing probability of the identified inducing combinations |
|---|---|---|---|---|---|
| Siemens Suite | printtokens | 2 | 4 | 2.75 | 0.95 |
| | printtokens2 | 2 | 7 | 2.14 | 1 |
| | schedule | 2 | 5 | 2 | 0.86 |
| | schedule2 | 2 | 2 | 2 | 1 |
| | tcas | 2 | 8 | 5.12 | 0.33 |
| | | 3 | 1 | 6 | 0.02 |
| | totinfo | 2 | 9 | 4.67 | 1 |
| flex | flex1 | 2 | 4 | 2 | 1 |
| | flex2 | 2 | 5 | 2 | 1 |
| | flex3 | 2 | 5 | 2 | 1 |
| | flex4 | 2 | 2 | 2 | 1 |
| gzip | gzip1 | 2 | 4 | 2 | 1 |
| | gzip2 | 2 | 1 | 2 | 1 |
| | gzip5 | 2 | 1 | 2 | 1 |
| sed | sed2 | 2 | 3 | 2.33 | 0.85 |
| | sed3 | 2 | 4 | 2 | 1 |
| | sed5 | 2 | 4 | 2 | 1 |
| | sed6 | 2 | 3 | 2 | 1 |
| | sed7 | 2 | 4 | 2.25 | 1 |

TABLE 22
RESULTS FOR MULTIPLE-FAULT VERSIONS

| Programs | | Test strength (t) | # of tests in t-way test set | # of detected versions | Avg size of inducing combination | Avg # of tests for identifying inducing combination | Avg # of times the core member does not fail | Avg # of tests executed for generating derived members | Avg # of times derived members selected from initial test set | Avg # of test runs traced for coverage collection | Avg # of statements inspected to find actual fault | Avg percentage of statements inspected to locate actual fault |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Siemens Suite | printtokens | 2 | 170 | 4 | 2.75 | 17.5 | 0 | 5 | 0 | 6 | 1.25 | 0.66 |
| | printtokens2 | 2 | 170 | 7 | 2.14 | 11.43 | 0 | 3.14 | 0 | 4.14 | 1.86 | 0.92 |
| | replace | 2 | 193 | 8 | 2.5 | 13 | 0.12 | 1.87 | 0 | 2.99 | 12.25 | 5.06 |
| | schedule | 2 | 64 | 5 | 2 | 10 | 0.2 | 2.60 | 0 | 3.80 | 8.2 | 5.32 |
| | schedule2 | 2 | 64 | 2 | 2 | 10 | 0 | 4 | 0 | 5 | 45.5 | 25.83 |
| | tcas | 2 | 100 | 8 | 5.12 | 31.75 | 0.50 | 14.37 | 0 | 15.87 | 3.62 | 5.57 |
| | | 3 | 405 | 1 | 6 | 22 | 1 | 23 | 0 | 25 | 11 | 16.92 |
| | totinfo | 2 | 30 | 9 | 4.67 | 36.67 | 0 | 9.78 | 0 | 10.78 | 8.67 | 7.05 |
| flex | flex1 | 2 | 26 | 4 | 2 | 10 | 0 | 2.25 | 0.75 | 4 | 127.5 | 3.76 |
| | flex2 | 2 | 26 | 5 | 2 | 10 | 0 | 2 | 0 | 3 | 11 | 0.28 |
| | flex3 | 2 | 26 | 5 | 2 | 10 | 0 | 2.2 | 0 | 3.2 | 63 | 1.60 |
| | flex4 | 2 | 26 | 2 | 2 | 10 | 0 | 2 | 0.5 | 3.5 | 429 | 10.82 |
| grep | grep1 | 2 | 121 | 4 | 2.5 | 15 | 0 | 5.5 | 0 | 6.5 | 107.5 | 3.49 |
| | grep3 | 2 | 121 | 7 | 3.14 | 15.71 | 0.57 | 3.14 | 0 | 4.71 | 32.86 | 1 |
| | grep4 | 2 | 121 | 1 | 2 | 10 | 0 | 3 | 0 | 4 | 23 | 0.69 |
| gzip | gzip1 | 2 | 21 | 4 | 2 | 8 | 0 | 2 | 1 | 4 | 88.75 | 5.21 |
| | gzip2 | 2 | 21 | 1 | 2 | 10 | 0 | 2 | 1 | 4 | 32 | 1.60 |
| | gzip5 | 2 | 21 | 1 | 2 | 4 | 0 | 2 | 1 | 4 | 83 | 4.16 |
| sed | sed2 | 2 | 58 | 3 | 2.33 | 11.33 | 0 | 2.67 | 0 | 3.67 | 64.67 | 1.91 |
| | sed3 | 2 | 58 | 4 | 2 | 8.75 | 0 | 6.25 | 0 | 7.25 | 5.75 | 0.26 |
| | sed5 | 2 | 58 | 4 | 2 | 2.75 | 0 | 2 | 0.75 | 3.75 | 3.25 | 0.07 |
| | sed6 | 2 | 58 | 3 | 2 | 5 | 0 | 2 | 1 | 4 | 38 | 0.84 |
| | sed7 | 2 | 58 | 4 | 2.25 | 10.75 | 0 | 5.5 | 0 | 6.5 | 17.5 | 0.36 |

size of inducing combination is smaller. Inducing combinations of smaller size are less expensive to identify than those of larger size. This is because the smaller the inducing combination is, the fewer times the *identify* algorithm is called to identify the combination. Moreover, the number of candidate sets equals the size of inducing combination. Thus, the smaller the inducing combination is, the fewer derived candidate sets and therefore the fewer tests are generated in the second phase.

### 5.2.3. Comparison with Tarantula and Ochiai

We compared BEN to two spectrum-based approaches, Tarantula and Ochiai, in terms of effectiveness and efficiency. Experiments suggest that Tarantula and Ochiai perform best among spectrum based approaches [1][23][31]. Recall that effectiveness is measured by the percentage of executable code that must be examined to guide the programmer to the faulty statement, and efficiency is measured by the number of tests executed, the number of tests runs traced for coverage collection, and the execution time.

Since Tarantula and Ochiai do not deal with test generation, we applied them using the initial combinatorial test set.

Tables 23 and 24 show the comparison results for single-fault versions and multiple-fault versions, respectively. We used average to aggregate the results of all the detected faulty versions for each program. The third column shows the average size of the combinatorial test sets used in the testing stage for each program. The fourth column shows the number of detected faulty versions.

The average number of test runs traced for BEN is shown in the sixth column. For Tarantula and Ochiai, every test run needs to be traced, so the average number of test runs traced is the same as the number shown in the third column. As shown in Tables 23 and 24, BEN needs to trace only a very small number of tests in comparison with the other two approaches. However, BEN generates and executes a number of tests (in addition to the initial test set) to identify the inducing combination. This cost is shown in the fifth column of Tables 23 and 24, and it equals the seventh column of Tables 20 and 22.

We also report the execution time of BEN, Tarantula and Ochiai for four large programs *flex*, *grep*, *gzip* and *sed*. Experiments are conducted on a server that has an Intel(R) Xeon(R) E5410 @ 2.33GHz (4-cores) processor and 4 GB memory and that runs Red Hat Enterprise Linux

TABLE 23
EFFICIENCY COMPARISON RESULTS FOR SINGLE-FAULT VERSIONS

| Program | | Avg # of tests executed in the testing stage* | # of detected versions | Avg # of tests generated and executed in fault localization stage by BEN | Avg # of test runs traced for coverage collection by BEN | Avg execution time (in seconds) | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Tarantula | Ochiai | BEN |
| Siemens Suite | printtokens | 170 | 3 | 20 | 11 | - | - | - |
| | printtokens2 | 170 | 9 | 16.67 | 11.89 | - | - | - |
| | replace | 193 | 32 | 19.37 | 5.57 | - | - | - |
| | schedule | 64 | 7 | 17.14 | 7.57 | - | - | - |
| | schedule2 | 64 | 3 | 10 | 5.33 | - | - | - |
| | tcas | 461.05 | 36 | 27.44 | 22.58 | - | - | - |
| | totinfo | 103.5 | 12 | 32.67 | 13.67 | - | - | - |
| flex | flex1 | 26 | 12 | 10 | 3.5 | 15.61 | 15.61 | 4.93 |
| | flex2 | 26 | 6 | 9.5 | 4.17 | 18.74 | 18.78 | 5.83 |
| | flex3 | 32.67 | 6 | 10.67 | 5 | 18.78 | 18.78 | 6.89 |
| | flex4 | 26 | 9 | 11.55 | 3.33 | 18.79 | 18.83 | 4.88 |
| | flex5 | 26 | 1 | 10 | 6 | 18.96 | 18.87 | 7.05 |
| grep | grep1 | 121 | 3 | 13.33 | 12 | 94.82 | 94.65 | 14.85 |
| | grep3 | 121 | 4 | 17.5 | 6.75 | 99.05 | 99.06 | 29.32 |
| | grep4 | 121 | 2 | 10 | 6 | 98.96 | 99.06 | 7.65 |
| gzip | gzip1 | 21 | 3 | 8.67 | 3.67 | 14.93 | 14.94 | 7.52 |
| | gzip2 | 21 | 2 | 9 | 4 | 14.69 | 14.69 | 8.65 |
| | gzip5 | 21 | 2 | 8 | 4 | 16.86 | 16.84 | 8.11 |
| sed | sed2 | 58 | 4 | 9 | 6.25 | 37.58 | 37.52 | 11.67 |
| | sed3 | 58 | 3 | 6.33 | 11 | 32.75 | 32.87 | 12.18 |
| | sed5 | 58 | 3 | 3.67 | 3.67 | 49.66 | 49.69 | 10.09 |
| | sed6 | 58 | 6 | 6.17 | 8.67 | 49.73 | 49.79 | 14.10 |
| | sed7 | 58 | 3 | 13.67 | 8.67 | 52.96 | 53.03 | 15.30 |

** In Tarantula and Ochiai, all the test runs are traced. Thus, the Avg # of test runs traced for coverage collection is the same as the Avg # of tests executed in the testing stage."*

Server 6.5 (Santiago) (64 bit). Moreover, BEN uses Choco [7] as a constraint solver.

The last three columns of Tables 23 and 24 show the time comparison results for single-fault versions and multiple-fault versions, respectively. We used average to aggregate the results of all the detected faulty versions for each program. The seventh and eighth columns of Tables 23 and 24 show the average time in seconds to run Tarantula and Ochiai, which includes executing tests to collect their spectra and computing the statement ranks based on Tarantula or Ochiai formula, respectively. The last column of Tables 23 and 24 indicates the average execution time of BEN. This time includes time spent in both phases, including inducing combination identification and faulty statement localization.

Note that the execution time in Tables 23 and 24 does not include the time needed for test evaluation. Recall from Section 5.1.2, in our experiments，a test run is evaluated by comparing its output to the output produced by running the same test against the error-free version. In practice, however, we do not have access to the error-free version of a program. Thus, it could be misleading to include the test evaluation time. As discussed in Section 2.2, the test oracle problem is common to many testing and fault localization approaches. BEN is most effective when there exists an automated test oracle or when test evaluation could be performed quickly.

As shown in Tables 23 and 24, the seventh and eighth columns, are almost equal, for single and multiple-fault versions of the four programs, *flex*, *grep*, *gzip* and *sed*. For all these programs *flex, grep, gzip* and *sed*, single-fault and multiple-fault versions, BEN is faster than Tarantula and Ochiai. The best case happens in multiple-fault versions of *grep4* where BEN is 17.3 times faster than Ochiai.

In [23][40], a score is used to compare different fault localization methods. The score is defined based on the percentage of code that must be examined to find the faulty statement. The percentage is based on executable code, i.e., non-executable code is excluded. Tables 25 and 26 show the percentage of all the program versions that achieve each score for single-fault and multiple-fault versions, respectively. The results of BEN, Tarantula and Ochiai for the Siemens programs are aggregated and shown in the "Siemens Suite" rows, and the results of these three approaches for the *flex*, *grep*, *gzip* and *sed* programs are aggregated in their corresponding rows.

For single-fault versions (Table 25), on the first score, i.e., 99-100%, which means only 1% or less than 1% of code must be inspected to find the first faulty statement, BEN outperforms Tarantula for the Siemens, *flex* and *grep* programs, while both have the same results for the *gzip* and *sed* programs.

BEN performs better than Ochiai for the Siemens programs and the same for the *grep* and *gzip* programs on

TABLE 24
EFFICIENCY COMPARISON RESULTS FOR MULTIPLE-FAULT VERSIONS

| Program | | Avg # of tests executed in the testing stage* | # of detected versions | Avg # of tests generated and executed in fault localization stage by BEN | Avg # of test runs traced for coverage collection by BEN | Avg execution time (in seconds) | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | Tarantula | Ochiai | BEN |
| Siemens Suite | printtokens | 170 | 4 | 17.5 | 6 | - | - | - |
| | printtokens2 | 170 | 7 | 11.43 | 4.14 | - | - | - |
| | replace | 193 | 8 | 13 | 2.99 | - | - | - |
| | schedule | 64 | 5 | 10 | 3.80 | - | - | - |
| | schedule2 | 64 | 2 | 10 | 5 | - | - | - |
| | tcas | 133.89 | 9 | 30.67 | 16.88 | - | - | - |
| | totinfo | 30 | 9 | 36.67 | 10.78 | - | - | - |
| flex | flex1 | 26 | 4 | 10 | 4 | 15.71 | 15.68 | 5.60 |
| | flex2 | 26 | 5 | 10 | 3 | 18.75 | 18.76 | 5.11 |
| | flex3 | 26 | 5 | 10 | 3.2 | 18.64 | 18.65 | 5.05 |
| | flex4 | 26 | 2 | 10 | 3.5 | 18.78 | 18.77 | 5.51 |
| grep | grep1 | 121 | 4 | 15 | 6.5 | 94.32 | 94.42 | 9.88 |
| | grep3 | 121 | 7 | 15.71 | 4.71 | 98.95 | 98.98 | 32.86 |
| | grep4 | 121 | 1 | 10 | 4 | 98.67 | 98.79 | 5.71 |
| gzip | gzip1 | 21 | 4 | 8 | 4 | 12.63 | 12.63 | 7.94 |
| | gzip2 | 21 | 1 | 10 | 4 | 15.37 | 15.40 | 9.12 |
| | gzip5 | 21 | 1 | 4 | 4 | 14.8 | 14.8 | 8.88 |
| sed | sed2 | 58 | 3 | 11.33 | 3.67 | 37.34 | 37.34 | 10.55 |
| | sed3 | 58 | 4 | 8.75 | 7.25 | 32.79 | 32.63 | 10.16 |
| | sed5 | 58 | 4 | 2.75 | 3.75 | 49.61 | 49.74 | 9.72 |
| | sed6 | 58 | 3 | 5 | 4 | 49.74 | 49.64 | 10.17 |
| | sed7 | 58 | 4 | 10.75 | 6.5 | 52.89 | 52.88 | 12.30 |

** In Tarantula and Ochiai, all the test runs are traced. Thus, the Avg # of test runs traced for coverage collection is the same as the Avg # of tests executed in the testing stage."*

the first score. However, Ochiai outperforms BEN for the *flex* and *sed* programs, in terms of the first score of single-fault versions. Note that on the second score, i.e., 90-99%, BEN outperforms Ochiai for the *flex* program.

For multiple-fault versions of all the programs (Table 26), on the first score, BEN outperforms Tarantula, except for the *gzip* program. For the *gzip* program, all three approaches, BEN, Tarantula and Ochiai, produce the same score. BEN also outperforms Ochiai for the Siemens, *flex*, *grep* and *sed* programs. Moreover, the improvement of BEN in comparison with the other approaches is greater for the multiple-fault versions compared to the single-fault versions. The reason is that BEN first identifies one inducing combination and it is likely that each inducing combination corresponds to one faulty statement. In the second phase, BEN generates a group of tests with one failing test, i.e., the core member, which likely includes one inducing combination and executes only one faulty statement. Therefore, even when there is more than one fault in the program, BEN focuses on one of them. However, when Tarantula and Ochiai are applied on multiple-fault programs, they use the initial test set that likely includes several failing tests corresponding to different faulty statements. Moreover, Tarantula and Ochiai do not perform any nearest neighbor analysis. Thus, it is likely that very different execution traces are compared to each other, which reduces their effectiveness of locating the faulty statement.

Tables 27 and 28 show the comparison between BEN, Tarantula, and Ochiai for single-fault and multiple-fault versions, respectively, based on number of outperformed versions. There are two groups of columns that show the comparison between BEN and Tarantula and the comparison between BEN and Ochiai, respectively.

In each group, the first two columns show cases that BEN outperforms the other approach, Tarantula or Ochiai (positive numbers). The first column shows the number of detected faulty versions that BEN outperforms the other approach, and the next one shows the average percentage of the differences. For example in the 19 out of 36 detected single-fault versions of the *tcas* program, BEN inspects 7.94% (of executable code) less than Tarantula.

The third column of each group shows the number of detected faulty versions that BEN and the other approach, Tarantula or Ochiai, produce the same results. In addition, the last two columns of each group show the number of versions that the other approach outperforms BEN and the average percentage of the differences (negative numbers). For example for five out of 36 detected single-fault versions of the *tcas* program, BEN inspects about 3.38% (of executable code) more than Tarantula.

Five rows, *Siemens suite*, *flex*, *grep, gzip* and *sed*, are added to summarize the results of all the Siemens programs, all the different versions of *flex*, *grep, gzip* and *sed* versions, respectively.

TABLE 25
COMPARISON RESULTS FOR SINGLE-FAULT VERSIONS BASED ON PERCENTAGE OF CODE INSPECTED

| Program | Approach | Score | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 99-100% | 90-99% | 80-90% | 70-80% | 60-70% | 50-60% | 40-50% | 30-40% | 20-30% | 10-20% |
| Siemens Suite | BEN | 23.53 | 30.39 | 22.55 | 4.90 | 3.92 | 9.80 | 1.96 | 1.96 | 0.98 | 0 |
| | Ochiai | 20.59 | 34.31 | 14.71 | 11.76 | 4.90 | 5.88 | 5.88 | 1.96 | 0 | 0 |
| | Tarantula | 18.63 | 33.33 | 16.67 | 11.76 | 3.92 | 3.92 | 8.82 | 0.98 | 1.96 | 0 |
| flex | BEN | 35.30 | 50.00 | 11.76 | 2.94 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 55.88 | 23.53 | 14.71 | 5.88 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 32.36 | 44.12 | 8.82 | 8.82 | 5.88 | 0 | 0 | 0 | 0 | 0 |
| grep | BEN | 66.67 | 11.11 | 22.22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 66.67 | 11.11 | 22.22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 55.56 | 11.11 | 22.22 | 11.11 | 0 | 0 | 0 | 0 | 0 | 0 |
| gzip | BEN | 14.29 | 57.14 | 14.29 | 14.28 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 14.29 | 57.14 | 14.29 | 14.28 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 14.29 | 57.14 | 14.29 | 14.28 | 0 | 0 | 0 | 0 | 0 | 0 |
| sed | BEN | 78.95 | 21.05 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 89.47 | 10.53 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 78.95 | 15.79 | 5.26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

TABLE 26
COMPARISON RESULTS FOR MULTIPLE-FAULT VERSIONS BASED ON PERCENTAGE OF CODE INSPECTED

| Program | Approach | Score | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 99-100% | 90-99% | 80-90% | 70-80% | 60-70% | 50-60% | 40-50% | 30-40% | 20-30% | 10-20% |
| Siemens Suite | BEN | 38.64 | 40.91 | 18.18 | 0 | 0 | 0 | 0 | 0 | 2.27 | 0 |
| | Ochiai | 31.82 | 52.27 | 13.64 | 0 | 0 | 0 | 0 | 2.27 | 0 | 0 |
| | Tarantula | 31.82 | 61.36 | 4.55 | 0 | 0 | 0 | 0 | 0 | 2.27 | 0 |
| flex | BEN | 56.25 | 37.5 | 6.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 43.75 | 25 | 6.25 | 18.75 | 6.25 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 37.5 | 62.5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| grep | BEN | 91.67 | 0 | 8.33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 75.00 | 16.67 | 8.33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 58.33 | 33.33 | 8.33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| gzip | BEN | 50 | 33.33 | 16.67 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 50 | 33.33 | 16.67 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 50 | 33.33 | 16.67 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sed | BEN | 94.44 | 5.56 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Ochiai | 88.89 | 11.11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | Tarantula | 83.33 | 16.67 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

For single-fault versions (Table 27), BEN outperforms Tarantula in all the five cases, *Siemens suite*, *flex*, *grep, gzip* and *sed*, which is consistent with Table 25. However, the difference between BEN and Tarantula is very small (less than one percent) for the *gzip* program, and thus it is not reflected in Table 25. According to Table 27, BEN outperforms Ochiai for the Siemens programs, while Ochiai works better than BEN for the *flex, grep, gzip* and *sed* programs, for single-fault versions. Note that the difference between BEN and Ochiai is very small for the *grep* and *gzip*, and it is not reflected in Table 25.

For multiple-fault versions (Table 28), BEN outperforms Ochiai for three cases, *Siemens Suite*, *grep* and *gzip*, although the difference between the two approaches is very small for the Siemens programs. For the *flex* program, Ochiai works better than BEN in more versions while BEN makes greater average of difference percentage than Ochiai.

In Siemens, *flex* and *sed* programs, Tarantula is more effective than BEN; however, BEN is much more effective

TABLE 27
COMPARISON RESULTS FOR SINGLE-FAULT VERSIONS BASED ON NUMBER OF OUTPERFORMED VERSIONS

| Program | | # of detected versions | Tarantula | | | | | Ochiai | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | BEN > Tarantula | | BEN = Tarantula | BEN < Tarantula | | BEN > Ochiai | | BEN = Ochiai | BEN < Ochiai | |
| | | | # of versions | Avg of Difference Percentages | | # of versions | Avg of Difference Percentages | # of versions | Avg of Difference Percentages | | # of versions | Avg of Difference Percentages |
| Siemens Suite | printtokens | 3 | +1 | +8.51 | 1 | -1 | -0.53 | +1 | +4.79 | 1 | -1 | -0.53 |
| | printtokens2 | 9 | +3 | +5.97 | 2 | -4 | -3.98 | +1 | +9.45 | 4 | -4 | -7.21 |
| | replace | 32 | +14 | +8.56 | 4 | -14 | -9.80 | +14 | +8.21 | 4 | -14 | -11.16 |
| | schedule | 7 | +2 | +1.30 | 1 | -4 | -13.47 | +2 | +1.30 | 1 | -4 | -13.47 |
| | schedule2 | 3 | +2 | +5.91 | 1 | 0 | 0 | 0 | 0 | 1 | -2 | -3.54 |
| | tcas | 36 | +19 | +7.94 | 12 | -5 | -3.38 | +19 | +7.61 | 12 | -5 | -3.38 |
| | totinfo | 12 | +4 | +27.85 | 6 | -2 | -13.82 | +3 | +4.07 | 6 | -3 | -10.03 |
| Siemens Suite | | 102 | +45 | +9.40 | 27 | -30 | -8.40 | +40 | +7.21 | 29 | -33 | -8.90 |
| flex | flex1 | 12 | +9 | +5.39 | 2 | -1 | -5.30 | +1 | +8.52 | 1 | -10 | -2.12 |
| | flex2 | 6 | +3 | +0.75 | 2 | -1 | -0.64 | 0 | 0 | 4 | -2 | -1.72 |
| | flex3 | 6 | +3 | +5.64 | 0 | -3 | -0.12 | +2 | +6.42 | 0 | -4 | -1.21 |
| | flex4 | 9 | +8 | +8.53 | 0 | -1 | -0.66 | +2 | +8.20 | 1 | -6 | -1.14 |
| | flex5 | 1 | 0 | 0 | 0 | -1 | -0.10 | 0 | 0 | 0 | -1 | 0.10 |
| flex | | 34 | +23 | +5.91 | 4 | -7 | -1.01 | +5 | +7.52 | 6 | -23 | -1.57 |
| grep | grep1 | 3 | +1 | +3.77 | 2 | 0 | 0 | 0 | 0 | 1 | -2 | -1.14 |
| | grep3 | 4 | +1 | +6.80 | 1 | -2 | -0.09 | +1 | +0.24 | 1 | -2 | -0.09 |
| | grep4 | 2 | +1 | +3.08 | 0 | -1 | -0.12 | 0 | 0 | 0 | -2 | -0.98 |
| grep | | 9 | +3 | +4.55 | 3 | -3 | -0.10 | +1 | +0.24 | 2 | -6 | -0.73 |
| gzip | gzip1 | 3 | +1 | +1.11 | 1 | -1 | -0.35 | +1 | +1 | 1 | -1 | -1 |
| | gzip2 | 2 | +2 | +1.10 | 0 | 0 | 0 | +1 | +0.25 | 0 | -1 | -1.45 |
| | gzip5 | 2 | +2 | +0.25 | 0 | 0 | 0 | 0 | 0 | 0 | -2 | -0.75 |
| gzip | | 7 | +5 | +0.76 | 1 | -1 | -0.35 | +2 | +0.62 | 1 | -4 | -0.98 |
| sed | sed2 | 4 | +2 | +8.42 | 0 | -2 | -2.62 | 0 | 0 | 0 | -4 | -1.81 |
| | sed3 | 3 | +1 | +0.69 | 1 | -1 | -0.32 | 0 | 0 | 2 | -1 | -0.32 |
| | sed5 | 3 | +2 | +0.09 | 1 | 0 | 0 | +1 | +0.04 | 1 | -1 | -0.11 |
| | sed6 | 6 | +3 | +0.17 | 0 | -3 | -5.22 | 0 | 0 | 0 | -6 | -2.78 |
| | sed7 | 3 | +1 | +0.73 | 0 | -2 | -0.15 | 0 | 0 | 0 | -3 | -0.16 |
| sed | | 19 | +9 | +2.11 | 2 | -8 | -2.69 | +1 | +0.04 | 3 | -15 | -1.65 |

in the *grep* and *gzip* programs.

We investigated all the four versions of *totinfo* in which Tarantula outperforms BEN. In all cases the faulty statement localized by BEN is different from the one localized by Tarantula. The faulty statement detected by Tarantula is not even executed by the core member generated by BEN. Thus, it is not considered suspicious by BEN. The same situation happens for two out of three versions of the *tcas* program that Tarantula outperforms BEN (Table 28).

As we mentioned, BEN focuses on one inducing combination, which is likely due to one faulty statement. BEN stops searching for inducing combinations, as soon as the first one is identified, in the first phase. Therefore, BEN localize the faulty statement related to the identified inducing combination.

### 5.2.4. Threats to Validity

Threats to internal validity are factors that may be responsible for the experimental results, without our knowledge. One of the key steps in our experiments is modeling the input parameters, which may affect the correctness of the result. To reduce this threat, for three programs, *flex*, *gzip* and *sed*, we used the models from [39]. For the other programs, we have modeled the input parameters by using the program specifications and if they are not available, the error-free versions, without having any knowledge about the faults. All the models, except the *grep* model, have been used in other studies [18][15]. In [15], the models are used to compare the effectiveness of combinatorial testing and random testing.

In addition, we automated the experimental procedure as much as possible, as an effort to remove human errors. In particular, all the steps are automated except counting the number of statements that should be inspected to find the faulty statement. Further, consistency of the results has been carefully checked to detect potential mistakes made in the experiments. For example, the higher the average of inducing probability, the more likely the core member fails. In the extreme case, if the inducing probability is 1,

TABLE 28
COMPARISON RESULTS FOR MULTIPLE-FAULT VERSIONS BASED ON NUMBER OF OUTPERFORMED VERSIONS

| Program | | # of detected versions | Tarantula | | | | | Ochiai | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | BEN > Tarantula | | BEN = Tarantula | BEN < Tarantula | | BEN > Ochiai | | BEN = Ochiai | BEN < Ochiai | |
| | | | # of versions | Avg of Difference Percentages | | # of versions | Avg of Difference Percentages | # of versions | Avg of Difference Percentages | | # of versions | Avg of Difference Percentages |
| Siemens Suite | printtokens | 4 | 0 | 0 | 3 | -1 | -0.53 | 0 | 0 | 4 | 0 | 0 |
| | printtokens2 | 7 | +6 | +3.73 | 1 | 0 | 0 | +3 | +1.33 | 4 | -2 | -0.50 |
| | replace | 8 | 0 | 0 | 1 | -7 | -4.25 | +2 | +3.10 | 1 | -5 | -4.96 |
| | schedule | 5 | +1 | +2.60 | 1 | -3 | -5.41 | +4 | +2.27 | 0 | -1 | -9.74 |
| | schedule2 | 2 | +1 | +3.94 | 1 | 0 | 0 | 0 | 0 | 1 | -1 | -0.79 |
| | tcas | 9 | +1 | +4.62 | 5 | -3 | -4.62 | +2 | +2.31 | 4 | -3 | -4.10 |
| | totinfo | 9 | +2 | +2.85 | 3 | -4 | -11.18 | +4 | +10.30 | 3 | -3 | -1.36 |
| Siemens Suite | | 44 | +11 | +3.57 | 15 | -18 | -5.84 | +15 | +4.34 | 17 | -15 | -3.51 |
| flex | flex1 | 4 | +2 | +3.45 | 0 | -2 | -3.58 | +2 | +23.56 | 0 | -2 | -3.08 |
| | flex2 | 5 | +4 | +3.80 | 0 | -1 | -0.61 | +4 | +20.93 | 1 | 0 | 0 |
| | flex3 | 5 | 0 | 0 | 0 | -5 | -1.04 | 0 | 0 | 0 | -5 | -0.85 |
| | flex4 | 2 | +1 | +1.21 | 0 | -1 | -16.95 | 0 | 0 | 0 | -2 | -5.69 |
| flex | | 16 | +7 | +3.33 | 0 | -9 | -3.32 | +6 | +21.81 | 1 | -9 | -2.42 |
| grep | grep1 | 4 | +3 | +0.64 | 1 | 0 | 0 | +2 | +13.97 | 1 | -1 | -4.00 |
| | grep3 | 7 | +4 | +2.13 | 0 | -3 | -0.10 | +4 | +0.24 | 0 | -3 | -0.10 |
| | grep4 | 1 | 0 | 0 | 0 | -1 | -0.12 | 0 | 0 | 0 | -1 | -0.12 |
| grep | | 12 | +7 | +1.49 | 1 | -4 | -0.10 | +6 | +4.82 | 1 | -5 | -0.89 |
| gzip | gzip1 | 4 | +1 | +1.23 | 3 | 0 | 0 | +1 | +1 | 3 | 0 | 0 |
| | gzip2 | 1 | +1 | +0.5 | 0 | 0 | 0 | +1 | +0.25 | 0 | 0 | 0 |
| | gzip5 | 1 | +1 | +1.96 | 0 | 0 | 0 | +1 | +0.45 | 0 | 0 | 0 |
| gzip | | 6 | +3 | +1.23 | 3 | 0 | 0 | +3 | +0.57 | 3 | 0 | 0 |
| sed | sed2 | 3 | +2 | +5.13 | 0 | -1 | -3.27 | +2 | +2.33 | 0 | -1 | -1.77 |
| | sed3 | 4 | 0 | 0 | 2 | -2 | -0.41 | +1 | +0.05 | 1 | -2 | -0.41 |
| | sed5 | 4 | +1 | +0.13 | 3 | 0 | 0 | 0 | 0 | 1 | -3 | -0.06 |
| | sed6 | 3 | +2 | +0.18 | 0 | -1 | -0.22 | 0 | 0 | 0 | -3 | -0.47 |
| | sed7 | 4 | +1 | +0.73 | 0 | -3 | -0.26 | 0 | 0 | 0 | -4 | -0.33 |
| sed | | 18 | +6 | +1.91 | 5 | -7 | -0.72 | +3 | +1.57 | 2 | -13 | -0.42 |

the core member must fail. To check the consistency of the results, we checked the inducing probability whenever the core member did not fail. For instance, in one out of seven detected faulty versions of the *schedule* program, the core member did not fail. We checked the inducing probability for this version, which is relatively small, 0.25.

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from the *Siemens suite* [11]; these programs are created by a third party and have been used in other studies [23][40][31]. However, the subject programs are programs of relatively small size with seeded faults. To mitigate this threat, the *flex*, *grep*, *gzip*, and *sed* programs were added to the experiments, but more experiments on larger programs with real faults can further reduce this threat.

Each of the Siemens program has multiple versions, each of which has a single-fault. However, programs in practice could have multiple faults. To mitigate this threat, we created several multiple-fault versions that combined randomly selected faults and conducted an experiment on these versions. More experiments on programs with real faults can further reduce this threat.

## 6. RELATED WORK

In this section, we first discuss existing work on identifying failure-inducing combination, i.e., the first phase of BEN. Then, we focus on existing work on fault localization, which is related to the second phase of BEN.

### 6.1. Related Work on Identifying Inducing Combinations

Existing approaches to identifying inducing combinations can be classified into two groups. The first group takes as input a single failing test and tries to identify inducing combinations in the test.

Two techniques, called FIC and FIC_BS [57], try to identify all the inducing combinations contained in a failing test. These approaches take one failing test from a combinatorial test set, then generate and execute a small number of tests in a systematic manner to identify

inducing combinations in the failing test. New tests are generated such that one value, $v_i$, of the failing test is changed to another possible value. When the newly generated test passes, $v_i$ is part of an inducing combination because its removal makes the test pass. FIC generates k tests, where k is the number of parameters, for each failure inducing combination.

FIC_BS is the binary search version of FIC. To generate a new test, FIC_BS changes the values of $k/2$ parameters of the failing test. If the newly generated test passes, FIC_BS searches for inducing combinations in the changed values ($k/2$). The process continues until all inducing combinations are found. FIC and FIC_BS assume that no new inducing combinations are introduced when a value is changed to create a new test.

Li et al. [30] introduced two approaches, RI and SRI, for identifying inducing combinations. These techniques use a method called delta debugging [56] in an iterative framework. The RI approach takes one failing test from the initial combinatorial test set, and adopts a similar approach to FIC_BS to generate a small number of tests. The SRI approach is an improved version of RI, and it takes as input one failing test, f. Then it tries to generate a passing test similar to f. SRI uses the fact that the inducing combination appeared in the failing test f, but not in the similar passing test. Therefore, it focuses on the parameters, which are different in the failed and passing tests. SRI could identify inducing combination by generating fewer tests than RI.

The second group of approaches for identifying inducing combinations takes a set of tests as well as their execution statuses.

The AIFL and InterAIFL approaches in [45][49] first identify a set A of suspicious combinations as candidates for being inducing. Second, it generates a group of tests for each failing test using the SOFOT strategy [34]. Let k be the number of parameters. For each test f, the SOFOT strategy generates k tests by changing the value of one parameter at a time. Each test is different from the original test f in one value; the value is selected randomly from the corresponding parameter's domain. After executing the newly generated tests, combinations that appeared in the passing tests are removed from the suspicious set A. The InterAIFL approach improves AIFL by adopting a framework in which the suspicious set A is iteratively generated and refined until it becomes stable.

BEN also tries to identify inducing combinations in a combinatorial test set, instead of a single failing test. Thus, BEN belongs to the second group. There are two advantages resulting from using the whole test set rather than a single test. First, a test set contains more information than a single test. Second, it would be possible to identify inducing combinations that appear in different tests.

BEN identifies suspicious combinations in the same way as AIFL and Inter-AIFL. However, BEN produces a ranking of suspicious combinations and focuses on the most suspicious combinations. Moreover, BEN significantly differs from AIFL and Inter-AIFL in the way of generating new tests. BEN generates tests for a top-ranked suspicious combinations based on the notions of combination suspiciousness and environment suspiciousness. This is in contrast with the SOFOT strategy used in AIFL and Inter-AIFL.

We mention that Yilmaz et al. proposed a machine learning approach to identify failure-inducing combinations [54]. The approach analyzes the combinatorial test set and tests statuses and builds a classification tree. The classification tree is used to predict inducing combinations. Shakya et al. in [43] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

### 6.2. Related Work on Fault Localization

In Section 5, we already mentioned two fault localization approaches, Tarantula [23][24] and Ochiai [1]. Similar to BEN, Tarantula and Ochiai use statement coverage information to compute suspiciousness of each statement. Statement coverage is computed by multiple execution traces of failed and passing tests.

In Tarantula, the suspiciousness value of each statement is the ratio of failing tests that execute the statement divided by the ratio of failing tests that execute the statement plus the ratio of passing tests that execute the statement. Ochiai computes the suspiciousness value of each statement by dividing the number of failing tests that execute the statement by the square root of the product of the number of all failing tests and the number of all tests that execute the statement.

Then, Tarantula and Ochiai look for the faulty statement in a non-increasing order of their suspiciousness values.

Three spectrum-based approaches, set union, set intersection and nearest neighbor, are proposed by Renieris and Reiss [40]. These approaches assume that there is one failed run (the spectrum of a failing test) and a large number of passed runs (the spectra of passing tests).

Each of the three approaches has a different way to identify highly suspicious statements for being faulty, and these statements are then checked to find the actual faults. Let f be the program spectrum of a failing run and S be a set of program spectra of passed runs. The set union method computes $f - \cup_S s$, where $\cup_S s$ is the union spectra of a set of passed runs. The statements in the spectrum of the failed run but not in the union spectra of the passed runs are highly suspicious. In the intersection method, the highly suspicious statements are in the intersection spectra of a set of passed runs but not in the spectrum of the failed run, $\cap_S s - f$.

In the nearest neighbor approach, one passed run whose spectrum is the most similar to the failed spectrum is selected from $S$. The statements in the difference set of these two spectra have the highest suspiciousness of being faulty.

If the fault is not found in the highly suspicious statement set, the program dependence graph is built. The nodes corresponding to the highly suspicious statements are marked as blamed nodes. Then, in both directions, backward and forward, a breadth-first search is performed from the blamed nodes. The statements corresponding to the nodes at a distance of one are also suspicious and must

be checked. This process is repeated until the faulty statement is found.

Empirical evaluation in [23] shows that for the *Siemens suite*, Tarantula is more effective and efficient than the other methods, including set union, set intersection, and nearest neighbor. Lucia et al. in [31] reported the experiments that show Tarantula and Ochiai are comparable to each other for the Siemens programs. However, the work reported in [1] suggests that Ochiai outperforms Tarantula. The former work used statement coverage spectra while the latter used branch coverage spectra. Both works, i.e., [1] and [31], applied fault localization methods using the test pools provided for each program by the benchmark [11].

Our experimental results also show that Ochiai is slightly better than Tarantula. BEN used combinatorial test sets and statement coverage spectra.

The fundamental difference between BEN and the above spectrum-based approaches is that BEN systematically generates a small group of tests, and then analyzes their spectra to produce a ranking of statements. The existing approaches do not deal with test generation. Instead, they assume an existing test set that is generated randomly or using other techniques. In addition, they require every test execution to be traced. As a result, they cannot utilize the testing results if the test executions were not traced. In contrast, our approach is designed to work after normal testing is performed where test executions are not traced. Our approach only needs to trace the execution of a small number of tests that are generated in the second phase of our approach. As shown in Section 5, our approach can significantly reduce the number of tests that need to be traced but still produce results that are competitive to or better than Tarantula and Ochiai.

One approach, called LCEC [32], was reported that also leverages the result of combinatorial testing to localize the faulty statement. LCEC was published after our original work in [18][15]. LCEC selects a failing test from the initial combinatorial test set, and generates a group of passing tests by changing values of failing test involved in the inducing combination. The execution traces of failed and passing tests are analyzed to derive cause-effect chains of statements. A depth-first search is performed for all cause-effect chains to locate the faulty statement. Then, if the faulty statement is not found, a breath-first search is performed in the dynamic backward slice, associated with the incorrect output value. LCEC is applied to four small programs, with a maximum of 220 lines of code, including the *tcas* program. The cost of applying LCEC is not reported in [32]. The LCEC tool is not publicly available.

We mention several other publications in fault localization literature. Roßler et al. [41] propose a technique, BUGEX, which adopts a dynamic symbolic execution approach to generate tests with a minimal difference from a single failing test in terms of facts, i.e., branches or state predicates. Based on the generated tests, the facts that are executed by more failing tests but fewer passing tests are more likely to cause the failure. The proposed approach is different from BEN, as BEN does not analyze code to generate tests. Instead, BEN generates tests in a black box manner and uses an input model to find values for each parameter.

In [52], Xuan and Monperrus proposed an approach to purify test cases that are written as xUnit-style test methods. A test method typically contains one or more assertions that are used to evaluate each test run. In their approach, when a test method containing multiple assertions fails, it generates purified test methods of this failing test method such that each of these purified methods only contains one assertion and relevant statements, i.e., statements that may affect this assertion. These purified test methods are executed to refine the statement ranking generated by spectrum based fault localization approaches. Their work is complementary to ours in that additional tests generated by our approach can be written as xUnit-style test methods to which their approach can be applied. Note that while both approaches generate additional tests for fault localization, these tests are generated in very different ways. Specifically, the approach in [52] generates additional tests based on assertions, whereas our approach generates additional tests based on inducing combinations.

Metallaxis [37], is a fault localization approach based on mutation analysis. The basic idea is the following: The more failing tests that kill a mutant, the more likely the statement that is changed by the mutant is faulty. In Metallaxis, a suspiciousness value is computed for each mutant, and thus for its corresponding statement, using the same Tarantula formula, except that the number of failing (passing) tests that kill the mutant is used, in place of the number of failing (passing) tests that execute the statement, in the formula.

Baah et al. [3] presented a PPDG, a probabilistic graphical model based on the program dependence graph to capture the statistical dependences among program elements. The PPDG could be used to analyze the program behavior and then generate a ranking of statements for fault localization.

Le et al. [28] proposed a multi-modal technique called AML, that considers bug reports and program spectra to locate bugs. AML uses Vector Space Model [28] and Tarantula as the information retrieval and spectrum-based technique, respectively. Then, it integrates their output and produces the final ranking.

BEN is different from the above approaches as it is a spectrum-based fault localization approach based on combinatorial testing. BEN does not perform any program dependency analysis.

In [55] a new approach to prioritize tests for efficient fault localization is proposed. It used Tarantula as a fault localization approach. After finding the first failing test, it prioritizes tests such that tests that could potentially produce greater suspiciousness for the faulty statement are executed first. Moreover, Xia et al. [51] proposed a test case selection strategy to maximize the effectiveness of the Ochiai approach, while minimizing the cost of test oracle construction. These approaches are complementary to BEN in that they could be used to further improve the effectiveness of BEN.

## 7. Conclusion

In this paper, we presented an approach called BEN to localizing faults that leverages the result of combinatorial testing. Our approach combines black-box combinatorial test generation with white-box spectrum analysis for fault localization. Our approach consists of two phases. The first phase identifies a failure-inducing combination, which is used in the second phase to localize the faulty statement in the source code.

In the first phase, BEN adopts an iterative framework that ranks suspicious combinations and generates new tests repeatedly until an inducing combination is identified. The novelty of this phase lies in the fact that we rank suspicious combinations and generate new tests based on the notions of combination suspiciousness and environment suspiciousness. The higher the combination suspiciousness of a combination, the lower its environment suspiciousness, the higher this combination is ranked. New tests are generated for a user-specified number of top-ranked suspicious combinations such that the environment suspiciousness of a combination is minimized in each test. Our approach starts with searching for inducing combinations whose size is equal to the strength t of the initial test set. If it is not found, the approach expands its search to combinations whose size is greater than t.

The key idea of the second phase of BEN is that we systematically generate a group of tests from an inducing combination such that the spectra of these tests can be analyzed quickly to identify the faulty statement. This group of tests consists of a core member that is a failing test and a number of derived members that are passing tests but are very similar to the core member. The suspiciousness values of statements are computed by analyzing the spectra of the core member and the derived members.

We applied BEN to the Siemens suite and also the flex, grep, gzip and sed programs. Our experimental results show that our approach requires a very small number of tests to be generated while significantly reducing the number of statements to be inspected for fault localization. In particular, our approach achieves results that are competitive to or better than those of Tarantula [24] and Ochiai [1] while requiring significantly fewer test runs to be traced.

We emphasize that our approach has an important advantage over existing spectrum-based approaches such as Tarantula and Ochiai. Existing spectrum-based approaches require every test execution be traced. If a test set is already executed without being traced, the test set must be re-executed to collect traces before they can be used by approaches like Tarantula and Ochiai. In contrast, our approach only requires a small number of tests generated in the second phase of our approach to be traced. Our approach is designed to work after normal testing is performed where test executions do not need to be traced.

We plan to conduct more empirical studies to further evaluate the performance of our approach. In particular, we plan to evaluate our approach using other metrics such as acc@N and also compare our approach to information retrieval [28][42][58] or learning-to-rank based approaches [27][47][53]. We also plan to investigate how to adapt our approach to work with an arbitrary test set. Our current approach assumes that a combinatorial test set is used to test a program. This will further increase the applicability of our approach. That is, we will try to identify inducing combinations from an arbitrary test set and then use them to generate tests for fault localization. The challenge is to deal with the fact that unlike a combinatorial test set, an arbitrary test set does not guarantee that all t-way combinations are covered. This might reduce the effectiveness of our approach.

## 8. Acknowledgment

This work is partly supported by three grants (70NANB12H175, 70NANB10H168, and 70NANB15H199) from Information Technology Lab of National Institute of Standards and Technology (NIST).

*Disclaimer*: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

## 9. References

[1] R. Abreu, P. Zoeteweij, and A. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization", In Proceedings of 12th Pacific Rim International Symposium on Dependable Computing, pp.39,46, Dec. 2006.

[2] Advanced Combinatorial Testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html, 2015.

[3] G. K. Baah, A. Podgurski and M. J. Harrold, "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis," in IEEE Transactions on Software Engineering, 36(4): 528-545, 2010.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in IEEE Transactions on Software Engineering, 41(5), pp. 507-525, May 1 2015.

[5] BEN: a combinatorial tesing-based fault localization tool, http://barbie.uta.edu/~laleh/BEN.html, 2015.

[6] M. N. Borazjany, Y. Linbin, Y. Lei, R. Kacker, and D. R. Kuhn, "T-way testing of ACTS: A Case Study", In Proceedings of the IEEE fifth International Conference on Software Testing, Verification and Validation, pp.591-600, 2012.

[7] The Choco Constraint Solver, http://www.emn.fr/z-info/choco-solver/index.html

[8] D. Cohen, S. Dalal, M. Fredman, and G. Patton. "The AETG system: An approach to testing based on combinatorial design", In IEEE Transactions on Software Engineering, 23(7):437–444, 1997.

[9] M. B. Cohen, P. B. Gibbons, W.B. Mugridge, C.J. Colbourn. "Constructing test suites for interaction testing", In Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), pages 38-48, 2003.

[10] D. Coppit and J. M. Haddox-Schatz, "On the use of specification

based assertions as test oracles," In Proceedings of Annual IEEE/NASA Software Engineering Workshop, pp. 305–314, 2005.

[11] H. Do, S. Elbaum, and G. Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact", In Empirical Software Engineering. 10(4):405-435, 2005.

[12] Empirical study on combinatorial testing, http://barbie.uta.edu/~laleh/research.html, 2015.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," In Proceedings of Science of Computer Programming, vol. 69, no. 1, pp. 35–45, 2007.

[14] GCC online documentation, https://gcc.gnu.org/onlinedocs, 2015.

[15] L. Ghandehari, M. N. Borazjany, Yu Lei, Raghu Kacker, Richard Kuhn, "Applying Combinatorial Testing to the Siemens Suite", In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICSTW), 2013.

[16] L. Ghandehari, J. Czerwonka, Y. Lei; S. Shafiee, R. Kacker, R. Kuhn, "An Empirical Comparison of Combinatorial and Random Testing," In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp.68-77, 2014.

[17] L. Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, Richard Kuhn, "BEN: A Combinatorial Testing-Based Fault Localization Tool", In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICSTW), Graz, Austria, April, 2015.

[18] L. Ghandehari, Y. Lei, D. Kung, R. Kacker, R, Kuhn. "Fault localization based on failure-inducing combinations," In Proceeding of the IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), 168-177, 2013.

[19] L. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. "Identifying Failure-Inducing Combinations in a Combinatorial Test Set", In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST), 370-379, 2012.

[20] GNU Grep, http://www.gnu.org/software/grep/manual/grep.html, 2015.

[21] GNU Gzip, http://www.gnu.org/software/gzip/manual/gzip.html, 2015.

[22] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization", In Proceedings of the 38th International Conference on Software Engineering ACM, pp. 523-534, 2016.

[23] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique", In Proceeding IEEE/ACM Automated software engineering, 2005, 273-282.

[24] J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization", In Proceedings of International Conference on Software Engineering, 2002, 467-477.

[25] D. R. Kuhn and V. Okum. "Pseudo-Exhaustive Testing for Software", In Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW '06). IEEE Computer Society, 2006, 153-158.

[26] D.R. Kuhn, D.R. Wallace, A.M. Gallo. "Software fault interactions and implications for software testing", In Proceedings of the IEEE Transaction on Software Engineering, 2004, 30(6), 418–421, 2004.

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

[28] T.D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," In Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering, pp: 579–590, 2015.

[29] Y. Lei, R. Kacker, D. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOD: Efficient test generation for multi-way software testing", In Journal of Software Testing, Verification, and Reliability, 18(3):125-148, Sept. 2008.

[30] J. Li; C. Nie, and Y. Lei, "Improved Delta Debugging Based on Combinatorial Testing," In Proceedings of International Conference on Quality Software (QSIC), pp.102,105, 2012.

[31] Lucia, D. Lo, L. Jiang, A. Budi, "Comprehensive evaluation of association measures for fault localization", In Proceedings of the IEEE International Conference on Software Maintenance, 1-10, 2010.

[32] C. Ma, Y. Zhang, J. Liu, and M. Zhao, "Locating Faulty Code Using Failure-Causing Input Combinations in Combinatorial Testing," In Proceedings of 4th World Congress on of Software Engineering (WCSE), pp.91,98, 2013.

[33] C. Nie and H. Leung. "A survey of combinatorial testing", In ACM Computing Surveys (CSUR), 43(2):11: 1-11: 29, January 2011.

[34] C. Nie, H. Leung, and B. Xu. "The minimal failure-causing schema of combinatorial testing", In ACM Transactions on Software Engineering and Methodology, 20(4) , September 2011.

[35] B. Ozcelik and C. Yilmaz, "Seer: A Lightweight Online Failure Prediction Approach," In IEEE Transactions on Software Engineering, 42(1), pp. 26-46, Jan. 1 2016.

[36] A. Panichella, R. Oliveto, M. D. Penta and A. De Lucia, "Improving Multi-Objective Test Case Selection by Injecting Diversity in Genetic Algorithms," In IEEE Transactions on Software Engineering, 41(4), pp. 358-383, April 1 2015.

[37] M. Papadakis and Y. L. Traon, "Metallaxis-FL: mutation-based fault localization", In proceeding of Software Testing, Verification and Reliability, 25(5-7): 605-628, 2015.

[38] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," In Proceedings of Foundations of Software Engineering, pages 26–36, 2013.

[39] J. Petke, M. B. Cohen, M. Harman and S. Yoo, "Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection," In IEEE Transactions on Software Engineering, 41( 9): 901-924, 2015.

[40] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries", In Proceedings of the International Conference on Automated Software Engineering, 2003.

[41] J. Röβler, G. Fraser, A. Zeller, and A. Orso. "Isolating Failure Causes Through Test Case Generation", In Proceeding of the International Symposium on Software Testing and Analysis, pp. 309–319, 2012.

[42] R. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. Proceedings of IEEE/ACM 28th International Conference on Automated

Software Engineering (ASE), pp. 345–355, 2013.

[43] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn, "Isolating Failure-Inducing Combinations in Combinatorial Testing Using Test Augmentation and Classification," In proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), pp.620-623, 2012.

[44] P.J. Schroeder, P. Bolaki, V. Gopu, "Comparing the fault detection effectiveness of n-way and random test suites," In Proceeding of the International Symposium on Empirical Software Engineering, pp.49-59, 2004.

[45] L. Shi, C. Nie, B. Xu. "A software debugging method based on pairwise testing", In Proceedings of the International Conference on Computational Science (ICCS2005), pages 1088-1091, 2005.

[46] Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php, 2012.

[47] J. Sohn and S. Yoo, "FLUCCS: using code and change metrics to improve fault localization". In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 273-283, 2017.

[48] D. R. Wallace, D. R. Kuhn, "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data", In Proceeding of the ACS/ IEEE International Conference on Computer Systems and Applications, pp. 301-311, 2001.

[49] Z. Wang, B. Xu, L. Chen, and L. Xu. "Adaptive interaction fault location based on combinatorial testing", In Proceedings of the 10th International Conference on Quality Software (QSIC 2010), pages 495–502, 2010.

[50] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. IEEE Transactions on Software Engineering, 42(8):707–740, 2016.

[51] X. Xia, L. Gong, T. D. B. Le, D. Lo, L. Jiang, and H. Zhang: "Diversity maximization speedup for localizing faults in single-fault and multi-fault programs", In proceedings of Automated Software Engineering, 23(1): 43-75, 2016.

[52] J. Xuan and M. Monperrus. "Test case purification for improving fault localization", In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 52-63, 2014.

[53] J. Xuan and M. Monperrus. "Learning to Combine Multiple Ranking Metrics for Fault Localization", In proceedings of the IEEE International Conference on Software Maintenance and Evolution, pages 191-200, 2014.

[54] C. Yilmaz, M. B. Cohen, A. A. Porter. "Covering arrays for efficient fault characterization in complex configuration spaces", In Proceedings of the IEEE Transaction on Software Engineering, 2006, 32(1): 20-34.

[55] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches". ACM Transactions on Software Engineering and Methodology, 29 pages, 2013.

[56] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input", In Proceedings of the IEEE Transactions on Software Engineering, 2002, pages 183–200.

[57] Z. Zhang, and J. Zhang. "Characterizing failure-causing parameter interactions by adaptive testing", In Proceedings of ACM International Symposium on Software Testing and Analysis (ISSTA 2011), pp. 331-341, 2011.

[58] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. Proceedings of the 34th International Conference on Software Engineering, pp. 14–24, 2012.

[59] H. Zhu, "A note on test oracles and semantics of algebraic specifications," In Proeedings of Conference on Quality Software, pp. 91–98, 2003.

**Laleh Sh. Ghandehari** received the PhD degree in computer science from University of Texas at Arlington in 2016. She is an Adjunct Professor in computer science and engineering department of University of Texas at Arlington. Her research interests includes software testing, fault localization, automatic debugging and program analysis.

**Yu Lei** is a Professor in Department of Computer Science and Engineering at the University of Texas at Arlington. His research interests are in the area of software analysis, testing and verification, with a special focus on combinatorial testing. He was a Member of Technical Staff in Fujitsu Network Communications, Inc. for about three years. He received his PhD degree in Computer Science from North Carolina State University.

**Raghu Kacker** is a mathematical statistician in the National Institute of Standards and Technology (NIST). His current interests include development and use of combinatorial methods for testing software and systems. He has co-authored over 140 refereed publications and one book. He has a Ph.D. and has worked in academia (Virginia Tech), and industrial (AT&T Bell Laboratories) and government (NIST) research laboratories. He is a Fellow of the American Statistical Association and a Fellow of the American Society for Quality. He has received the Distinguished Technical Staff Award from AT&T Bell Labs, and Bronze medal and Silver medal from the US Department of Commerce.

**Rick Kuhn** is a computer scientist in the Computer Security Division of the National Institute of Standards and Technology and is a Fellow of the IEEE. He has co-authored three books and more than 150 papers on information security, empirical studies of software failure, and combinatorial methods in software testing, and co-developed the role based access control model used worldwide. His awards include the IEEE Innovation in Societal Infrastructure Award and Gold medal for scientific/technical achievement from the US Department of Commerce. Before joining NIST, he worked as a software developer with NCR Corporation and the Johns Hopkins University Applied Physics Laboratory. He received an MS in computer science from the University of Maryland College Park.

**Tao Xie** is a full professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at UrbanaChampaign, USA. His research interests are software testing, software analytics, software security, and intelligent software engineering. He is a Fellow of the IEEE.

**Dr. David Kung** is a professor of Department of Computer Science and Engineering and director of Software Engineering Research Center at The University of Texas at Arlington. He received his BS in mathematics from Beijing University, and MS and Ph.D. degrees in computer science from The Norwegian University of Science and Technology. He has more than 40 years of combined research, education and industry experiences, and collaborates extensively with US industry. He has published four books titled "Object-Oriented Software Engineering: An Agile Unified Methodology," "Testing Object-Oriented Software," "Introduction to Information Systems Engineering" and " The Role of AI in Databases and Information Systems." He has also published more than 100 technical articles in journals and conference proceedings.