The Journal of Systems and Software 90 (2014) 3–17

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices

Xiaolin Ju[a,b], Shujuan Jiang[a,*], Xiang Chen[b,c], Xingya Wang[a], Yanmei Zhang[a], Heling Cao[a]

[a] School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, China
[b] School of Computer Science and Technology, Nantong University, Nantong, China
[c] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 30 January 2013
Received in revised form 29 October 2013
Accepted 21 November 2013
Available online 9 December 2013

Keywords:
Dynamic slicing
Execute slicing
Fault localization

ABSTRACT

Most of the existing fault localization approaches use execution coverage of test cases to isolate the suspicious codes that likely contain faults. Program slicing can extract the dependencies of program entities with respect to a specific criterion. Therefore this technique is expected to have a beneficial effect on fault localization. In this paper, we propose a novel approach using a hybrid spectrum of full slices and execution slices to improve the effectiveness of fault localization. In particular, our approach firstly computes full slices of failed test cases and execution slices of passed test cases respectively. Secondly it constructs the hybrid spectrum by intersecting full slices and execution slices. Finally it computes the suspiciousness of each statement in the hybrid slice spectrum and generates a fault location report with descending suspiciousness of each statement. We also implement our proposed approach in our prototype tool HSFal by Java programming language. To verify the effectiveness of our approach, we performed an empirical study by the prototype on several widely used open source programs. Our approach is compared with eight representative coverage-based and slice-based fault localization approaches. Final experimental results show that our proposed approach is more effective in fault localization than other compared approaches, and can reduce almost 2.98–31.79% of the average cost of examined code significantly.

1. Introduction

Locating faults in a program is one of the most tedious and time-consuming tasks in program debugging (DeMillo et al., 1997; Renieris and Reiss, 2003; Wong and Qi, 2004; Jones and Harrold, 2005). It usually aims to narrow the search domain of the bugs to improve its efficiency, and therefore decreases debugging cost. One intuitive way of fault localization is to find the statements which cause abnormal output directly or indirectly. Nowadays a more popular way is to create a ranking list according to the likelihood of containing a bug for entities (e.g., statements, predicates, or basic blocks). This way can guide the developer to search for bugs along the ranking list (Renieris and Reiss, 2003; Wong and Qi, 2004; Jones and Harrold, 2005; Wong et al., 2012; Jones et al., 2002).

In the last decade, researchers have proposed many fault localization approaches. Most of these approaches are coverage-based, and they compute the suspiciousness of each statement using the coverage of program executions (Jones and Harrold, 2005; Wong et al., 2012; Jones et al., 2002; Santelices et al., 2009). The assumption of these approaches is that program executing coverage could approximate fault causality. This assumption is generally made to facilitate software testing automation. In practice, the coverage information of test cases can be easily obtained and stored by code instrumentation. In addition, experimental evaluation indicates that coverage-based fault localization approaches can be more effective and only need to examine less than 20% of program codes (Ali et al., 2009). However, the effectiveness and efficiency of fault localization would decrease with the increase of faults, and the root cause of this issue is the cumulative impact on the same statement by the different faults.

Program slicing, which can extract the data and/or control dependencies of program entities, has been used to improve the effectiveness of fault localization (Lyle, 1984). Existing program slicing techniques can be classified into two categories: the static slicing and the dynamic slicing. The static slicing analyzes the dependencies without running the program while the dynamic slicing obtains the dependencies along the execution path. Although both the static and dynamic slicing can be used in fault localization, the former has a greater time complexity and may produce false positives, while the latter has a higher space complexity and may generate false negatives (Gyimóthy et al., 1999; Agrawal and Horgan, 1990; Agrawal et al., 1995; Al-Khanjari et al., 2005; Zhang

* Corresponding author.
E-mail addresses: shjjiang@cumt.edu.cn, Ju.xl@ntu.edu.cn (S. Jiang).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.11.1109

et al., 2004). Besides, most program slicing techniques (e.g., Agrawal and Horgan, 1990) tend to suffer from the high cost of creating a dependence graph when using backward traversal algorithms. However, we construct slices by using a forward computation algorithm which does not need to construct the dynamic dependence graph. Therefore, the cost of slicing would be reduced. In addition, we conjecture that ranking the statements of a slice might be helpful to provide guidance for the developer and further improve the fault localization effectiveness.

In this paper, we propose a novel fault localization approach HSS (hybrid slice spectrum), based on a combination of full slices and execution slices. The intuition of our approach is that only the faulty program entities on which the output dynamically depends can trigger the failure. Thus, the idea of our approach is to exclude the coverage of those program entities whose execution does not return the wrong output by dynamic slicing. So the coverage slices of our approach might be smaller than that of coverage-based approaches because of the removal of the fault-irrelevant statements by program slicing with slice criteria of any known faulty statements. Therefore, HSS might have a better performance in suspiciousness computing for fault localization. We also propose a suspiciousness evaluation formula, which is theoretically proved to be maximal in performance of fault localization, for ranking the statements of HSS to provide a guidance for the developer. Firstly, we compute full slices with respect to abnormal output. Secondly, we trace and record the execution slices of successful execution. Thirdly, we intersect the union of full slices with execution slices, and then combine with full slices to form a hybrid slice spectrum. Finally, we compute and rank the suspiciousness of each statement in the intersection of full slices and execution slices in descending order. Therefore, programmers can examine the code along with the ranking list. We also developed a prototype tool HSFal (hybrid slice spectrum fault locator) to incorporate our proposed approach. To demonstrate the effectiveness of our proposed approach, we designed and performed an empirical study on 14 open source programs which are widely used by other researchers in their empirical studies. The results from our study suggest that our approach can reduce almost 2.98% up to 31.79% of the average cost of examined code.

The main contributions of this paper can be summarized as:

- A novel fault localization approach (HSS) based on compounding full slices and execution slices.
- A maximal risk evaluation formula which is theoretically proved, for calculating suspiciousness based on a hybrid spectrum of full slices and execution slices.
- Effectiveness and efficiency of HSS are evaluated across various open source programs, the LOC of these programs ranges from 181 to 22,318, compared approaches include most of the representative coverage-based and slice-based fault localization techniques. The experimental results show that HSS is more effective in fault location than the compared techniques.

The rest of this paper is organized as follows. Section 2 provides the background of fault localization and program slicing. Section 3 firstly gives a motivating example to show the intuition of our approach, secondly presents the framework and the implementation of our approach. Section 4 describes our empirical study including experiment subjects, experiment design, data analysis, and threats to validity. Section 5 summarizes the related work of fault localization and emphasizes the contribution of our research. Finally, Section 6 provides the conclusion and future work.

$$T_{cover} = \begin{array}{c} \begin{matrix} s_1 & s_2 & \cdots & s_n \end{matrix} \\ \left[\begin{array}{cccc} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{k1} & c_{k2} & \cdots & c_{kn} \\ \hdashline c_{l1} & c_{l2} & \cdots & c_{ln} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{array}\right] \end{array} \begin{array}{c} t_{p1} \\ t_{p2} \\ \cdots \\ t_{pk} \\ \hdashline t_{fl} \\ \cdots \\ t_{fm} \end{array}$$

Fig. 1. Coverage of m executions of program.

Table 1
Notations widely used in suspiciousness calculating.

Notation	Value	Description
N	m	The number of test cases
N_F	$m-k$	The number of failed test cases
N_S	k	The number of passed test cases
N_{CF}	$\sum_{i=l}^{m} C_{ij}$	The number of failed test cases that can cover the statement s_j
N_{CS}	$\sum_{i=1}^{k} C_{ij}$	The number of passed test cases that can cover the statement s_j
N_C	$N_{CF}+N_{CS}$	The number of test cases that can cover the statement s_j
N_{UF}	$\sum_{i=l}^{m}(1-C_{ij})$	The number of failed test cases that cannot cover the statement s_j
N_{US}	$\sum_{i=1}^{k}(1-C_{ij})$	The number of passed test cases that cannot cover the statement s_j
N_U	$N_{UF}+N_{US}$	The number of test cases that cannot cover the statement s_j

2. Background

2.1. Fault localization

Fault location aims to locate the wrong instruction, process, or data definition in programs. Existing fault location approaches can be divided into two categories: static analysis based approaches and dynamic testing based approaches. The locating granularity can be a statement, basic block, function, or class. In this paper, we mainly focus on dynamic testing based fault location approaches and the locating granularity is statement.

Suppose we have a program $P=\{s_1, s_2, \ldots, s_n\}$, and this program consists of n executable statements. Though we consider program elements to be statements in this paper. Without loss of generality, the program component can be also set to be predicates, basic blocks or functions. At the same time, we suppose corresponding test suite $T=\{t_1, t_2, \ldots, t_m\}$ that consists of m distinct test cases. After instrumenting the program under test and executing each of these test cases or just monitoring the running of program through debugging interface, we can gather the execution traces and the running result (i.e., passed or failed) conveniently.

According to the running result, we further divide T into T_p and T_f. All the test cases in $T_p=\{t_{p1}, t_{p2}, \ldots, t_{pk}\}$ lead to successful executions, and all the test cases in $T_f=\{t_{fl}, t_{f(l+1)}, \ldots, t_{fm}\}$ lead to failed executions (here $l=k+1$). In Fig. 1, we use T_{cover} to indicate the coverage of statements collected after running m test cases. The value of c_{ij} is 1 if the statement s_j is covered by the test case t_i. Otherwise the value of c_{ij} is 0.

We summarize nine notations commonly used in suspiciousness computing as shown in Table 1. The value of all these notations

Table 2
Five maximal suspiciousness evaluation formulas.

Group	Name	Formula description
ER1	Naish1	$Naish1(s) = \begin{cases} -1, & \text{if } N_{CF}(s) < N_F \\ N_S - N_{CS}(s), & \text{if } N_{CF}(s) = N_F \end{cases}$ (1)
	Naish2	$Naish2(s) = N_{CF}(s) - \frac{N_{CS}(s)}{N_{CS}(s) + N_{US}(s) + 1}$ (2)
	Wong1	$Wong1(s) = N_{CF}(s)$ (3)
ER2	Russel&Rao	$Russel\,\&\,Rao(s) = \frac{N_{CF}(s)}{N_{CF}(s) + N_{UF}(s) + N_{CS}(s) + N_{US}(s)}$ (4)
	Binary	$Binary(s) = \begin{cases} 0, & \text{if } N_{CF}(s) < N_F \\ 1, & \text{if } N_{CF}(s) = N_F \end{cases}$ (5)

can be obtained from the matrix shown in Fig. 1. Moreover, N_{CF}, N_{CS}, N_{UF} and N_{US} are the most essential factors among these nine factors due to all the rest factors can be obtained by them. For example, $N_C = N_{CF} + N_{CS}$, $N_U = N_{UF} + N_{US}$, $N_F = N_{CF} + N_{UF}$, $N_S = N_{CS} + N_{US}$, $N = N_C + N_U = N_S + N_F$.

In this paper, we define the suspiciousness of the statement s_i as the likelihood of containing a fault. Most of the existing coverage-based fault localization techniques calculate suspiciousness of each statement based on the matrix T_{cover} in Fig. 1 (e.g., Renieris and Reiss, 2003; Wong et al., 2012, 2008; Jones et al., 2002; Agrawal et al., 1995; Weiglhofer et al., 2009; Abreu et al., 2009; Jones, 2004). After analyzing previous research work, we propose four assumptions for our work as follows:

- The more a statement is executed by failed test cases, the more suspicious it should be, therefore, the greater suspiciousness it should be assigned to. That is to say, the suspiciousness of a statement should be directly proportional to the number of failed executions that can cover it.
- The more a statement is executed by passed test cases, the less suspicious it should be, therefore, the lower suspiciousness it should be assigned to. That is to say, the suspiciousness of a statement should be inversely proportional to the number of passed executions that can cover it.
- The more a statement is not executed by failed test cases, the less suspicious it should be, therefore, the lower suspiciousness it should be assigned to. That is to say, the suspiciousness of a statement should be inversely proportional to the number of failed executions that cannot cover it.
- The more a statement is not executed by passed test cases, the more suspicious it should be, therefore, the greater suspiciousness it should be assigned to. That is to say, the suspiciousness of a statement should be directly proportional to the number of passed executions that cannot cover it.

Based on the above assumptions, most coverage-based fault localization techniques calculate suspiciousness by the notations listed in Table 1. Researchers usually performed empirical investigations on their proposed approaches (Jones and Harrold, 2005; Abreu et al., 2006; Wong et al., 2007, 2012) have further compared 12 similarity coefficient based techniques. However, Xie et al. (2013) conducted a theoretical analysis of 30 risk evaluation formulas for coverage-based fault localization based on four presupposed assumptions recently. Among all the 30 investigated formulas, they have proved two groups of maximal formulas listed in Table 2, and they also indicated that the formulas from the two groups are not equivalent in fault locating performance.

2.2. Program slicing

Program slicing, firstly proposed by Weiser, can select all the statements that can affect the value of a variable in a statement directly or indirectly (Weiser, 1982). The set of selected statements is called a slice of the program with respect to the variable. Program slicing, which indicates the dependencies among program elements, has been widely used in program analysis and software testing (Gyimóthy et al., 1999; Agrawal and Horgan, 1990; Sun et al., 2007). In this section, we give some related definitions of program slicing as follows.

Definition 1 *(A Ref set).* A Ref set of the statement s is a set of variables that are used in statement s. We use notation $Ref(s)$ to denote the Ref set of statement s.

Definition 2 *(A Def set).* A Def set of the statement s is a set of variables that its value is changed in s. We use notation $Def(s)$ to denote the Def set of statement s.

Definition 3 *(Data dependency).* A data dependency between two statements s_2 and s_1 satisfies three conditions (1) s_1 is executed before s_2, (2) the variable $v \in Ref(s_2)$, $v \in Def(s_1)$, and (3) there are no other statements between s_1 to s_2 on the execution trace which changes the value of v. Notation $s_2 \xrightarrow[v]{DD} s_1$ means s_2 data depends on s_1 by the variable v.

Let $\{s' | s \xrightarrow[v]{DD} s' \wedge v \in Ref(s)\}$ to be a set of statements on which the statement s data depends.

Definition 4 *(Control dependency).* A control dependency between two statements s_2 and s_1 satisfies two conditions (1) s_1 is executed before s_2 and (2) the s_2's execution is conditionally guarded by the former executed statement s_1. Notation $s_2 \xrightarrow{DD} s_1$ means s_2 control depends on s_1.

Let $\{s' | s \xrightarrow{CD} s'\}$ to be a set of statements on which the statement s control depends.

Definition 5 *(Full slice).* A full slice (FS)[1] is the transitive closure of statements that directly or indirectly influence the slice criteria statement through chains of dynamic data and control dependencies (Zhang et al., 2005).

Let $D^0(s) = \{s\}$, $D^1(s) = \{s'' | s'' \in DD(s') \cup CD(s') \wedge s' \in D^0(s)\}$, $D^2(s) = \{s'' | s'' \in DD(s') \cup CD(s') \wedge s' \in D^1(s)\}$, ..., $D^n(s) = \{s'' | s'' \in DD(s') \cup CD(s') \wedge s' \in D^{n-1}(s)\}$. Furthermore, let $D^*(s) = D^1(s) \cup D^2(s) \cup \ldots$ to be the transitive closure of $D(s)$, then the full slice of statement s can be denoted as $FS(s) = \{s' | s' \in D^*(s)\}$.

Definition 6 *(Execution slice).* An execution slice is a set of executed statements after test case execution.

Let $ES(t) = \{s | s \in P \wedge s$ is covered by $t\}$ to be a set of statements covered by the test case t.

By program slicing, all of the statements can be partitioned into the correct statement set and suspicious statement set. Although

[1] In this paper, we used terms "full slice" and "dynamic slice" interchangeably.

```
   foo ( ) {
      int x, y, z, m, ret ;
s1    read (x, y, z);
s2    x = x/y;  // correct: x = x * y
s3    m = x + y;
s4    if (x > 1) {
s5        m = x − 2;
s6        x = x * y;
s7        z = 2 * y;
      }
s8    if (m > 0) {
s9        ret = x − z;
      } else {
s10       ret = y − z;
      }
s11   return ret;
   }
```

	coverage of 8 tests							
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
s_1	●	●	●	●	●	●	●	●
s_2	●	●	●	●	●	●	●	●
s_3	●	●	●	●	●	●	●	●
s_4	●	●	●	●	●	●	●	●
s_5		●	●	●	●	●	●	●
s_6		●	●	●			●	●
s_7		●	●	●			●	●
s_8	●	●	●	●	●	●	●	●
s_9	●		●		●		●	●
s_{10}		●		●		●		
s_{11}	●	●	●	●	●	●	●	●
results	F	F	P	P	P	P	F	P

Fig. 2. A motivating example.

we can simply regard both full slice and execution slice of failed running as the suspicious statement set, the scale of these slices usually could be too large to guide the programmers in locating faults effectively in real world software debugging. Therefore, most of the slice-based fault localization techniques locate faults using different slices (e.g., dicing (Chen and Cheung, 1997), execution slice (Agrawal et al., 1995; Wong and Qi, 2006)) for narrowing the domain of suspicious statements.

3. Our approach

We will explain our approach by using an example program shown in Fig. 2. The function `foo()` in Fig. 2 returns the calculated results of three input integers. Considering all possible combinations, we can generate eight test cases (dubbed t_1 to t_8), which input values are as followed (1, 2, 1), (4, 2, 1), (3, 1, 1), (2, 1, 1), (0, 1, 1), (1, − 1, 1), (1, 2, 0), (8, 2, 1). We also manually seed a fault into statement s_2 as shown in Fig. 2.

The overall goal of our work is to give an effective solution for fault localization. Since program slicing can extract a set of statements from the program with respect to a certain slice criterion, we compute all the full slices with the fault-related criteria of failed runs by program slicing. Illuminated by coverage-based approaches, we also compute execution slices and combine them with full slices to form HSS in our approach. Comparing with execution coverage matrix used by coverage-based approaches, HSS might have fewer rows and columns and then it can potentially reduce the time cost of suspiciousness computing. Moreover, unlike some slice-based approaches (e.g., Agrawal's (Agrawal and Horgan, 1990)), our approach computes the full slices based on a static Program Dependence Graph (PDG) which is constructed by the previous analysis and is used during all the tests, while those dynamic slicing methods need to compute a dynamic dependence graph during each test. Furthermore, our approach computes full slice of statements with a forward computing algorithm. Based on HSS, suspiciousness of statements in fault-related slices are computed with a proposed maximal evaluation formula. Here we give some related definitions which are used in our approach.

Definition 7. $O_f = \{s_j | \textit{the output of statement}(s_j) \textit{ is unexpected under some test cases}\}$ is a set of the statements which outputs mismatch the expected output.

Definition 8. $FS(o_i, t_{fj})$ is a function that returns the full slice on statement o_i under test case t_{fj}, where $o_i \in O_f$ and $t_{fj} \in T_f$.

3.1. The framework of our approach

The framework of our approach is shown in Fig. 3. Our approach is mainly composed of three modules: `Pre-Processing module`, `Slice computing module`, and `Fault locating module`. The initial input of our approach is the source code of the program P under test and a test suite T. After preprocessing by analyzing the source code statically and running all the test cases in T, we divide the test cases into two groups: passed test cases T_p and failed test cases T_f. At the same time, we also compute the O_f. In the slice computing module, based on Program Dependence Graph (*PDG*) constructed by the previous static analysis, we first compute the full slice (*FS*) by a forward computing algorithm using P and T_f with respect to O_f. Then we compute the execution slice (*ES*) by a forward computing algorithm using P and T_p. With full slices and execution slices obtained, we compute the hybrid slice spectrum (*HSS*) by a hybrid slice spectrum algorithm (HSSC). Finally, in the last module, we compute the suspiciousness of each statement in *HSS* based on a proposed formula and we generate a fault localization report by sorting the suspiciousness in descending order. Next, we will introduce the implementation of these modules in sequence.

3.2. Pre-processing module

The first phase of our approach focuses on preprocessing. Firstly, we statically analyze the program under test and transform the source code into intermediate code. Specifically, we obtain the Jimple code,[2] a 3-address code representation of Java bytecode, for performing analysis. By analyzing Jimple code, we can compute the static Program Dependence Graph (*PDG*) of the program, which is beneficial for dynamic full slice computing in the following steps. In reality, we utilize *PDG* to identify dynamic dependencies in the computation of dynamic slices. Secondly, we run all test cases in T and divide the test cases into two groups: passed test cases T_p and failed test cases T_f. We also compute the O_f when the running of

[2] More information about Jimple code can be found at the website: http://www.sable.mcgill.ca/soot/.

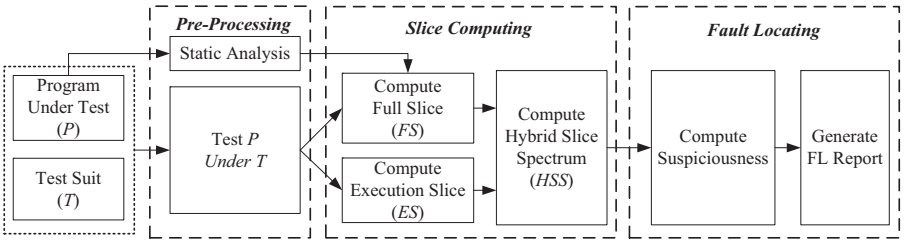

Fig. 3. The framework of our approach.

test cases results in the failure. After the preprocessing phase, we can obtain Jimple code and *PDG* of program *P*, two test case groups T_p and T_f, and the set of faulty output statements which are useful for the next slice computation O_f.

3.3. Slice computing module

In this module, we first conduct the computation of full slice (*FS*) and execution slice (*ES*) using *P*, T_f, T_p, *PDG* and O_f. After obtaining full slices and execution slices, we can compute the hybrid slice spectrum by the proposed hybrid slice spectrum algorithm (*HSSC*).

3.3.1. Execution slice computing

Execution slice describes the set of statements that executed in a program run. In our approach, we use execution slices of passed runs to construct hybrid spectrum slice (HSS) by combining with full slices of failed runs. Generally, the execution slice generation method can be classified into two categories: one category is to trace and record executed statements by program instrumentation. Another category is to monitor and record the memory map through the shadow memory. Both of them are required to measure the program behavior without interfering with the normal operation of the program. We developed our tool HSFal using the second way to trace the runtime information. In particular, we utilize the interrupter mechanism of the JVM through JDI (Java Debug Interface) to obtain the data related to the current executing statement.

We run all test cases to test the function `foo()` and gather their statement coverage. The detail of execution coverage and test results are shown in Fig. 2. The faulty program fails when running test cases t_1, t_2, and t_7 with different execution slices, and passes when running the rest of test cases. Specially, the function `foo()` has the same coverage under the test case t_3 and t_8. That is to say, $ES(t_3) = ES(t_8)$.

3.3.2. Full slice computing

In our approach, full slices of failed runs are also used to combine hybrid spectrum slice (HSS). Full slices are the transitive closure of statements on which the slice criteria data or control depend. When a test case triggers a failure in the program, developers are more concerned about the full slices on the undesired results. The intuitive method of computing full slice is to travel from the faulty outputs back along the Program Dependence Graph (*PDG*), but this method needs to construct the *PDG*. In our prototype tool HSFal, we adopted the forward computing full slice algorithm (Zhang et al., 2005) to compute both the data dependency sets and control dependency sets with *PDG* along the program executing from the entrance of the program. Although full slices of all statements are computed, only the latest slices of statements are used. Our tool HSFal continuously computes full slices as statements are executed until the statement $s'(s' \in O_f)$ is reached.

We use the example in Fig. 2 to explain the forward computing the full slice algorithm. The program fails after running test cases $t_1(1, 2, 1)$, $t_2(4, 2, 1)$ and $t_7(3, 1, 1)$. Let ${}^i s$ be the ith execution of the statement s. Table 3 shows the process of forward computation for full slice under the test case t_7. The return value is determined by the statement ${}^1 s_{11}$ and the output of faulty program is 2 which is mismatched with the expected output value −2. Looking up the full slice of the variable *ret* at ${}^1 s_{11}$, we will find the real faulty statement s_2. In other words, the fault in s_2 can be located in $FS({}^1 s_{11})$. Obviously, the size of $FS({}^1 s_{11})$ is smaller than that of $ES(t_7)$ by removing statements s_3 and s_6.

Table 3
Forward computation of full slices of faulty program under test case t_7.

${}^i s$	$Def(s)$	$Ref(s)$	$CD(s)$	$FS(s)$
${}^1 s_1$	$\{x, y, z\}$	$\emptyset$	$\emptyset$	$\{s_1\}$
${}^1 s_2$	$\{x\}$	$\{x, y\}$	$\emptyset$	$\{s_1, s_2\}$
${}^1 s_3$	$\{m\}$	$\{y\}$	$\emptyset$	$\{s_1, s_3\}$
${}^1 s_4$	$\emptyset$	$\{x\}$	$\emptyset$	$\{s_1, s_2, s_4\}$
${}^1 s_5$	$\{m\}$	$\{x\}$	$\{s_4\}$	$\{s_1, s_2, s_4, s_5\}$
${}^1 s_6$	$\{x\}$	$\{x, y\}$	$\{s_4\}$	$\{s_1, s_2, s_4, s_6\}$
${}^1 s_7$	$\{z\}$	$\{y\}$	$\{s_4\}$	$\{s_1, s_2, s_4, s_7\}$
${}^1 s_8$	$\emptyset$	$\{m\}$	$\emptyset$	$\{s_1, s_2, s_4, s_5, s_8\}$
${}^1 s_9$	$\{ret\}$	$\{y, z\}$	$\{s_8\}$	$\{s_1, s_2, s_4, s_5, s_7, s_8, s_9\}$
${}^1 s_{11}$	$\emptyset$	$\{ret\}$	$\emptyset$	$\{s_1, s_2, s_4, s_5, s_7, s_8, s_9\}$

3.3.3. Hybrid slice spectrum computing

We calculate the hybrid slice spectrum (HSS) using full slices and execution slices obtained from the previous two steps. A union of all full slices is calculated firstly. Then the union is respectively intersected with both full slices and execution slices to construct the hybrid slice spectrum. The coverage detail of HSS is shown in Fig. 4.

Let U_F denotes the union of $FS(o_i, t_{fj})$, $o_i \in O_f$, $t_{fj} \in T_p$. In fact, U_F can be denoted by a set of statements $\{s'_1, s'_2, \ldots, s'_m\}$ shown in Fig. 4.

$$S_{cover} = \begin{array}{c} \begin{matrix} s'_1 & s'_2 & \cdots & s'_m \end{matrix} \\ \left[\begin{array}{cccc|c} d_{11} & d_{12} & \cdots & d_{1m} & t_{f1} \\ d_{21} & d_{22} & \cdots & d_{2m} & t_{f2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ d_{l1} & d_{l2} & \cdots & d_{lm} & t_{fl} \\ \hline e_{v1} & e_{v2} & \cdots & \cdots & t_{pv} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ e_{u1} & e_{u2} & \cdots & e_{um} & t_{pu} \end{array}\right] \end{array}$$

Fig. 4. Coverage of hybrid spectrum slices (HSS).

Table 4
Hybrid slice spectrum for `foo()` after running all test cases.

Test cases	Pass/fail	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}
t_1	Fail	1	1	1	1	0	0	0	1	1	0	1
t_2	Fail	1	1	0	1	1	1	1	1	0	1	1
t_7	Fail	1	1	0	1	1	0	1	1	1	0	1
$t_3(t_8)$	Pass	1	1	1	1	1	1	1	1	1	0	1
t_4	Pass	1	1	1	1	1	1	1	1	0	1	1
t_5	Pass	1	1	1	1	0	0	0	1	1	0	1
t_6	Pass	1	1	1	1	0	0	0	1	0	1	1

The elements d_{ij} and e_{ij} which denote the coverage of full slices and execution slices are computed by formula (6), respectively.

$$d_{ij} = \begin{cases} 1, & \text{if } s'_j \in FS(o_k, t_{fi}) \\ 0, & \text{otherwise} \end{cases} \quad \text{and} \quad e_{ij} = \begin{cases} 1, & \text{if } s'_j \in ES(t_{pi}) \cap (\bigcup_{o_h \in O_f} FS(o_h, t_{fk})) \wedge t_{pi} \in T_p \wedge t_{fk} \in T_f \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

We use Algorithm 1 to explain the computation of hybrid slice spectrum based on full slices and execution slices. $FS(O_f)$ denote the set of all the full slices $FS(o_i, t_{fj})$, $o_i \in O_f$, $t_{fj} \in T_p$. The lines 1–3 fetch each slice in $FS(O_f)$ and add them to HSS matrix. The function *Addline*(*fs*) is to add slice *fs* to HSS satisfying formula (6). The lines 5–7 intersect each slice in execution slices with the union set of all statements in $FS(O_f)$ and add the result to HSS. The function *Merge*(HSS) is to remove the redundancy execution slices that are same with others in HSS matrix.

Algorithm 1. [Hybrid slice spectrum computing algorithm: HSSC]

Require:
<$FS(O_f)$, $ES(T_P)$>
Ensure
HSS
1: **for** each fs in $FS(O_f)$ **do**
2: $HSS.Addline(fs)$;
3: **end for**
4: $ALLStmsInFS = \cup FS(O_f)$;
5: **for** each es in $ES(T_P)$ **do**
6: $HSS.Addline(es \cap ALLStmsInFS)$;
7: **end for**
8: $HSS = Merge(HSS)$;
9: return HSS;

In the motivating example shown in Fig. 2, considering that the function `foo()` fails after running test cases t_1, t_2 and t_7, we further compute the full slices with respect to statement s_{11}. With full slices and execution slices obtained, we can compute the hybrid slice spectrum (HSS) by Algorithm 1. Table 4 shows the hybrid slice spectrum of the example. It is demonstrated that the test case t_3 and t_8 have the same hybrid slice spectrum.

Generally speaking, both the number of rows and columns in S_{cover} computed by Algorithm 1 might be smaller than that of T_{cover} matrix shown in Fig. 1. That is to say, HSS can reduce the computational costs at the stage of suspiciousness computing. But we also notice that HSS would spend quite a number of time computing full slices, execution slices and hybrid slice spectrum. We will discuss the total computational costs in Section 4.

3.4. Fault locating module

In general, the suspiciousness of a statement is directly proportional to the number of covering by failed executions and uncovering by passed executions, and inversely proportional to the number of covering by passed executions and uncovering by failed executions. Moreover, for both passed tests and failed tests, the covered entities should make more contributions to fault localization than the uncovered entities do. Thus, the number of covering entities by tests is more influential and should carry a greater weight than that of uncovering entities in suspiciousness computing. Dozens of formulas for suspiciousness calculation were proposed and evaluated by empirical studies during the last decades. In our approach, based on the above intuitions, we first choose two maximal formulas (`Naish2` and `Russel&Rao`) within five formulas from each maximal group, theoretically proved by Xie et al. (2013). Since that the formulas from each group are not equivalent in performance of fault localization, we multiply them to construct a new formula shown as follows. Furthermore, we expect that the multiply operation of constructing new formula might hold the advantages of the two existing maximal formulas.

$$HSS^0(s) = \left(N_{CF}(s) - \frac{N_{CS}(s)}{N_{CS}(s) + N_{US}(s) + 1}\right) \times \left(\frac{N_{CF}(s)}{N_{CF}(s) + N_{UF}(s) + N_{CS}(s) + N_{US}(s)}\right) \tag{7}$$

In formula (7), notation $N_{CF}(s)$ and $N_{CS}(s)$ denote the number of covering by failed and passed slices in HSS, respectively. Similarly, notation $N_{UF}(s)$ and $N_{US}(s)$ denote the number of uncovering by failed and passed slices in HSS, respectively. Let $N_S(s)$ denote the sum of $N_{CS}(s)$ and $N_{US}(s)$, and let $N_F(s)$ denote the sum of $N_{CF}(s)$ and $N_{UF}(s)$. Though the formula (7) is mutated form Naish2 and Russel&Rao, it holds its trends of the two maximal formulas mathematically, and it also complies with the intuitions discussed above. The reason is that the assessed suspiciousness by both Naish2 and Russel&Rao are nonnegative, and formula (7) is the product of the two maximal formulas by multiplying. Furthermore, Considering $N_S(s) >= 0$, $N_F(s) >= 0$ and $N_S(s) + N_F(s) > 0$ in our scenario, we replace $N_{CS}(s) + N_{US}(s) + 1$ with N. Therefore, the formula (7) can be modified as follows:

$$HSS(s) = \frac{N_{CF}^2(s)}{N} - \frac{N_{CF}(s) \times N_{CS}(s)}{N^2} \tag{8}$$

Based on the HSS matrix in Fig. 4, all the parameters in the formula (8) can be computed as follows, $N_{CS}(s_j) = \sum_{i=v}^{u} e_{ij}$, $N_{CF}(s_j) = \sum_{i=1}^{l} d_{ij}$, $N_S = l$, $N_F = u - q + 1$ and $N = N_S + N_F$. Formula (8) is artificially constructed as an intuitively suspiciousness evaluation formula. But the underlying assumption of this formula is that the statement covered by more failed or less passed tests should have a greater suspiciousness value, and this assumption also complies with the assumptions adopted by other formulas.

Illuminated by the work of Xie et al. (2013), we will give a theoretical analysis to show the effectiveness of our proposed formula by comparing with two maximal formulas (i.e., Naish2 and Russel&Rao). In our analysis, we will use some notations and definitions appeared in Xie's paper. Such as $S_B^R = \{s_i | R(s_i) > R(s_f)\}$, $S_F^R = \{s_i | R(s_i) = R(s_f)\}$ and $S_A^R = \{s_i | R(s_i) < R(s_f)\}$, s_i is a statement of program, s_f is the real fault of program, R is a suspiciousness evaluation formula. Given two suspiciousness formulas R_1 and R_2, it is said that R_1 is better than R_2 in the effectiveness of fault localization (denoted as $R_1 \rightarrow R_2$) if and only if $S_B^{R1} \subseteq S_B^{R2}$ and $S_A^{R2} \subseteq S_A^{R1}$ (Xie et al., 2013). Here, the relation "$\rightarrow$" means "*better than*". Before presenting our theoretical analysis of the effectiveness of these formulas, we firstly propose one assumption as a complementary to Xie's four assumptions (Xie et al., 2013). Our assumption is that a test case will always

yield the result of fail if the real fault is executed under the test case, then we have $N_{CF}(s_f)=N_F$, and $N_{CS}(s_f)=0$. Then we will prove the following two propositions.

Proposition 1. *HSS → Russel & Rao.*

Proof. (We dubbed *Russel & Rao* as *R & R* in our proof)

(A) To prove that $S_B^{HSS} \subseteq S_B^{R\&R}$.

Assume $s_i \in S_B^{HSS}$. Then, according to the definition of S_B^{HSS}, we have

$$\frac{N_{CF}^2(s_i)}{N} - \frac{N_{CF}(s_i)\times N_{CS}(s_i)}{N^2} > \frac{N_{CF}^2(s_f)}{N} - \frac{N_{CF}(s_f)\times N_{CS}(s_f)}{N^2}. \tag{9}$$

Considering our proposed assumption, we have $N_{CS}(s_f)=0$, then the above inequality (9) can be re-arranged to

$$\frac{N_{CF}^2(s_i)}{N} - \frac{N_{CF}(s_i)\times N_{CS}(s_i)}{N^2} > \frac{N_{CF}^2(s_f)}{N}. \tag{10}$$

Also considering $N_{CF}(s_i)\geq 0$, $N_{CS}(s_i)\geq =0$, we have

$$\frac{N_{CF}(s_i)\times N_{CS}(s_i)}{N^2} > 0. \tag{11}$$

Therefore, the inequality (10) can be re-arranged to $N_{CF}^2(s_i) > N_{CF}^2(s_f)$. Also because $N_{CF}(s_f)>0$ and $N=N_{CF}(s_i)+N_{CS}(s_i)+N_{UF}(s_i)+N_{US}(s_i)=N_{CF}(s_f)+N_{CS}(s_f)+N_{UF}(s_f)+N_{US}(s_f)$, we have

$$\frac{N_{CF}(s_i)}{N_{CF}(s_i)+N_{CS}(s_i)+N_{UF}(s_i)+N_{US}(s_i)} > \frac{N_{CF}(s_f)}{N_{CF}(s_f)+N_{CS}(s_f)+N_{UF}(s_f)+N_{US}(s_f)}. \tag{12}$$

Thus, $s_i \in S_B^{R\&R}$ according to the definition of $S_B^{R\&R}$. Therefore, we have $S_B^{HSS} \subseteq S_B^{R\&R}$.

(B) To prove that $S_A^{R\&R} \subseteq S_A^{HSS}$.

Assume $s_i \in S_A^{R\&R}$. Then, according to the definition of $S_A^{R\&R}$, we have

$$\frac{N_{CF}(s_i)}{N_{CF}(s_i)+N_{CS}(s_i)+N_{UF}(s_i)+N_{US}(s_i)} < \frac{N_{CF}(s_f)}{N_{CF}(s_f)+N_{CS}(s_f)+N_{UF}(s_f)+N_{US}(s_f)}. \tag{13}$$

Considering $N_{CF}(s_i)>=0$, $N_{CS}(s_i)>=0$ and $N=N_{CF}(s_i)+N_{CS}(s_i)+N_{UF}(s_i)+N_{US}(s_i)=N_{CF}(s_f)+N_{CS}(s_f)+N_{UF}(s_f)+N_{US}(s_f)$, we have $(((N_{CF}(s_i)\times N_{CS}(s_i))/N^2)>0)$ and the inequality (13) can be re-written into

$$\frac{N_{CF}^2(s_i)}{N} < \frac{N_{CF}^2(s_f)}{N}. \tag{14}$$

Following the above assumptions, we have $((N_{CF}(s_f)\times N_{CS}(s_f))/N^2)=0$. Then, the inequality (14) can be re-arranged into

$$\frac{N_{CF}^2(s_i)}{N} - \frac{N_{CF}(s_i)\times N_{CS}(s_i)}{N^2} < \frac{N_{CF}^2(s_f)}{N} - \frac{N_{CF}(s_f)\times N_{CS}(s_f)}{N^2}. \tag{15}$$

Thus, $s_i \in S_A^{HSS}$ according to the definition of S_A^{HSS}. Therefore, we have $S_A^{R\&R} \subseteq S_A^{HSS}$.

In conclusion, we have $S_B^{HSS} \subseteq S_B^{R\&R}$ and $S_A^{R\&R} \subseteq S_A^{HSS}$. So it is sufficient to prove *HSS → Russel & Rao*. In an other words, HSS is better than Russel&Rao.

Proposition 2. *HSSandNaish2arethemaximalformulas.*

Proof. First, we will prove that *HSS → Naish2* does not hold. (We dubbed *Naish2* as *Na* in our proof). Assume *HSS → Naish2*, then we have $(S_B^{HSS} \subseteq S_B^{Na})$ and $(S_A^{Na} \subseteq S_A^{HSS})$.

$\forall s_i \in S_B^{HSS}$, we have $s_i \in S_B^{Na}$ because of $S_B^{HSS} \subseteq S_B^{Na}$. Then, according to the definitions of $S_B^{HSS} and S_B^{Na}$, we have

$$\begin{cases} \frac{N_{CF}^2(s_i)}{N} - \frac{N_{CF}(s_i)\times N_{CS}(s_i)}{N^2} > \frac{N_{CF}^2(s_f)}{N} - \frac{N_{CF}(s_f)\times N_{CS}(s_f)}{N^2} \\ N_{CF}(s_i) - \frac{N_{CS}(s_i)}{N_{CS}(s_i)+N_{US}(s_i)+1} > N_{CF}(s_f) - \frac{N_{CS}(s_f)}{N_{CS}(s_f)+N_{US}(s_f)+1} \end{cases} \tag{16}$$

According to our assumptions, we have $N_{CF}(s_f)=N_f$, $N_{CS}(s_f)=0$ and $N_{CS}(s_i)+N_{US}(s_i)=N_{CS}(s_f)+N_{US}(s_f)=N_S$. Thus, the above inequality group (16) can be re-arranged to

$$\begin{cases} N_{CF}(s_i) > N_{CF}(s_f) \\ N_{CF}(s_i) - \frac{N_{CS}(s_i)}{N_S+1} > N_{CF}(s_f) \end{cases}$$

Thus, $\frac{N_{CS}(s_i)}{N_S+1} < N_{CF}(s_i) - N_{CF}(s_f)$ due to $N_{CF}(s_i) - N_{CF}(s_f) <=0$ and $N_S+1>0$. Then, we will get the result $N_{CS}(s_i)<0$. But in fact $N_{CS}(s_i)>0$. Therefore, the relation ("$S_B^{HSS} \subseteq S_B^{Na}$") does not hold. In conclusion, *HSS* is not "better" ("→") than *Naish2*.

Secondly, we will prove that *Naish2 → HSS* does not hold. Assume *Naish2 → HSS*, then we have $(S_B^{Na} \subseteq S_B^{HSS})$ and $(S_A^{HSS} \subseteq S_A^{Na})$. Similar to the first step of our proof, we can prove that the relation ("$S_B^{Na} \subseteq S_B^{HSS}$") does not hold. Therefore, *Naish2* is also not "better" ("→") than *HSS*.

Following Propositions 1 and 2, we can conclude that both our proposed formula *HSS* and *Naish2* are the maximal formulas.

Next we will give a mini experiment to illustrate the effectiveness of our approach based on the motivating example shown in Table 5. The suspiciousness of each statement in `foo()` are computed by formula (8) and three other formulas using coverage and hybrid slice spectrum, respectively. It shows that our proposed formula (HSS) shares the same effectiveness with the other two maximal formulas, namely Naish2 and Russel&Rao (dubbed R&R), based on both coverage and hybrid slice spectrum. Moreover, the formula (8) always performs better than Tarantula does. For example, based on hybrid slice spectrum, we need to examine 1 up to 5 statements using formulas HSS, Naish2, or Russel&Rao. However, we need to examine 4 up to 8 statements using formula Tarantula.

In addition, using Xie et al.'s two sample programs *PG1* and *PG2* (Xie et al., 2013) with the test suite *TS*3, we also compute the suspiciousness of each statement in PG1 and PG2 with our formula and two maximal formulas (Russe&Rao and Naish2) to make a comparison. The experiment result shows that the rank of the faulty statement by our formula is always the same as that of Naish2 and smaller than that of Russel&Rao. Thus, we can consider that the performance of our formula is as good as Naish2 and is always better than Russel&Rao which is exactly brought into correspondence with our previous theoretical analysis.

However, in real program debugging scenarios, the assumptions of our theoretical analysis are not always satisfied. Therefore, we need more empirical studies to evaluate our approach. In the next section, we will use more complex programs and conduct empirical studies to show the effectiveness and efficiency of our approach and the proposed formula.

4. Empirical study

To evaluate the effectiveness and efficiency of our approach (HSS), we implemented it in a prototype tool HSFal and applied the tool to 14 subject programs and corresponding faults. In our empirical study, we want to investigate the following research questions:

- RQ1: Can the hybrid slice spectrum of our approach perform better than other refined spectrum based on the same suspiciousness evaluation formula? If so, by how much?

Table 5
Suspiciousness of statements for `foo()` computed by different approaches.

Statement	Coverage based				Hybrid slice spectrum based			
	Tarantula	Naish2	R&R	HSS	Tarantula	Naish2	R&R	HSS
s_1	0.5	2.17	0.38	0.89	0.5	2.2	0.38	0.91
s_2	0.5	2.17	0.38	0.89	0.5	2.2	0.38	0.91
s_3	0.5	2.17	0.38	0.89	0.25	0.2	0.13	0.05
s_4	0.5	2.17	0.38	0.89	0.5	2.2	0.38	0.91
s_5	0.53	1.5	0.25	0.41	0.57	1.6	0.25	0.43
s_6	0.53	1.5	0.25	0.41	0.4	0.6	0.13	0.09
s_7	0.53	1.5	0.25	0.41	0.57	1.6	0.25	0.43
s_8	0.5	2.17	0.38	0.89	0.5	2.2	0.38	0.91
s_9	0.53	1.5	0.25	0.41	0.57	1.6	0.25	0.43
s_{10}	0.45	0.67	0.13	0.09	0.4	0.6	0.13	0.09
s_{11}	0.5	2.17	0.38	0.89	0.5	2.2	0.38	0.91
Fault rank	5–10	1–6	1–6	1–6	4–8	1–5	1–5	1–5
Loc. cost (%)	45.5–90.9	9.1–54.5	9.1–54.5	9.1–54.5	36.4–72.7	9.1–45.5	9.1–45.5	9.1–45.5

Table 6
The characteristics of subjects.

Subject	Description	LOC	Tests	Faults
Print_tokens1	Lexical analyzer	478	1200	5
Print_tokens2	Lexical analyzer	410	1200	10
Schedule1	Priority scheduler	290	2650	9
Schedule2	Priority scheduler	317	2710	9
Tot_info	Information measure	283	1052	10
Jtcas	Collision avoidance	181	1201	12
Sorting	Five sorting algorithms	222	100	3
NanoXML v1	XML parser	3497	214	7
NanoXML v2	XML parser	4009	214	6
NanoXML v3	XML parser	4608	216	4
NanoXML v5	XML parser	4782	216	8
XML-sec v1	XML encryption	21,613	92	8
XML-sec v2	XML encryption	22,318	94	6
XML-sec v3	XML encryption	19,895	84	7

- RQ2: Can the ranking of HSS computed by our proposed maximal formula improve the effectiveness of fault localization significantly? If so, by how much?
- RQ3: Since HSS is a heavyweight fault localization technique, how about the time cost of HSS used for fault localization?

4.1. Experiment subjects

We summarize the characteristics of the subjects used in our empirical study in Table 6. These subjects are implemented by Java programming language. For each subject, it provides a brief description (column 2), the number of executable lines of code (column 3), the number of test cases (column 4) and the number of faults studied (column 5). The former five subjects were Java version of Siemens programs which were translated from C version by Santelices et al. (2009),[3] and the rest of subjects were downloaded from the Subject Infrastructure Repository (SIR) (Do et al., 2005). Specially, `NanoXML` and `XML-security` are large scale subjects which we treat as representative of real-world subjects in our empirical study. Moreover, we studied four releases of `NanoXML` and three releases of `XML-security`.

The numbers of executable lines of all the subjects are ranging from hundreds of lines to tens of thousands of lines (column 3). All test cases for these subjects excluding `Sorting` were downloaded from the two above web site. We only use parts of them (the numbers of test cases are listed in column 4 of Table 6), and we randomly generated 100 groups of various integers for testing the subject program `Sorting`. Each version of the subjects listed in Table 6 has been seeded single fault. All the faults in these subjects excluding `Sorting` were manually seeded by other researchers. For `Sorting`, we manually seeded three faults and the considered fault types include predicate based fault and assignment based fault. For `Jtcas`, we randomly selected 12 faults from 41 faulty versions to avoid biasing the average cost toward this subject. In all, we studied 104 different faults in our experiment.

4.2. Variables and measures

The primary goal of this study is to evaluate the effectiveness and efficiency of the fault localization approach proposed in this paper. To accomplish this, we utilize one independent and four dependent variables. The independent variable is the fault-localization technique. We examine our proposed approach comparing with a set of fault-localization techniques. The first dependent variable is the *cumulative number of statements examined*, the second dependent variable is the *EXAM* score, the third dependent variable is the *average costs*, and the last dependent variable is the *time overhead* of each fault localization technique used in locating faults.

Intuitively, we consider the *cumulative number of statements examined* with respect to all faulty versions as the effectiveness of a fault localization technique. For each approach, the *cumulative number of statements examined* is recorded and computed for comparing the effectiveness with each other. Additionally, we compute *EXAM* score defined as the percentage of statements that has to be examined in order to find the fault. Thus, the effectiveness of a fault localization technique can also be illustrated by *EXAM* score. In other words, Technique `A` is considered to be more effective than technique `B` if its *EXAM* score assigned is less than that of technique `B`. Furthermore, we compute the *average costs* on each subject defined as the percentage of examined statements per fault. Finally, we collect the time used in each phase of a fault localization to illustrate its efficiency.

We compare HSS with both other coverage-based and slice-based fault localization techniques, based on the above four dependent variables (i.e., *cumulative number of statements examined*, *EXAM*-score, *average costs*, *time overhead*). Since we intend to demonstrate that HSS is more effective than other compared fault localization techniques, we use the one-tail alternative hypothesis to evaluate that HSS requires examining fewer statements than that of other techniques.

4.3. Experimental set up

To verify the effectiveness of our approach, we compare our approach HSS with both four coverage-based approaches (i.e., Tarantula (Jones, 2004), DStar (Wong et al., 2012), Naish2, and

[3] These subjects can be downloaded from his personal website: http://www3.nd.edu/ rsanteli/subjects.

Russel&Rao (Xie et al., 2013)) and four slice-based approaches (i.e., Set Intersection (Renieris and Reiss, 2003), Set Union (Renieris and Reiss, 2003), ADBS (Lei et al., 2012), and Wen (2012)). We select these approaches by considering that Tarantula is widely compared in most literatures, Naish2 and Russel&Rao are proved to be the maximal evaluation formulas by Xie et al. (2013) while DStar is not included in Xie's theoretical analysis work. Tarantula, which evaluation formula is $(N_{CF}/N_F)/((N_{CS}/N_S)+(N_{CF}/N_F))$, takes the intuition that the program entities which are primarily covered by failed runs are more likely to be faulty than those by passed runs. DStar, which evaluation formula is $N_{CF}^*/(N_{UF}+N_{CS}, (*=2, 2.5, \ldots))$, evaluates the suspiciousness of statements by modifying the *Kulczynski* coefficient $(a/(b+c))$ to DStar (i.e., by adding an exponent to its numerator). Since we cannot theoretically prove that other values of its exponent shall always be more efficient in fault localization, without loss of generality, the exponent of DStar is set to 2 for simplified calculation in our empirical study. Naish2 and Russel&Rao gather the coverage of all the executions and calculate the suspiciousness of statements by the evaluation formulas which are listed in Table 2. With each proposed risk evaluation formula, these coverage-based approaches can calculate the suspiciousness of all statements. Finally these approaches can generate a fault localization report with descending suspiciousness of each statement.

`Set Intersection` (dubbed Inter) and `Set Union` (dubbed Union) which are described in Renieris and Reiss (2003) locate faults based on execution slices. Different from those statistical approaches, these two approaches do not need to compute suspiciousness. Given a random failed execution slice (*fs*) and a set of passed execution slices (*PS*), `Set Intersection` computes the difference $(\bigcap_{ps\in PS} ps - fs)^C$ and `Set Union` computes the difference $(fs - \bigcup_{ps\in PS} ps)$ to form the search domain. Then the developer can inspect the search domain according to the appearance order of statements in the program under test until finding the fault. Considering that Set Intersection and Set Union approaches may generate false negatives, we count all statements in execution slices as search domain if these approaches missed the actual fault. Approximate dynamic backward slicing (ADBS) approach (Zhang et al., 2007) computes the statements in the intersection of the static backward slice and the set of execution slices as the search domain of faults. Furthermore, we apply a statistical approach to ADBS for suspiciousness computing by Tarantula formula (Lei et al., 2012) in our experiment. Wen's approach (Wen, 2012) combines full slices of failed runs and execution slices of passed runs to a matrix which presents the frequency of each statement executed by each test case. Based on this matrix, we compute the corresponding suspiciousness for each statement by a Tarantula style formula (Wen et al., 2011) represented as $(\sum Per_{CF}(s_i, t_{fl})/N_F)/((\sum Per_{CS}(s_i, t_{pk})/N_S)+(\sum Per_{CF}(s_i, t_{fl})/N_F))$, where $\sum Per_{CF}(s_i, t_{fl})$ and $\sum Per_{CS}(s_i, t_{pk})$ denote the sum of the percentage of statement-frequency by failed test and passed test, respectively. Let $fre(s_i, t_k)$ denotes the frequency of s_i that is executed by the test case t_k, then we have $Per_{CF}(s_i, t_{fl}) = (fre(s_i, t_{fl})/\sum fre(s_j, t_{fl}))$ where s_i is covered by a failed test case t_{fl} and $Per_{CS}(s_i, t_{pk}) = (fre(s_i, t_{pk})/\sum fre(s_j, t_{pk}))$ where s_i is covered by a passed test case t_{pk}.

To answer our three research questions, we conduct an empirical study to examine the behavior of faulty software and assess the four assumptions previously discussed in Section 2. Main part of HSFal includes analyzing the Java bytecode, tracing executions of each subject program on the fly, and computing the ranking list based on HSS. In practice, we firstly select a single fault subject program listed in Table 6, and exclude multi-fault versions. Secondly, we run the faulty subjects by the test cases which is randomly selected from the test suite. Thirdly, HSFal monitors each execution through Java Debug Interface (JDI) with full slices and execution slices computed at the same time. Finally, HSFal composes all slices to HSS and generates a fault localization report based on the ranking list computed by formula (8). For each faulty version, we compute the *EXAM* score based on fault localization report considering the seeded faults accordingly. Then, we compare *the percent of code examined* and *the percent of fault located* of our technique with other fault localization techniques, respectively.

Comparing with these above approaches, our approach (HSS) which locates faults with a new refined spectrum might be smaller than that of ADBS's and Wen's because of the removal of repeated execution slices from the final spectrum. Moreover, our approach adopts a maximal risk evaluation formula to compute the suspiciousness of statements. In summary, we have designed three experiments to make a comparison between our approach and these approaches discussed above. The designed experiments are described as follows:

- The first experiment is to make an evaluation of the hybrid spectrum by comparing our approach with ADBS and Wen's with the same suspiciousness evaluation formula adopted. In this comparison, we use the three maximal formulas (i.e., our formula, Naish2, and Russel&Rao) to compute the suspiciousness of statements, respectively.
- The second experiment is to make an evaluation of our approach by comparing with coverage-based approaches.
- The third experiment is to make an evaluation of our approach by comparing with slice-based approaches.

Note that we compute the suspiciousness of statements with each formula proposed in original literatures in the latter two experiments. In addition, we make a comparison of time cost between coverage-based fault localization (CBFL) and our approach (HSS).

The experiments were run on an Intel(R) Xeon(R) CPU at 3.07 GHz with 16G memory, and Ubuntu 12.04 64-bit operating system with Open-JDK 1.7 installed.

4.4. Data analysis

We conduct the first experiment described in Section 4.3 and present the results in Table 7. In this experiment, we employ three maximal formulas (i.e., Naish2, Russel&Rao, and our proposed formula) and compare the cumulative statements need to be examined by the refined spectrum (i.e., HSS, ADBS, and Wen's) using the same evaluation formula, respectively. As shown in Table 7, the column of *Naish2* illustrates the fault localization costs per subject based on three compared spectra when employing the formula Naish2 to evaluate the suspiciousness of statements. Similar to column *Naish2*, the latter two columns, *Russel & Rao* and *Ourformula*, represent the cumulative costs based on three refined spectra when employing Russel&Rao and our formula, respectively.

From Table 7, we can observe that, our spectrum can always need to examine fewer statements than that of the other two spectra (i.e., ADBS and Wen's) whatever formula is employed. For example, when employing formula *Naish2*, the total cost of HSS, which is only 9082, is much lower than the costs of other two compared spectra which all exceed 10,000 lines. That is to say, our hybrid slice spectrum (HSS) performs better in fault localization comparing with ADBS and Wen's. Besides, we can also observe that, the costs of fault localization employing our formula are always smaller than other two formulas (i.e., *Naish2* and *Russel & Rao*) whatever spectra is employed. For example, the total cost of HSS which is only 8844 when employing our formula, is much lower than the costs of HSS when employing other two formulas (i.e., *Naish2* and *Russel & Rao*). It means that in the empirical study, our formula can perform better than the compared formulas. This result also complies with the two propositions of our theoretical analysis in Section 3.4.

Table 7
Comparison on the cumulative statements need to be examined by HSS, ADBS and Wen's based on maximal formulas.

Subject	Naish2			Russel&Rao			Our formula		
	ADBS	Wen's	HSS	ADBS	Wen's	HSS	ADBS	Wen's	HSS
Print_tokens1	154	149	141	173	153	143	164	147	136
Print_tokens2	387	315	203	419	323	204	302	292	191
Schedule1	279	187	124	309	212	137	261	175	102
Schedule2	323	256	261	348	287	273	312	236	244
Tot_info	395	337	322	401	357	323	381	346	317
Jtcas	226	179	185	251	234	211	212	185	182
Sorting	101	98	79	147	116	81	126	117	74
NanoXML v1	1232	1152	912	1315	1172	924	1253	1098	842
NanoXML v2	1583	1285	1156	1630	1467	1227	1561	1275	1151
NanoXML v3	975	942	891	1247	1152	1017	987	965	873
NanoXML v5	1431	1327	1173	1495	1372	1269	1483	1247	1150
XML-sec v1	1356	1203	1017	1326	1267	1052	1367	1185	986
XML-sec v2	1417	1173	1064	1392	1205	1083	1408	1115	1053
XML-sec v3	2743	2212	1554	2815	2247	1659	2763	2138	1543
Overall cost	12,602	10,815	9082	13,268	11,564	9603	12,580	10,521	8844

To evaluate the significance of improvement by our hybrid slice spectrum (HSS), we applied the paired Wilcoxon tests between HSS and other two spectra (i.e., ADBS and Wen's) when using three maximal formulas. In order to show that our spectrum (HSS) is more effective in fault localization than the two compared spectra, we carried out the one-tailed alternative hypothesis to verify that HSS requires the least examination of statements than the compared spectra employing the formulas in Table 7, respectively. The p-values of all tests between HSS and ADBS range from 0.08178 to 0.1137, and the p-values of all tests between HSS and Wen's range from 0.179 to 0.2412. Therefore, we can accept the hypothesis with confidence level 0.8863 of the test between HSS and ADBS, and can accept the hypothesis with confidence level 0.7588 of the test between HSS and Wen's. In summary, comparing with spectra ADBS and Wen's with the same formula, our hybrid slice spectrum (HSS) can improve the effectiveness of fault localization even though these improvements are not very significant.

Next, we conduct the second and the third experiments to evaluate the effectiveness of our approach with formula (8) comparing with coverage-based approaches and slice-based approaches with each formula proposed in original literatures. We present the results of the *cumulative number of statements examined* over all faulty versions for each subject in Tables 8 and 9, when employing Tarantula, DStar, Naish2, Russel&Rao and our approach HSS. Due to the real faulty statement may share the same suspiciousness with others, the programmer might need to examine only one of these statements fortunately (best cases) or to examine all these statements to locate the real fault unfortunately (worst cases).

As shown in Table 8, the column of *Best cases* illustrates the best fault-localization costs per subject (i.e., minimal cumulative costs over all faults in all subjects) and the column of *Worst cases* shows the worst fault-localization costs per subject (i.e., maximal cumulative costs over all faults in all subjects).

From Table 8, we can observe that, under both best cases and worst cases, the *cumulative number of statements examined* by Tarantula and DStar are quite similar with each other. The *cumulative number of statements examined* by Russel&Rao (dubbed R&R) are smaller than three techniques (Tarantula, DStar and Nasish2) under best cases. Excepting for the subjects *Jtcas* and *Sorting* under the best cases, the *cumulative number of statements examined* by HSS are much smaller than compared techniques. The last row overall cost of Table 8, illustrates the total number of statements needed to be examined over all subjects by each approach. We can also observe that the total cost of HSS under best case, which is only 8299, is much lower than the costs of compared techniques which all exceed 14,000 lines, and the same in worst case. Therefore, it is obvious that the total costs of our approach HSS are significantly lower than that of the compared approaches either the best or the worst cases.

Table 9 indicates the comparison of slice-based approaches with our proposed approach. Columns 2–6 shows the *cumulative number of statements examined* by Set Intersection (dubbed Inter), Set Union (dubbed Union), ADBS, Wen and our approach (HSS), respectively. Columns 7–10 shows the improvement ratio of HSS comparing with other approaches.

From Table 9, we can observe that the cumulative statements by our approach (HSS) are always fewer than that by other compared approaches. Although the cumulative number of examined statements by our approach of each subject are close to that by Wen's, HSS, which overall cost is 8844, nearly reduced by one-third of cost than that of Wen's approach which is 12,755. All in all, our proposed approach, HSS, reduced the cumulative statements greatly than that of compared approaches.

Next, as shown in Fig. 5, we present the *EXAM* scores over all faulty versions by employing different fault localization techniques considering the best, the worst, and the average cases. Fig. 5 illustrates the *EXAM* score of our proposed approach and other compared fault localization approaches using six subplots. For each subplot, the horizontal axis represents the *percentage of code examined* in all subjects. Along the vertical axis we represent the *percentage of fault located* in all faulty versions. From subplot (a), (b), (c) and (d), we can observe that, regardless of the best or worst case is considered, HSS is always the most effective one than Tarantula, DStar, Naish2 and Russel&Rao when we examine less than 60% of the code, though Naish2 and Russel&Rao are proved to be the maximal risk evaluation methods (Xie et al., 2013). Besides that, Russel&Rao is less effective than other approaches while examining code is less than 70% in the worst case. In summary, HSS can always obtain the lower *EXAM* score than that of compared approaches (i.e., Tarantula, DStar, Naish2 and Russel&Rao). We present the average *EXAM* score of both coverage-based approaches and slice-based approaches in subplot (e) and (f), from which we can observe that HSS is more effective than others, and the *EXAM* score of Tarantula, DStar, Naish2 and Russel&Rao are much closed with each other in subplot (e), and the *EXAM* score of Inter and Union are also similar to each other. For example, to detect 70% of faults, HSS only needs to examine less than 10% of code while the other techniques except Wen need to examine more than 20%. As shown in subplot (f), HSS is slightly better than Wen in the *EXAM* score of locating faults. In brief, HSS is the most effective one among all the compared approaches in our experiments.

Furthermore, we computed the average costs of locating faults employing HSS together with other compared techniques on each subject as shown in Table 10. From Table 10, we can observe that,

Table 8
Comparison on the cumulative statements need to be examined by coverage-based approaches for all faulty versions of each subject.

Subject	Best cases					Worst cases				
	Tarantula	DStar	Naish2	R&R	HSS	Tarantula	DStar	Naish2	R&R	HSS
Print_tokens1	580	516	521	284	126	665	677	550	676	174
Print_tokens2	780	590	388	163	110	1465	1247	904	1408	312
Schedule1	126	144	68	44	39	234	261	252	458	134
Schedule2	770	784	867	364	209	1116	1176	1063	1699	286
Tot_info	550	500	421	386	314	650	750	804	973	467
Jtcas	194	135	144	12	84	521	453	459	730	253
Sorting	27	23	33	9	21	238	245	294	294	168
NanoXML v1	1636	1694	1771	1406	673	1881	1823	1891	2503	1815
NanoXML v2	1372	1193	1468	1350	1017	1560	1481	1756	1545	1493
NanoXML v3	1468	1246	970	967	548	1592	1348	1072	1061	858
NanoXML v5	2680	2289	2456	2096	1017	2993	2615	2769	3061	1582
XML-sec v1	3326	2847	2997	2820	1712	3925	3228	3540	4175	2124
XML-sec v2	2550	2310	2686	2652	1014	2718	2617	3061	3496	1556
XML-sec v3	4368	4438	4425	3673	1415	4790	4850	4813	6092	2935
Overall cost	20,427	18,709	19,215	16,226	8299	24,348	22,771	23,228	28,171	14,157

Table 9
Comparison on the cumulative statements need to be examined by slice-based approaches for all faulty versions of each subject.

Subject	Inter	Union	ADBS	Wen	HSS	$\frac{\Delta I-H}{\#HSS}$	$\frac{\Delta U-H}{\#HSS}$	$\frac{\Delta F-H}{\#HSS}$	$\frac{\Delta W-H}{\#HSS}$
Print_tokens1	956	606	181	172	136	6.03	3.46	0.33	0.26
Print_tokens2	2050	1835	681	394	191	9.73	8.61	2.57	1.06
Schedule1	1472	1427	472	183	102	13.43	12.99	3.63	0.79
Schedule2	1585	1868	490	243	244	5.50	6.66	1.01	0
Tot_info	1208	1103	415	374	317	2.81	2.48	0.31	0.18
Jtcas	1086	784	248	160	182	4.97	3.31	0.91	−0.12
Sorting	331	424	197	174	74	3.47	4.73	1.66	1.35
NanoXML v1	10,240	3080	1683	1428	842	11.16	2.66	1	0.70
NanoXML v2	9327	3828	1690	1485	1151	7.10	2.33	0.47	0.29
NanoXML v3	6316	3882	1021	1013	873	6.23	3.45	0.17	0.16
NanoXML v5	12,127	7823	1327	1319	1150	9.55	5.80	0.15	0.15
XML-sec v1	11,718	11,009	2013	1965	986	10.88	10.17	1.04	0.99
XML-sec v2	8953	3765	1869	1369	1053	7.50	2.58	0.77	0.30
XML-sec v3	11,601	5870	3683	2476	1543	6.52	2.80	1.39	0.60
Overall cost	78,970	47,304	15,970	12,755	8844	7.93	4.35	0.81	0.44

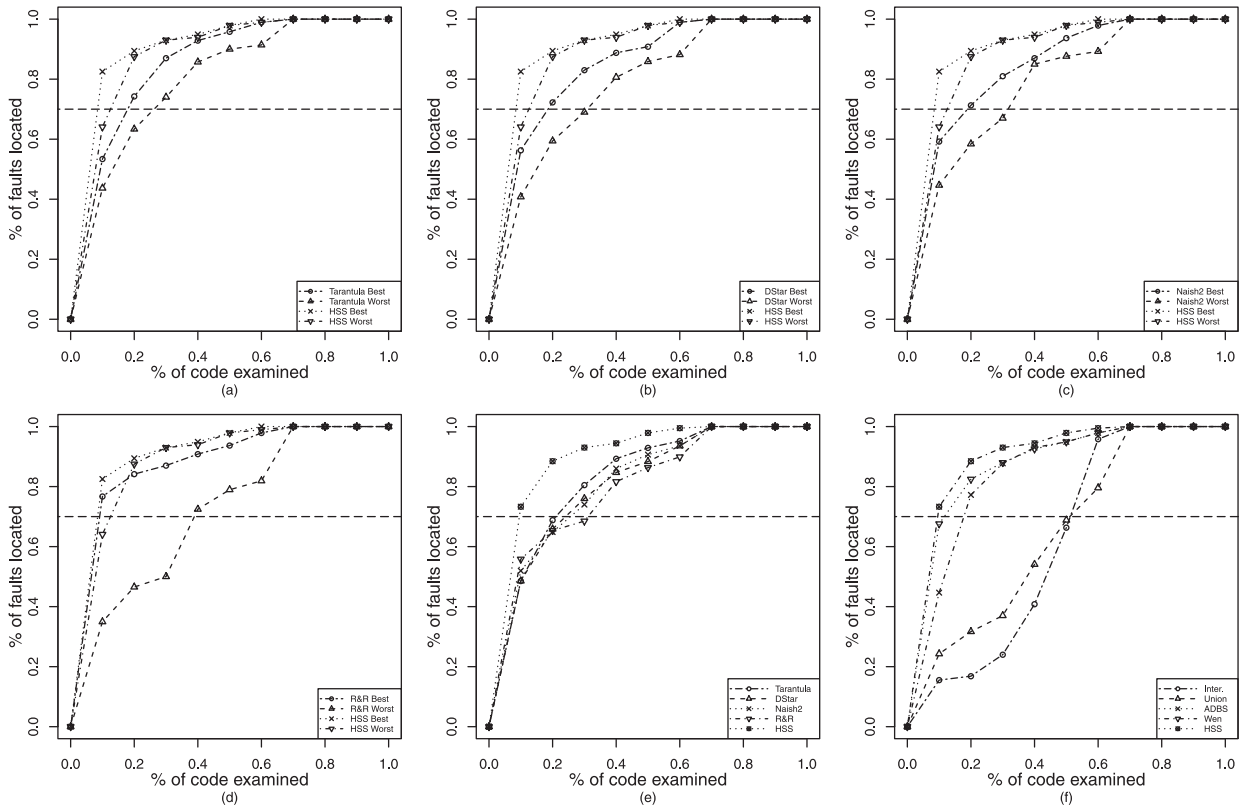

Fig. 5. *EXAM* score-based comparison on all subjects.

Table 10
Average fault-localization costs (the percentages of examined code per fault).

Subject	Tarantula	DStar	Naish2	R&R	Inter	Union	ADBS	Wen	HSS
Print_tokens1	26.05	24.96	22.41	20.08	40	25.36	7.57	7.20	5.06
Print_tokens2	27.38	22.40	15.76	19.16	50	44.76	16.61	9.61	4.66
Schedule1	6.90	7.76	6.13	9.62	56.40	54.67	18.08	7.01	3.91
Schedule2	33.05	34.35	33.82	36.15	55.56	65.47	17.17	8.52	8.55
Tot_info	21.20	22.08	21.64	24.01	42.69	38.98	14.66	15.22	11.2
Jtcas	16.46	13.54	13.88	17.08	50	36.10	16.02	7.37	8.38
Sorting	19.89	20.12	24.55	22.75	49.70	63.66	29.58	26.13	11.11
NanoXML v1	7.18	7.18	7.48	7.98	41.83	12.58	6.88	6.93	3.44
NanoXML v2	6.09	5.56	6.70	6.02	38.78	15.91	7.03	6.17	3.79
NanoXML v3	8.30	7.04	5.54	5.50	34.27	21.06	5.54	5.50	2.84
NanoXML v5	7.41	6.41	6.83	6.74	31.7	20.45	4.47	4.45	2.21
XML-sec v1	2.10	1.76	1.89	2.02	6.78	6.37	1.96	1.85	0.57
XML-sec v2	1.97	1.84	2.15	2.30	6.69	2.81	1.80	1.62	0.79
XML-sec v3	3.29	3.33	3.32	3.51	8.33	4.21	2.64	1.78	1.11
Average cost	13.38	12.74	12.29	13.07	36.62	29.46	10.72	7.81	4.83
Standard dev.	10.42	10.23	9.91	10.24	17.49	21.46	8.11	6.36	3.61

for all the subjects, the average fault localization costs of HSS are much smaller than that of compared approaches except for Wen's approach. In addition, all subjects except for `Schedule2` and `Jtcas`, the costs of HSS are slightly lower than that of Wen's approach. Complying precisely with the trend illustrated in subplot (e) and (f) in Fig. 5, the cost of Tarantula is very close to that of DStar, Naish2 and Russel&Rao (R&R in our table), and the costs of Inter and Union are very similar and almost twice more than that of the techniques as well as Tarantula. The last but one row of Table 10 illustrates the average costs on all subjects of each technique. It shows that, for all subjects, the average cost of HSS (which is only 4.83%) is less than half of Tarantula, DStar, Naish2, R&R and ADBS which are 13.38%, 12.74%, 12.29%, 13.07% and 10.72%, respectively. Furthermore, the cost of HSS is much smaller than that of Inter and Union, which are 36.62% and 29.46%. That is to say, HSS is actually more effective than these compared techniques in our experiments. The last row of Table 10 illustrates the standard deviation of costs on different subjects of each technique. The standard deviation value of HSS, which is only 3.61%, is the smallest one in that of all the nine compared approaches. In other words, HSS is more stable in effectiveness on different subjects than other approaches in our experiments.

Additionally, we applied the paired Wilcoxon test between column HSS and rest of eight columns in Table 10, respectively. In order to show that HSS is indeed more effective than the compared fault localization approaches, we carried out the one-tailed alternative hypothesis to verify that HSS requires the least examination of statements than the compared approaches. The p-values of all tests are less than 0.05 except for Wen's approach which p-value is 0.08906. Therefore, we can accept the alternative hypothesis with confidence level 0.90 of the test between HSS and Wen's approach, and can accept the alternative hypothesis for the rest tests with confidence level 0.95. In summary, compares with other approaches in our experiment, there is a statistically significant reduction of the cost on the set of subjects by HSS. Hence, HSS is the most effective one among all the fault localization approaches in Table 10.

Finally, we present the time overhead of our approach HSS comparing with coverage-based approaches (CBFL) in Table 11. Without loss of generality, we measured the time overhead of Tarantula as a representative of CBFL approaches. Column `test&trace` illustrates the time overhead spent for all test cases by CBFL and HSS, column `dyn.slicing` presents the time required for computing full slices by HSS, column `susp.computing` presents the computational time of suspiciousness, column `others` presents the time costs of rest tasks (i.e., writing and reading traced files), and the last column presents the division of the total time required by HSS and CBFL to illustrate the high time cost of our approach. We obtain the running information through JDI, so the time of HSS and CBFL required in testing and tracing are much longer than that of the running of the original subjects. Due to the complexity of the full slice computing algorithm, our approach requires additional time to compute full slices and construct hybrid slice spectrum. For example, our approach requires 2953.92 seconds, which is almost more than half of the running time by CBFL, to compute the dynamic slices of Print_tokens1. However, HSS requires a slightly less time in suspiciousness computing than that of CBFL owing to the smaller scale of the hybrid slice spectrum. Column *# HSS/# CBFL* in Table 11 shows that the time overhead of HSS for all subjects are less than 1.8 times of CBFL in our experiment.

Based on these above results, for this set of subjects, test suites and faults used in our empirical study, we can answer the three proposed research questions as follows:

- For RQ1, we can conclude that our hybrid slice spectrum can improve the performance of fault localization to a certain degree. In our experiments, comparing with ADBS and Wen's spectra, the costs of our spectrum can be reduced by 15.9% up to 27.9% at the 75.88% confidence level.
- For RQ2, we can conclude that HSS improves the effectiveness of locating faults significantly. In our experiments, comparing with eight fault localization techniques, the total costs of HSS are reduced by 2.98% up to 31.79% at the 90% confidence level.
- For RQ3, we can conclude that HSS requires less than 1.8 times of time-cost of CBFL approaches.

4.5. Threats to validity

Like any empirical study, there are some threats to the validity of our experiment. The external validity of this work lies in generalizing our empirical results. We only perform the empirical study on four releases of the medium-size program (NanoXML) and three releases of the large-scale program (XML-security), while the other subjects are small-size. Therefore it is difficult to guarantee the representativeness of our experimental results. However, all of these subject programs are widely used in most fault localization studies (e.g., Wong et al., 2012, 2008, 2010; Santelices et al., 2009; Abreu et al., 2009, 2009; Lei et al., 2012; Zhang et al., 2006; Yu et al., 2011). Another external validity lies in the using of seeded faults. However, we only manually seeded 3 faults in the second subject program `Sorting` by ourselves, and the other seeded faults in the rest of programs are carefully designed by other researchers (Do et al., 2005). Although seeded faults are not naturally-occurring faults, they are widely used by many researchers. A recent study by Ali et al. (2009) indicates that mutants were often found with the same effect in coverage-based techniques such as Tarantula.

The internal validity of this work lies in the accuracy of the slicing result computed by our tool HSFal. However, we adopted

Table 11
Time costs of coverage-based fault localization (CBFL) and our approach (HSS) spent on all subjects.

Subject	Approach	Test &trace(s)	Dyn. slicing(s)	Susp. computing(s)	Others(s)	Total(s)	$\frac{\#HSS}{\#CBFL}$
Print_tokens1	CBFL	5357.27	0	356.50	52.25	5766.02	1.76
	HSS	6785.88	2953.92	353.94	58.40	10,152.14	
Print_tokens2	CBFL	19,229.60	0	1428.69	155.54	20,813.83	1.37
	HSS	26,097.32	749.46	1421.93	173.84	28,442.55	
Schedule1	CBFL	3913.95	0	335.37	81.27	4330.59	1.80
	HSS	5479.53	1923.40	327.43	83.63	7813.99	
Schedule2	CBFL	15,933.90	0	2265.40	260.85	18,460.15	1.44
	HSS	17,808.48	6251.04	2227.89	280.17	26,567.58	
Tot_info	CBFL	813	0	214.52	23.27	26,567.58	1.54
	HSS	917	462.51	210.21	27.43	1617.15	
Jtcas	CBFL	724.71	0	14.79	2.11	741.61	1.74
	HSS	879.62	395.24	14.42	2.13	1291.41	
Sorting	CBFL	72.29	0	3.52	0.47	76.28	1.19
	HSS	78.52	8.48	3.41	0.53	90.94	
NanoXML v1	CBFL	1389.58	0	72.06	16.41	1478.05	1.31
	HSS	1768.56	73.24	71.69	16.86	1930.35	
NanoXML v2	CBFL	1469.01	0	83.51	12.84	1565.36	1.17
	HSS	1736.10	5.35	83.38	13.59	1838.42	
NanoXML v3	CBFL	1749.24	0	78.42	14.79	1842.45	1.19
	HSS	1955.03	137.72	78.11	15.28	2186.14	
NanoXML v5	CBFL	1523.16	0	94.04	12.04	1629.24	1.41
	HSS	1762.44	435.55	93.69	12.72	2304.40	
XML-sec v1	CBFL	2170.27	0	83.18	14.08	2267.53	1.16
	HSS	2237.34	302.41	82.24	14.71	2636.70	
XML-sec v2	CBFL	2078.65	0	95.12	11.81	2185.58	1.15
	HSS	2139.73	278.90	93.44	12.26	2524.33	
XML-sec v3	CBFL	1793.25	0	78.57	14.51	1886.33	1.21
	HSS	1831.78	355.78	78.62	14.98	2281.16	

and implemented a forward full slice computing algorithm similar to the algorithm proposed by Zhang et al. (2005). Another internal validity lies in the evaluation formula we proposed for suspiciousness computing. Although formula (8) is mutated from two formulas that were theoretically proved to be maximal by Xie et al. (2013), it complies with the general expectation as other widely adopted formulas. Furthermore, we have conducted a theoretical analysis of our evaluation formula under a complementary assumption, and proved it to be a maximal formula. Finally to avoid faults in our tool implementation, we prepared our data carefully and tested our tool HSFal with simple programs.

5. Related work

There have been a number of lightweight and heavyweight fault localization approaches proposed over the last decade. In this section, we mainly focus on coverage-based and slicing-based fault localization techniques that most similar to our approach (HSS) on behalf of lightweight and heavyweight approaches, respectively. Then, we briefly survey both of these studies.

5.1. Coverage-based fault localization techniques

Coverage-based fault localization techniques, which compute the suspiciousness of program entities with the execution traces, are widely applicable because they do not require any knowledge of the program.

Renieris and Reiss (2003) presented a nearest neighbor queries (NNQ) method for performing fault localization using similar coverage of program. NNQ contrasts the failed execution trace with a passed execution trace which is most similar to the failed one, and produces a report of "suspicious" parts of the program. Different from NNQ, our approach uses more than one execution slice of both passed and failed execution.

There are numerous coverage-based approaches using statistical measures for fault localization. Chen et al. (2002) introduced a statistical fault-localization algorithm, Jaccard, which predicts the location of a fault by computing the percentages of passed and failed tests that execute that statement. Similar to Jaccard, Jones et al. (2002), Jones (2004), and Jones and Harrold (2005) proposed the Tarantula technique that computes *suspiciousness* for each statement and ranks those statements by the *suspiciousness*. Tarantula is widely used and compared in the subsequent researches. Abreu et al. (2006) applied the Ochiai coefficient in software fault localization for locating single mistakes. Comparing with Tarantula, their experiments indicated that the Ochiai coefficient consistently outperforms Tarantula etc., and improves 5% on average over the compared techniques in terms of the *EXAM*-score. A later study by Abreu et al. (2009) evaluated the effectiveness of Tarantula and other proposed methods, their experiments indicated that Ochiai performs the best for statement coverage independent of test cases.

Tarantula and some similar fault location techniques equally treat the each occurrence of statement execution. However, Wong et al. (2007, 2010) proposed a code coverage-based fault localization method that can automatically adjust the weight for suspiciousness of a statement considering the execution times of passed test. They also gave some heuristics for reducing the search domain for fault locating. Wong et al. (2008) proposed a crosstab-based statistical approach using the coverage of each executable statement and the execution result (passed or failed) with respect to each test case. In recent work, Wong et al. (2012) proposed a technique named DStar(D*) which computed the suspiciousness of statements by modifying the form of the *Kulczynski* coefficient in the task of fault localization.

Furthermore, many researchers exploited the correlations between program features and the coverage of program entities as the clue in fault localization. Santelices et al. (2009) extended the Tarantula technique for different entity types (i.e., branch and du-pair) and combined these entity types by using the Ochiai coefficient for more effective fault localization. Masri (2010) present a fault localization technique which assigned the rank of statements associated with an information flow by contrasting the percentage of failed runs to the percentage of passed runs. Wang et al. (2009) proposed a technique to locate faults by strengthening the correlation between faulty statements and failed runs with control-flow and data-flow pattern. Unlike their techniques, HSS only needs

to statistic the occurrences of statements in hybrid slice spectrum for suspiciousness computing.

Moreover, a recent research by Xie et al. (2013), which provided a theoretical investigation on the effectiveness of the formulas based on four assumptions, have identified five maximal listed in Table 2. Specially, our evaluation formula is mutated from the combination of two maximal formulas, namely Naish2 and Russel&Rao. Comparing with coverage-based techniques, our approach narrows the search domain of fault localization by slicing the program, and develops a better formula for suspiciousness computing in fault localization.

5.2. Slicing-based fault localization techniques

Weiser (1982) first proposed program slicing for debugging, Lyle (1984) evaluated variations on program slicing for debugging in his Ph.D. thesis. Since then, many slicing algorithms (both static and dynamic) are proposed for program debugging.

Since then, dynamic slicing has been introduced into program debugging for narrowing the search domain of faults. Agrawal and Horgan (1990) investigated the concept of *dynamic slicing* and examined several approaches for computing dynamic slices. A later study by Agrawal et al. (1995) introduced a fault localization method using execution slices and dicing based on different test cases. Wong and Qi (2004, 2006) presented a combined approach using execution slices and inter-block data dependency to locate the program faults effectively. All these approaches above are based on execution slices which somewhat similar to the coverage of statements during a test.

A lot of interesting studies other than execution slice based have been carried on in fault localization. Al-Khanjari et al. (2005) studied the effectiveness of the critical slicing technique. Zhang et al. (2007) proposed an approach using dynamic slices to locate execution omission errors by switching outcomes of relevant predicates. Zhang et al. (2006) also developed a technique that prunes the dynamic slice on analyzing the values of the variables involved in dynamic slice. Rather than pruning dynamic slices of failed runs, HSS takes advantage of the both full slices of failed runs and execution slices of passed runs for isolating the fault-irrelevant part of the program automatically.

There are also some researches on fault localization using hybrid slices. Lei et al. (2012) applied a new approximate dynamic backward slice (ADBS) to statistical approach which is compared in our empirical study. Although ADBS, which is the intersection of the static backward slice (SBS) and the execution slice, will not lose any faulty code presented in SBS once the faulty code is executed, it usually has a larger size than our hybrid slice spectrum (HSS). Yu et al. (2011) proposed the LOUPE model for fault localization by integrating two suspiciousness that were computed on control dependence model and data dependence model, respectively. Zhang et al. (2009) proposed the CP model to locate fault by computing the propagation of suspicious program states through control flow edges with the suspiciousness score assessed. Unlike HSS, LOUPE computes two suspiciousness and mixes them to construct a new suspiciousness of statements, and CP locates faults along the control flow graph (CFG) while HSS searches faults by the program dependence graph (PDG).

Sun et al. (2007) proposed a heuristic approach using a dynamic slice of one failed and execution slices of three passed test cases. Unlike their approach, our approach requires more than one dynamic slice of failed executions and more than three execution slices. Moreover, our approach requires only one step to locate faults after the suspiciousness are computed by HSS, while Sun's approach needs two phases: refining phase and augmenting phase. Similar to our work, Wen et al. (2011) and Wen (2012) proposed a statistical approach using a mixed slice spectrum to improve the effectiveness of fault localization. The main difference between HSS and Wen's approach can be summarized as follows. Firstly, for all statements in hybrid spectrum, Wen's approach uses the frequency of statements while HSS uses the coverage of statements. Secondly, HSS removes the same execution slice from hybrid slice spectrum while Wen's approach does not. Finally, HSS calculates the suspiciousness using a formula mutated from two maximal evaluation formulas while Wen's approach evaluates suspiciousness similar to Tarantula using the frequency of statements.

6. Conclusions and future work

In this paper, we propose a novel approach (HSS) which can improve the effectiveness of fault localization using hybrid full slices and execution slices. To compare with coverage-based fault localization approaches (i.e., Tarantula, DStar, Naish2, and Russel&Rao) and slice-based approaches (i.e., Set Intersection, Set Union, ADBS, and Wen's), we implemented a prototype tool HSFal, and then designed and conducted an empirical study to answer three research questions. These three questions are designed to evaluate the effectiveness and efficiency of HSS. In summary, the empirical results illustrate that our proposed approach (HSS) is more effective than those compared ones.

In the future, we want to further consider the following issues. Firstly, we want to apply info-flow analysis to our approach to promote the accuracy of our slicing algorithm. Secondly, we want to adopt the methods like roBDDs to deal with the massive information of slice computing, so that we can improve our tool HSFal to locate faults in more large-scale and real-world applications. Last but not least, we want to apply our approach to more subjects written by other programming languages such as C++, C# and conduct more detailed empirical studies; for example, we want to further evaluate the effectiveness of the proposed formula and other formulas for comparisons based on the same spectra, such as our proposed spectra or other spectra.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful and constructive comments to improve the quality of this paper. This work is supported in part by the NSFC Project under Grant Nos. 61202006 and 60970032, the Fundamental Research Funds for the Central Universities under Grant No. 2013QNB17, the Qinlan Project of Jiangsu Province, the Nantong Application Research Plan under Grant Nos. BK2011025 and BK2012023, the University Natural Science Research Project of Jiangsu Province under Grant No. 12KJB520014, the Graduate Training Innovative Projects Foundation of Jiangsu Province under Grant No. CXZZ12_0935, and the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University under Grant No. KFKT2012B29.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2006. A evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006), pp. 39–46.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2009. Spectrum-based multiple fault localization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 88–99.

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C., 2009. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software 82 (11), 1780–1792.

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. SIGPLAN Not 25 (6), 246–256.

Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of the 6th

International Symposium on Software Reliability Engineering (ISSRE 1995), pp. 143–151.
Ali, S., Andrews, J.H., Dhandapani, T., Wang, W., 2009. Evaluating the accuracy of fault localization techniques. In: Proceedings of the 24th IEEE/ACM International Conference on ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 76–87.
Al-Khanjari, Z.A., Woodward, M.R., Ramadhan, H., Kutti, N.S., 2005. The efficiency of critical slicing in fault localization. Software Quality Journal 13 (2), 129–153.
Chen, T.Y., Cheung, Y.Y., 1997. Dynamic program dicing. Journal of Software Maintenance-Research and Practice 9 (1), 33–46.
Chen, M., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks (DSN 2002), pp. 595–604.
DeMillo, R.A., Pan, H., Spafford, E.H., 1997. Failure and fault analysis for software debugging. In: Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC 1997), Washington, DC, USA, pp. 515–521.
Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Software Engineering 10 (4), 405–435.
Gyimóthy, T., Beszédes, Á., Forgács, I.,1999. An efficient relevant slicing method for debugging. In: Proceedings of the 2nd Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 1999), Vol. 1687. Springer, Berlin/Heidelberg/Toulouse, France, pp. 303–321.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, pp. 273–282.
Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL, USA, pp. 467–477.
Jones, J.A., 2004. Fault localization using visualization of test information. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, UK, pp. 54–56.
Lei, Y., Mao, X., Dai, Z., Wang, C., 2012. Effective statistical fault localization using program slices. In: Proceedings of the 36th Annual International Computer Software and Applications Conference (COMPSAC 2012), pp. 1–10.
Lyle, J.R., 1984. Evaluating variations on program slicing for debugging. University of Maryland (Ph.D. thesis).
Masri, W., 2010. Fault localization based on information flow coverage, Software Testing. Verification and Reliability 20 (2), 121–147.
Renieris, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada, pp. 30–39.
Santelices, R., Jones, J.A., Yanbing, Y., Harrold, M.J., 2009. Lightweight fault-localization using multiple coverage types. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada, pp. 56–66.
Sun, J.R., Li, Z.S., Ni, J.C., Yin, F., 2007. Software fault localization basel on testing requirement and program slice. In: Proceedings of International Conference on Networking, Architecture, and Storage (NAS 2007), pp. 168–174.
Wang, X.M., Cheung, S.C., Chan, W.K., Zhang, Z.Y., 2009. Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada, pp. 45–55.
Weiglhofer, M., Fraser, G., Wotawa, F., 2009. Using spectrum-based fault localization for test case grouping. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 630–634.
Weiser, M., 1982. Programmers use slices when debugging. Communications of the ACM 25 (7), 446–452.
Wen, W., Li, B., Sun, X., Li, J., 2011. Program slicing spectrum-based software fault localization. In: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), pp. 213–218.
Wen, W.,2012. Software fault localization based on program slicing spectrum. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012). IEEE Press, Zurich, Switzerland, pp. 1511–1514.
Wong, W.E., Qi, Y., 2004. An execution slice and inter-block data dependency-based approach for fault localization. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), Busan, Korea, pp. 366–373.
Wong, W.E., Qi, Y., 2006. Effective program debugging based on execution slices and inter-block data dependency. Journal of Systems and Software 79 (7), 891–903.
Wong, W.E., Qi, Y., Zhao, L., Cai, K.Y., 2007. Effective fault localization using code coverage. In: Proceedings of the 31st Annual International Computer Software and Applications Conference, Vol. I (COMPSAC 2007), Beijing, China, pp. 449–456.
Wong, W.E., Wei, T., Qi, Y., Zhao, L., 2008. A crosstab-based statistical method for effective fault localization. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008), pp. 42–51.
Wong, W.E., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software 83 (2), 188–208.
Wong, W.E., Debroy, V., Li, Y.H., Gao, R.Z., 2012. Software fault localization using dstar (d*). In: Proceedings of the 6th IEEE International Conference on Software Security and Reliability (SERE 2012), Gaithersburg, MD, USA, pp. 21–30.
Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology 22 (4), 1–40.
Yu, K., Lin, M., Gao, Q., Zhang, H., Zhang, X.,2011. Locating faults using multiple spectra-specific models. In: Proceedings of the 26th ACM Symposium on Applied Computing (SAC 2011). ACM, TaiChung, Taiwan, pp. 1404–1410.
Zhang, X., Gupta, R., Zhang, Y., 2004. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, UK, pp. 502–511.
Zhang, X., He, H., Gupta, N., Gupta, R.,2005. Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging (AADEBUG 2005). ACM, pp. 33–42.
Zhang, X.Y., Gupta, N., Gupta, R., 2006. Pruning dynamic slices with confidence. SIGPLAN Not 41 (6), 169–180.
Zhang, X., Tallam, S., Gupta, N., Gupta, R., 2007. Toward locating execution omission errors. SIGPLAN Not 42 (6), 415–424.
Zhang, Z.Y., Chan, W.K., Tse, T.H., Jiang, B., Wang, X.M., 2009. Capturing propagation of infected program states. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009), Amsterdam, Netherlands. Association for Computing Machinery, pp. 43–52.

Xiaolin Ju is a Ph.D. candidate in computer science at China University of Mining and Technology and a lecturer at Nantong University, 221116, Xuzhou, China. He received his B.S. degree in information science from Wuhan University in 1998, and M.Sc. degree in computer science from Southeast University in 2004. His research interests are mainly in software testing, such as combinatorial testing, regression testing, and fault localization.

Shujuan Jiang is a professor at School of Computer Science and Technology, China University of Mining and Technology, 221116, Xuzhou, China. She received her Ph.D. degree in computer science from Southeast University in 2006, and was a visiting scholar at the Georgia Institute of Technology in 2008-2009. Her current research interests include software dependability and reliability, software maintenance and software development methodology.

Xiang Chen received the B.Sc. degree at school of management from Xi'an Jiaotong University, China in 2002. Then he received the M.Sc. and Ph.D. degrees in computer science from Nanjing University, China in 2008 and 2011. Now he joined the Department of Computer Science and Technology at Nantong University as an assistant professor. His research interests are mainly in software testing, such as combinatorial testing, regression testing, and fault localization.

Xingya Wang is a Ph.D. candidate in computer science at China University of Mining and Technology, 221116, Xuzhou, China. He received his B.S. degree in computer science from China University of Mining and Technology. His current research interests include program slicing, software testing and analysis.

Yanmei Zhang is assistant professor at China University of Mining and Technology, 221116, Xuzhou, China. She received her Ph.D. degree from China University of Mining and Technology, China in 2012. Her current research interests include Software Analysis and Testing, Web Application Testing, Fault Localization Technique.

Heling Cao is a Ph.D. candidate in computer science at China University of Mining and Technology, 221116, Xuzhou, China. She received her B.E. and M.E. degree in computer science from Zhengzhou University. Her current research interests include program slicing, software testing and analysis, and software debugging.