The Journal of Systems and Software 122 (2016) 144–154

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Fault localization using disparities of dynamic invariants

Xiaoyan Wang[a,b,c], Yongmei Liu[b,*]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
[b] Department of Computer Science, Sun Yat-sen University, Guangzhou, China
[c] Department of Information Management and Information System, Nanjing Audit University, Nanjing, China

ARTICLE INFO

Article history:
Received 19 April 2015
Revised 8 September 2016
Accepted 12 September 2016
Available online 13 September 2016

Keywords:
Software debugging
Fault localization
Dynamic invariant
Program analysis

ABSTRACT

Violations of dynamic invariants may offer useful clues for identifying faults in programs. Although techniques that use violations of dynamic invariants to detect anomalies have been developed, some of them are restrained by the high computational cost of invariant detecting, false positive filtering, and redundancy removing, and others can only discover a few specific types of faults under a complete monitoring environment. This paper presents a novel fault localization approach using disparities of dynamic invariants, named FDDI. To make more efficient use of invariant detecting tools, FDDI first selects highly suspect functions via spectrum-based fault localization techniques, and then applies invariant detecting tools to these functions one by one. For each suspect function, FDDI uses variables that are involved in dynamic invariants that do not simultaneously hold in a set of passed and a set of failed tests to do further analysis, which reduces the time cost in filtering false positives and redundant invariants. Finally, FDDI locates statements that are data-related to these variables. The experimental results show that FDDI is able to locate 75% of 360 common faults in utility programs when examining up to 10% of the executed code, while Naish2, Ochiai and Jaccard all locate around 53%.

1. Introduction

Dynamic invariants are relations among variables that are observed to hold at certain locations in some runs of a program (Nguyen et al., 2012). They can be inserted as assertion statements to detect abnormal behaviors of programs, collected to generate likely documentation and formal specifications, and used in program understanding (Zeller, 2009), etc. In particular, the violation of dynamic invariants can offer useful clues for fault localization and give better explanation for the localization result. However, current applications of dynamic invariants in automated program debugging are not efficient due to the high computational cost of detecting dynamic invariants. For Daikon (Ernst et al., 2001), it maintains an invariant pattern library that limits the detection of invariants with rich expressive power. GenInv (Nguyen et al., 2012) combines mathematical techniques to bring new capabilities to find more expressive dynamic invariants while increasing the complexity of the detection. For Diduce (Hangal and Lam, 2002) and ClearView (Perkins et al., 2009), they can only detect and patch specific types of errors due to the monitoring mechanisms they require. Currently, the work of Sahoo et al. (2013) combines dynamic program invariants with more sophisticated filtering techniques to identify a set of dynamic invariants that hold in selected passed runs but do not hold in failing tests, and returns the locations of the dynamic invariants as the localization results.

In this paper, we present a novel automated fault localization approach via using disparities of dynamic invariants, named *FDDI*, to locate the root causes of faulty programs via disparities of two sets of dynamic invariants generated from passing and failing test cases respectively. The intuition behind the idea is that a variable is likely to be related to the root cause if it is involved in relations that do not simultaneously hold in a certain number of failed runs and passed runs. In FDDI, spectrum-based fault localization (SBFL) techniques are first applied in function level to find suspect functions. Dynamic invariants are then yielded in these functions one by one. After detecting dynamic invariants, how does FDDI further locate the source code lines with bugs? In the following segment, we demonstrate how to use disparities of dynamic invariants to locate faults by using a snippet that contains a bug at line 5 where "if(d < 6)" should actually be "if(d < 5)", as shown in Fig. 1. Consider the function `f()` in Fig. 1 and suppose we have only one invariant schema: i < j, where i and j are metavariables. At the entry of `f()`, instantiating this schema produces 12 concrete potential invariants:

```
a<b, a<c, a<d, b<a, b<c, b<d, c<a, c<b, c<d, d<a, d<b, d<c.
```

* Corresponding author.
E-mail addresses: wangxy25@mail2.sysu.edu.cn, xywang@nau.edu.cn (X. Wang), ymliu@mail.sysu.edu.cn (Y. Liu).

http://dx.doi.org/10.1016/j.jss.2016.09.014
0164-1212/

	int f(int a, int b, int c, int d)	After <a=2, b=1, c=2, d=4>:
	{	a<d, b<a, b<c, b<d, c<d
①	int x, y;	After <a=1, b=2, c=5, d=3>:
②	if (c < 5)	a<d, b<c, b<d
③	x = a + b;	After <a=4, b=1, c=1, d=4>:
	else	b<d (passed tests)
④	x = a - b;	
⑤	if (d < 6) // d < 5	After <a=2, b=1, c=3, d=5>:
⑥	y = a * b;	a<c, a<d, b<a, b<c, b<d, c<d
	else	After <a=5, b=4, c=4, d=5>:
⑦	y = a / b;	b<a, b<d, c<d
⑧	return x + y;	After <a=4, b=4, c=4, d=5>:
	}	b<d, c<d (failed tests)

Fig. 1. Motivational example.

The upper right portion of Fig. 1 shows the set of invariants that have not been falsified by any of the preceding passed tests. Therefore, the last potential invariant "b < d" survived all three passed executions of function `f()`. Similarly, as shown in the bottom right portion of Fig. 1, the last potential invariant "b < d" and "c < d" survived all three failed executions of `f()`. As we can see, two likely invariant sets {b < d} and {b < d, c < d} are yielded via respectively running a passed test suite and a failed test suite, and the disparity between these two sets is {c < d}. FDDI extracts variable "c" and "d" from the disparity and can locate suspect statements line 2 and line 5 via using these variables. As a result, line 5 that is exactly the root cause is captured by our method.

FDDI is neither for certain error types nor under any monitoring environment. To be effective, it consists of two stages in its application. In the first stage, the block hit spectrum based technique is applied to rank functions by their suspiciousness, and *n* most suspicious functions will be selected for further analysis. By doing this, our method can concentrate on a small portion of the program at a time. Also, this overcomes the problem that a large number of variables in a program will bring existing invariant detecting techniques to their knees. In the second stage, for each suspicious function, two sets of dynamic invariants are first yielded by running a passed and a failed test case suite respectively. The statement-based reduction strategy (Yu et al., 2008) is applied to generate the passed and the failed test case suite. Variables in the difference of the two sets are then used to find data-related statements in the function by static analysis, which reduces the computational cost of filtering redundant and spurious invariants. These statements will be returned to developers in the order of the suspiciousness of the functions where they appear and in the order in which they appear in the same function.

The main contributions of this paper include: (1) We propose to use variables in the disparity of two dynamic invariant sets respectively generated from a failed and a passed test suite to locate bugs in a faulty program. (2) To reduce the high computational cost of current invariant detecting methods, FDDI employs existing dynamic invariant detecting tool, like Daikon, to generate dynamic invariants in the scope of one highly suspect function each time, and block hit spectrum based techniques are applied to rank these functions of a faulty program. (3) The experimental result shows that FDDI locates 75% of 360 common faults in 6 real-life utility programs when examining up to 10% of the executed code, while Naish2, Ochiai and Jaccard all locate around 53%.

The reminder of this paper is organized as follows: Section 2 investigates the proposed approach in detail. Section 3 evaluates the proposed approach and presents the experiment results. Section 4 presents the related work. Section 5 concludes this paper and highlights some future work.

2. FDDI

In this section, we first illustrate the top-level view of FDDI and present primary parts of FDDI. We then describe and analyze the algorithm of FDDI.

2.1. Top level view of FDDI

Fig. 2 depicts the overview of FDDI. The inputs of FDDI include a faulty program *app*, a test case suite *TS* and the component granularity *FL*. The output is a debugging report *R*. As we can see from Fig. 2, FDDI has eight primary components.

- *call DCC*: call DCC to calculate the suspiciousness of each executed function in *app*. DCC (Perez et al., 2014) is a dynamic coverage based multiple granularity fault localization technique. To employ DCC to rank functions, we need to set both the initial granularity and finial granularity to be the function level. As a result, a function sequence $\vec{F}$ is generated from this part.
- *select function*: select a function *f* from $\vec{F}$ in decreasing order of suspiciousness. FDDI will give deeper analysis of *f* via its dynamic invariants. FDDI proceeds to "generate report" part, if all functions with expected suspiciousness are selected out from $\vec{F}$.
- *refine tests*: find test cases that invoke *f* and remove redundant tests that cover the same code of *f*. FDDI distinguishes a test suite *RTS* that all test cases in it execute the code in *f* from the original test suite *TS*, because not all the tests in *TS* invoke the highly suspect function *f*. To save running time, FDDI removes redundant test cases that cover the same code of *f*, because these tests may not refine the generated dynamic invariant set of *f*. Besides, FDDI identifies a passed test suite RTS_p and a failed test suite RTS_f in *RTS*.
- *analyze call site*: analyze call sites in function *f*, and find the locations *CS* of loop and return statements and variable set *V* for each block in *f*. Instrumenting new call sites in *f* is required to obtain more dynamic invariants, because Daikon is applied in FDDI and it only generates dynamic invariants in the entrance and exit sites of a function by default.
- *instrument f*: instrument a dummy procedure for each executed loop in *f*, and one call site of the dummy procedure is instrumented in the loop head and another call site is added in the loop tail. Parameters of the dummy procedure are variables that hold in the loop. For each return statement, a dummy procedure is constructed and its call site is before the return statement. Detailed description can be seen in Section 2.2.
- *detect dynamic invariant*: detect two dynamic invariant sets *S* and S' for function *f* through respectively running the passed test suite RTS_p and the failed test suite RTS_f in Daikon.
- *analyze disparity*: find different dynamic invariants between *S* and S', and extract variables involved in these dynamic invariants. If any, FDDI finds suspect statements in *f* that are data dependent with these variables. Otherwise, FDDI collects all executed statements in *f* in a random selected failed test of *app*. FDDI returns back to "select function" part for the next suspect function.
- *generate report*: generate a debugging report of *app*. It consists of suspect statements in each selected function and corresponding variables in disparities. All suspect statements are returned in the report not only in decreasing order of suspiciousness of their functions but also in increasing order of appearance in their functions. Moreover, these variables with disparities could assist users in understanding the bug when they observe localization results in each function.

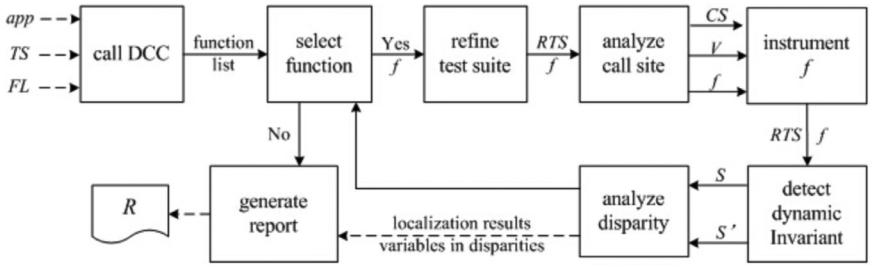


Fig. 2. Top-level view of FDDI.

2.2. Algorithm of FDDI

In this section, we show the algorithm of FDDI and describe how to construct a dummy procedure and insert call sites for it.

As shown in Algorithm 1 , FDDI_Debugging is the algorithm of FDDI and it requires two inputs: the faulty program *app* and its a test suite *TestSuite*. $\mathcal{R}$ stores the fault localization results and is initialized as an empty list (line 1).

$\mathcal{M}$ is a sorted list of functions and obtained by DCC (line 2). The initial granularity and final granularity are set to functions, as long as FDDI calls DCC. In Perez et al. (2014), DCC is implemented by incorporating Ochiai approach (see formula 2.2.1), a spectrum-based fault localization method. In our experiments, we apply two other spectrum-based fault localization methods Naiish2 and Jaccard to implement DCC (see formula (2.2.2) and (2.2.3)),

$$Ochiai(f) = \frac{a_{11}(f)}{\sqrt{(a_{11}(f)+a_{01}(f)) \times (a_{11}(f)+a_{10}(f))}} \tag{2.2.1}$$

$$Naish2(f) = a_{11}(f) - \frac{a_{10}(f)}{a_{10}(f)+a_{00}(f)+1} \tag{2.2.2}$$

$$Jaccard(f) = \frac{a_{11}(f)}{a_{11}(f)+a_{01}(f)+a_{10}(f)} \tag{2.2.3}$$

where $a_{ij}(f) = |\{e | x_{ef} = i \wedge r_e = j\}|$, $i, j \in \{0, 1\}$ and f is a function in program P. Also, $x_{ef} = i$ indicates whether function f has been invoked ($i = 1$) in the execution of run e or not ($i = 0$). Similarly, $r_e = j$ represents whether a run e was faulty ($j = 1$) or not ($j = 0$). This similarity coefficient ranks functions of *app* with respect to their likelihood of containing bugs in *app*. The comparison results of these three methods show whether the improvement made by FDDI can be noticeable when using different SBFL formulas for function selection. In Section 3, we evaluate these three different function selection methods.

Algorithm 1: FDDI_Debugging.

input : *app*: the source code of a faulty application
TestSuite: all test cases
n: the number of suspect functions.
output: $\mathcal{R}$: a debugging report

1 $\mathcal{R} \leftarrow [\ ]$
2 $\mathcal{M} \leftarrow$ *DCC(app, TestSuite, function_level, function_level)*
3 $\mathcal{F} \leftarrow$ selectSusp($\mathcal{M}$, *n*)
4 **foreach** f ***in*** $\mathcal{F}$ **do**
5 $(T_p, T_f) \leftarrow$ refineTests(*TestSuite*, f)
6 $A_f \leftarrow$ runTests(T_p, T_f)
7 $(\mathcal{C}, \mathcal{V}) \leftarrow$ getCallSite(f, A_f)
8 instrument(f, $\mathcal{C}$, $\mathcal{V}$)
9 $(S, S') \leftarrow$ detectInvariant(f, T_p, T_f)
10 **if** $(l, d) \in S \wedge (l', d') \in S' \wedge l == l' \wedge d \neq d'$ **then**
11 $(\mathcal{C}', \mathcal{V}) \leftarrow$ analyzeDisparity(S, S')
12 $\mathcal{L} \leftarrow$ locateStatement($\mathcal{V}$)
13 $\mathcal{R}$.append($\mathcal{L}$, $\mathcal{C}'$, $\mathcal{V}$)
14 **return** $\mathcal{R}$

n suspect functions are then selected from the sorted function list $\mathcal{M}$ and stored in $\mathcal{F}$ (line 3). In our experiments, n is set according to statistics about the faulty function localization, as shown in Section 3.2.1. For each suspect function f in $\mathcal{F}$, FDDI does the following (lines 5–13).

Firstly, in *TestSuite*, FDDI finds a test suite in which the code of f is executed in its all tests. Besides, test cases in the suite are divided into two parts: a passed test suite T_p and a failed test suite T_f, and T_p and T_f are simplified according to the statement-based reduction strategy (Yu et al., 2008) (line 5).

Secondly, obtain coverage information A_f of function f via running tests in T_f and T_p (line 6). FDDI then identifies a call site set $\mathcal{C}$ in line with the A_f, it tells us where to generate dynamic invariants in f. Locations at the beginning of executed loop bodies and followed the loop body are collected into $\mathcal{C}$. Also, locations just before executed return statements are collected as well. Meanwhile, variables used in these executions are extracted into set $\mathcal{V}$ (line 7), it tells us which variables in f will be involved in the following generated dynamic invariants.

Thirdly, FDDI employs Daikon to detect dynamic invariants for f. Since Daikon only generates dynamic invariants in the entrance and the exit site of f by default, FDDI uses dummy procedures to construct more dynamic invariant generating sites (line 8). To obtain dynamic invariants in an expected location, we present how to use the dummy procedure in the following snippet,

```
int get_token(tokenstream str) {
    ...
    statements;
    ...
    // you desire to compute an invariant here
    statements;
    ...
}
```

if we want to obtain more dynamic invariants from a location that we desire to, then the original code has to be changed into

```
// dummy procedure
void calculate_invariants(int x, token stream, float y) {}
int get_token(tokenstream str) {
    ...
    statements;
    ...
    calculate_invariants(x, stream, y); //desired position
    statements;
    ...
}
```

A dummy procedure `calculate_invariants` is defined before function `get _token`. There is no function body for `calculate_invariants`, and its parameters are variables that are expected to be observed in `get_token`. By doing this, Daikon (Ernst et al., 2001) will produce properties that held at the desired call site. With instrumented f, FDDI invokes Daikon to generate a binary tuple set S (S') by running test in T_p (T_f) (line 9). For each element (l, d) in S (S'), l is a location of f and d is a generated dynamic invariant in l.

At last, for $(l, d) \in S$ and $(l', d') \in S'$, if $l == l'$ and $d \neq d'$ then locations as l (l') are stored in $\mathcal{C}'$ (line 11). As we can see, these disparities between S and S' are likely caused by the root cause, therefore, FDDI captures statements in f that are data dependent with variables in $\mathcal{V}$ via both forward and backward slicing. The intersection of all locating results for reference variables will be treated as the localization result In the debugging report $\mathcal{R}$, located statements by FDDI will be reported at first. Then, for each suspect function f, call sites in $\mathcal{C}'$ and variables in $\mathcal{V}$ are also returned, which can assist the user to understanding why localized these statements.

2.3. Complexity of FDDI

The time complexity of FDDI_Debugging is determined by three aspects:

Firstly, refine test cases for each function f (line 5). FDDI needs to identify a passed test suite T_p and a failed test suite T_f for each selected function f from test suite T, and the rule of the selection for test cases in both T_p and T_f is that the more similarity of test cases the more accurate localization results will be obtained. Divide T into T_p and T_f according to the test result (passed/failed), and eliminate redundant test cases with the same code coverage for T_p and T_f respectively. Its time complexity is $O((|T| + |T_p| + |T_f|) \times l) \leq O(|T| \times l)$, l is the average number of the code lines in selected functions.

Secondly, the time complexity of detecting dynamic invariants in f is $O(p \times |\mathcal{V}|^3 \times |\mathcal{C}|) \leq O(p \times |\mathcal{V}|^3 \times l)$ (line 9), in which p is the size of the invariant pattern of Daikon, $\mathcal{V}$ is the set of variables in f, $\mathcal{C}$ is the set of call sites in f and $|\mathcal{C}|$ is less than the number of statements of function f, i.e. l. As we know, Daikon uses the brute force solution to obtain instantiations (dynamic invariants) from its pattern library, and each pattern in the current library involves three variables at most (Ernst, 2000; Zeller, 2009). Therefore, the complexity of this part is $O(p \times |\mathcal{V}|^3 \times l)$.

Thirdly, the time complexity of finding dynamic invariants with disparities in the same call site in f is $O(|S| \times |S'|)$, S and S' are generated dynamic invariants of f by running T_p and T_f respectively. Suppose that all patterns hold for all variables in each call site of f, then FDDI needs to compare $p \times |\mathcal{V}|^3 \times l$ times, in such extreme case, to obtain dynamic invariants with differences. Hence, its time complexity is $O(|S| \times |S'|) \leq O(p \times |\mathcal{V}|^3 \times l)$ (line 11).

As a result, the time complexity of FDDI_Debugging is $O(n \times (|T| \times l + p \times |\mathcal{V}|^3 \times l + p \times |\mathcal{V}|^3 \times l)) = O(n \times l \times (|T| + p \times |\mathcal{V}|^3))$, n is the number of selected suspect functions. As we can see, the time complexity of FDDI is

The space complexity of FDDI_Debugging is primarily decided by the average number of code lines l in selected functions and the number of dynamic invariants ($|S| + |S'|$) in f.

3. Experiments and results

In this section, we evaluate the performance of the FDDI approach for real projects. First, we present experimental setup. Then, we discuss results of our empirical study, and demonstrate explanatory capabilities of FDDI through two examples. Finally, we finish this section with a threats to validity discussion.

Table 1
Experimental subjects.

Subject	Version	KLOC	#Fun	#Avg	Test cases	Coverage
sed	4.1.5	7.2	255	28	370	64.21%
gzip	1.3.13	5.7	104	55	215	64.17%
space	ORACOLO2	9.2	136	68	174	50.03%
grep	2.6	13.3	146	91	669	59.73%
flex	2.5.33	14.3	162	92	525	74.6%
make	3.8.1	36	268	135	307	63.45%

Table 2
Fault injectors used.

Name	Function
Negate decision (ND)	Negate the condition in an if or loop statement
Replace constant (RC)	Off-by-one replacement of integer constant
Delete statement (DS)	Delete a statement
Replace operator (RO)	Replace an operator by another operator
Replace variable (RV)	Replace a variable in a statement
Multiple fault (MF)	A real fault or a composition of above injectors

3.1. Experimental setup

In this part, subjects under analysis and evaluation metrics are first presented. The comparison methods are then presented.

3.1.1. Subjects

For our empirical study, six subjects written in C program language were considered:

- sed[1] - a stream editor used to filter text.
- gzip[2] - a compression utility.
- space[3] - an interpreter for an array definition language (ADL).
- grep[4] - a text search tool used to find one or more input files for lines containing a match to a specified pattern.
- flex[5] - a tool for generating scanners.
- make[6] - a tool controls the generation of executables.

Table 1 presents the details of each subject. The number of lines of code for each subject is listed in KLOC column. *#Fun* is the number of functions in each subject. *#Avg* is the average number of code lines in functions of each subject. As we can see, the scale of selected subjects ranges from 28 to 135 lines of code per function in average. Test count and coverage percentage were collected with GCOV, a test code coverage tool being used in conjunction with GCC.

To evaluate the performance of FDDI when tracking a single fault and multiple simultaneous faults, our experiments were performed using 60 faulty versions per program. Since the programs are bug-free, we injected common mistakes in the programs using fault injectors shown in Table 2, by reference to injectors in Steimann et al. (2013). Each fault injector was used to create 10 random injections into each subject. This gave us up to 60 fault injections per subject. In total, we evaluated 360 faulty versions.

Multiple faults are considered in our experiments as shown in Table 2. For single and multiple faults, there is almost no difference in FDDI. The reason is shown as following. First, for multiple faults, we select failed test cases that executed all the statements with faults. Second, the ranking result of a multiple faults is the average of the sum of the ranking result of each fault.

[1] sed - http://www.gnu.org/software/sed/.
[2] gzip - http://www.gnu.org/software/gzip/.
[3] space - http://sir.unl.edu/portal/bios/space.php.
[4] grep - http://www.gnu.org/software/grep/.
[5] flex - http://flex.sourceforge.net.
[6] make - https://www.gnu.org/software/make/.

3.1.2. Metrics

The metrics gathered were the percentage of the number of LOCs needed to be inspected until the fault is located. Therefore, the ranking of the statement containing the fault must be calculated at first. It assumes that the user do the inspection in an ordered manner, starting from the statements ranked highly. There are two special types of errors that we have to emphasize the calculation of their rankings. For the omission fault, the ranking of the root cause is calculated by averaging the ranking of statements that around the root cause. With respect to multiple faults, the final ranking result is represented by the average of the sum of the rankings of each root cause.

Although such metrics is questioned about its usefulness for final localization (Parnin and Orso, 2011), it is a better way for us to compare the efficiency of FDDI with that of other efficient methods, especially in large-scale. In the further application, we would like to use some visualization techniques, like HTML5 (Gouveia et al., 2013), to aid users' understanding of the localization results.

3.1.3. Comparison methods

FDDI has three variants via respectively combining Daikon with Naish2, Ochiai and Jaccard. In this way, we can see whether the improvement made by FDDI can be noticeable when combining with more effective spectrum-based block hit techniques. We show the comparison results of these three invariants in Section 3.2.1. In addition, we compared FDDI with Naish2, Ochiai and Jaccard, because Naish2, Ochiai and Jaccard are not equivalent relations, which are proved by Lee et al. (2011) . Besides, the recruited approaches apply to locating common faults illustrated in Section 3.1.1, and their localization results both can be evaluated by the metrics mentioned in Section 3.1.2.

From the developers perspective, our method and the spectrum-based fault localization method are comparable. The goal of a fault localization tool is to let the developer find the root cause as quickly as possible and know why it is the root cause as clearly as possible. Therefore, we compare the ranking result of FDDI with the spectrum-based fault localization techniques. Besides, disparities of dynamic invariants in our method provide explanations for the localization results.

3.2. Results and discussion

The results of our experiments are illustrated in this section. Firstly, we show the evaluation of the dependence of outcomes on the *n* highest ranked functions and the evaluation of time requirements. Also, overall results are presented by comparing FDDI with Naish2, Ochiai and Jaccard. Secondly, individual results of FDDI technique are depicted by comparing with that of the Naish2. Lastly, we present the comparison results of FDDI with Naish2 on top few lines of code examination effort.

3.2.1. Overall results

FDDI depends on two steps: preselecting the n most suspicious functions and dynamic invariant detection.

For selection of functions, we contrast the results of Naish2, Ochiai and Jaccard applied in calculating the risk of each function in our experiment. Naish2, Ochiai and Jaccard are identified as different equivalent relations by Naish, and Naish2 is one of the five maximal formulas under the single-fault scenario (Xie et al., 2013a). Fig. 3 shows the comparison of the function-localization results of FDDI by respectively employing Naish2, Ochiai and Jaccard. The x-axis of each plot in Fig. 3 represents the percentage of functions of each faulty version to be examined, and the y-axis means the percentage of faults located within the given function examination range. Naish2(f), Ochiai(f) and Jacard(f) mean that

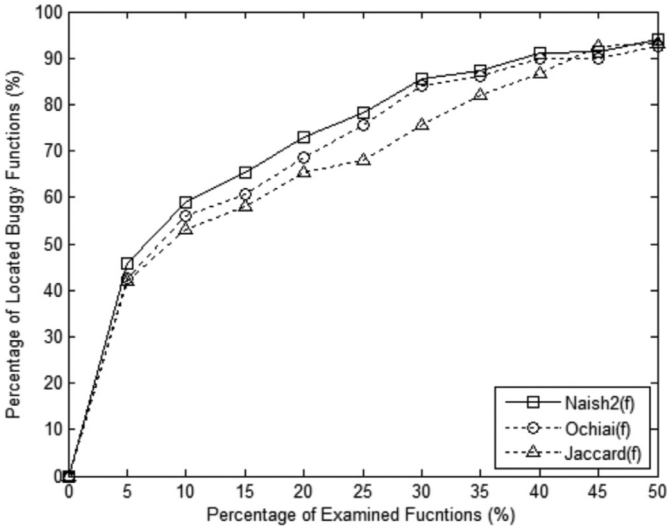


Fig. 3. Comparison of three SBFL-based function selecting methods.

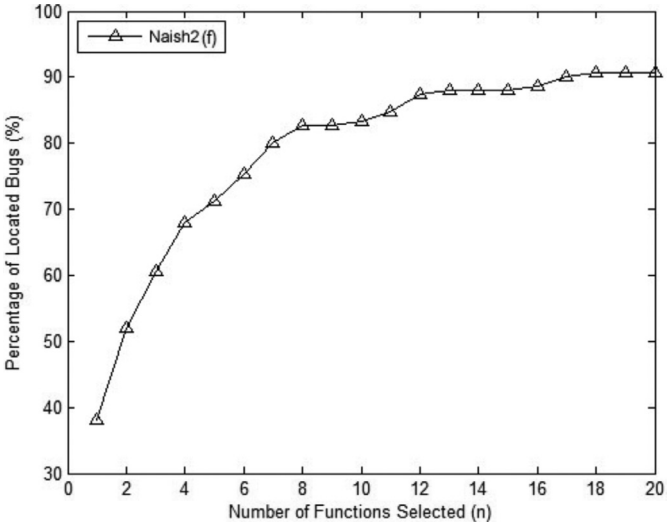


Fig. 4. Evaluation of the dependence of the accuracy outcome on the threshold *n*.

we respectively apply the corresponding fault localization methods to locate suspect functions. As we can see from Fig. 3, the improvement made by function selection is not noticeable when using maximal formulas in the preselecting stage, but Naish2 takes slightly advantage over Ochiai and Jaccard in the function examination range from 0 to 43%. However, Jaccard performs a little better than Naish2 and Ochiai in the function examination range from 43% to 47%. As a result, Naish2 is employed to incorporate in FDDI for function selection.

In addition, Fig. 4 shows the evaluation of the dependence of accuracy on the threshold n. The x-axis of each plot in Fig. 4 represents the number of functions to be examined, and the y-axis means the percentage of faults located within the selected number of suspicious functions. As we can see, FDDI can locate 83% of the 360 evaluated common bugs, when at most 8 suspicious functions are selected by Naish2(f) that analyzing coverage traces for functions. In this stage, the time cost is less than 200 seconds in average in our experiment, because the number of test cases used in each faulty version is less than 150 in average.

For dynamic invariant detection, Table 3 shows a comparison for the average execution time for both FDDI and directly using Daikon on complete faulty programs. It can be seen that FDDI is efficient. Besides, In FDDI, the Daikon result is the intermediate

Table 3
Execution time.

Subject	#Avg_f	#Avg_c	Avg_v (%)	FDDI-s2 (m)	Whole-s2 (m)	FDDI time reduction (%)
sed	11	28	33.6	7.1	55.2	87.3
gzip	6	55	31.4	5.6	43.7	87.2
space	7	68	20.0	4.4	60.7	92.8
grep	13	91	22.5	28.2	398.5	92.9
flex	19	92	20.3	33.1	529.4	93.7
make	23	135	18.3	49.7	1214.9	95.9

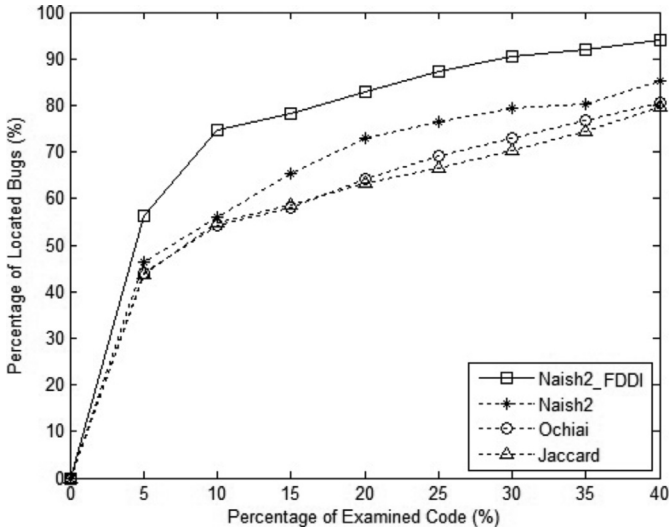


Fig. 5. Overall results.

result and we use them to find disparities which assist FDDI finds suspicious statements and helps users to identify the root causes. In Table 3, #Avg_f represents the average number of functions detected until the faults are located for each subject. #Avg_c is the average number of executed lines of the code per function and the function scales are ranges from 28 to 135. #Avg_v is the proportion of program variables selected for each function according to FDDI. In addition, FDDI-s2 and Whole-s2 respectively represents the time cost of FDDI in the dynamic invariant detection stage and the time cost of detecting dynamic invariants for complete faulty programs. On average, it can be seen that applying Daikon to complete programs takes much more time than using FDDI. The FDDI approach reduces the execution time of the second stage at least 87.2% for 6 subjects, and it reduces time more than 92% especially for bigger programs.

In Fig. 5, we show the results of FDDI by incorporating Naish2 for the function selection and the comparison results of FDDI with Naish2, Ochiai and Jaccard on 360 versions of 6 programs. The x-axis of each plot in Fig. 5 means the percentage of executed statements of each faulty version to be examined, and the y-axis represents the percentage of faults located within the given code examination range. As we can see, FDDI takes advantage over Naish2, Ochiai and Jaccard in the showed entire examined code range. Moreover, when 10% of the code is examined, FDDI captures 75% of the bugs while Naish2, Ochiai and Jaccard all capture about 55% of the bugs. In Fig. 5, Naish2 performs better than Ochiai and Jaccard in the code examined range from 0% to 40%. Also, Ochiai slightly outperforms Jaccard in the entire examined code range. This is consistent with the empirical study results of Abreu et al. (2007) and the theoretical study results of Xie et al. (2013a).

3.2.2. Individual results

To emerge more concrete results, we then show the comparison results on four individual subjects, sed, space, flex, and make in Fig. 6 and Fig. 7. These four subjects have different function scales as shown in column *Avg_c* of Table 3.

As we can see from Table 1, sed, space, flex, and make respectively consist of 255, 136, 162 and 268 functions and have 28, 68, 92 and 135 lines of the code per function in average. Since FDDI performs better via incorporating Naish2, we only show the results of such invariant of FDDI in the following results. Besides, we compared the results of FDDI only with Naish2, because Naish2 outperforms both Ochiai and Jaccard in the overall result as shown in Fig. 5. For clarity, we only show the comparison results between our method and Naish2 for each subject. In Fig. 6, FDDI takes advantage over Naish2 for each subject, and FDDI is basically consistent with the growth trend of the Naish2.

In Fig. 6(c), it shows the results of FDDI, Naish2, Ochiai and Jaccard on 60 faulty versions of flex. FDDI exhibits observably advantage over Naish2 in the code examination range from 0 to 40%, because high coverage of functions is achieved for faulty versions of flex. As shown in Table 1, the code coverage of flex is 74.6% that is higher than other subjects. When a programmer would like to examine at most 10% of the code, FDDI identifies about 72% of the bugs while Naish2 captures 44% of the faults in flex.

Let us move on to other three subjects. In Fig. 6(a), for sed, it can be seen that FDDI takes advantage over Naish2 in the code examination range forms 0% to 50%. FDDI catches about 66% of the bugs when examining up to 10% of the executed code, while Naish2 locates 57% of the bugs. Also, FDDI catches about 93% of the bugs when examining up to 50% of the executed code, while Naish2 locates 70% of the bugs. For space in Fig. 6(b), FDDI outperforms Naish2 in the examined code range from 0 to 50%. When examining up to 5% of the code, FDDI identifies 80% of the bugs, while Naish2 captures 70% of the bugs. For make in Fig. 6(d), the localization results of FDDI with examining less percentage of the execute code is much worse than that for space, because the function preselecting results of space is 7 in average that is much better than that of make, as shown in Table 3. For example, if we expect to locate 70% of the bugs of make, we have to examine up to about 15% of the executed code, while only 4% of the executed code needed to be examined for space.

Although FDDI obtains better localization results than Naish2 for subjects with different function scales, there exists different gaps between FDDI and the maximal formula Naish2 for different subjects. This may be caused by following reasons: first, the function coverage impacts the ranking of the functions containing the faulty code. For flex, the average percentage of covered functions is 75%, while that is around 45% for the other programs. Since the higher function coverage, the more executed code for each faulty version. Second, when the functions with the fault has highly risk value of being faulty, it may be located by examining less lines of code by FDDI than other SBFL techniques.

For SBFL techniques, the number of statements needed to be examined until we hit the bug is the less the better. Therefore, we show the results on top 10, 20 and 30 lines of code examination effort in Fig. 7. Overall, FDDI obtains better localization results than Naish2 when examining at most 10 lines of the executed code for these four subjects. For flex and space, FDDI shows observably advantage over Naish2 when we only expect to examine less 20 or 30 lines of the executed code, and the improvement made by FDDI reaches up to 23% at most. For sed and make, FDDI and Naish2 perform almost comparably in above cases except on top 10 lines.

3.3. Explanatory capabilities

Statements related to the disparity of dynamic invariants respectively generated by a failed and a passed test suites could be good inspecting start points for developers to do further localization. Therefore, FDDI has potential to provide explanation for the

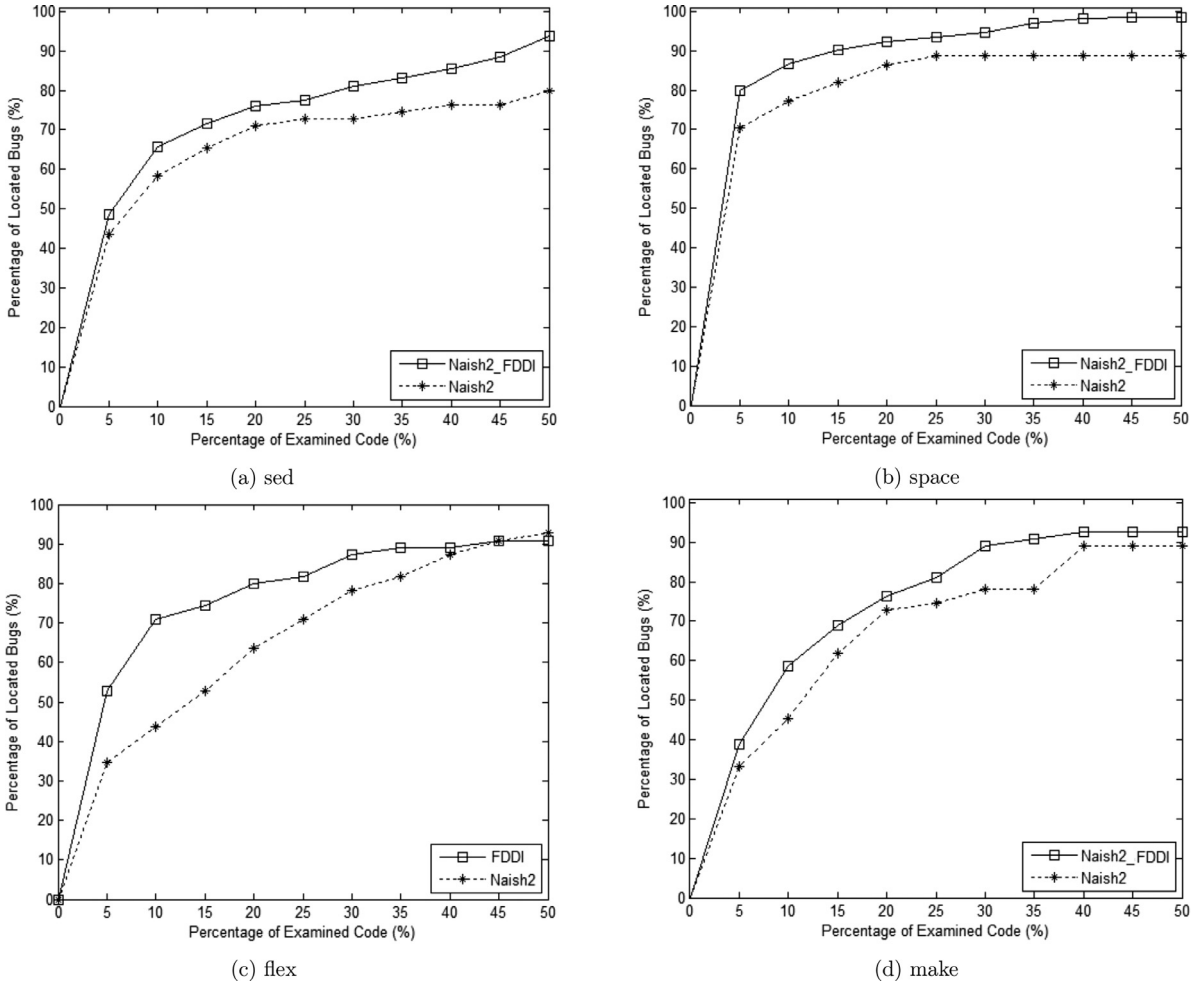


Fig. 6. Individual results.

located statements, and the explanation may offer useful clues for finally pinpointing the root causes. In this section, we illustrate this with two bugs from our experiments.

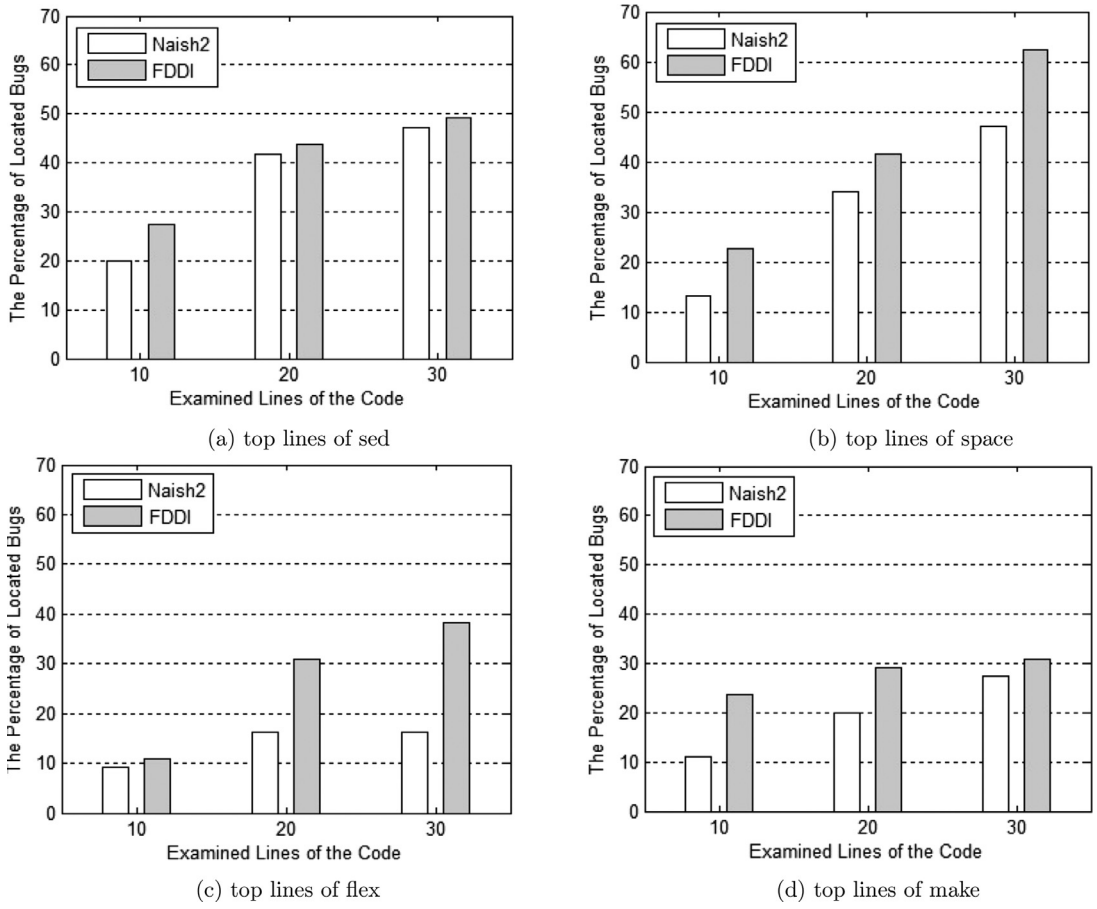
The following a snippet contains an assignment bug at line 3403 where `curr_ptr=tp;` should actually be `*curr_ptr=*tp;`.

```
3393:int GetReal(..., struct charac ** tp)
     {
3403:   curr_ptr=tp; // correct version: *curr_ptr=*tp;
        ...
3412:   ch=TapeGet(curr_ptr);
3416:   if((isdigit(ch)==0)&&(ch != '+')&&...)
            return 13;
3419:   num[i]=ch;
        ...
3426:   ch=TapeGet(curr_ptr);
3432:   while((isdigit(ch)||...)&&((*curr_ptr)!=NULL)) {
            ...
3451:       ch=TapeGet(curr_ptr);
        }
        ...
3456:   *reale=atof(num);
        ...
3464:   *tp=*curr_ptr;
        ...
        }
```

In our example, for function `GetReal`, the disparity of its two invariant sets is shown as following:

```
> tp[]->LINE_NUM == tp[]->PREV->PREV->LINE_NUM
-----------------------------------------------
> tp[]->LINE_NUM < tp[]->PREV->PREV->info
-----------------------------------------------
< reale[] elements one of { 0.0, 90.0 }
> reale[] elements one of { 0.0 }
-----------------------------------------------
tp[]->PREV->PREV->info > tp[]->PREV->PREV->LINE_NUM
-----------------------------------------------
< tp[]->PREV->info elements one of { 32, 80, 97 }
> tp[]->PREV->info elements one of { 32, 97 }
-----------------------------------------------
< tp[]->info > orig(tp[]->PREV->PREV->LINE_NUM).
```

As we can see, six dynamic invariants with differences are separated with dash lines. Symbol "<" (">") at the beginning of some dynamic invariants represents that this dynamic invariants yielded by failed (passed) tests. Via difference analysis, point variables `*tp` and `*reale` can be extracted from them. For `*reale`, according to its data dependency, we can find that `*reale` (line 3456)→ `num` (line 3419)→ `ch` (line 3412)→ `curr_ptr` (line 3403), "A→ B" represents that A is data-dependent on B. Accordingly, line 3403 containing the bug is easy to be observed by the developer. According to our method, statement at line 3403 is ranked the 17th, while it is ranked the 62th via Naish2 on the same test suite. Besides, variable `*tp` and `*reale` in `GetReal` can be presented in the bug report as auxiliary information for users to understand why line 3403 is treated as a suspicious statement by FDDI,



(a) top lines of sed

(b) top lines of space

(c) top lines of flex

(d) top lines of make

Fig. 7. Results on the top 10, 20 and 30 lines.

which may enhance the explanatory ability of the localization result (Parnin and Orso, 2011).

The following another snippet illustrates a "DS" bug Table 2 at line 2453:

```
2377:int fixselem(struct Elem * elem_ptr)
2379:{
       ...
2430:   if (elem_ptr->NPORTS==0)
        {
2433:       port_ptr=(struct Port *)
                    malloc(sizeof(struct Port));
            ...
2451:       port_ptr->PHEPOL_UNIT
                   =DEF_ELEM_LIN_POL_ANGLE_UNIT;
            /*omission error:
2453:       port_ptr->OMIT_POL=YES; */
2455:       port_ptr->NEXT=NULL;
            ...
2458:       elem_ptr->PORT_PTR=port_ptr;
2459:  }
       ...
    }
```

For variable `elem_ptr` in function `fixselem`, FDDI returns a following set of different dynamic invariants:

```
< elem_ptr.PORT_PTR->AMP elements one of { 1.0 }
> elem_ptr.PORT_PTR->AMP elements one of { 1.0, 66.0 }
-----------------------------------------------------
< elem_ptr.PORT_PTR->PHEPOL elements == 90.0
-----------------------------------------------------
> elem_ptr.PORT_PTR->PPA elements one of { 0.0, 451.0 }
-----------------------------------------------------
> elem_ptr.PORT_PTR->OMIT_POL elements == 1
```

Via difference analysis, variable `elem_ptr->PORT_PTR` can be extracted from them. In this example, data dependencies of `elem_ptr->PORT_PTR` can help us understand that there is likely missing an assignment for element `OMIT_POL` of `port_ptr`. From the dynamic invariants with disparities, we suspect element `AMP`, `PHEPOL`, `PPA` and `OMIT_POL` of variable `port_ptr` that are dependent with `elem_ptr->PORT_PTR` caused the bug. Then we observe statements related to variable `port_ptr` in function `fixselem`, and we find that there is no any assignment for the element `OMIT_POL` of `port_ptr` in function `fixselem`, which exactly causes the bug.

3.4. Threats to validity

We summarize the threats to validity in brief.

Internal validity: Firstly, FDDI ranks functions first and does further analysis on functions with high suspiciousness. Therefore, statements outside of functions are not take into account by FDDI, like global variables initialization. Moreover, FDDI analyzes each function separately after function selecting, a global variable cannot be associated with its related statements in other functions. Secondly, although 360 injected and real bugs are recruited in our experiments, those faulty version may not represent all types of bugs in reality. Thirdly, to locate more faults, the metrics used in our method requires to report all executed statements in a function when the disparity of its two sets of dynamic invariants cannot be found. In such cases, the ranking results of the root cause is mainly determined by the function selecting results, and the disparity of dynamic invariants is primarily used to filter statements that are not data-dependent with variables in the disparity

and offer explanations for the located results. One solution to mitigate this problem is to employ more powerful dynamic invariant detecting techniques to generate more expressive invariants and omit statements in the above mentioned functions.

External validity: It mainly caused by applying existing dynamic invariant detecting tools. For example, Daikon has limited ability to yield dynamic invariants with its current built into library. In our experiments, there are 16 errors that FDDI can only return executed statements as suspicious statements, because the functions they belong to are fed into Daikon without obtaining any distinct invariants for the failed and passed test case suites. Accordingly, to make FDDI more practical, some properties will be analyzed and added to extend Daikon's invariant library. However, the more properties to be checked, the longer invariant detection time Daikon has to take. Likewise, other existing invariant detecting tools have the same problem.

4. Related work

Fault localization has been an active area of research for the past decades. Since many techniques are emerged for automated program debugging, we only present some works that are highly relevant with our work.

Spectrum-based fault localization (SBFL) is a testing based program debugging approach. It utilizes program spectra and testing results collected during software testing, and applies risk evaluation formulas on each block to calculate a real value that indicates the risk of being faulty for the block. Many risk evaluation formulas have been proposed to rank the faulty block as high in the result list as possible, which included Tarantula (Jones et al., 2002), Jaccard (Chen et al., 2002) used by the Pinpoint tool, Sober (Liu et al., 2005), the cooperative bug isolation (Liblit et al., 2003; 2005), Ochiai (Abreu et al., 2006) known from the biology domain, Wong1, Wong2 and Wong3 (Wong et al., 2007), Holmes (Chilimbi et al., 2009), the causal inference approach (Baah et al., 2010), Naish1 and Naish2 (Lee et al., 2011), DCC (Perez et al., 2014), and so on. The block entities could be statements, predicates, branch, paths, components, etc. DCC (Perez et al., 2014) begins with analyzing spectra of large components of the programs and progressively increases the instrumentation details of components until reached the statement level. With increasingly risk formulas emerged, empirical approaches were conducted to compare their performance by Jones and Harrold (2005) and Abreu et al. (2007); 2009). In these studies, even through various methods are used to control the threats to validity, the empirical studies can only be considered as sampled observations. Accordingly, the performance of risk evaluation formulas are studied from the theoretical perspective (Lee et al., 2009; 2011; Xie et al., 2013a). As a result, Naish2 is found to be one of the maximal formulas. In our paper, we first analyze the spectra of procedures of programs with Naish2 (Lee et al., 2011). To improve not only the effectiveness but also the explain ability of localization results, we then use the disparity of dynamic invariants to locate likely faulty statements in each highly suspect procedure. Recently, DStar (Wong et al., 2014) is proposed to automatically locate faults without requiring any prior information on program structure or semantics.

Dynamic invariants has been used to identify anomalies and provide patches for faulty programs. Diduce technique (Hangal and Lam, 2002) was the first work to use dynamic invariants for bug diagnosis. It monitors a restricted set of variables in a program on-the-fly and attempts to give a manifestation of bugs by looking at the difference between the previous and current values of the variable. Screeners, i.e. simple bit-mask and range invariants, Abreu et al. (2008); 2008) are considered to diagnose program defects in the operational phase. ClearView technique (Perkins et al., 2009) identifies violations of dynamic invariants obtained from first learning phase, generates candidate repair patches, and observes the continued executions to select the most possible patch for the faulty program. This approach repairs errors in softwares with high availability requirements and restricts to eliminate faults that its monitor can detect. As we can see, these two techniques both require complete monitoring environment, like a controlled test environment. In contrast, Daikon (Ernst et al., 2001) technique maintains a library of invariant patterns over variables and constants. Each pattern will be instantiated with different variables and constants in a trace. Without under any monitor environment, Perkins et al. (2009) used the difference between likely invariants of passed runs and invariant spectrum of a failed run to localize faults . However, they did not locate bugs for variants of Siemens suite. Recently, Sahoo et al. (2013) used training inputs close to the failing input to generate such likely invariants and applied filtering techniques to reduce the size of the invariant candidate set. In our paper, we fist apply DCC (Perez et al., 2014) to find highly suspect components, like functions, and then only use variables in the difference between the dynamic invariants model of passed runs and the dynamic invariants model of failed runs to do further fault localization. In this way, our approach can avoid massive work on filtering and removing redundant and spurious invariants.

Methods based on seeking difference have been proposed by researchers ten years ago. For example, Delta debugging (Zeller and Hildebrandt, 2002; Zeller, 2002) technique identifies state differences between a failed and passed run of a faulty program, and detects cause effect chains leading to the root cause (Cleve and Zeller, 2005). Its diagnostic accuracy has been greatly improved by Sumner et al. (2010). Similarly, NN (Renieres and Reiss, 2003) selects a correct run that most resembles a specific faulty run, compares the spectra of these two runs, and produces a set of suspicious blocks of the program to developers. However, the approaches generally provide no hint as to which parts of the suspicious block are more likely to be faulty, and they are not effective when the set of the difference decreases in size. Therefore, FDDI combines the statistical technique to calculate the risk of each function in faulty programs at first. It then do further analysis at statement level from the highly suspect function to the lowly one. As a result, the risk of each candidate faulty statement is determined by that of its function and the sequence it appeared in the function.

The integration of SBFL with other techniques aims to improve debugging effectiveness and explain ability. For effectiveness, Yu et al. (2008) found that fault-localization effectiveness varies depending on distinct test-suite reductions. Jiang et al. (2009) studied the input of test case prioritization on SBFL techniques, and they found that test cases of random ordering can be more effective than distribution-based techniques. Later, they conducted a controlled experiment in Jiang et al. (2012) to investigate factors that have significant impacts on such combined technique, and they found that utilizing the results of Modified Condition/Decision Coverage (MC/DC)-adequate test suites are more effective than branch-adequate test suites that are more effective than statement-adequate test suites. As we know, test cases used in SBFL techniques are divided into two categories: failed and passed. That means there exists a mechanism, called test oracle, to determine whether a test finally failed or passed. Actually, this does not always hold. Therefore, Xie et al. (2011); 2013b) proposed to use expected characterizations of programs to determine whether an execution satisfied or not, which enhances the applicability of SBFL techniques. In our paper, we combined SBFL with dynamic invariants. For explain ability, Parnin and Orso (2011) showed that developers usually need context and explanations to understand a candidate root cause. Zuddas et al. (2014) combined input generation techniques with identifying the violations of a built and normal behavioral model of the faulty program. These violations are treated

as the failure causes and may offer explanations for understanding the bug. Similarly, we explore to use the disparity of dynamic invariants to enhance the explain ability of the reported faulty code.

5. Conclusions and future work

In this paper, we proposed a novel automated fault localization method via using variables in the disparities of two dynamic invariant sets, generated by running a suite of failed and passed test cases respectively. The underlying idea is that variables may be related to the fault if they are involved in dynamic invariants that do not simultaneously hold in a set of failed and passed tests. FDDI combines the spectrum-based fault localization method with the dynamic invariant detecting technique, and gives priority to reporting localization results in the highly suspect components. Since FDDI only captures variables in the disparity of these two dynamic invariant sets to do further localization, it can avoid massive work on filtering spurious invariants and removing redundant invariants. In addition, to make more efficient use of existing invariant detecting tools, FDDI only applies such a tool to a function each time to avoid the high computational cost of detecting dynamic invariants. Our experimental results show that FDDI not only improves the localization accuracy via a comparison with Naish2, but also reduces the average execution time by at least 87.2%.

Our future work will include employing some other dynamic invariant detecting tools that could generate more expressive dynamic invariants, and extend the vocabulary of Daikon to accommodate more types of errors. With the current library, Daikon did not yield any disparity of dynamic invariant sets for 16 functions with errors in our experiments. Another future direction is to investigate the influence of the size and coverage of failed and passed test suites on the performance of our method.

Acknowledgements

The authors wish to thank Kailun Luo for his valuable suggestions for improving this paper. This work is supported by State Key Laboratory of Software Development Environment Open Fund (SKLSDE-2012KF-08).

References

Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J., 2008. Automatic software fault localization using generic program invariants. In: Proceedings of the 2008 ACM Symposium on Applied computing. ACM, pp. 712–717.
Abreu, R., Gonzalez, A., Zoeteweij, P., Van Gemund, A.J., 2008. On the Performance of Fault Screeners in Software Development and Deployment. Technical Report.
Abreu, R., Zoeteweij, P., Golsteijn, R., J.C. van Gemund, A., 2009. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82 (11), 1780C–1792.
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Sysmposium on Dependable Computing (PRDC'06). IEEE, pp. 39–46.
Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. IEEE, pp. 89–98.
Baah, G.K., Podgurski, A., Harrold, M.J., 2010. Causal inference for statistical fault localization. In: Proceedings of the 2010 International Symposium on Software Testing and Analysis (ISSTA'10). ACM, pp. 73–84.
Chen, M.Y., 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'02). IEEE, pp. 595–604.
Chilimbi, T.M., Liblit, B., Mehra, K., Nori, A.V., Vaswani, K., 2009. Holmes: Effective statistical debugging via efficient path profiling. In: Proceedings of 31st IEEE International Conference on Software Engineering (ICSE'09). IEEE, pp. 34–44.
Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th international conference on Software engineering (ICSE'05). ACM, pp. 342–351.
Ernst, M.D., 2000. Dynamically discovering likely program invariants PhD thesis.
Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.
Gouveia, C., Campos, J., Abreu, R., 2013. Using html5 visualizations in software fault localization. In: Software Visualization (VISSOFT), 2013 First IEEE Working Conference on. IEEE, pp. 1–10.
Hangal, S., Lam, M.S., 2002. Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th international conference on Software engineering (ICSE'02). ACM, pp. 291–301.
Jiang, B., Zhai, K., Chanc, W.K., Tse, T.H., Zhang, Z., 2012. On the adoption of mc/dc and control-flow adequacy for a tight integration of program testing and statistical fault localization. Inf. Softw. Technol. 55 (5), 897–917.
Jiang, B., Zhang, Z., Tse, T., Chen, T.Y., 2009. How well do test case prioritization techniques support statistical fault localization. In: Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC'09). IEEE, pp. 99–106.
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'05). ACM, 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'02), pp. 467–477.
Lee, H.J., Naish, L., Ramamohanarao, K., 2009. Study of the relationship of bug consistency with respect to performance of spectra metrics. In: Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT'09). IEEE, pp. 501–508.
Lee, N., Huajie, L., Kotagiri, R., 2011. A model for spectra-based software diagnosis. In: ACM Trans. Softw. Eng. Methodol. ACM, p. 20 Issue 3.
Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I., 2003. Bug isolation via remote program sampling. In: ACM SIGPLAN Notices, vol. 38. ACM, pp. 141–154.
Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. ACM SIGPLAN Notices 40 (6), 15–26.
Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P., 2005. Sober: statistical model-based bug localization. ACM SIGSOFT Softw. Eng. Notes 30 (5), 286–295.
Nguyen, T., Kapur, D., Weimer, W., Forrest, S., 2012. Using dynamic analysis to discover polynomial and array invariants. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'02). IEEE, pp. 683–693.
Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11). ACM, pp. 199–209.
Perez, A., Abreu, R., Riboira, A., 2014. A dynamic code coverage approach to maximize fault localization efficiency. J. Syst. Softw. 90 (5), 18–28.
Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al., 2009. Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09). ACM, pp. 87–102.
Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03). IEEE, pp. 30–39.
Sahoo, S.K., Criswell, J., Geigle, C., Adve, V., 2013. Using likely invariants for automated software fault localization. In: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS'13). ACM, pp. 139–152.
Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13). ACM, pp. 314–324.
Sumner, X., William, N., Zhang, 2010. Memory indexing: canonicalizing addresses across executions. In: Proceedings of the 17th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'10). ACM.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
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(COMPSAC 2007). IEEE, pp. 449–456.
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 Trans. Softw. Eng. Methodol. 22 (4), Article31.
Xie, X., Wong, W.E., Chen, T.Y., Xu, B., 2011. Spectrum-based fault localization: Testing oracles are no longer mandatory. In: Proceedings of the 11th International Conference on Quality Software (QSIC'11). IEEE.
Xie, X., Wong, W.E., Chen, T.Y., Xu, B., 2013. Metamorphic slice: An application in spectrum-based fault localization. Inf. Softw. Technol. 55 (5), 866–C879.
Yu, Y., Jones, J.A., Harrold, M.J., 2008. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th international conference on Software engineering (ICSE'08). ACM, pp. 201–210.
Zeller, A., 2002. Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (FSE'02). ACM, pp. 1–10.
Zeller, A., 2009. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann.
Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28 (2), 183–200.
Zuddas, D., Jin, W., Pastore, F., 2014. Mimic: locating and understanding bugs by analyzing mimicked executions. In: Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14). ACM.

Xiaoyan Wang is a Lecturer of School of Management Science and Engineering at Nanjing Audit University, China. She received her PhD in School of Information Science Technology of Sun Yat-sen University in 2015. Her research interests include software engineering issues on program analysis, testing, debugging and web service composition.

Yongmei Liu is a Professor of Computer Science at Sun Yat-sen University, China. She received her PhD in Computer Science from University of Toronto in 2006. Her research interests lie in Artificial Intelligence, knowledge representation and reasoning, cognitive robotics, program verification and debugging.