Leveraging Contextual Information from Function Call Chains to Improve Fault Localization

Árpád Beszédes
Software Engineering Department
University of Szeged
Szeged, Hungary
beszedes@inf.u-szeged.hu

Ferenc Horváth
Software Engineering Department
University of Szeged
Szeged, Hungary
hferenc@inf.u-szeged.hu

Massimiliano Di Penta
Department of Engineering
University of Sannio
Benevento, Italy
dipenta@unisannio.it

Tibor Gyimóthy
Software Engineering Department
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

***Abstract*—In Spectrum Based Fault Localization, program elements such as statements or functions are ranked according to a suspiciousness score which can guide the programmer in finding the fault more efficiently. However, such a ranking does not include any additional information about the suspicious code elements. In this work, we propose to complement function-level spectrum based fault localization with *function call chains* – i.e., snapshots of the call stack occurring during execution – on which the fault localization is first performed, and then narrowed down to functions. Our experiments using defects from Defects4J show that (i) 69% of the defective functions can be found in call chains with highest scores, (ii) in 4 out of 6 cases the proposed approach can improve Ochiai ranking of 1 to 9 positions on average, with a relative improvement of 19-48%, and (iii) the improvement is substantial (66-98%) when Ochiai produces bad rankings for the faulty functions.**

***Index Terms*—Spectrum Based Fault Localization, Function Call Chains, Call Stack Traces, Testing and Debugging.**

I. Introduction

Debugging and related activities are among the most difficult and time-consuming ones in software development [1]. This activity involves human participation to a large degree, and many of its sub-task are difficult to automate.

A relevant debugging sub-task is *fault localization* (FL), in which the root causes of an observed failure are sought. Fault localization is notoriously difficult, and any (semi)automated method, which can help the developers and testers in this task, is welcome. There exist a class of approaches to aid FL which are popular among researchers, but have not yet been widely adopted by the industry: Spectrum-Based Fault Localization (SBFL), also known as Statistical Fault Localization (SFL) [2], [3], [4], [5], [6].

The basic intuition behind SBFL is that code elements (statements, blocks, paths, functions, etc.) exercised by comparably more failing test cases than passing ones are considered as "suspicious" (i.e., likely to contain a fault), while non-suspicious elements are traversed mostly by passing tests. Suspiciousness can be expressed in different ways, usually assigning one value to each code element (called the *suspiciousness score*), which can then be used to *rank* the code elements. The idea is that by inspecting this ranked list a developer would find the fault near the beginning of the list, hence being more productive in localizing the fault.

A possible approach to measure the effectiveness of a SBFL method is to investigate the average rank position of the actual faulty element relative to the total number of code elements [2], *i.e.,* the number of elements that have to be investigated before finding the fault (we refer this as the *Expense* metric). Further studies revealed that Expense is crucial to the adoption of the method in practice. In particular, research showed that if the faulty element is beyond the 10th element (or even the 5th according to some other studies), the method will not be used by practitioners because they need to investigate too many elements in vain [5], [7], [8], [9].

The state-of-the-art approach to SBFL is to use the so-called "hit-based" spectra [10] with statements as basic code elements. Researchers proposed many different scoring mechanisms, but these are essentially all based on counts of passing/failing and traversing/non-traversing test cases in different combinations [2], [5], [11]. Popular suspiciousness scores are Tarantula [12], Ochiai [13], and DStar [14], among others.

One reason why an SBFL formula may fail is what is referred as *coincidental correctness* [15], [16], [17]. This is the situation when a test case traverses a faulty element without failing. This can happen quite often since not all exercised elements may have an impact on the computation performed by a test case [18], and if there are relatively more such cases than traversing and failing ones, the suspiciousness score will be negatively affected [16].

In this work, we propose to enhance traditional SBFL with *function call chains* on which the FL is performed. Function call chains are snapshots of the call stack occurring during execution and as such can provide valuable context to the fault being traced. Call chains (and call stack traces) are artifacts occurring during program execution which are well-known to programmers who perform debugging, and can show, for instance, that a function may fail if called from one place and perform successfully when called from another. There is empirical evidence that stack traces help developers fix bugs [19], and Zou *et al.* [20] showed that stack traces can be used to locate *crash-faults*.

More specifically, we propose a novel SBFL algorithm, that computes ranking on all occurring call chains during execution, and then selects the suspicious functions from these

Accepted for publication by IEEE. © 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

ranked chains using a function-level (*i.e.,* method-level for object-oriented languages like Java) spectrum-based algorithm, Ochiai in particular [13]. An example of the overall approach is presented in Section III.

Our approach works at a higher granularity than statement-level approaches (previous work suggests that function-level is a suitable granularity for the users [21], [20]). At the same time, we provide more context in the form of the call chains, and therefore have the potential to show better performance in terms of Expense.

We empirically evaluated the proposed approach using 404 real defects from the Defects4J suite [22]. Results indicate that except for the two outliers (Chart and Closure) the call chain-based FL approach can improve the localization effectiveness of 1 to 9 positions (with a relative improvement of 19-48%), compared to a hit-based function-level approach (Ochiai [13]). In the case of defects with ranks worse than 10, this ratio increased even more (66-98%) on all programs. Furthermore, the defective element could be located in 69% of the cases in the highest-ranked call chains, which turned out to be relatively short on average. Last, but not least, we provide qualitative evidence that, besides improved performances, the proposed approach can provide useful information to the developer performing a debugging task.

II. Context and Related Work

A. Spectrum-Based Fault Localization

The use of execution profiles – program spectra – for FL purposes has been proposed for the first time in a study on the Y2K problem aimed to discover date-dependent computations [6], although the first mention of the idea appeared already in 1987 [23]. Since then, SBFL emerged as one of the main approaches to software FL [24], [4], [3].

SBFL approaches are still finding their way to be employed in practice [25], [26], [9], [13]. For instance, most studies are carried out using artificial faults [5], and still the faulty element is usually placed far from the top of the rankings [7], [8]. Abreu *et al.* [27] investigate the accuracy of SBFL approaches in practice. Le *et al.* show that there is a gap between theoretical and practical results [25].

Different types of program spectra have been proposed by Harrold *et al.* (hit-based, count-based, counting branches, paths, data dependencies, etc.) [28], [10], however, the most commonly adopted approach uses individual statements or functions as the basic program elements.

While several SBFL suspiciousness formulae have been proposed, in this work we complement (and compare with) Ochiai, which proved to be among the best performing [13]. A detailed analysis of suspiciousness formulae is reported in the survey by Wong *et al.* [2], in the theoretical analysis of Xie *et al.* [11], and in the study of Pearson *et al.* [5].

B. Improvements to SBFL Approaches

One of the main reasons for the suboptimal performance of SBFL, in general, is coincidental correctness. This has been the focus of several works [18], [29], [30]. Wang *et al.* used context patterns for common fault types, which can strengthen the correlations between program failures and the coverage of faulty program entities [29]. Bandyopadhyay and Ghosh proposed an approach to assign weights to test cases for representing their importance in FL based on the proximity to the failing test cases [30]. They also proposed an approach to iteratively predict and remove coincidentally correct test cases based on user feedback in small programs [31].

Xie *et al.* present an informative overview of suspiciousness score assignment approaches [11]. Yoo presents an automatic approach to derive risk evaluation formulae [32] using genetic programming. Renieris and Reiss use nearest neighbor measures of program spectra for FL [33]. Several researchers have used the *learning to rank* model [34], to combine different FL algorithms [35], [21], [20].

Gong *et al.* [36] propose to complement SBFL techniques with the users' feedback, showing how this could produce significant improvements on the FL accuracy.

Finally, to support developers in visualizing the output of spectrum-based FL, Jones *et al.* developed Gammatella, a tool that visualizes statement suspiciousness using color maps [37].

C. Stack Trace-Based and Related Approaches

Schröter *et al.* provide strong evidence that the stack traces have a great value to developers during the debugging process [19], and as other works show, they carry important information that can help in the localization process [20], [38]. Zou *et al.* discuss that stack trace analysis may be used for FL by examining the depth of an element in the stack trace for assigning the suspiciousness score [19], [20]. In particular, they suggest to assign a score to the function which is reciprocal to the depth of its first occurrence in the stack trace. They found that this approach was the most successful in localizing the crash faults (which accounts for about 25% of the defects in Defects4J). Wu *et al.* extend the call stack with static call graph information and then calculate the suspiciousness scores for the functions [38]. These approaches use stack traces in a limited way, for instance only on crash failures. Our approach is to leverage the information from all appearing stack trace instances (the call chains) and apply the basic SBFL techniques on them.

Similar concepts to function call chains have been explored by other researchers [39], [40], however not in this detail and not with FL application in focus.

Involving a different type of context to the FL process has been recently proposed by de Souza *et al.* [41]. They introduce different "roadmaps" to the user based on additional information from the code and the execution. One of their approaches uses method call relationships, which is similar to our idea.

D. Non-SBFL Approaches

We focus on the statistical analysis of dynamic test case executions, but there have been other approaches proposed for FL as well. These include slicing-based [42]; statistics-based [43], [44]; and mutation-based approaches [45], [46].

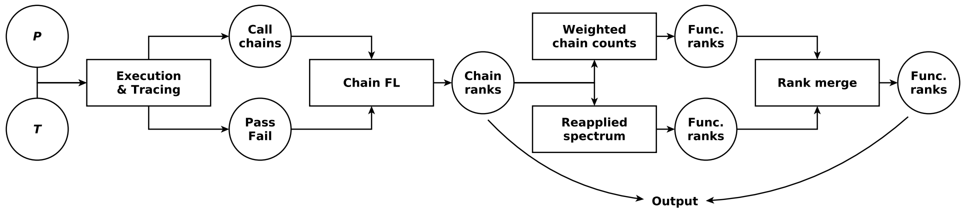

Fig. 1. Call chain based FL overview.

Machine learning and data mining techniques are employed for FL as well [47], [48]. Researchers have also proposed model-based FL approaches [49], [50], as well as state-based approaches [51], [52]. We refer to the surveys of Wong *et al.* [2], [24] and Parmar *et al.* [3].

A recent paper by Zou *et al.* [20] presents an empirical comparison of different FL families of algorithms that include SBFL, mutation-based approaches, program slicing, stack trace analysis, predicate switching, and history-based approaches. They also combine these mechanisms using learning to rank [34].

Other debugging techniques involve record-replay and user interaction [53], [54], [55], [56]. These are not directly related to our work, although call chains could support user interaction during debugging.

E. Contribution

The novelty of our approach with respect to the above is that, without requiring additional user input, we are utilizing call chain (stack trace) data at function-level for FL, which includes much more information compared to individual functions (or statements). Call chains provide a *context* about the possible failures, which can complement the basic ranking lists of program elements or, in some cases, replace them.

III. Fault Localization on Call Chains

Fig. 1 provides a high-level overview of our approach. Using a given set of test cases T, the subject program P is executed while collecting the necessary execution trace information. This is used to produce the function call chains, as well as the test case pass/fail outcomes (more on this in Section III-A). Based on that, we compute the call chain level program spectrum information, which is used to calculate the ranking of the chains according to their suspiciousness levels (discussed in more detail in Section III-B). In the next step, two algorithms are applied to compute the ranking of the functions for FL, which are then merged to produce the final ranking (see Section III-C).

A. Function Call Chains

Let F be the set of functions in a program P, and T a set of test cases used to test P. Then, a *Call Chain* c is a sequence of functions $f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_n$ ($f_i \in F$), which occur during the execution of some test case $t \in T$, and for which:

- f_1 is the entry point called by t,
- each f_i directly calls f_{i+1} ($0 < i < n$), and
- f_n returns without calling further functions in that sequence.

In other words, c is one of the possible deepest *call stack* states occurring during the execution of t. Call stacks and the associated stack traces are well-known structures used in everyday work by programmers during debugging. They describe a particular state during program execution and help understand the context that led to that state. At the same time, they are very concise as well because no previous state is maintained, and typically the function call nesting levels are not very deep. Statement-level control or data flow information is much more complex and more difficult to produce.

Call chains can be efficiently produced from test case executions because only the function entry and exit events need to be recorded and stored in a stack structure. One thing to note here is that the used instrumentor method must be able to handle any non-structured call events such as exceptions and multi-threaded execution.

In our method, we collect all distinct call chains occurring during the execution of T, which will be referred to as the call chain set C. We also maintain a set of chains $C(t)$ occurring for each individual test case t (we say that t *executes* c if $c \in C(t)$). Finally, the set of functions occurring in a chain c will be denoted by $F(c)$.

Fig. 2 contains a simple code snippet for illustrating these concepts with the associated test cases in Fig. 3. In it, we can identify four test cases $\texttt{t1} \ldots \texttt{t4}$. $\texttt{t1}$ and $\texttt{t2}$ are passing, while the other two fail due to an error in function g. These test cases produce altogether five different call chains: $\texttt{a} \rightarrow \texttt{f}$, $\texttt{a} \rightarrow \texttt{g}$, $\texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}$, $\texttt{b} \rightarrow \texttt{g}$ and $\texttt{b}$, which will constitute the set C. Then, $C(\texttt{t1}) = \{\texttt{a} \rightarrow \texttt{f}, \texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{g}\}$, $C(\texttt{t2}) = \{\texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{g}\}$, $C(\texttt{t3}) = \{\texttt{a} \rightarrow \texttt{g}, \texttt{b}\}$ and $C(\texttt{t4}) = \{\texttt{a} \rightarrow \texttt{f}, \texttt{a} \rightarrow \texttt{g}, \texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}\}$.

This example is constructed so that the benefits of our method are visible. In particular, we set the fault to be in function g, and it will be manifested if invoked directly from b but not when invoked from a. This way, both elements, the caller and the callee, are important from the localization point of view, and this is what the call chains will capture. As we will see, the fault will be located at the first ranking position with our approach, while a function-level hit-based suspiciousness score (*e.g.,* Ochiai) will give priority to some other code element. The call chain information is also useful because it will allow the programmer to find other possible fixes for the failure such as modifying the call site if that is more appropriate.

A more realistic example is provided at the end of Section V, which is an actual fault from our benchmark.

```
public class ChainFLExample {
  private int _x = 0;
  private int _s = 0;
  public int x() {return _x;}

  public void a(int i) {
    _s = 0;
    if (i==0) return;
    if (i<0)
      f(i);
    else
      g(i);
  }

  public void b(int i) {
    _s = 1;
    if (i==0) return;
    if (i<0)
      a(i);
    else
      g(i);
  }

  private void f(int i) {
    _x -= i;
  }

  private void g(int i) {
    _x += (i+_s); // error: should be _x += i;
  }
}
```

Fig. 2. Example for illustrating call chains.

```
public class ChainFLExampleTest {
  @Test public void t1() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(-1);
    tester.a(1);
    tester.b(1);
    assertEquals(3, tester.x());
  }

  @Test public void t2() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(1);
    tester.b(1);
    assertEquals(2, tester.x());
  }

  @Test public void t3() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(1);
    tester.b(0);
    assertEquals(1, tester.x());
  }

  @Test public void t4() {
    ChainFLExample tester = new ChainFLExample();
    tester.a(-1);
    tester.a(1);
    tester.b(-1);
    assertEquals(3, tester.x());
  }
}
```

Fig. 3. Test cases for the example.

B. Chain-Based FL

The first phase of our approach is FL on the call chains. This takes as inputs the test case execution outcomes (pass/fail) and uses a program spectrum representation with the chains as code elements. The output is a ranked list of call chains with the associated suspiciousness scores.

We apply a traditional program spectrum representation based on binary matrices. Let $\mathbf{S}^{ch}$ denote the chain based spectrum, whose rows represent test cases (elements of T), and columns contain the call chains (elements of C):

$$\mathbf{S}^{ch} = t_i \left\{ \overbrace{\begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix}}^{c_j} \right. \quad \mathbf{R}^{ch} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix}$$

$\mathbf{S}^{ch}(i,j) = 1$ means that the call chain c_j will occur at least once in the execution of test case t_i.

The vector $\mathbf{R}^{ch}$ denotes the test case execution results vector. It is a record of the outcomes of test case runs, namely *pass* (0) or *fail* (1). Fig. 4 shows the spectrum with the matrix and the result vector for our example from Fig.s 2 and 3 ($c_1 \ldots c_5$ denote the chains $\texttt{a} \rightarrow \texttt{f}$, $\texttt{a} \rightarrow \texttt{g}$, $\texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}$, $\texttt{b} \rightarrow \texttt{g}$ and $\texttt{b}$, respectively).

For the call chains, any basic SBFL suspiciousness score could be used. In this work we used the Ochiai score [13], used in recent work [5], [20], [21] and proved to outperform

$$\mathbf{S}^{ch} = \begin{array}{c|ccccc} & c_1 & c_2 & c_3 & c_4 & c_5 \\ \hline \texttt{t1} & 1 & 1 & 0 & 1 & 0 \\ \texttt{t2} & 0 & 1 & 0 & 1 & 0 \\ \texttt{t3} & 0 & 1 & 0 & 0 & 1 \\ \texttt{t4} & 1 & 1 & 1 & 0 & 0 \end{array} \quad \mathbf{R}^{ch} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Fig. 4. Chain-based spectrum for the example.

other popular formulae in many situations. To calculate the suspiciousness scores, many formulae rely on some or all of these four basic statistics for each code element c (chain, in our case): $ef(c)$, $nf(c)$, $ep(c)$ and $np(c)$, which count the number of test cases that execute call chain c and fail, do not execute c and fail, execute c and pass, do not execute c and pass, respectively. The Ochiai score does not use $np(c)$ and is computed as:

$$\mathcal{O}(c) = \frac{ef(c)}{\sqrt{(ef(c) + nf(c)) \cdot (ef(c) + ep(c))}}.$$

This way, each call chain c will be assigned a suspiciousness score between $[0,1]$ according to the formula. For our example, $\mathcal{O}(\texttt{a} \rightarrow \texttt{f}) = \frac{1}{2}$, $\mathcal{O}(\texttt{a} \rightarrow \texttt{g}) = \frac{1}{\sqrt{2}}$, $\mathcal{O}(\texttt{b} \rightarrow \texttt{a} \rightarrow \texttt{f}) = 0$, $\mathcal{O}(\texttt{b} \rightarrow \texttt{g}) = 1$ and $\mathcal{O}(\texttt{b}) = 0$. This, in itself, might be a useful output for the programmer seeking the faulty code element because the high ranked chains could lead her attention to the faulty element and the context in which it was invoked (in our case the call chain $\texttt{b} \rightarrow \texttt{g}$). However, we proceed to compute also the most suspicious functions, as described in the following.

C. Locating Functions

A trivial approach for the user to locate the defective function (and statement, respectively) is to consider the highest-ranked call chains and investigate the functions occurring in them (according to our experimentation, this can be successful quite often). But we also propose an approach to produce a ranked list for functions as well based on the call chain scores.

We experimented with various algorithms for this purpose and eventually found out that different strategies may produce good results in different cases. Hence, we decided to use the two best performing strategies and then combine their results, as explained in the following.

1) Weighted Chain Counts: The basic idea with this strategy is to count the number of occurrences of each function in the chains weighted by the respective chain scores from the previous phase. The intuition behind this is that functions frequently occurring in highly ranked chains will be more suspicious. More precisely, for each function $f \in F$ we compute the score $\mathcal{W}$ as:

$$\mathcal{W}(f) = \sum_{c \in C(f)} \mathcal{O}(c)\,, \text{where } C(f) = \{c \mid f \in F(c)\}\,.$$

Note that this score will not fall in the interval $[0,1]$ which is typical for many other scoring mechanisms. However, this does not affect other parts of the approach since only the relative ranks are subsequently used.

For our example, the scores will be the following: $\mathcal{W}(\mathtt{a}) = \frac{1}{2} + \frac{1}{\sqrt{2}}$, $\mathcal{W}(\mathtt{b}) = 1$, $\mathcal{W}(\mathtt{f}) = \frac{1}{2}$ and $\mathcal{W}(\mathtt{g}) = 1 + \frac{1}{\sqrt{2}}$. This leads to the defective function with the highest score.

2) Reapplied Spectrum: The second idea for computing function-level scores is to re-apply the spectrum-based approach, but this time on the functions using the call chains in place of the test cases. For this purpose, we treat a call chain as a "proxy" to a test case in the following manner. If its score is greater than a threshold $\mathtt{z} \in [0,1)$ it is treated as "failing" otherwise as "passing." Hence, our function-level spectrum has the call chains in its rows and the functions in the columns:

$$\mathbf{S}^{fn} = c_i \left\{ \overbrace{\begin{bmatrix} 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ & & \ddots & & \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 & 0/1 \end{bmatrix}}^{f_j} \right. \quad \mathbf{R}^{fn} = \begin{bmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \\ 0/1 \end{bmatrix}$$

In this case, a 1 at the matrix position (i,j) means that $f_j \in F(c_i)$, and the entry in the vector $\mathbf{R}^{fn}$ for a chain c_i is 1 if $\mathcal{O}(c_i) > \mathtt{z}$. By adjusting $\mathtt{z}$, one can regulate how "strictly" a suspicious call chain should be considered as faulty. We experimented with different thresholds, but in the following, we will set $\mathtt{z} = 0$ as it provided the best results.

The final scores in this case will be computed by re-applying the Ochiai formula to this function-level spectrum, which will be denoted by $\mathcal{R}(f)$ for a function f. In the example, $\mathcal{R}(\mathtt{a}) = \frac{2}{3}$, $\mathcal{R}(\mathtt{b}) = \frac{1}{3}$, $\mathcal{R}(\mathtt{f}) = \frac{1}{\sqrt{6}}$ and $\mathcal{R}(\mathtt{g}) = \frac{2}{\sqrt{6}}$, which again ranks g to the first position.

```
Input R1 {rank according to W scores}
Input R2 {rank according to R scores}
Output {combined rank}
Repeat
  f := next element in R1
  If f is not output yet
    Output f
  f := next element in R2
  If f is not output yet
    Output f
Until R1=empty and R2=empty
```

Fig. 5. Rank merging algorithm.

Note, that the simple function-level Ochiai formula scores function f with $\frac{1}{2}$ and the other three with $\frac{1}{\sqrt{2}}$, which makes this approach not very useful in this particular case. It is also interesting to note, that the statement-level Ochiai FL will locate the call statement to g in function b, which is also informative. However, our approach provides more context in a general case because the whole call stack is presented and not only individual code elements. Also, with our second phase not only the chain ranks but the function ranks will be available as well.

3) Merging the Ranks: The reason for the two ranking methods to behave differently can be traced back to the mentioned coincidental correctness, which – apparently – can affect the chain-based approach as well.

Most SBFL formulae poorly perform when the defective element f has a high $ep(f)$ value compared to $ef(f)$. In the case of our reapplied spectrum technique, this means that f is found in many chains that have $\leq \mathtt{z}$ score, while in fewer chains that have $> \mathtt{z}$ score. This, in turn, can happen if some passing test cases are complex enough to generate a lot of different chains, contrary to the failing ones. We observed that this can happen often in the case of the reapplied spectrum, so in this case, the weighted chain counts technique will perform better because it is not affected by the many passing chains.

Since we do not know in advance which of the two function-level scores will lead to better results in a particular case, so we merged the two ranked lists by alternatively selecting the next element from each of the two lists. The algorithm in Fig. 5 depicts the approach more precisely.

The algorithm has the following property: if the rank position of the faulty element is r_1 in R_1 and r_2 in R_2, in the worst case it will be found in $2 \cdot \min(r_1, r_2)$ steps. This means that if one of the scoring mechanisms is poor compared to the other, the result will depend on the better one. Moreover, if the two ranks are similar, the output will also be similar to them and the mentioned worst case will not be reached. Note that this algorithm does not explicitly handle ties, situations when elements with the same score are ranked subsequently in an arbitrary order. Also, the rank list with which the processing is started is arbitrary. Depending on how these are implemented, the algorithm could produce different final outputs.

Several researchers have used machine learning, such as learning to rank [34], to combine different FL algorithms [35], [21], [20]. We selected to use the above simple approach instead because it guarantees the minimum required rank position and it is practically the same approach the user would follow if she is given two ranked lists to analyze in parallel.

Consider again, our running example. The weighted chain counts approach produces the following ranked list of functions: `gabf`. At the same time, the function-based spectrum results in `gafb`. Finally, the merging step outputs either `gabf` or `gafb`, depending on which rank list is the processing started on. The faulty element is in the first position in either case.

IV. Empirical Evaluation

The *goal* of the study is to assess the proposed approach based on the combination of call chain scores and function-level SBFL. The *quality focus* is the effectiveness of the approach, compared with state-of-the-art SBFL. The *context* consists of 404 bugs from the Defects4J suite [22].

More specifically, the study aims at addressing the following research questions:

RQ1 *What are the properties of the occurring call chains and, in particular, of chains that contain faulty elements?* We measure how often do the faulty elements appear in the top-ranked chains. Also, we collect the number and length of the chains because this highly contributes to the usefulness of the chain ranking.

RQ2 *How much improvement can the call chain-based approach achieve compared to basic function-level fault localization?* We measure this property using the fault localization Expense measure (RQ2a). At this point, we also compare the two function-level scoring mechanisms of the second phase of our approach and measure how often is each of them better than the other (RQ2b).

In the following, we describe the details of the experiment setup and the evaluation methodology.

A. Study Settings

Our study considered different SBFL formulae *i.e.,* Ochiai, Tarantula and DStar. Since they provided similar performance in the traditional setting and due to space limitations we report only Ochiai results. However, the online appendix [57] includes the results of the other formulae as well.

We performed the experiments on real defects from the Defects4J suite (v1.4.0) [22]. We selected this benchmark because it can be seen as the state of the art in FL research for Java (see *e.g.,* [21], [5], [20], [58], [59], [60] and many others), and it includes real defects and programs with non-trivial size and complexity. The dataset provides the fix for each bug as a patch set (called a *version*). Using the patch sets we were able to create change sets that contain data about which functions (Java methods) were affected by each bug fix.

By default, Defects4J utilizes Cobertura [61] to measure code coverage. However, since call chains are needed for our approach we had to use different technology. We used a bytecode instrumentation tool based on Javassist [62] to collect execution traces. This tool uses a compact data structure which was carefully engineered to handle recursive calls and

TABLE I
MAIN PROPERTIES OF THE DEFECTS USED IN THE EXPERIMENTS.

Program	KLOC	Tests	Bugs	Functions	Chains
Chart	96	2 187	25	5 235	41k
Closure	91	7 867	173	8 379	889k
Lang	22	2 270	60	2 353	6k
Math	84	4 371	92	6 351	228k
Mockito	11	1 331	28	1 433	11k
Time	28	4 019	26	3 627	150k
Total	**332**	**22 045**	**404**	**27 378**	**1 325k**

the exceptional amount of data that is generated during the execution of real life programs.

Unfortunately, some tests cases fail if the code is instrumented. These tests assert things that the instrumentation changes *e.g.,* structure of an object, runtime, contents of the classpath, etc. Since the unexpectedly failing tests would affect the suspiciousness of the covered code elements, we excluded those bugs that include this kind of tests. Finally, we considered only those faults for which there is at least one failing and traversing test case. The final set of programs and defects from the Defects4J dataset we used in our experiments is reported in Table I. Numbers regarding size (lines, tests, functions) vary from version to version, here data from the last versions are provided. The last column contains the number of chains generated (also for the last version), which will be explained later.

To store the spectrum information matrices and compute the various scores and ranks, we used the SoDA framework [63]. Apart from that only various scripts and spreadsheet editors were used for the calculations.

B. Measuring the Chain Properties

Compared to a basic function-level SBFL, the proposed approach requires:

1) To compute the call chains besides simple code coverage information.
2) A larger spectrum matrix, as its columns include the different chains rather than functions.
3) An additional step to locate the functions, which is composed of two ranking algorithms and a merging phase. The function-level spectrum requires a matrix whose rows are composed of the different chains which are typically more numerous than the test cases.

To account for these differences, we recorded their basic properties such as the number and size (number of function occurrences) of the chains.

C. Evaluation of Fault Localization Effectiveness

Several strategies have been proposed in the literature for measuring the effectiveness of SBFL approaches, but they are practically all based on looking at the rank position of the actual faulty element within the list of all possible program elements. One strategy is to express this as the number of elements that need to be investigated by the programmer before finding the fault [2], and another is the opposite: elements that do not need to be investigated [33]. This is usually expressed in relative terms compared to the length of the rank list (program size). However, Parnin and Orso argued that absolute rankings are more helpful in practical situations [8].

Another issue with these mechanisms is the handling of ties [64], because in many cases different program elements may get assigned the same suspiciousness scores. Some approaches select the first (best case), last (worst case) or middle (expected case) element for expressing this value, while others simply treat the elements with the same values as all belonging to one position.

For computing the effectiveness of an SBFL approach, we follow the strategy to look at "elements that need to be investigated" using the "expected case" in the case of ties and express this in a set of measures called *Expense*. We use two variants of the measure: an absolute one expressed in the number of code elements (E) and a relative version compared to the length of the rank list (E'). The following formulae express precisely how to calculate this value (following [27]):

$$E = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2},\ E' = \frac{E}{N} \cdot 100\ [\%],$$

where N is the number of code elements, for $i \in \{1, \ldots, N\}$ s_i is the suspiciousness score of the ith code element and f is the index of the faulty code element.

To compare our approach to traditional SBFL techniques, we will compute the Expense metric for both approaches and compare them in terms of change relative to traditional SBFL, using both absolute values and relative improvements.

Apart from the general average change, we define the notion of *enabling improvement*, an improvement in which the traditional SBFL algorithm ranks the faulty element beyond the 10th position but the proposed approach reaches it in at most 10 steps. This way, from a practically "hopeless" localization scenario, our approach enables the user to localize the fault by inspecting only the top elements in the list.

Finally, we compare the rankings achieved by Ochiai with those achieved by the proposed, combined approach. To this aim, a Wilcoxon sign-rank test [65] should normally be used. However, in the context of FL, especially for the easy matches, the test could encounter ties, *e.g.,* when both approaches report a first rank for a function. To overcome this limitation, we use, instead, the Wilcoxon-Pratt test [65] which copes with the ties. Since multiple tests are performed (one per program), the p-values have been adjusted using the Holm's correction [66]. We complement the Wilcoxon-Pratt test with the Cliff's d effect size measure [67].

V. Results

In this section, we present the results of our experimental evaluation following our research questions from above.

A. Properties of Call Chains

As described in Section IV-B, we recorded the different properties of data structures during the execution of the experiments and used these to compare our approach to the basic function-level FL. Table I includes some basic statistics, *i.e.,* number of functions, tests and call chains, of the considered programs (their last versions).

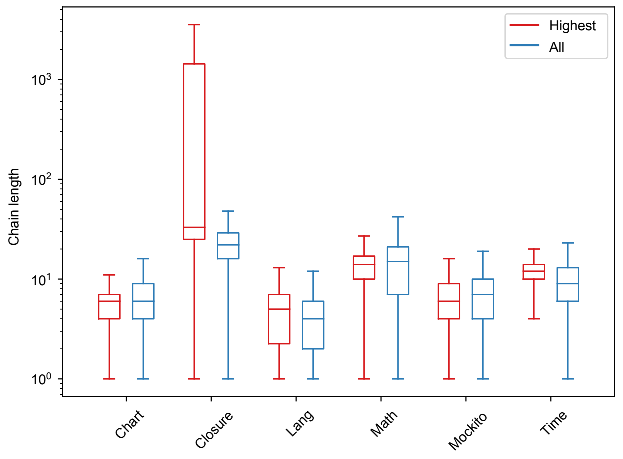

Fig. 6. Properties of highest-ranked and all chains.

TABLE II
FAULTY ELEMENTS IN HIGH RANKED CHAINS AND AVG. CHAIN LENGTHS.

Program	Faulty in High	Length All	Length High
Chart	19 (73%)	8.3	5.7
Closure	98 (56%)	26.0	849.3
Lang	56 (88%)	4.4	5.1
Math	70 (75%)	14.8	13.9
Mockito	20 (69%)	7.8	58.6
Time	21 (78%)	10.1	11.7
Total / Average	**284 (69%)**	**24.8**	**749.2**

The distribution of chain lengths is depicted in the blue boxplots (*i.e.,* the second for each program) of Fig. 6 (note that outliers are excluded). Although the call chains can be very long (about 3,500 functions), Closure tend to have shorter chains, i.e., about 4 to 26 functions. The average call chain length is 24.8 for the whole dataset and 12.4 if we exclude Closure. (More details can be seen in Column 3 of Table II.)

We now investigate what is the relationship between the faulty elements and the content of the highly-ranked chains produced in the first phase of our approach. The second column of Table II shows the number of times (and their ratio) the faulty element can be located in the call chains from the very beginning of the ranked list. In particular, we considered the chains with the highest suspiciousness scores. It is interesting to note that the highest score was in many cases 1. We can observe from the data that, for all programs, 69% of the defective elements are found in the highest-ranked chains, which is a very high ratio.

It is also interesting to investigate whether these fault-containing chains are any different in terms of their sizes from the general statistics. The red boxes in Fig. 6 (left-side for each program) depict the length distribution of such chains. As we can observe, the maximum length of these call chains varies from program to program. In general, not considering the outlier Closure, the average length of chains with the

TABLE III
FAULT LOCALIZATION EFFECTIVENESS COMPARISON (AVERAGES SHOWN).

Program	**Bugs**	**Ochiai** $E(E')$	**Combined** $E(E')$	**Difference** $E(E')$	**Relative change**	**Ochiai** > 10	**Enabling improvements**	**Relative improvement**
Chart	25	8.3 (0.19%)	10.8 (0.25%)	2.4 (0.06%)	29%	5	2 (8%)	-19.0 (-76%)
Closure	173	99.5 (1.33%)	131.4 (1.77%)	31.9 (0.44%)	32%	106	16 (9%)	-58.8 (-93%)
Lang	60	4.7 (0.23%)	3.5 (0.17%)	-1.1 (-0.05%)	-24%	7	4 (7%)	-15.4 (-66%)
Math	92	11.0 (0.29%)	7.3 (0.19%)	-3.7 (-0.10%)	-34%	27	17 (18%)	-28.1 (-87%)
Mockito	28	25.6 (2.47%)	20.6 (1.98%)	-5.0 (-0.49%)	-19%	9	3 (11%)	-92.0 (-98%)
Time	26	18.3 (0.53%)	9.5 (0.27%)	-8.8 (-0.26%)	-48%	7	2 (8%)	-49.2 (-94%)
Total / Average	**404**	**49.3 (0.89%)**	**61.1 (1.00%)**	**11.9 (0.11%)**	**24%**	**161**	**44 (11%)**	**-43.0 (-91%)**

TABLE IV
RESULTS OF THE WILCOXON-PRATT TEST AND CLIFF'S d EFFECT SIZE WHEN COMPARING OCHIAI VS. COMBINED APPROACH (CLIFF'S d IS POSITIVE WHEN IN FAVOR OF THE COMBINED APPROACH).

Program	**Whole evaluation set**			**Bugs ranked > 10 by Ochiai**		
	p**-value**	d	**Magn.**	p**-value**	d	**Magn.**
Chart	<0.001	-0.05	negligible	0.01	0.12	negligible
Closure	<0.001	-0.27	small	<0.001	-0.01	negligible
Lang	<0.001	0.04	negligible	<0.001	0.59	large
Math	<0.001	0.15	small	<0.001	0.66	large
Mockito	<0.001	-0.25	small	<0.001	0.47	medium
Time	<0.001	-0.04	negligible	<0.001	0.61	large
Overall	**<0.001**	**-0.08**	**negligible**	**<0.001**	**0.14**	**negligible**

highest score is 13.8. Column 4 in Table II shows the related average values. This finding indicates that the investigation of only the resulting call chains may often lead to finding the fault. However, this process may be supported by the ranked functions in the second phase.

Answer to RQ1: 69% of the faulty elements appear in the chains with the highest score values, and these chains contain about 14 functions on average. These two factors contribute to the usefulness of the chain ranks for FL.

B. Localization Effectiveness

Table III reports the results for FL effectiveness. Columns "Ochiai" and "Combined" show the absolute and relative Expense values for function-level Ochiai and for the proposed approach, respectively. Column "Difference" reports the difference between the average rankings, while column "Relative change" expresses the same as percentage increase/decrease with respect to Ochiai. Column "Ochiai > 10" reports the number of defects in the programs for which the ranking position is more than 10. "Enabling improvement" indicates how many defects were successfully moved to the 10th or below position by our approach (the percentage is relative to bug number), and the last column shows the average absolute and relative difference of rankings for such cases.

For Lang, Math, Mockito and Time, the improvement is measurable in terms of the Expense metric: this ranges from 1 to about 9 ranking positions on average with relative change of 19-48%. For Chart and Closure, the proposed algorithm yields ranking positions that are worse by 29-32% on average compared to Ochiai. Note that, the average ranking that Ochiai scores on the bugs of Closure is 99.5, which is already impractical as developers would unlikely investigate such a large number of functions. The reason for this result could be that Closure is different from the other programs in the dataset. It is a JavaScript compiler, which means that it has a very specific code structure and test suite as well. Also note that, despite the poor average performance on Closure, our approach can still deliver enabling improvements in 16 (9%) cases and the improvement is very high -58.8 (-93%) in these cases.

The left-side of Table IV addresses the comparison between the Combined approach and Ochiai from a statistical point of view. Results show that, on the whole evaluation dataset, while the results of the statistical tests show significant differences, the effect size is negligible to small, and in favor of Ochiai.

However, if we look at the results obtained when Ochiai scores a bad ranking position of the correct recommendation, *i.e.,* $> 10th$ (right-side of Table III and IV), we can observe that 44 of 161 (27%) of defects with ranks higher than 10 could be reduced to below 10 and the average reduction in terms of ranking positions is 43 which is 91% relative improvement. From a statistical point of view, except for Closure, results are in favor of the proposed approach. Also, the effect size is large in three cases (Lang, Math, Time), and medium for Mockito.

Answer to RQ2a: In 4 out of 6 cases the call chain-based FL approach could improve the localization effectiveness of Ochiai of 1 to 9 positions on average, with a relative improvement of 19-48%. Also, about 27% of the defects with ranks higher than 10 could be reduced to below 10 with an average reduction of 91%, with statistically significant differences and medium to large effect size.

Our final set of experiments regarding the localization effectiveness deals with the two function localization algorithms that work on the ranked chains, which we introduced in Section III-C. As described, the two techniques performed well in different situations, and it was difficult to predict which approach would be better for a particular case. Hence, we follow the described merging approach, which produces an overall better result than the two individually (in each particular case, twice the minimum is guaranteed). Table V includes the comparison of these two techniques summarized for each program, with the overall average shown in the last row.

In columns 2 and 3 of the table, we report the average absolute Expense metrics for the respective techniques, while

TABLE V
COMPARISON OF WEIGHTED CHAINS VS REAPPLIED SPECTRUM (AVERAGES SHOWN).

Program	E Weigh.	Reapp.	Comb.	Weigh. better	Reapp. better
Chart	13.5	10.6	10.8	12	9
Closure	143.6	149.9	131.4	59	112
Lang	3.7	4.0	3.5	23	19
Math	9.3	7.0	7.3	17	52
Mockito	21.0	36.3	20.6	13	15
Time	16.5	8.0	9.5	9	13
Total / Average	**67.5**	**70.1**	**61.1**	**133**	**220**

column 4 includes the same data for the merged outcome. The last two columns include the counts when the respective technique performed better than the other. We can conclude from the data that the combined algorithm indeed is useful because there is a similar number of cases when one of the two rankings is better. We also checked the correlation between the scores produced by the two function-level techniques, and we found that it is close to zero. As expected, the combined approach produced an overall better result than any of the other two, however, both approaches are quite close to the combined.

Answer to RQ2b: When comparing function-level rankings, we found that the reapplied spectrum outperforms the weighted chain counts in more cases (220 vs. 133). It also yields better average scores than the combined approach in some cases, though its overall average is not as low as the scores of the combined rank.

C. Discussion

Besides the ranking improvement, we argue that the additional information provided by the call chains (stack traces) could help the developer even in the situations when the function itself will be further in the rank. As shown in RQ1, the faulty element is typically found among the highest-ranked chains. The chains are relatively short (12-13 functions) on average, so investigating the chains themselves in more detail is a good approach during the localization process.

TABLE VI
FUNCTION-LEVEL OCHIAI (BASE).

Method	ef	ep	nf	np	Ochiai
forTimeZone	1	6	0	3822	0.3780
getConvertedId	1	6	0	3822	0.3780
getZone	1	131	0	3697	0.0870
getID	1	528	0	3300	0.0435
setDefault	1	3157	0	671	0.0186
getDefault	1	2884	0	944	0.0178

For better illustrating the support provided by call chains during FL, let us consider a real case from our benchmark. Bug number 23 from the Joda-Time Defects4J subject[1] can be located in the method `DateTimeZone.getConvertedId`. This causes one test case, `TestDateTimeZone.testForID_String_old`, to fail.

[1]https://github.com/JodaOrg/joda-time/commit/14dedcb

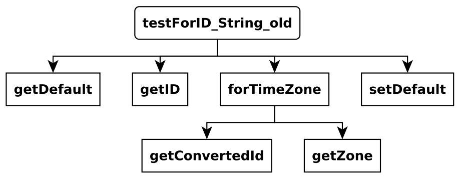

Fig. 7. Call graph of `TestDateTimeZone.testForID_String_old`.

TABLE VII
CHAIN-LEVEL OCHIAI.

Chain	ef	ep	nf	np	Ochiai
forTimeZone→getZone	1	2	0	3826	0.5774
forTimeZone→getConvertedId	1	2	0	3826	0.5774
getID	1	9	0	3819	0.3162
setDefault	1	2882	0	946	0.0186
getDefault	1	2887	0	941	0.0186

The base function-level Ochiai-based FL approach provides the localization scores as shown in Table VI. Apart from the mentioned faulty element, all other functions are listed that have a score > 0. It can be seen that all functions are executed by the single failing test case and several passing ones as well. However, two of them are executed by fewer passing tests, *i.e.,* the faulty one and `DateTimeZone.forTimeZone`, which makes them the most suspicious but indistinguishable from each other.

Fig. 7 shows the relationship of the mentioned functions, which is an excerpt of a call-graph belonging to this program. `DateTimeZone.forTimeZone` is the main function called by the test case, which apart from the faulty `DateTimeZone.getConvertedId` calls `ZoneInfoProvider.getZone` as well. The other directly called functions are setup and tear-down helper functions for the test case. The reason the base algorithm cannot distinguish between `forTimeZone` and `getConvertedId` is that the latter is always called (both in failing and passing test cases) by the former, and no additional information is available to the algorithm.

By introducing the concept of the call chains, `forTimeZone` → `getConvertedId` can be investigated separately along with all other call chains. In particular, `forTimeZone` → `getZone` is interesting as it also relates to the suspicious `forTimeZone` but represents a different context. Table VII presents the localization scores calculated by the first phase of the proposed approach, namely the suspiciousness scores for the chains. Similarly, we only show those chains that have > 0 scores. We can observe that apart from the two mentioned chains, the other (one-element) chains are represented as well because they are also part of the failing test run (and also of some passing runs as well).

Again, the two highest-ranked chains cannot be distinguished from each other because both are executed in the same situations by the failing and passing test cases. However, the next phase of our approach can pinpoint the faulty elements, because it combines the information about suspicious chains with the functions they contain. Namely, in the reapplied

TABLE VIII
FUNCTION-LEVEL OCHIAI (REAPPLIED).

Chain	ef	ep	nf	np	Ochiai
getConvertedId	1	4	4	87692	0.2000
forTimeZone	2	50	3	87646	0.1240
setDefault	1	78	4	87618	0.0503
getZone	1	77	4	87619	0.0506
getID	1	376	4	87320	0.0230

spectrum technique, we treat all suspicious call chains as "failing" and by counting their frequency for each function and the frequency of non-suspicious chains for the same, we can select the most suspicious function. Table VIII shows the statistics for this phase. As can be seen, the highest score is given to `getConvertedId`, followed by `forTimeZone`. The explanation for this can also be seen in the corresponding numbers used by the Ochiai formula. Although `forTimeZone` can be found in more suspicious chains than `getConvertedId` (2 vs. 1) it is found in much more non-suspicious chains as well (50 as opposed to 4). `forTimeZone` is a common method called by many test cases, passing and failing, and present in many different chains, but its specific branching to the faulty `getConvertedId` is less frequent and is typical to the failing test case.

This example is realistic and shows one possible benefit of the approach. However, we had to limit its complexity to be able to clearly explain it. The ranking positions 1 and 2, used in the example, are equally good in practical situations, but in more complex cases, the context provided by the call chains could be much more useful.

In summary, the two outputs produced by our approach (*i.e.,* the ranked list of most suspicious call chains in the first phase and the merged ranked list of functions in the second) can be used in different scenarios — to be empirically evaluated in future work through user studies — to complement hit-based approaches like Ochiai. In a first scenario, the user can start localizing the fault by observing the ranked chains. If the fault is located this way, the context of the investigated chains also informs about the possible ways to fix the defect. If there are many high ranked chains with equally high scores, the user can rely on the final result of the ranked functions from the second phase, and focus on those functions only. In a second scenario, the user starts from the ranked list of functions from the second phase, and if the defect is not easily found, looks at the highest-ranked call chains (and the functions with high ranks in them) for clues about the possible contexts leading to the failed test cases.

D. Threats to Validity

Concerning *construct validity*, we relied on a widely used measurement for SBFL, *i.e.,* the Expense metric. The likelihood of errors in the dataset is limited, as the Defect4J suite is widely used in research, and we carefully reviewed the code for call chain extraction.

The main threat to *internal validity* of this study is that, as explained in Section IV-A, we used 404 defects out of total 438 that were available in Defect4J suite version we used, because we could not compute call chains for all of them. However, the selection was not based on the performances of the proposed algorithm and of Ochiai, and the comparison was in any case performed on the same dataset. Since this limitation was mainly technological due to the nature of the programs in the dataset we used, it will not affect use cases when one could apply the approach in a real usage scenario.

Conclusion validity of this study is supported by the use of appropriate statistical procedures, namely a non-parametric test suitable to deal with ties (Wilcoxon-Pratt test), the use of Holm's correction to avoid fishing the error rate, and Cliff's d effect size.

Concerning *external validity*, while the evaluation of the study has been performed on 404 real defects from Java programs, it is still desirable to replicate the work on a larger and possibly more diverse dataset.

VI. Conclusions

This paper investigates the use of *function call chains* for Spectrum Based Fault Localization (SBFL). Call chains are instances of call stack traces, and these are useful artifacts occurring at runtime which can often help developers in debugging.

Results indicate that except for the two outliers the proposed approach can achieve a significant improvement in terms of the FL expense, about 19-48%, which is even higher in the case of worse ranking positions (over 10).

The highest-ranked call chains provide useful information for a better understanding of the context of the defect (in 69% of the cases, the defective element is in the highest-ranked chains), and could even provide hints for the fixation of the bug. For instance, the call chain indicates which function invokes the defective function when the fault manifests in a failure.

In future work, we plan to conduct user studies to evaluate the practical usefulness of call chains, following scenarios as the ones described in Section V-C. In addition, it could be interesting to evaluate our approach on more benchmarks *e.g.,* Bugs.jar [68], BugsJS [69], etc. The rank combination approach we used could be replaced by a more sophisticated approach taking into account other properties of the spectrum, or by the learning-to-rank model.

The interested reader can find more information and access the benchmark on our website: https://chainfl.github.io

Acknowledgements

This work was partially supported by grants 2018-1.2.1-NKP-2018-00004 "Security Enhancing Technologies for the IoT" funded by the Hungarian National Research, Development and Innovation Office, by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things," and by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary.

REFERENCES

[1] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[3] P. Parmar and M. Patel, "Software fault localization: A survey," *International Journal of Computer Applications*, vol. 154, no. 9, pp. 6–13, 2016.

[4] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *CoRR*, vol. abs/1607.04347, 2016.

[5] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," *Proceedings of the 39th International Conference on Software Engineering*, pp. 609–620, 2017.

[6] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 432–449, Nov. 1997.

[7] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, oct 2016, pp. 267–278.

[8] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209.

[9] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[10] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[11] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. of International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[13] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[14] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Trans. Reliability*, vol. 63, pp. 290–308, 2014.

[15] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 717–727, Aug. 1992.

[16] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, ser. DEFECTS '09. New York, NY, USA: ACM, 2009, pp. 1–5.

[17] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *28th international conference on Software engineering*, ser. ICSE '06. ACM, 2006, pp. 82–91.

[18] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 8:1–8:28, Feb. 2014.

[19] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 118–121.

[20] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," arXiv:1803.09939 [cs.SE], Feb. 2018.

[21] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 177–188.

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[23] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *Managing Requirements Knowledge, International Workshop on(AFIPS)*, vol. 00, 12 1899, p. 539.

[24] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

[25] T.-D. B. Le, F. Thung, and D. Lo, "Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 380–383.

[26] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 314–324.

[27] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[28] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proc. of the 1998 ACM SIGPLAN-SIGSOFT workshop PASTE '98*. ACM, 1998, pp. 83–90.

[29] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–55.

[30] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov 2011, pp. 420–423.

[31] ——, "Tester feedback driven fault localization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 41–50.

[32] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Proceedings of the 4th international conference on Search Based Software Engineering*, ser. SSBSE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 244–258.

[33] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. IEEE Computer Society, 2003, pp. 30–39.

[34] H. LI, "A short introduction to learning to rank," *IEICE Transactions on Information and Systems*, vol. E94.D, no. 10, pp. 1854–1862, 2011.

[35] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 191–200.

[36] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 67–76.

[37] A. Orso, J. A. Jones, M. J. Harrold, and J. T. Stasko, "Gammatella: Visualization of program-execution data for deployed software," in *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, 2004, pp. 699–700.

[38] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 204–214.

[39] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in java," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007514

[40] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, p. 85–96, 1997. [Online]. Available: http://doi.acm.org/10.1145/258916.258924

[41] H. A. de Souza, D. Mutti, M. L. Chaim, and F. Kon, "Contextualizing spectrum-based fault localization," *Inf. Softw. Technol.*, vol. 94, no. C, pp. 245–261, Feb. 2018.

[42] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.

[43] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[44] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[45] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, aug 2015.

[46] ——, "Effective fault localization via mutation analysis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*. New York, New York, USA: ACM Press, 2014, pp. 1293–1300.

[47] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[48] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Formal concept analysis enhances fault localization in software," *Lecture Notes in Computer Science*, vol. 4933, pp. 273–288, 2008.

[49] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 128–137.

[50] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically," in *IEA/AIE*. Springer, 2002, pp. 746–757.

[51] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[52] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[53] H. Agrawal, R. A. D. Millo, and E. H. Spafford, "An execution-backtracking approach to debugging," *IEEE Software*, vol. 8, no. 3, pp. 21–26, May 1991.

[54] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, 2004, pp. 151–158.

[55] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 393–403.

[56] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 485–495.

[57] Á. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization - online appendix," https://chainfl.github.io.

[58] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 261–272.

[59] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[60] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 201–211.

[61] "Cobertura," http://cobertura.github.io/cobertura/, last visited: 2019-10-25.

[62] "Javassist," http://jboss-javassist.github.io/javassist/, last visited: 2019-10-25.

[63] "SoDA library," https://github.com/sed-szeged/soda, last visited: 2019-10-25.

[64] X. Xu, V. Debroy, W. E. Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, pp. 803–827, 2011.

[65] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[66] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal on Statistics*, vol. 6, pp. 65–70, 1979.

[67] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[68] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world Java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: ACM, 2018, pp. 10–13.

[69] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Árpád Beszédes, R. Ferenc, and A. Mesbah, "BugJS: A benchmark of javascript bugs," in *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019.