IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 10, 2019, accepted October 16, 2019, date of publication October 18, 2019, date of current version October 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2948366

An Automatic Localization Tool for Null Pointer Exceptions

JING DUAN[1], SHUJUAN JIANG[2], (Member, IEEE), QIAO YU[3], KAI LU[2], XU ZHANG[2], AND YIWEN YAO[2]

[1]College of Computer and Information, Hohai University, Nanjing 211100, China
[2]School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China
[3]School of Computer Science and Technology, Jiangsu Normal University, Xuzhou 221116, China

Corresponding authors: Jing Duan (jrbduanjing@163.com) and Shujuan Jiang (shjjiang@cumt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61673384.

ABSTRACT Null pointer exceptions are common software faults, but they are dangerous and can cause a program to crash if they occur. In addition, it is hard to find them by simply running test cases. When a null pointer exception occurs, the stack trace stored by the Java runtime environment can help us to locate the cause of the exception. In this paper, firstly, we propose an automatically null pointer exception localization approach guided by stack trace, from the null pointer dereference to the null value assignment point. Secondly, an empirical study is designed to evaluate the effectiveness of the tool on eight open source projects. The experimental results show that the tool is effective in locating the null pointer exception.

INDEX TERMS Null pointer exception, fault localization, stack trace.

I. INTRODUCTION

Exception handling is the main mechanism to improve the software reliability in most programming language. When a semantic constraint of the programming language is violated (e.g., divided by zero), an exception is raised to reveal this error. In this case, the normal control flow of this program is interrupted, and there is an attempt to recover from the error by a specific handler. If the recovery is successful, the program continues the normal control flow, otherwise the program will terminate.

The exceptions in Java can be divided into two categories: application exceptions raised by exceptional conditions in an application, and runtime exceptions raised by Java runtime environment. At present, there are a lot of researches on application exceptions [1], [2], such as the analysis of application exception to provide valuable information for developers. However, there are a few researches on runtime exceptions, although they are often raised in the execution of Java programs [4]. If one program has raised a runtime exception, it indicates that this program contains errors, such as trying to access an array element outside the scope of the index and dereferencing an object that is null. Because Java does not require that methods specify or catch such exceptions, when they are raised during execution, the developers usually do not provide exception handler to handle them. As a result, the program may terminate.

A program execution failed implies that the program under test contains at least a fault. Then, the program debugging should be performed for locating and fixing the fault. In order to precisely and quickly locate the fault, developers should alleviate the output information as much as they can. However, the difference the expression of failure (e.g. assertion not satisfied, null pointer exception) it is, the difference the information they can alleviate. When a null pointer exception occurs, the stack trace stored by the Java runtime environment can help us to locate the cause of this exception. A null pointer exception indicates that a null value variable is dereferenced in an execution. To locate the fault that results in the exception, a conventional step is to find the statement of Null Pointer Assignment (NPA). The NPA helps developer to analyze whether a null value assignment is appropriate, which indicates the post-processing is not correct, or not, which indicates the statements correspond to NPA should be reexamined. Therefore, finding NPA is an essential step in locating the faults that result in null pointer exception.

Null pointer exceptions are common software faults, but they are dangerous and can cause a program to crash if it occurs. And it's also hard to find by running test cases.

When a null pointer exception occurs, the stack trace stored by the Java runtime environment can help us to locate the cause of this exception. The information of stack trace includes the code line that throws the exception and the

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

method that contains this line of code. The information of stack trace also contains all methods currently on the run-time stack, along with the statements in those methods at which the method calls were made. Furthermore, the methods that have been called during the execution and have already normal returned will not appear in the stack trace. However, stack trace does not contain the information about control flow through the method. Thus, during the inspection of the methods to locate the origin of the exception, the developer may miss those methods that were involved in the execution, but not in the stack trace, or may not understand correctly the complex control flow in the program. Thus, to locate the origin of the exception, the developer must inspect the execution manually and attempt to understand the control flow through the calling methods.

Program slicing [3] is widely applied in program analyzing, testing and debugging. It can be classified as static slicing and dynamic slicing. Static slicing computes with respect to the point of failure and works backward, identifying all statements that could affect the behavior at the point of failure until the bug is found. Dynamic slicing [24] computes with respect to the execution trace of this executing, identifies the statements that could affect the behavior at the point of failure for the given inputs.

The program slicing can effectively separate the fault-related statements by identifying the set of statements in the program that affect the specific position variables, thus becoming an effective tool for assisting software fault location [22].

Fault localization can use both static backward slicing and dynamic backward slicing, but dynamic backward slicing comes with a price: The program must be instrumented to gather an execution trace, and dynamic slicing itself can be very slow, which may be unacceptable for large programs [20].

To evaluate the effectiveness of our tool on null pointer exception localization, we conduct experiments on 15 open source projects. The empirical results show that our tool is effective in locating the null pointer exception.

The contributions of the paper include:

(1) It presents an automatic localization tool for null pointer exceptions guided by stack trace.
(2) An empirical study is designed to evaluate the effectiveness of our tool for localization null pointer exceptions.

II. BACKGROUND

This section summarizes some relevant background information on program analysis and fault localization.

A. DEFINITIONS

Definition 1 (Control Flow Graph (CFG)): A CFG G is a graphical representation of the logic execution of the program, and can be presented in a tetrad (N, E, Entry, Exit). N is the set of nodes presenting statements, $E \subseteq N \times N$ is the set of the edges presenting the control relations between statements, Entry is entry node in the program, and Exit is the exit node in the program.

For each intra-procedure CFG, we add a call edge and a return edge according to the function call sequence, and then construct the system CFG which is ICFG.

Definition 2: Let n_1, n_2 be the nodes in a CFG, whether n_2 executing is determined by the executing status of n_1, then n_2 control dependent on n_1, denoted by CD (n_2, n_1).

Definition 3: Let n_1, n_2 be the nodes in a CFG and$v$be a variable. If v defined in n_1 may be directly used in n_2 during their execution; there is an executable path between n_1 and n_2, and there is no re-definition of v from n_1 to n_2, then n_2 is directly data dependent on n_1 about v, denoted by DD (n_2, n_1).

Definition 4 (Program Dependency Graph (PDG)): A PDG of program P is a directed graph, which can be represented by a binary group (N′, E′), where N′ is a node set, $N' = N$ (N is the set of nodes presenting statements in the CFG of P)); edge set E′ represents the dependencies between nodes. That is, $\langle n_1, n_2 \rangle \in E'$ represents CD (n_1, n_2) or DD (n_1, n_2).

To construct System Dependence Graph (SDG), for each PDG, we need add the edges of control dependence, which are calling edge and return edge according to the function call sequence, and the edges of data dependence, which is due to subroutine calls, parameter passing, and global variables.

Definition 5 (Program Slicing): in general, program slicing which can meet two requirements below is M. Weiser's program slicing [3]:

1) A program slicing has a specific slicing criteria presented as $\langle n, V \rangle$. V presents the set of variables used or defined in the point n, and n presents a node in the program (usually a statement in the program).
2) The program slicing S of the program P can be achieved by deleting zero or several statements in P. However, we must be sure that the behaviors of program P and the slice S regarding to slicing criteria $\langle n, V \rangle$ are the same.

B. ASSIGNMENTS AND DEREFERENCES

In Java, a null pointer exception is thrown at a statement that dereferences a variable or a field that has a null value, such as statement $x.m$, where $x = \text{null}$. A null pointer assignment (NPA) is a statement where a null value is generated, such as "$x = \text{null}$", "return null" and "foo(null)" (a null value for an actual parameter at a method call).

Figure 1 illustrates the information that our approach provides. An null pointer exception was raised at reference statement $S_r(\varepsilon)$, there is exactly one statement $S_a(\varepsilon)$ at which the null value that caused the exception is assigned.

For example, for a null pointer exception at statement $x.m$ was raised, $S_a(\varepsilon)$ is the statement that assigned a null value to x. The goal is to locate this source statement $S_a(\varepsilon)$.

Our approach performs static backward data flow analysis guided by stack trace, starting at $S_r(\varepsilon)$. If the analysis obtains a unique NPA, this one is the definite NPA. If it obtains more than one NPA, there may be multiple possible NPAs that

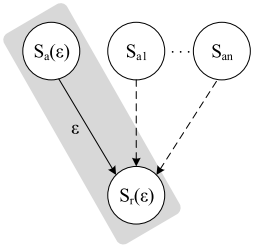


FIGURE 1. NPAs identified by our approach.

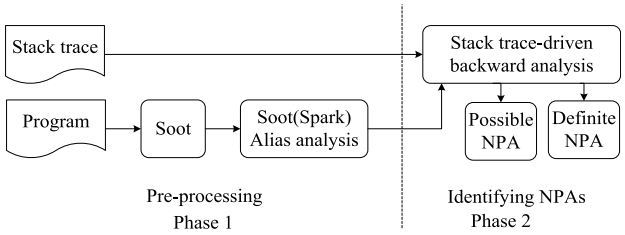


FIGURE 2. The framework of our approach.

could have caused the exception. Since stack trace does not provide control flow information within the method, the analysis cannot determine which NPA caused the exception. But it can conclude that at least one of them caused the exception.

In fig.1, the shaded area represents the execution on which the null pointer exception was thrown at $S_r(\varepsilon)$. The figure shows that the analysis was able to identify the exact statement $S_a(\varepsilon)$ that caused the null pointer exception at $S_r(\varepsilon)$. Thus, $S_a(\varepsilon)$ is a definite NPA. Moreover, we may also obtain more than one NPAs, such as $S_a(\varepsilon)$, S_{a1}, S_{a2}, ..., S_{an}. These NPAs are possible NPAs.

III. OUR APPROACH

In this section, we first introduce the localization approach for null pointer exceptions guided by stack trace. Then we give a case study to show how to locate the null pointer exception using our approach.

A. OVERVIEW

Fig.2 presents an overview of our approach. Fig.2 presents an overview of our approach. Phase 1 is a pre-processing, we first use Soot [5] framework and Spark pointer analysis to get the information of source program, such as control flow information, data flow information and alias information. Phase 2 performs a backward data flow analysis, guided by stack trace, to identify the definite NPA and possible NPAs.

Soot [5] is a static analysis framework developed by Mcgill University, Canada. Here, this paper uses it to perform lexical analysis and syntax analysis, and get the information that needs in phase 2. Spark (Soot Pointer Analysis Research Kit) is a part of Soot, which is used to perform alias analysis.

1) PHASE 1: PRE-PROCESSING

When more than one variable name can be used to access a data location in memory, the names are called aliases. Aliasing is harmful to readability because modifying the data through one name implicitly modifies the values associated with all aliased names, which may not be expected by the developer. Because there may be many aliases in a program, this makes it very difficult to understand and analyze programs in practice.

First, we use Soot to perform lexical analysis and syntax analysis on the program with a null pointer exception, and construct the ICFG. Then we can get the variable that raised the null pointer exception according to the stack trace information. And then we use Spark tool to perform aliases analysis on the variable that throws the null pointer exception.

2) PHASE 2: IDENTIFYING DEFINITE AND POSSIBLE NPAS

When runtime exceptions occur, the information in the stack trace can help us to locate the causes of exceptions. However, the granularity of the information in the stack trace may be too coarse to locate the causes. The stack trace contains only the methods involved in the execution and the statement where a method call was made, but does not contain information about the control flow through the method. It can reduce some branches and methods that do not traverse in the execution according to the stack trace information, and then perform analysis guided by stack trace. Meanwhile, this paper performs aliases analysis to improve the accuracy of localization. This section discusses the state constraint transformation and null pointer localization algorithm.

B. STATE CONSTRAINT TRANSFORMATION

The focus of the null pointer exception localization algorithm is to put the variable that raised a null pointer exception in state constraints. During the analysis of program, the state constraint continuously converts until it converts to null. This means that we find the null pointer assignment statement. In this approach, the state constraint transition can be divided into five types, as shown in Table 1. For each type, the table shows type number and description (columns 1-2). Then it shows example and transition way (columns 3-4).

For Type 1: if the state constraint variable is a simple assignment variable, for example, $a = b$, then transforms a to b.

For Type 2: if the state constraint variable is a parameter of a function, then transforms actual parameter to formal parameter when entering the function, and transforms formal parameter to actual parameter when return from the function.

For Type 3: if the state constraint is a return value of a function, for example, $a =$ method( ), then transforms a to the return value of method( ).

For Type 4: if the state constraint is an array variable, for example, $b = c$, $a = b[2]$, then transforms a to b and then to $c[2]$, and add b to *States* set, where *States* is a set of variables that raised an exception.

For Type 5: if the state constraint is a dereference of a member variable, for example, $b = c$, $a = b.name$, then transforms a to $b.name$ and then to c.name, and add b to *States* set.

TABLE 1. State constraint transition.

Type	Description	Example	Transition way
1	State constraint variable is a simple assignment variable	*a = b*	Transform *a* to *b*
2	State constraint variable is a parameter of a function	*method(a); method(StringBuffer b) {b.append(2);}*	When entering method(), transform *a* to *b*; when method() finished, transform *b* to *a*
3	State constraint is a return value of a function	*a = method()*	Transform *a* to the return value of method()
4	State constraint is an array variable	*b = c; a = b[2]*	Transform *a* to *b* and then to *c*[2]; add *b* to *States*
5	State constraint is a dereference of a member variable	*b = c; a = b.name*	Transform *a* to *b*.name and then to *c*.name; add *b* to *States*

C. NULL POINTER EXCEPTION LOCALIZATION ALGORITHM

Algorithm INPA-ST in Fig. 3 presents an overview of our null pointer localization algorithm.

Starting at a node in ICFG that raised an exception, this approach performs a backward and path-sensitive analysis to determine whether v is assigned as null (lines 1-4). First, the algorithm abstracts information from stack traces that we stored in a file when an exception occurs and gets *Element* (*Unit*, *States*), where *Unit* is a node in ICFG (*UnitGraph), States* is a set of variable v that raises an exception and its state constraint that is $\langle v = \text{null}\rangle$. The algorithm starts with a state constraint that is $\langle v = \text{null}\rangle$, and updates states during the path traversal. If the updated state becomes inconsistent, the path is infeasible and the analysis stops traversing the path. When the algorithm encounters a *null* assignment to *v,*the state constraint becomes $\langle null = null\rangle$, which is represented an ⟨true⟩. This means that we find the origin of *null* value.

If the algorithm obtains a NPA, then the NPA is the definite NPA that raised the exception; if it obtains more than one NPAs, then the NPA that raised the exception must be one of the NPAs; if it does not obtain any NPA, then the variable that raised the exception is not initialized.

Starting at a statement that dereferences variable v, Analysis (*Element*) performs a backward and path-sensitive analysis to determine whether state constraint transition in a function (lines 5-39). The algorithm initializes the *worklist* with *Element*, and then performs backward analysis on each *predElement* that is a predecessor of *Element* (lines 7-34).

If the *Unit* of *predElement* is an assignment statement, then the algorithm transforms state constraint according to type 1, type 4 and type 5 in Table 1(lines 11-12). If the *Unit* of *predElement* is a call statement that invokes M, which M has been analyzed, then the algorithm transforms State constraint according to *OutgoingStates*. If M has not been analyzed, the algorithm pushes M to function call stack and recursive calls the algorithm (lines 14-23). If the *Unit* of *predElement* is a return statement, then the algorithm transforms state constraint according to type 3 in Table 1 when it is a return statement (lines 24-26). If the *Unit* of *predElement* is the entry of a method and not analyzed, then the algorithm assigns the *States* of *predElement* to *returnStates*, otherwise adds the *States* of *predElement* to *returnStates*(lines 27-31).

If the function call stack is empty and the stack trace is not empty, the algorithm pops a statement from stack trace and assigns the statement to the *Unit* of element. Meanwhile, assigns the *returnStates* of the call function to the States of element. And then recursive call the algorithm (lines 35-37). If *returnStates* is ⟨true⟩, then add *preElement.Unit* to NPA, which means that we find the origins of null values (line 38).

D. A CASE STUDY

In this section, we performs a case study to show our approach how to locate the null pointer exception when there is a null assignment statement, more than one null assignment statements, and no any null assignment statement. It also shows how to locate the fault when there is an alias in the variables that raised null pointer exceptions. Fig.4 shows an example program, and Fig.5 shows its ICFG.

1) ONE NULL ASSIGNMENT STATEMENT

Based on the above example and its ICFG in Fig.5, if a null pointer exception occurs without any input when executing the program, its stack trace is listed as follows:

```
Exception in thread "main" java.lang.NullPointerException
    at example.ExampleException.
       method3(ExampleException. java:35)
    at example. ExampleException.
       method2(ExampleException.java:32)
    at example. ExampleException.
       method0(ExampleException.java:22)
    at example. ExampleException.
       main(ExampleException.java:5)
```

The stack trace shows that the null pointer exception is raised in method3() at line 35 (i.e. the dereferenced field *bigInt1* was null). The stack trace also shows that method2(), method0() and main() are still on the stack when the exception occurs. The function call sequence is that main() called method0() at line 5, mehod0() called method2() at line 22, method2() called method3() at line 32.

```
Algorithm INPA-ST
Input:  EST: Stack trace when an exception raised
        UnitGraph: System control flow graph
Output: NPA: A set of node or statement that assign a null to a variable
        NotInitializedStates: A variable that is not initialized
Declare: States: A set of variable v that raised an exception and its state constraint, that v is null
         Element (Unit, States): where Unit is a node in UnitGraph,States is{v,<v=null>}
         CS: Call stack of methods
         CallRecords(calledMethod, OutgoingStates): Record a called method and its analysis result OutgoingStates
         VisitRecords: Record a node has been analyzed
         predElement: a predecessors of Element
Begin
1.   NPA=Φ;
2.   Scan stack trace EST and get Element(Unit, States) according to Soot;
3.   Initialize Element.States={v,<v=null>}; //v is a variable that raised a null pointer exception
4.   NPA = Analysis (Element); // If we find a NPA, then the NPA is the definite NPA that raised the exception; if we find more than one NPAs, then the NPA
      that raised the exception must be one of the NPAs
End
Function States Analysis(Element)
Input: Element(Unit, States)
Output: NPA - A node that assign a null to a variable
        returnStates- A set of states constraint after analyzing a method, which contains some null value assignments or NotInitializedState
Declare: int i=5;
Begin
5.   Initialize worklist with Element;
6.   Initialize returnState=NotInitializedStates;
7.   while worklist ≠ Φ do
8.     Remove Element from worklist and add Element to visitRecords;
9.     predElement=get predecessors of Element; // obtain from UnitGraph
10.    for each predElement do
11.    switch(i) { case 0: // if predElement.Unit is an assignment statement then
12.                  Transform1(predElement.States); // State constraint transition type 1,type 4 and type 5 in Table 1
13.                  break;
14.              case 1: // if predElement.Unit is a call node that invokes M then
15.                  search M in CallRecords(calledMethod, OutgoingStates);
16.                  if found then Element.States=OutgoingStates;
17.                  else CS.push(M)
18.                       Element.Unit = exit node of M;
19.                       Transform2(PredElement.States.); // State constraint transition type 2 in Table 1
20.                       predElement.States.=Analysis (Element);
21.                       add (M, States) to CallRecords;
22.                       CS.pop(M); end if
23.                  break;
24.              case 2: // if predElement.Unit is a return statement then
25.                   Transform3(PredElement.States); // State constraint transition type 3 in Table 1 endif
26.                  break;
27.              case 3: // if predElement.Unit is the entry of a method then
28.                  if the entry of a method is visited firstly then
29.                       returnStates= predElement.States;
30.                  else add predElement.States to returnStates end if
31.                  break; }
32.          if predElement is not in VisitRecords then add predElement to worklist end if
33.        end for
34.    end while
35.  if CS=Φ and EST ≠Φ then
36.    pop(EST) ; Element.Unit = callSite of Unit;
37.    Element.State = returnStates; Analysis (Element); end if
38.  if returnStates = <true> then add preElement.Unit to NPA end if
39.  return returnStates;
End
```

FIGURE 3. The algorithm of locating NPA.

First, we can find that variable *bigInt1*raises a null pointer exception, and *Element.States*(in algorithm INPA-ST) is initialized with $\{bigInt1, \langle bigInt1 = null\rangle\}$. Along with edge (35->34), the algorithm reaches the entry of method3(). Pop up the stack, the algorithm traverses line 32. Along with edge (32->31), the algorithm reaches line 31 and the value of *Element.States*is changed from $\{bigInt1, \langle bigInt1 = null\rangle\}$ to $\{bigInt2, \langle bigInt2 = null\rangle\}$. Along with edge (31->30), the algorithm reaches the entry of method2(). Pop up the stack, the algorithm traverses line 22. Along with edge (22->21), the algorithm reaches line 21, where the value of *Element.States* is not changed. Along with edge (21->20), the algorithm reaches the entry of method0(). Pop up the stack, the algorithm traverses line 5. Along with edges (5->4->3), the algorithm reaches line 3 and enters a constructor where it finds that *bigInt2* is assigned *null* at line 17, the value of *Element.States* is changed from $\{bigInt2, \langle bigInt2 = null\rangle\}$ to $\{bigInt2, \langle true\rangle\}$. At the same time, the stack trace is empty. The algorithm terminates and it has only found one *null* assignment statement that is line 17.

2) MORE THAN ONE NULL ASSIGNMENT STATEMENTS

If a null pointer exception occurs with input x when executing the program, its stack trace is listed as follows:

```
package example;
public class Item {
1    public String name = "a";
          }
  public class ExampleException {
     public Integer bigInt1, bigInt2, bigInt3;
     public int i;
2     public static void main(String[ ] args) {
3           ExampleExceptionexample1 = new
                  ExampleException (args.length);
4       if(args.length==0){
5           example1.method0();}
6       else if (args.length==1){
7           example1.method1((args.length)/2);}
8       else if (args.length==2) {
9           bigInt3.intValue();}
         else {
10          Item b =new Item();
11          Item a = b;
12          b.name=null;
13          System.out.println(a.name.length());}
14    }
15    public ExampleException (int i) {
16        bigInt1 = null;
17        bigInt2 = null;
18        this.i=i;
19    }
20    public void method0(){
21        bigInt1 = new Integer(i);
22        method2();
23    }
24    public void method1(int number){
25        if(number>1){
26         bigInt1 = new Integer(number);}
          else{
27         bigInt2 = null;
           }
28        method2();
29    }
30    public void method2(){
31        bigInt1 = bigInt2;
32        method3();
33    }
34    public void method3(){
35        bigInt1.intValue();
36    }
  }
```

FIGURE 4. An example program.

```
Exception in thread "main" java.lang.NullPointerException
    at example. ExampleException.
       method3(ExampleException.java:35)
    at example. ExampleException.
       method2(ExampleException.java:32)
    at example. ExampleException.
       method1(ExampleException.java:28)
    at example. ExampleException.
       main(ExampleException.java:7)
```

The stack trace shows that the null pointer exception is raised in method3() at line 35(i.e. the dereferenced field *bigInt1* was *null*). First, we can obtain that variable *bigInt1*raises a null pointer exception, and *Element.States* is initialized with $\{bigInt1, \langle bigInt1 = null\rangle\}$. Along with edge (35->34), the algorithm reaches the entry of method3(). Pop up the stack, the algorithm reaches statement 32.

On traversing edge (32->31), the algorithm reaches statement 31 and the value of *Element.States* is changed from $\{bigInt1, \langle bigInt1 = null\rangle\}$ to $\{bigInt2, \langle bigInt2 = null\rangle\}$. Along with edge (31->30), the algorithm reaches the entry of method2().

Pop up the stack, the algorithm reaches statement 28, which has two predecessors (statements 26 and 27). Because statement 27 is a *null* assignment statement, the value of *Element.States* is changed from $\{bigInt2, \langle bigInt2 = null\rangle\}$ to $\{bigInt2, \langle true\rangle\}$, the algorithm terminates on the path and has found a *null* assignment statement that is line 27.

Following back along edges (26->25->24), the algorithm reaches the entry of method1(). Pop up the stack, the algorithm traverses statement 7. On traversing edges (7->6->4->3), the algorithm reaches statement 3 and enters a constructor where it finds that *bigInt2*is assigned a *null* at line 17, the value of *Element.States* is changed from $\{bigInt2, \langle bigInt2 = null\rangle\}$ to $\{bigInt2, \langle true\rangle\}$. At the same time, the stack trace is empty. The algorithm terminates and it has found another *null*assignment statements. So we have found two possible origins of null values that are statements 17 and 27.

3) NO ANY NULL ASSIGNMENT STATEMENT

To illustrate, consider the example program in Fig. 4. Executing the program with input 12 results in the following null pointer exception and stack trace:

```
Exception in thread "main" java.lang.NullPointerException
atexample.ExampleException.
      main(Example.java:9)
```

The stack trace shows that the null pointer exception is raised in main( ) at line 9. First, we can obtain that variable *bigInt3*raises a null pointer exception, and *Element.States* is initialized with $\{bigInt3, \langle bigInt3 = null\rangle\}$. Along the edges (9->8->6->4->3), the algorithm reaches statement 3 and enters a constructor where it does not find any null assignment statements to *bigInt3*.Then following back along edge (3->2), the algorithm reaches the entry of main(), meanwhile the stack trace is empty. So the reason of raising the null pointer exception is that variable *bigInt3* is not initialized.

4) EXIST ALIASES

To illustrate, consider the example program in Fig. 4. Executing the program with inputs 1, 2, 3 results in the following null pointer exception and stack trace:

```
Exception in thread "main" java.lang.NullPointerException
        at example.ExampleException.
     main(Example.java:13)
```

The stack trace shows that the null pointer exception is raised in main() at line 13. If the algorithm Follows back along edges (13->12->11->10->8->6->4->3->2), which reaches the entry of main(), it will not find any null assignment statements.

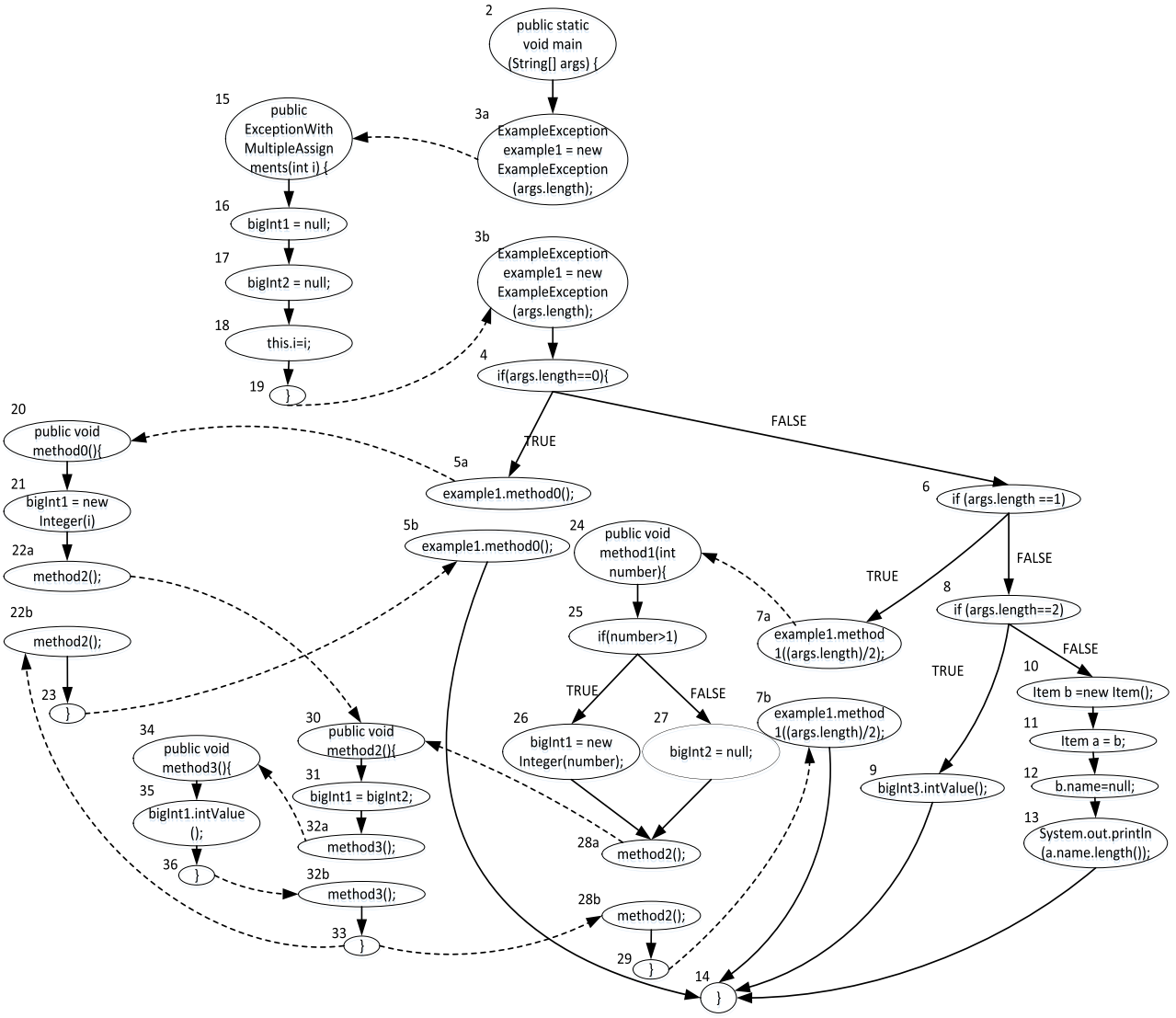


FIGURE 5. Control flow graph of the example program.

The variable *a.name*raised the exception at line 13, but it assigned a null to *b.name* at line 12. When performing backward analysis, the algorithm only searches variable *a.name* whether it is assigned a null. Therefore, it cannot find the null assignment statement.

If the algorithm performs aliases analysis firstly, it will find the null assignment statement easily. The analysis begins from statement 13, and *Element.States* is initialized with $\{a.name, \langle a.name = null\rangle\}$. From aliases analysis, we obtain that variables *a.name* and *b.name* are aliases after line 12. Following back along edge (13->12), the algorithm reaches statement 12, where assigns a null to *b.name,*and the value of *Element.States* is changed from $\{a.name, \langle a.name = null\rangle\}$ to $\{b.name, \langle true\rangle\}$. The statement 12 is the result that we search. Following back along edges (12->11->10->8->6->4->3->2), which reaches the entry of main(), it will not find any additional null assignment statements. Meanwhile the stack trace is empty, the algorithm terminates and it obtains the *null* assignment statement 12.

IV. EMPIRICAL STUDY

To evaluate the effectiveness of our tool, we implemented three tools: the first tool is a program slicing tool JSST which is guided by stack trace; the second tool is a localization tool SSoot that using Soot [5] to locate the NPA, without using stack trace information; The third tool is our localization tool INPA-ST for null pointer exceptions that describes in section III.

We also conducted an empirical study on open-source projects SourceForge [6], the BUGZILLA defect reports for the projects, and the null pointer data set in [7]. This section describes the experimental subjects, experimental design and implementation, and then presents the results of the study.

TABLE 2. Subjects for the experiment.

Project Release	Loc	the Class in Which Raised a Null Pointer Exception	The Line in Which a Null Pointer Exception raised
Ant 1.6.0	174,225	GenericDeploymentTool	788
Ant 1.6.2	152,483	DOMElementWriter	174
Ant 1.6.3	157,646	DirectoryScanner	875
Ant 1.6.5	157,678	ImportTask	96
Tomcat 5.0.18	148,255	DataSourceRealm	369
Tomcat 5.0.28	151,772	FileStore	217
Ant 1.4	77,980	UnknownElement	148
Ant 1.5.1	131,419	CBZip2InputStream	327
Ant 1.6.1	195,306	DOMElementWriter	209
Ant 1.6.2	152,483	CBZip2InputStream	285
Ant 1.7.1	179,889	DefaultLogger	339
JFreeChart 1.0.2	223,869	Bug2	13
Checkstyle 4.2	47,871	ASTFactory	266
JFreeChart 1.0.2	223,869	StackedXYAreaRenderer$Stacked--XYAreaRendererState	138
JRefactory 2.9.18	231,338	PrettyPrintVisitor	1,177
Freemaker	64,442	NodeModel	373
Mckoi 1.0.6	94,681	StackedXYAreaRendererState.java	138
Jode 1.0.9	44,937	ConstantPool	221

A. EXPERIMENTAL SUBJECTS

All experimental subjects come from open-source projects SourceForge [6] and the null pointer data set in [7]. We looked through the BUGZILLA repository for defect reports that show appearance of null pointer exceptions. This paper chooses those that have stack trace, the source line numbers, and can be analyzed by Soot. Table 2 lists the 18 subjects from eight projects which are used for the study. The table shows, project release and Loc (columns 1-2), the class in which raised a null pointer exception (column 3), and the line number the null pointer exception raised (column 4). The sizes of the applications in terms of the number of nodes in the ICFG vary from approximately 44,937 for Jode 1.0.9 to over 223,869 for JFreeChart 1.0.2. Our experiments are conducted on Windows Server 2008 R2 with 32GB of RAM, Open JDK 1.7.

B. EXPERIMENTAL DESIGN AND IMPLEMENTATION

To evaluate the effectiveness of our tool, we design two research questions (RQ1 and RQ2). To evaluate the correctness of our tool, we design the third questions (RQ3).

- RQ1: Which one is more effective in locating the NPAs, JSST or INPA-ST ?
- RQ2: Can the stack trace affect the efficiency in locating the NPAs?
- RQ3: Are the results of our approach correct in locating the NPAs?

1) JSST

Weiser [3] suggests that when developers debug a program, they often use backward slicing: They start from the statement of failure and work backward, following data and control dependences, examining all codes that might affect the behavior of the failure statement until they find the bug. Susan horwitz [20] studied many kinds slicing and showed that callstack-sensitive slicing can dramatically decrease slice sizes: On average, a callstack-sensitive slice is about 0.31 time the size of the corresponding full slice, down to just 0.06 time in the best case.

When a null pointer exception occurs, the stack trace stored by the Java runtime environment includes the code line that throws the exception and the method that contains this line of code. The information of stack trace also contains all methods currently on the runtime stack, along with the statements in those methods at which the method calls were made. Furthermore, the methods that have been called during the execution and have already normal returned will not appear in the stack trace.

Therefore we can perform callstack-sensitive slicing using stack trace information. First, we infer program execution path according to stack trace, and classify the methods and statements as may-execute, partial-execute and not-execute.

- May-execute: All of the statements in these methods maybe executed, but we cannot determine the execution path within these methods and whether a specific statement is executed or not. There are two reasons for this case: only relying on the stack trace information, the execution paths within the method cannot be inferred; due to the polymorphic feature of object-oriented languages, even when we are sure that one invoke statement is executed, we still could not know the callee method belongs to which class if it is executed and normally return. Although the points-to analysis can be used to determine the pointer type, the result is approximate but not accurate.
- Partial-execute: In this condition, the methods are definitely executed, but some statements in the method maybe executed while the other part of statements are definitely not executed. These methods should be called only once in the execution, and terminated by the raised exception, so they must appear at the stack trace. As a result, by observing the termination nodes, we can determine which statements in the method are executed and which are not.
- Not-execute: These methods are certainly not executed during the program execution. They are just the methods that are never called in the may-execute methods and the statements which may be executed in partial-execute methods. Since they are not executed at all and have nothing to do with the cause of exception, they will be ignored in the slicing phase afterwards.

We implemented program slicing tool JSST. Firstly, we used Soot [5] to handle Java bytecode of the program and construct the CFGs and call graph. Then we mapped the methods in the stack trace and their corresponding lines into the source code, and then identified the execution state of all methods in the system (i.e. may-execute, partial-execute and not-execute). Finally, we performed control dependence and data dependence analysis to construct the simplified SDG which only contained may-execute, partial-execute methods, and then used our improved slicing algorithm to perform backward program slicing. By doing this, not only the size of slice will be reduced without affecting the content we

really cared about, but also the SDG would be simplified. The detail algorithm had been presented in our previous research work [23].

For JSST tool, program slicing criteria is C = (x, *v*) where *x* is a statement in program P that raised a null pointer exception and *v* is the variable in *x* that raised a null pointer exception, where *v* = null.

2) SSOOT

We implemented a localization tool SSoot. Firstly, we performed program slicing using traditional methods [3] from the point that raised a null pointer exception. Secondly, we performed the null pointer analysis and alias analysis on the sliced programs using Soot.

3) INPA-ST

We also implemented our localization tool INPA-ST. Firstly, we adopted Soot to construct the CFG, and then constructed the ICFG according to the method calling graph. Secondly, we used Spark tool to perform aliases analysis on the variables that raised the null pointer exception. Finally, we implemented the algorithm INPA-ST to perform null dereference analysis and locate the NPA.

To answer RQ1, we compare the results of JSST with that of INPA-ST.

To answer RQ2, we compare the results of SSoot with that of INPA-ST.

To answer RQ3, we manually check all the results to verify the correctness.

C. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we answer the three proposed research questions through analyzing the experimental results to verify the effectiveness and correctness of our approach.

RQ1 (Which one is More Effective in Locating NPA, JSST or INPA-ST?): Program slicing is an effective tool for assisting software fault location. Susan horwitz [20] studied many kinds of program slicing and showed that callstack-sensitive slicing could dramatically decrease slice sizes. When a null pointer exception raised, the stack trace includes all methods currently on the runtime stack, along with the statements in those methods at which the method calls were made. Therefore, we can using the method calling information to perform program slicing which is very similar to callstack-sensitive slicing. The difference is that the stack trace does not include the methods that have been called during the execution and have already normal returned.

We then compare the result of JSST with the result of INPA-ST.

Table 3 presents the data that we collected. The Table shows, for each subject, project release, Loc (columns 1-2), the number of code lines that need to examine and the time it needs for the NPA using JSST approach (columns 3-4), the number of code lines that need to examine and the time it needs for the NPA using INPA-ST (columns 5-6), and the time that INPA-ST needs / the time of JSST needs rate (column 7).

TABLE 3. The comparison of JSST with INPA-ST.

Project release	LOC	JSST size[a]	JSST time[b] (ms)	ST[c] size	ST[d] time (ms)	Rate[e]
Ant 1.6.0	174,225	15,525	67,147	1	13,590	20%
Ant 1.6.2	152,483	16,372	24,713	1	14,031	57%
Ant 1.6.3	157,646	17,051	73,934	2	15,244	21%
Ant 1.6.5	157,678	16,885	40,846	1	15,056	37%
Ant 1.4	77,980	1,390	24,432	6	6,334	26%
Ant 1.5.1	131,419	263	20,332	1	3,698	18%
Ant 1.6.1	195,306	1,789	25,512	1	13,223	52%
Ant 1.6.2	152,483	2,634	31,226	1	14,276	46%
Ant 1.7.1	179,889	8,541	55,839	1	14,450	26%
JFreeChart 1.0.2	223,869	92	32,561	1	4,526	14%
Checkstyle 4.2	47,871	4,758	73,856	13	11,093	15%
JFreeChart 1.0.2	223,869	1,513	32,187	1	6,835	21%
JRefactory 2.9.18	231,338	14,510	176,021	4	19,379	11%
Freemaker	64,442	2,607	110,831	3	17,733	16%
Mckoi 1.0.6	94,681	76	55,686	1	6,125	11%
Jode 1.0.9	44,937	5,054	129,403	1	16,822	13%

[a]JSST size: number of code lines that need to examine for the NPA using JSST tool.
[b]JSST time: the time that needs for the NPA using JSST tool.
[c]ST size: number of code lines that need to examine for the NPA using INPA-ST.
[d]ST time: the time that needs for the NPA using INPA_ST.
[e]Rate: ST time /JSST time

As listed in Table 3, we find that INPA-ST performs better than JSST. For the first project, the code that a programmer has to examine 15525 lines during locating the NPA using JSST, but the programmer only need examine one line using INPA-ST, and the time of INPA-ST needs is about 20% time that of JSST needs. For Mckoi 1.0.6 project, the code that a programmer has to examine 76 lines during locating the NPA using JSST, but the programmer only need examine one line using INPA-ST, and the time of INPA-ST needs is about 11% time that of JSST needs.

Therefore, INPA-ST is more effective in locating NPA than JSST.

RQ2 (Can the Stack Trace Affect the Efficiency in Locating the NPA?): In this section, we examine the effect of stack trace on locating null pointer exception.

This paper analyzed each execution using INPA-ST tool and computed definite and possible NPAs. Table 3 presents the data that we collected. The Table shows, for each subject, project release and Loc (columns 1-2), the number of code lines that need to examine and the time it needs for null pointer exception using SSoot approach (columns 3-4), the number of code lines that need to examine and the time it needs for null pointer exception using our method (INPA-ST) (columns 5-6), and the time that INPA-ST needs / the time of SSoot needs rate (column 7).

As listed in Table 4, we find that our approach performs better than the compared approach SSoot. For example, for the first project, the code that a programmer has to examine 117 lines during localization the null pointer exception using SSoot, but the programmer only need examine one line using our approach, and the time of INPA-ST needs is about the same as the time of SSoot needs. For Ant 1.7.1 project, the code that a programmer has to examine 6 lines during localization the null pointer exception using SSoot, but the

TABLE 4. The comparison of SSOOT with INPA-ST.

Project release	LOC	SS[a] size	SS[b] time (ms)	ST[c] size	ST[d] time (ms)	[e]Rate
Ant 1.6.0	174,225	117	13,917	1	13,590	97%
Ant 1.6.2	152,483	40	14,625	1	14,031	95%
Ant 1.6.3	157,646	124	15,321	2	15,244	99%
Ant 1.6.5	157,678	5	15,431	1	15,056	97%
Ant 1.4	77,980	44	6,350	6	6,334	99%
Ant 1.5.1	131,419	14	7,162	1	3,698	52%
Ant 1.6.1	195,306	70	13,823	1	13,223	96%
Ant 1.6.2	152,483	22	14,385	1	14,276	99%
Ant 1.7.1	179,889	6	20,454	1	14,450	71%
JFreeChart 1.0.2	223,869	1,163	8,223	1	4,526	55%
Checkstyle 4.2	47,871	3,285	11,499	13	11,093	96%
JFreeChart 1.0.2	223,869	1,039	8,164	1	6,835	84%
JRefactory 2.9.18	231,338	12,220	21,110	4	19,379	92%
Freemaker	64,442	41	17,956	3	17,733	99%
Mckoi 1.0.6	94,681	2	6,617	1	6,125	93%
Jode 1.0.9	44,937	79	17,043	1	16,822	99%

[a]SS size: number of code lines that need to examine for null pointer exception using SSoot approach.
[b]SS time: the time that needs for null pointer exception using SSoot approach.
[c]ST size: number of code lines that need to examine for null pointer exception using our approach (INPA-ST).
[d]ST time: the time that needs for null pointer exception using our approach (INPA_ST).
[e]Rate: ST time /SS time.

programmer only need examine one line using our approach, and the time of INPA-ST needs is about 71% time that of SSoot needs.

For the four projects in the middle of Table 4 (JFreeChart 1.0.2, Checkstyle 4.2, JFreeChart 1.0.2, JRefactory 2.9.18), the size of SS is much more lines than that of rest of the projects that we studied. The reason is that they are not performed aliases analysis due to not find the main class in the projects. Alias analysis by Soot must start from main class, so we do not perform aliases analysis for the four projects.

As listed in Table 4, we give the time that needs for locating the NPA using two different methods. We can find that the time our approach needs is less than that of the compared approach SSoot needs for all the project programs. Therefore, the stack trace do affect the efficiency on locating null pointer exception.

RQ3 (Are the Results of our Approach Correct in Locating the NPAs?): To answer the question, we manually checked all the results to verify the correctness of our approach. The result is that all the statements that have found using our tool conclude all the incorrect assignment statements that lead to null pointer exception occurred.

Consider project Ant 1.5.1 in Table 4 (the sixth row). The stack trace of that execution shows that the null dereference occurs at line 327 in method *bsR( )* of class *CBZip2InputStream:*

```
java.lang.NullPointerException
at org.apache.tools.bzip2.CBZip2InputStream.
      bsR(CBZip2InputStream.java:327)
at org.apache.tools.bzip2.CBZip2InputStream.
      bsGetUChar(CBZip2InputStream.java:346)
at org.apache.tools.bzip2.CBZip2InputStream.
      initBlock(CBZip2InputStream.java:232)
```

Line 327 of *bsR( )* contains a call of a method *read( )* of an object *bsStream*.

```
[320] private int bsR(int n) {
[321]    int v;
[322]    {
[323]      while (bsLive < n) {
[324]        int zzi;
[325]        char thech = 0;
[326]        try {
[327]          thech = (char) bsStream.read();
[328]        } catch (IOException e) {
[329]          compressedStreamEOF();
[330]        }. . .
```

To locate the NPA, we would have to trace back in method *bsR( )* to see where the value of *bsStream* comes from. Then we trace back into the caller method *bsGetUChar(),*where we do not find *bsStream,*and then we trace back into *initBlock(),* where we still do not find *bsStream.*We trace back to the declaration section of class *CBZip2InputStream,*where we find line 149, which defines a variable of type InputStream, *bsStream,* which is not assigned an initial value. In Java, its default value is null. Therefore, line 149 is the statement that we want to find.

```
[142]private int unzftab[] = new int[256];
[143]
[144]    private int limit[][] = new int[N_GROUPS]
          [MAX_ALPHA_SIZE];
[145]    private int base[][] = new int[N_GROUPS]
          [MAX_ALPHA_SIZE];
[146]    private int perm[][] = new int[N_GROUPS]
          [MAX_ALPHA_SIZE];
[147]    private int minLens[] = new int[N_GROUPS];
[148]
[149]private InputStream bsStream;
[150]
[151]    private boolean streamEnd = false;
```

If using our tool for this null pointer exception, it need only 4s to find the incorrect assignment statement.

D. DISCUSSION

The evaluation shows that, for the subjects that this paper studied, the approach is more efficient in locating the null pointer assignment statements than the compared methods.

To compare our approach with a baseline using a manual check by several experienced developers, we conducted an experiment. Our population consisted of six students in computer science (four) and software engineering (two) who were just about to get their degrees. These students included two PhD and four master students. Their programming experience ranged from three year up to nine years. We gave them the three projects, Freemaker, Mckoi 1.0.6 and Jode 1.0.9, and asked them to locate the error statements that caused the null pointer exceptions manually in IDEs. The results is as follows. Only two of six students find the three error statements within 30 minutes; three of six students find only two error statements within 30 minutes; and one of six students find only one statement within 30 minutes.

If the statement that raised the null pointer exception and the source statement at which an incorrect assignment was made is within one class, locating the incorrect assignment statement manually in IDE is relatively easy. Otherwise it is hard for developers.

However, there are several threats to the validity of the evaluation. Threat to internal validity is that our implementation could have some flaws that would affect the accuracy of the results. However, the implementation is based on the Soot [5] that has been used by many practitioners. Additionally, we have manually checked all the results to verify the correctness.

Threat to external validity is the ability to generalize the results for null pointer exceptions, based on the subjects we used. In our experiment, we use 18 programs of eight projects and our approach works well. But, we are unable to conclude that our approach will hold for subjects in general.

V. RELATED WORK

There are many related researches in fault localization, which can be divided into three categories: static analysis, dynamic analysis and combining dynamic with static analysis methods.

The static methods include: Hovemeyer [9] applied forward inter-procedure data flow analysis and annotations to find null pointer dereferences bugs. Evans [10] combined annotations to detect errors in C programs, such as memory leak and dangerous aliasing. Dong *et al.* [16] proposed a region-based memory model for detecting null pointer dereferences (NPDs) in C programs. Ma *et al.* [17] proposed a static analysis tool – LUKE for detecting NPDs in C programs, they also proposed a context and path sensitive algorithm for detecting NPDs [18]. Duan *et al.* [19] proposed a NPD verification approach for C programs. Nanda and Sinha [11] presented a context sensitive inter-procedural approach that could identify potential null pointer dereferences. Xie and Engler [12] used redundancy checkers to find null dereferences, potential deadlocks.

All above methods are purely static, thus they suffer from the common problem of imprecise analysis. Unlike this paper combine the dynamically generated information (from the stack trace) with the static analysis.

Hangal and Lam [13] attempted to find bugs by dynamic program invariant detection. Romano *et al.* [15] proposed a search-based test data generation approach to automatically identify the null pointer exceptions. However, gathering of dynamic invariants is an expensive work that may not suitable for large programs. In contrast, this paper's approach uses stack trace information to locate faults, and does not need to collect dynamic information.

Bond *et al.* [7] presented an approach to finding the origin of null and undefined value errors. Their idea is piggybacking: when the original program stores a null value, value piggybacking instead stores origin information in the spare bits of the null value. Unlike their approach, we perform analysis backward guided by stack trace.

In [8], we also presented a fault localization approach for null pointer exception based on stack trace and program slicing. Unlike the approach in this paper, that approach used Soot tool to find the null value assignment statements, which could give more than one statement. The approach in this paper, for most of projects, can find one or two statements that are possible origins of null values.

Tomb *et al.* [14] presented an approach to locating runtime exception in Java programs by combining static and dynamic analysis. The approach first performed forward inter-procedural symbolic execution to locate constraints that may reveal an exception, and then attempted to generate test cases to raise that exception. However, this paper's analysis is backward, and it starts from a statement where it raised an exception, and it uses the available runtime information, such as stack traces, to guide the backward analysis.

Sinha *et al.* [4] proposed an approach to locating and repairing faults in Java programs due to incorrect assignment of a value that finally leads to the exception. The approach performed a backward stack trace guided data flow analysis, starting at the point where the exception was thrown, to find the origin of null value for the exception. They implemented the tool for null pointer exceptions using the XYLEM tool, which used WALA [21] analysis infrastructure to construct the call graph and the ICFG. Our approach also leverages an inter-procedural path sensitive analysis. However, we use Soot analysis infrastructure to construct the call graph and the ICFG.

Horwitz *et al.* [20] evaluated call stack-sensitive slicing and slice intersection as applied to debugging. Their technique can help a programmer find the problem more quickly by reducing the size of the backward slice from the point of failure. To answer RQ2, the compared approach also performs program slicing using traditional methods, rather than guided by stack trace. And then performs the null pointer analysis and alias analysis on the sliced programs using Soot. In [20], they only discussed the effect of stack trace on program slicing. However, we discuss the effect of stack trace on locating the null pointer exceptions in this paper.

Program slicing [3], [22], [24] is a widely studied technique for debugging. For null pointer exception localization, a static program slice [3] could identify all statements that could affect the value of v whose value is null, which includes all NPAs. However, the developer would be faced with the task of examining the slice to identify the definite NPA. A dynamic slice [24] could compute with respect to the execution trace of a failing executing, excluding the NPAs that do not execute in this executing. Thus, it could exclude some of the possible NPAs identified by our INPA-ST. However, dynamic slicing requires the program to be rerun, with instrumentation, on the failing inputs. Moreover, dynamic slicing itself can be very slow, which would not be practical for large programs. Our approach uses readily available stack-trace information, and requires no re-execution of the program.

Wu *et al.* [25] proposed CrashLocator, a method to locate faulty functions using the crash stack information in

crash reports. Their work targeted at crashing fault localization by statistically analyzing a large amount of crash data collected from different users. Our work is different from them in that our approach locates the faulty statement not the faulty functions.

Xuan *et al.* [26] proposed NOPOL, an approach to automatic repair of buggy conditional statements, which took a buggy program as well as a test suite as input and generated a patch with a conditional expression as output. In their work, fault localization is used as a step of ranking suspicious statements to find out locations of bugs. Durieux *et al.* [27] presented a technique, called NPEfix, to explore the repair search space of null pointer exception bugs with metaprogramming. Different from fault repairing, our work is to locate the error source, which is the first step of repairing the fault.

Gu *et al.* [28] proposed an automatic approach, namely CraTer, which predicts whether a crashing fault resides in stack traces or not (referred to as predicting crashing fault residence). However, our work combines dynamic analysis (using stack-trace information) with static backward data-flow analysis to identify the source statement.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present an automatic localization tool for null pointer exception. The approach performs a backward data flow analysis, under guided by stack trace, starting at the point where the exception was thrown, to find the root causes of the exception. Then we conduct some experiments on 18 open source programs, and the experimental results show that our tool is effective in locating the null pointer exception.

There are several areas of future work which are planned to conduct. One is that it will apply the approach to other types of runtime exceptions that are based on incorrect assignment. The other is that it will extend the implementation to provide context information for repairing the exception.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and editors for suggesting improvements and for their very helpful comments.

REFERENCES

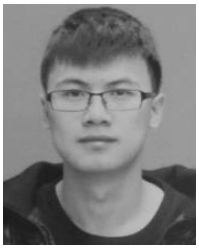
[1] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003.
[2] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 849–871, Sep. 2000.
[3] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[4] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions," in *Proc. Int. Symp. Softw. Test. Anal.*, Chicago, IL, USA, Jul. 2009, pp. 153–164.
[5] *Soot*. [Online]. Available: http://www.sable.mcgill.ca/soot/
[6] *SourceForge*. [Online]. Available: http://www.sourceforge.net/
[7] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley, "Tracking bad apples: Reporting the origin of null and undefined value errors," in *Proc. Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Appl.*, Montreal, QC, Canada, 2007, pp. 405–422.
[8] S. Jiang, W. Li, H. Li, Y. Zhang, H. Zhang, and Y. Liu, "Fault localization for null pointer exception based on stack trace and program slicing," in *Proc. Int. Conf. Qual. Softw.*, Xi'an, China, Aug. 2012, pp. 9–12.
[9] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 13–19, 2005.
[10] D. Evans, "Static detection of dynamic memory errors," *ACM SIGPLAN Notices*, vol. 31, no. 5, pp. 44–53, 1996.
[11] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for Java," in *Proc. Int. Conf. Softw.*, Vancouver, BC, Canada, 2009, pp. 133–143.
[12] Y. Xie and D. Engler, "Using redundancies to find errors," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 6, pp. 51–60, 2002.
[13] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. Int. Conf. Softw. Eng.*, Orlando, FL, USA, May 2002, pp. 291–301.
[14] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proc. Int. Symp. Softw. Test. Anal.*, London, U.K., 2007, pp. 97–107.
[15] D. Romano, M. Di Penta, and G. Antoniol, "An approach for search based testing of null pointer exceptions," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation*, Berlin, Germany, Mar. 2011, pp. 160–169.
[16] Y.-K. Dong, Y.-Z. Gong, and D.-H. Jin, "Null pointer dereference defect detected based on region-based memory model," *Acta Electron. Sinica*, vol. 42, no. 9, pp. 1744–1752, 2014.
[17] S. Ma, M. Jiao, S. Zhang, W. Zhao, and D. W. Wang, "Practical null pointer dereference detection via value-dependence analysis," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, Gaithersburg, MD, USA, Nov. 2015, pp. 70–77.
[18] S. Ma, W. Zhao, X.-Y. Xi, and D.-W. Wang, "Null pointer dereference detection based on value dependences analysis," *Acta Electron. Sinica*, vol. 43, no. 4, pp. 647–651, 2015.
[19] Z. Duan, C. Tian, and Z. H. Duan, "CEGAR based null-pointer dereference checking in C programs," *J. Comput. Res. Develop.*, vol. 53, no. 1, pp. 155–164, 2016.
[20] S. Horwitz, B. Liblit, and M. Polishchuk, "Better debugging via output tracing and callstack-sensitive slicing," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 7–19, Jan./Feb. 2010.
[21] [Online]. Available: http://wala.sourceforge.net
[22] X. Y. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Softw., Pract. Exper.*, vol. 37, no. 9, pp. 935–961, 2007.
[23] H. Zhang, S. Jiang, and R. Jin, "An improved static program slicing algorithm using stack trace," in *Proc. IEEE 2nd Int. Conf. Softw. Eng. Service Sci.*, Jul. 2010, pp. 563–567.
[24] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.
[25] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "CrashLocator: Locating crashing faults based on crash stacks," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, San Jose, CA, USA, Jul. 2014, pp. 204–214. doi: 10.1145/2610384.2610386.
[26] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. R. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017. doi: 10.1109/TSE.2016.2560811.
[27] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, Klagenfurt, Austria, Feb. 2017, pp. 349–358. doi: 10.1109/SANER.2017.7884635.
[28] Y. Gu, J. Xuan, H. Zhang, L. Zhang, Q. Fan, X. Xie, and T. Qian, "Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence," *J. Syst. Softw.*, vol. 148, pp. 88–104, Feb. 2019.

JING DUAN is currently pursuing the Ph.D. degree with the College of Computer and Information, Hohai University. Her main research interests include financial big data mining, software engineering, and so on.

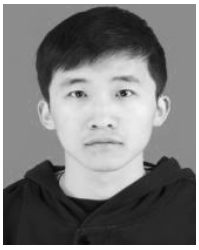
SHUJUAN JIANG received the B.S. degree in computer science from the Computer Science Department, East China Normal University, Shanghai, in 1990, the M.S. degree in computer science from the China University of Mining and Technology, Xuzhou, Jiangsu, in 2000, and the Ph.D. degree in computer science from Southeast University, Nanjing, Jiangsu, in 2006.

Since 1995, she has been a Teaching Assistant, a Lecturer, an Associate Professor, and a Professor with the School of Computer Science and Technology, China University of Mining and Technology. She is the author of more than 80 articles. Her research interests include software engineering, program analysis and testing, software maintenance, and so on.

QIAO YU received the Ph.D. degree from the China University of Mining and Technology, in 2017. She is a Lecturer with the School of Computer Science and Technology, Jiangsu Normal University. Her research interests include software analysis and testing, and machine learning.



KAI LU is currently pursuing the master's degree with the School of Computer Science and Technology, China University of Mining and Technology. His research interest includes software analysis and testing.



XU ZHANG is currently pursuing the master's with the School of Computer Science and Technology, China University of Mining and Technology. His research interest includes software analysis and testing.

YIWEN YAO is currently pursuing the master's degree with the School of Computer Science and Technology, China University of Mining and Technology. His research interests include fault localization and mutation testing.