Explaining Bug Provenance with Trace Witnesses

Jixiang Shen
The University of Sydney
Australia
jshe9611@sydney.edu.au

Xi Wu
The University of Sydney
Australia
xi.wu@sydney.edu.au

Neville Grech
University of Athens
Greece
me@nevillegrech.com

Bernhard Scholz
The University of Sydney
Australia
bernhard.scholz@sydney.edu.au

Yannis Smaragdakis
University of Athens
Greece
yannis@smaragd.org

Abstract

Bug finders are mainstream tools used during software development that significantly improve the productivity of software engineers and lower maintenance costs. These tools search for software anomalies by scrutinising the program's code using static program analysis techniques, i.e., without executing the code. However, current bug finders do not explain why bugs were found, primarily due to coarse-grain abstractions that abstract away large portions of the operational semantics of programming languages. To further improve the utility of bug finders, it is paramount to explain reported bugs to the end-users.

In this work, we devise a new technique that produces a program trace for a reported bug giving insight into the root cause for the reported bug. For the generation of the program trace, we use an abstracted flow-based semantics for programs to overcome the undecidability of the problem. We simplify the semantic problem by mapping an input program with a reported bug to a Constant Copy Machine (CCM) for the trace construction. Using CCM the semantics of the program can be weakened, and thus bug provenance can be solved in polynomial time, producing a shortest trace in the process which gives the shortest explanation. The technique is reified in the bug tracing tool *Digger* and is evaluated on several open-source Java programs.

CCS Concepts: • **Theory of computation** → **Program analysis**.

Keywords: Bug Provenance, Static Analysis, Trace Witness

ACM Reference Format:
Jixiang Shen, Xi Wu, Neville Grech, Bernhard Scholz, and Yannis Smaragdakis. 2020. Explaining Bug Provenance with Trace Witnesses. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '20), June 15, 2020, London, UK*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394451.3397206

1 Introduction

Bug finding tools have matured so that they are widely used in the software development life cycle in industry [1]. However, state-of-the-art bug-finding tools that employ static program analysis, mainly focus on the discovery of bugs without providing a comprehensive explanation for their existence.

To improve the utility of bug-finding tools, comprehensive explanations for reported bugs are of paramount importance to further the utility and acceptance of bug-finding tools. Bug-finding tools have developed ad-hoc notions of *provenance* that explain the root cause of a reported bug. However, the root cause of a bug is, in most cases, spatially and temporally removed from the reported line number of the bug itself. For a software engineer, it may not be immediately intuitive to understand the true existence of a reported bug without understanding the connection between the root cause of a bug and the reported line number. Hence, finding the provenance of a bug exposes this causal connection and is fundamental to improve the utility of a bug-finding tool. State-of-the-art tools provide weak notions of provenance such as recording partial paths and heuristics to expose the root cause of the bugs where most of the techniques are ad-hoc since the problem is inherently hard and undecidable if definite explanations are sought after [10].

In this work, we give insight into finding the causal connection between the root cause of a bug and the reported line number. We use an abstract interpretation framework to build the scaffolding for the provenance construction. The program state is abstracted via a new computational model, a Constant-Copy Machine (CCM), which is a decidable machine proposed for provenance construction. In the abstract interpretation framework, we provide a strong-update [6] semantics tracking the state of variables depending on the context. The provenance of a reported bug is provided in the form of a program path that, when executed in the abstracted strong-update semantics, exposes the bug. We call such a program path an abstract "*Trace Witness*".

Besides explaining the bug, the trace witness problem may also be used to sharpen a flow-insensitive static program analysis. For example, if a bug is reported using a flow-insensitive analysis, but no trace witness can be produced, we can assume that the bug is a false positive. Hence, the trace witness can also be seen as a post-mortem analysis [7] that introduces flow-sensitivity in a later stage of an analysis pipeline. However, the existence of a trace is a necessary but not sufficient condition for the existence of the bug. The contribution of this paper can be summarised as follows:

- We introduce Trace-Witness that exposes the causal connection between the root cause and the reported line number of bugs, which is a program path and resembles the provenance of a bug. (Section 2)
- We produce the trace-witness based on a novel translation scheme reducing the trace finding problem to a Constant Copy Machine (CCM) that has constant and copy assignments simulating state transfer in an input program. (Section 2)
- We provide a new trace witness generator for the CCM, which is based on shortest-path algorithms. (Section 3)
- We applied the trace generator on NPEs in java programs, which shows that around 80% of NPE bug reports were invalidated. (Section 4)

2 Problem Statement

The objective of this work is to find a path that leads to a bug in a program. The bug condition can be phrased as a condition at a given point of the program, denoted as an assertion. The trace witness of such program identifies a subset of nodes, from which a unique path from the entry point to the assertion of the program can be reconstructed while the bug condition holds. In this paper, we solve this problem for intra-procedural cases only. Inter-procedural cases are left as an exercise for future work.

2.1 The CCM Language

This section will introduce a simplified intra-procedural machine called the Constant Copy Machine (CCM). Elementary instructions in a CCM abstract the behaviour of a language with side-effects such as C, and mimick the control-flow of input programs via a Control-Flow Graph (CFG). In contrast to a concrete semantics, no conditions govern the flow of control, making the machine non-deterministic. The CCM is only used for analysis purposes and not as an actual execution semantics. For the sake of demonstration, we will omit the mapping of features such as memory management and predicate evaluation.

A program in CCM is represented as a CFG $G = (V, E, r, f)$, where V is the set of program statements and $E \subseteq V \times V$ is the set of control-flow edges. Flow edge (u, v) denotes the transfer of control from program statement $u \in V$ to program statement $v \in V$. There are two distinguished nodes r and f, where r is the start node of the CFG, and f is the final node.

A program path in CFG G is a finite sequence of statements, denoted by $p = \langle u_0, u_1, \ldots, u_k \rangle$, such that $(u_i, u_{i+1}) \in E$ for all $i \in [0, k-1]$. The empty program path is denoted by ϵ and the set of all program paths in a CFG G is denoted by P_G. The set $P_G(u, v) \subseteq P_G$ stands for the set of all program paths that emanate in statement u and terminate in statement v. We have a labelling function $\ell : V \longrightarrow \text{CCM}$ that assigns a statement to each node in the CFG. We have the following statements in the CCM:

$$\begin{aligned}
\textsc{ccm} ::=\ & x := n && \text{(constant assignment)}\\
\mid\ & x := y && \text{(copy assignment)}\\
\mid\ & \texttt{assert}\ (x = n) && \text{(assertion)}\\
\mid\ & \texttt{nop} && \text{(no-operation)}
\end{aligned}$$

The CCM has a finite set of variables *Var* whose values are natural numbers. It also has a fixed set of constants that can be assigned to variables and copied from variable to another variable via transfer statements. We use x and y to represent variables, which belong to the finite variable set *Var*, whereas $n \in \mathbb{N}$ stands for a constant. A statement in a node u can be either a constant assignment $x := n$ that assigns a constant value n to the variable x; or a copy assignment $x := y$ that copies the value stored in variable y to the variable x; or an assertion statement $\texttt{assert}\ (x = n)$ which is used to check whether the variable x has a certain value n in final node f; or a no-operation statement.

2.2 CCM Semantics

A trace witness is, in fact, a program path that emanates from the start node r and terminates in the final node f, i.e., the assertion statement. The trace witness problem asks for a trace witness (particular the shortest one) for which the assertion $\mathcal{A}$ in the final node f holds under the program context c in node f. The program context $c : Var \longrightarrow \mathbb{N}$ is a function that maps variables to values. The semantics function σ for statements in CCM $\sigma : \text{CCM} \longrightarrow (c \longrightarrow c')$ is given below:

- $\sigma[\![x := n]\!] \equiv \lambda c\ .\ c_{[x \leftarrow n]}$: the constant assignment updates the value of variable x with constant n in the program context c and keeps other variables unchanged.
- $\sigma[\![x := y]\!] \equiv \lambda c\ .\ c_{[x \leftarrow c(y)]}$: the copy assignment $x := y$ updates the value of variable x with the value of variable y in the program context c, and keeps other variables unchanged.
- $\sigma[\![\texttt{assert}\ (x = n)]\!] \equiv \lambda c\ .\ c$: the assertion statement does not affect the program context c, which keeps it unchanged.

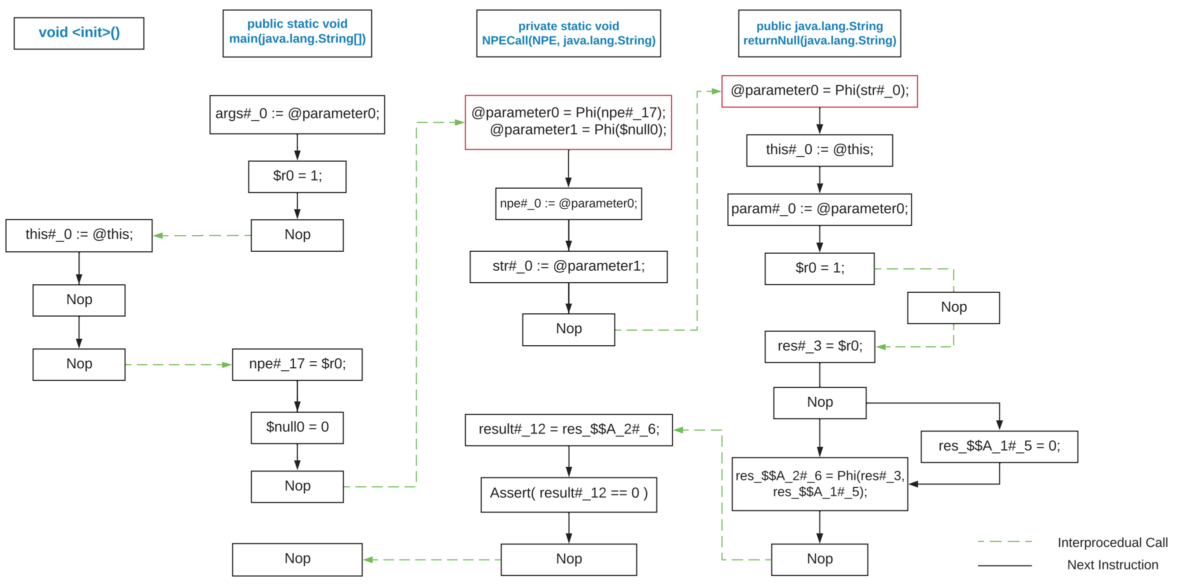
void <init>()
public static void main(java.lang.String[])
private static void NPECall(NPE, java.lang.String)
public java.lang.String returnNull(java.lang.String)
args#_0 := @parameter0;
$r0 = 1;
Nop
this#_0 := @this;
Nop
Nop
npe#_17 = $r0;
$null0 = 0
Nop
Nop
@parameter0 = Phi(npe#_17); @parameter1 = Phi($null0);
npe#_0 := @parameter0;
str#_0 := @parameter1;
Nop
result#_12 = res_$$A_2#_6;
Assert(result#_12 == 0)
Nop
@parameter0 = Phi(str#_0);
this#_0 := @this;
param#_0 := @parameter0;
$r0 = 1;
Nop
res#_3 = $r0;
Nop
res_$$A_1#_5 = 0;
res_$$A_2#_6 = Phi(res#_3, res_$$A_1#_5);
Nop
Interprocedual Call
Next Instruction

Figure 1. The CFG in CCM of NEP program in Listing 1

- $\sigma[\![\text{nop}]\!] \equiv \lambda c\ .\ c$: similar as the assertion statement, the no-operation statement also keeps the program context cunchanged.

For sake of simplicity, we overload the notation of semantics function σ and extend it for program paths, i.e., $\sigma : P_G \longrightarrow (c \longrightarrow c')$. The semantics function σ for program paths is defined as

$$\sigma[\![p]\!] \equiv \begin{cases} \lambda c\ .\ (\sigma[\![p']\!])\sigma[\![\ell(u)]\!], & \text{if } p \neq \epsilon \text{ and } p = u \cdot p' \\ \lambda c\ .\ c & \text{otherwise (i.e., } p = \epsilon) \end{cases}$$

The evaluation of a program path p is the semantics evaluation $\sigma[\![p]\!]c_0$ where c_0 represents the initial mapping for the variables in the program, i.e., all variables in the program are assumed to point to 0 initially. A bug condition can be expressed as an assertion denoted by $\mathcal{A}$, which is used to check whether variables have some specific values at the checking point for a context c that $\mathcal{A}(c)$ holds.

Definition 2.1 (Trace-Witness-Problem)**.** An instance of the trace witness problem is represented by the quadruple $(G, \mathcal{A}, Var, \ell)$ where G is the control flow graph of the program, $\mathcal{A}$ the assertion, Var the set of variables in the instance, and ℓ the labelling function. The solution of the trace-witness problem is the shortest program path p_s such that $\mathcal{A}(\sigma[\![p_s]\!]c_0)$ holds under the given context c_0.

3 Trace Witness Generation

To understand the trace-witness problem on imperative languages (e.g., OO-languages), we use a translation approach, known as a "gadget". The gadget translates input programs

```
1  class NPE{
2      public String returnNull(String param) {
3          String res = new String();
4          if(param == null) res = null;
5          return res;
6      }
7  }
8  public class Main {
9      private static void NPECall(NPE npe, String str) {
10         String result = npe.returnNull(str);
11         result.toString();
12     }
13     public static void main(String[] args) {
14         NPE npe = new NPE();
15         NPECall(npe, null);
16     }
17 }
```

Listing 1. An Example of NPE in Java Program

(e.g., OO programs) into CCM, especially it converts programs with inter-procedural calls into intra-procedural cases, so that we can solve the trace-witness problem in CCM. The gadget also maps CCM trace back to the original input programs; note that, the existence of a trace in CCM doesn't guarantee the existence of a trace in input programs because of unfeasible conditions. Its implementation makes use of the Doop [11] framework. Due to space limitations, we omit the details of the gadget. Instead, we provide a motivating example (Listing 1), a Java program snippet containing a Null Pointer Exception (NPE) at line 11 [1]. The translation of this program is shown as a CFG of the CCM in Figure 1. Two

[1]The variable `result` points to `null`, which was caused by passing `null` as a parameter into the static method `NPECall`.

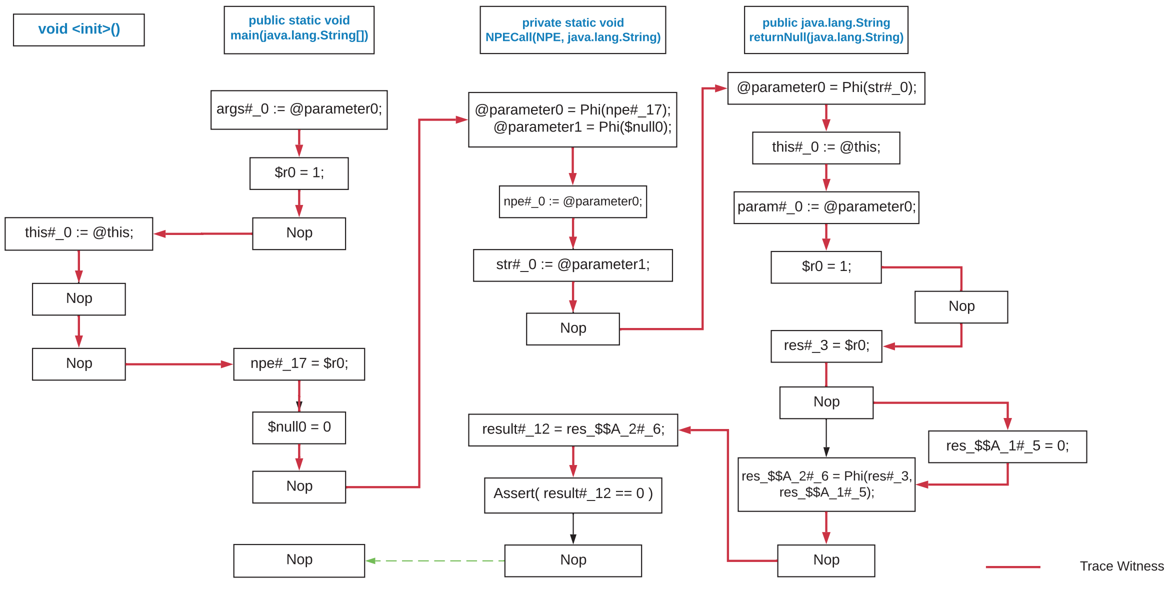
void <init>()
public static void main(java.lang.String[])
private static void NPECall(NPE, java.lang.String)
public java.lang.String returnNull(java.lang.String)
args#_0 := @parameter0;
$r0 = 1;
Nop
this#_0 := @this;
Nop
Nop
npe#_17 = $r0;
$null0 = 0
Nop
Nop
@parameter0 = Phi(npe#_17);
@parameter1 = Phi($null0);
npe#_0 := @parameter0;
str#_0 := @parameter1;
Nop
result#_12 = res_$$A_2#_6;
Assert(result#_12 == 0)
Nop
@parameter0 = Phi(str#_0);
this#_0 := @this;
param#_0 := @parameter0;
$r0 = 1;
Nop
res#_3 = $r0;
Nop
res_$$A_1#_5 = 0;
res_$$A_2#_6 = Phi(res#_3, res_$$A_1#_5);
Nop
Trace Witness

Figure 2. Trace Witness of NEP program in Listing 1

blocks in red are added by the gadget so that the original inter-procedural calls among methods are converted into intra-procedural connections (labeled in green dashed line).

In this section, we focus on presenting the new trace witness generator over a program CFG in CCM, which is based on Floyd-Warshall shortest path algorithm [2] [4] to compute the shortest distance from the initial node to a target node over all pairs of nodes in CFG, and generate the trace witness. For a given CCM program P and a bug report (i.e., an assertion) $\mathcal{A}$, we compute the Data Dependency Chain (DDC) of P, whose result contains the variables that may contribute to the assertion. Based on the Floyd-Warshall all pairs shortest path algorithm, we develop a shortest trace generation algorithm to compute the shortest trace distance between two data dependent nodes from DDC, and to record the previous data dependent node (or the initial node of CFG) of each data dependent node in DDC in the resulting pair. Finally, a new trace witness generator is proposed for the trace witness generation in CCM programs.

3.1 DDC Computation

Data dependency denotes the data of a program statement refers to the data of the preceding statements [8], and a DDC consists of a set of assignments where the right-hand side of the assignments is either a constant or a variable defined by a statement in the set, forming a topological order between assignments [3]. The construction of DDC abstracts the data flow in the program as a tree structure, with which the traversal becomes much easier to perform. Specifically, a function is designed to check whether a variable may contribute to the value in the assertion. If the variable is never assigned the value appeared in the assertion, then this variable will be pruned (i.e., definitely cannot appear in the DDC). We go through each statement s in the CCM program P and construct the DDC for the assertion $\mathcal{A}$.

[2]Multiple paths may be found but the shortest path is the best option for developers to easily find out the root cause of bugs.

3.2 Shortest Trace Generation

Floyd-Warshall algorithm is an efficient solution for finding the shortest path among all pairs of nodes, which leverages the principle of dynamic programming to achieve its simplicity and efficiency. Despite of its $O(n^3)$ time complexity, it allows the result to be queried multiple times without any re-computation, which may occur quite frequently during program analysis.

Based on the Floyd-Warshall algorithm and the DDC result, we propose a shortest trace generation algorithm to get the shortest trace distance between the initial node r and each data dependent node in DDC. Specifically, we go through all variables in DDC and get the corresponding node for each variable based on the CFG. If the statement on a node is a constant assignment, the distance of the shortest trace between the initial node r and the current node (e.g., denoted as u) is the distance of the shortest path between them, and we record r as the previous node of u. However, if the statement on a node is a copy assignment, the distance of the shortest trace between the initial node r and the current node u equals to the sum of the shortest distance between the initial node r and the node (i.e., middle node) whose statement contains the variable on the right hand side of the assignment, and the shortest distance between this middle node and the current node u. In this case, we regard this middle node as the previous node of u.

Algorithm 1: Trace Witness Generation

```
Data: CCM program CFG $G$, assertion $\mathcal{A}$ in node $f$, initial node $r$
      and context $c_0$
Result: Trace witness $t$ such that $t$ is the shortest path from the
        initial node $r$ to the final node $f$, which makes the
        assertion $\mathcal{A}(\sigma[\![t]\!]\ c_0)$ holds at node $f$

ShortestTrace(sNode, dNode) ← run Shortest Trace Generation
  algorithm to get the previous data dependent node of dNode, and
  the shortest distance between sNode and dNode;
ShortestPath(sNode,dNode) ← run Floyd-Warshall shortest path
  algorithm for all feasible nodes in $G$, which returns the previous
  node of dNode in $G$ and the shortest distance between sNode and
  dNode;

$(pred, dist)$ ← call ShortestTrace($r, f$);
$w \leftarrow (f, pred)$;
while $fst(w) \neq r$ do
    if $fst(w) \neq snd(w)$ then
        $w' \leftarrow w$;
        $fst(w')$ ← get the previous node by calling
          ShortestPath($r$,$fst(w)$);
    else
        $w'$ ← ($fst$(w), get the previous node by calling
                ShortestTrace($r$,$fst(w)$));
    add $fst(w)$ at the head of trace $t$;
    $w \leftarrow w'$;
```

3.3 Trace Witness Generation

Based on the algorithms above, we develop the trace witness generation algorithm in Algorithm 1. It will return all nodes on the trace witness, which is the shortest program path from the program initial node to the reported bug point. With the help of a trace witness, we can provide a better explanation for the root cause of the bug and avoid the false positive as much as possible.

In the trace witness generation algorithm, we firstly get the previous data dependent node *pred* of the bug reported node f and the shortest distance between the initial node r and f by invoking the shortest trace algorithm *Shortest-Trace*. A pair of the current node (i.e., at the beginning is the reported bug node f) and its previous data dependent node *pred* is assigned to the variable w. The algorithm finds out each node on the path from the current node to its previous data dependent node *pred* by invoking the shortest path algorithm *ShortestPath*. The current node in w will be updated by each node on the path until it becomes the same as the one in *pred*. In the meantime, the previous data dependent node is also updated according to the change of the current node. The iteration will finish until the current node becomes the initial node r and the final result of this algorithm is given in variable t. The trace witness result (labeled in red solid lines) of the motivating example can be found in Figure 2.

4 Experiments

As a static program analysis tool, Digger suffers from inherently high false-positive rate [5], which is difficult to trace the reported bug or identify whether the bug is a false positive. In this section, we will conduct an empirical study on some open-source Java programs to find valid traces of NPEs reported by Digger to fully evaluate 1) the performance of our algorithm in large-scale programs and 2) the application of the algorithm in false-positive invalidation.

4.1 Experimental Setup

The empirical experiment will be conducted on a list of open-source Java programs, which are obtained as Jar files from SourceForge at versions where NPEs are reported. The programs are chosen in various sizes to better evaluate the performance of our algorithm in relation to the size, including JSP-3.0.0, JBoss-1.1.1 and Sling-11 as shown in Table 1. All statistics in this table are reported by Doop and Soot [12].

Table 1. Problem Size

Benchmark	JSPWiki-3.0.0	JBoss-1.1.1	Sling-11
Class	490	143	19
Method Call	931	1049	284
Methods	345	390	121
Variables	2230	2697	664
Object Creation Sites	7628	1274	377
Instructions	90556	22134	5652
Main Methods	8	1	1

For each program, it is firstly translated via the gadget into CCM based program, in which our trace witness generator will take every NPE reported by Digger as an assertion, and produce a trace witness from the assertion back to the entry point of the program (i.e., the first statement in the main method). All tasks are run by Soufflé-1.6.2 compiler [9] in 6 threads except for Soot. All experiments are conducted on Fedora Server Version 30 with 187GB RAM and 32 Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz.

4.2 Results and Discussion

Results for the experiments can be viewed in Table 2. We are evaluating the runtime throughout stages in Gadget Translation, Shortest Path Computation and Trace Witness Generation. Since the majority of our algorithms are graph-based, the graph size is taken as the main factor that can affect our runtime performance.

From the results, we can observe that all experimental programs detected trace witnesses for their reported NPEs by Digger, with 14 out of 86 (16.2%) in JSPWiki, 2 out of 8 (25 %) in Sling and 3 out of 19 (15.7 %) in JBoss. The NPEs without a valid trace witness are either because all nodes are pruned due to unreachability or no trace can be found to make the assertion to be true. For example, JSPWiki is the largest code base program with 90556 statements before pruning and 8 main methods as mentioned in Table 1. However, out of all 8 main methods, only 2 of which the trace witness to the bug points can be found.

Table 2. Trace Witness Results of Experiments

Program	NPE	Assertion (File: Line)	Nodes	Edge	Gadget Runtime	Shortest Path Runtime	Trace Witness Runtime	Trace Length(Nodes)	Trace Length(Lines)
JSPWiki	86	Import: 167	2975	3295	3m 0s	13m 27s	6m 50s	11	7
		Import: 161			3m 1s	13m 25s	6m 31s	6	3
		WikiEngine: 580			2m 41s	11m 19s	5m 56s	236	106
		WikiEngine: 581			3m 1s	13m 28s	6m 32s	329	107
		WikiEngine: 582			2m 59s	13m 40s	6m 34s	331	108
		WikiEngine: 587			3m 0s	13m 39s	6m 31s	275	88
		WikiEngine: 618			3m 1s	13m 26s	6m 36s	275	88
		WikiEngine: 614			3m 0s	12m 57s	6m 32s	275	88
		WikiEngine: 579			2m 58s	13m 37s	6m 33s	325	105
		Tag: 115	1434	1778	29s	4m 54s	54s	327	89
		Tag: 111			28s	5m 9s	52s	235	89
		Tag: 107			28s	5m 0s	52s	229	87
		JspParser: 105			28s	5m 18s	52s	232	88
		JspParser: 933			28s	5m 9s	53s	232	89
Sling	8	Main: 557	1879	2116	42s	11m 45s	2m 46s	187	45
		Main: 566			45s	12m 19s	2m 55s	197	48
JBoss	19	PathUtils: 92	5223	5980	14m 39s	92m 57s	21m 28s	222	72
		ModuleLoader: 208			14m 35s	92m 50s	21m 15s	106	40
		ModuleLoader: 281			14m 29s	94m 21s	22m 34s	138	48

Even though our experiments invalidate nearly 80 % of bugs reported by Digger, it is unknown that the deemed invalid bugs are actually false positive. For example, the unreachable case can be explained by an existence of other entrances (e.g., a dependency on a third-party application) to the program that we are not aware.

Gadget translation among all three processes takes the least portion of total runtime for all graph sizes, whereas the Floyd-Warshall shortest path algorithm is always the slowest in terms of the performance. It is clear that the runtime grows by increasing the graph size of three processes, but the runtime of Floyd-Warshall shortest path algorithm grows exponentially while others stay in linear. The runtime growing trend indicates that the Floyd-Warshall shortest path algorithm we currently used may be the bottleneck of the performance once the graph becomes large enough.

The length of the trace witness is measured in the number of nodes (i.e., statements) and the number of code lines, which appear in a linear relation. It can be inferred that the length of the trace witness does not vary much among different programs, which accordingly implies the size of a program has little impact on the length of the trace witness.

In summary, this empirical study examined three open-source Java programs, with around 80% of bugs reported by Digger being eliminated by our technique and the rest being expressed with valid trace witnesses. Our algorithm is certainly efficient on small programs. As the size of the program grows, a better shortest path algorithm may be needed to reach a better performance.

5 Conclusion and Future Work

In this paper, we developed a trace-witness based bug provenance technique to expose the connection between the root cause of a bug and the reported line number of the bug to the end-users. We introduced a new computational model (CCM) for trace witness construction, which weakened the semantics of a standard imperative language to overcome the undecidability of computing a trace witness. Based on the Floyd-Warshall shortest path algorithm, we developed a new trace witness generator which has been applied on NPEs in Java programs, reified as the tool Digger. Our results show that around 80% of NPE bug reports were invalidated using our technique. Our results also illustrate the efficiency of our algorithms, which solve the bug provenance for complex programs in polynomial time.

As future work, we will enrich the CCM by adding conditional statements. Besides, we will investigate other shortest path algorithms (e.g., Dijkstra algorithm [2]) to improve the runtime performance of our trace witness generator. Moreover, inter-procedural cases will also be considered as an extension of this work.

References

[1] Cristina Cifuentes and Bernhard Scholz. 2008. Parfait - Designing a Scalable Bug Checker. In *Scalable Program Analysis*, Vol. 08161.

[2] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271.

[3] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349.

[4] Robert W. Floyd. 1962. Algorithm 97: Shortest Path. *Commun. ACM* 5, 6 (1962), 345.

[5] William Landi. 1992. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.* 1, 4 (Dec 1992), 323–337.

[6] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. *SIGPLAN Not.* 46 (2011), 3–16.

[7] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct 2004), 63–72.

[8] David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc.

[9] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proc. 25th International Conference on Compiler Construction.* ACM, 196–206.

[10] J. C. Shepherdson and H. E. Sturgis. 1963. Computability of Recursive Functions. *J. ACM* 10, 2 (April 1963), 217–255.

[11] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (Apr 2015), 1–69.

[12] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - A Java Bytecode Optimization Framework. In *Proc. of CASON '99.* IBM Press, 13.