HAL
open science

Casper: Debugging Null Dereferences with Dynamic Causality Traces

Benoit Cornu, Earl T. Barr, Lionel Seinturier, Martin Monperrus

To cite this version:

Benoit Cornu, Earl T. Barr, Lionel Seinturier, Martin Monperrus. Casper: Debugging Null Dereferences with Dynamic Causality Traces . [Research Report] hal-01113988, Inria Lille. 2015. hal-01113988v2

HAL Id: hal-01113988

https://hal.science/hal-01113988v2

Submitted on 19 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Casper: Debugging Null Dereferences with Dynamic Causality Traces

Benoit Cornu, Earl T. Barr†, Lionel Seinturier*, Martin Monperrus*
*University of Lille & INRIA
†University College London
contact:martin.monperrus@univ-lille1.fr

Abstract: Fixing a software error requires understanding its root cause. In this paper, we introduce "causality traces", crafted execution traces augmented with the information needed to reconstruct the causal chain from the root cause of a bug to an execution error. We propose an approach and a tool, called CASPER, based on code transformation, which dynamically constructs causality traces for null dereference errors. The core idea of CASPER is to replace nulls with special values, called "ghosts", that track the propagation of the nulls from inception to their error-triggering dereference. Causality traces are extracted from these ghosts. We evaluate our contribution by providing and assessing the causality traces of 14 real null dereference bugs collected over six large, popular open-source projects. Over this data set, CASPER builds a causality trace in less than 1 second.

```
Exception in thread "main" java.lang.NullPointerException
    at [..].BisectionSolver.solve(88)
    at [..].BisectionSolver.solve(66)
    at ...
```

Listing 1. The standard stack trace of a real null dereference bug in Apache Commons Math

```
Exception in thread "main" java.lang.NullPointerException
For parameter : f // symptom
    at [..].BisectionSolver.solve(88)
    at [..].BisectionSolver.solve(66)
    at ...
Parameter f bound to field UnivariateRealSolverImpl.f2
    at [..].BisectionSolver.solve(66)
Field f2 set to null
    at [..].UnivariateRealSolverImpl.<init>(55) // cause
```

Listing 2. What we propose: a causality trace, an extended stack trace that contains the root cause.

I. INTRODUCTION

In isolation, software errors are often annoyances, perhaps costing one person a few hours of work when their accounting application crashes. Multiply this loss across millions of people; consider that even scientific progress can be delayed or derailed by software error [10]: in aggregate, these errors are costly to society as a whole.

Fixing these errors requires understanding their root cause, a process that we call causality analysis. Computers are mindless accountants of program execution, yet do not track the data needed for causality analysis. This work proposes to harness computers to this task. We introduce *"causality traces"*, execution traces augmented with the information needed to reconstruct a causal chain from a root cause to an execution error. We construct causality traces over *"ghosts"*, an abstract data type that can replace a programming language's special values, like `null` or NaN. Ghosts replace such values and track operations applied to itself, thereby collecting a causality trace whose analysis reveals the root cause of a bug. To demonstrate the feasibility and promise of causality traces, we have instantiated ghosts for providing developers with causality traces for null deference errors, they are "null ghosts".

Anecdotally, we know that null dereferences are frequent runtime errors. Li et al. substantiated this conventional wisdom, finding that 37.2% of all memory errors in Mozilla and and Apache are null dereferences [9]. Kimura et al. [8] found that there are between one and four null checks per 100 lines of code on average. A null dereference runtime error occurs when a program tries to read memory using a field, parameter, or variable that points to nothing — "`null`" or "none", depending on the language. For example, on October 22 2009, a developer working on the Apache Common Math open source project encountered an null pointer exception and reported it as bug #305[1].

In low-level, unmanaged runtime environments, like assembly or C/C++, null dereferences result in a dirty crash, e.g. a segmentation fault. In a high-level, managed runtime environment such as Java, .NET, etc., a null dereference triggers an exception. Programs often crash when they fail to handle null dereference exceptions properly [2].

When debugging a null dereference, the usual point of departure is a stack trace that contains all the methods in the execution stack at the point of the dereference. This stack trace is decorated with the line numbers where each method was called. Listing 1 gives an example of such a stack trace and shows that the null dereference happens at line 88 of BisectionSolver.

Unfortunately, this stack trace only contains a partial snapshot of the program execution when the null dereference occurs, and not its root cause. In Listing 1, the stack trace says that a variable is null at line 88, but not when and what assigned "null" to the variable. Indeed, there may be a large gap between the symptom of a null dereference and its root cause [2]. In our evaluation, we present 7 cases where the patch for fixing the root cause of a null dereference error is

[1] https://issues.apache.org/jira/browse/MATH-305

not in any of the stack trace's method. This gap exactly is an instance of Eisenstadt's cause/effect chasm [4] for a specific defect class: null dereference errors.

This "null dereference cause/effect chasm" has two dimensions. The first is temporal: the symptom may happen an arbitrarily long time after its root cause, e.g. the dereference may happen ten minutes and one million method executions after the assignment to null. In this case, the stack trace is a snapshot of the execution at the time of the symptom, not at the time of the root cause. The second is spatial: the location of the symptom may be arbitrarily far from the location of the root cause. For example, the null dereference may be in package foo and class A while the root cause may be in package bar and class B. The process of debugging null dereferences consists of tracing the link in space and time between the symptom and the root cause.

A *causality trace* captures the complete history of the propagation of a `null` value that is incorrectly dereferenced. Listing 2 contains such a causality trace. In comparison to Listing 1, it contains three additional pieces of information. First, it gives the exact name, here f, and kind, here parameter (local variable or field are other possibilities), of the null variable. Second, it explains the origin of the parameter, the call to solve at line 66 with field f2 passed as parameter. Third, it gives the root cause of the `null` dereference: the assignment of null to the field f2 at line 55 of class UnivariateRealSolverImpl. Our causality traces contain several kinds of trace elements, of which Listing 2 shows only three: the name of the wrongly dereferenced variable, the flow of nulls through parameter bindings, and null assignment. Section II details the rest.

In this paper, we present a novel tool, called CASPER, to collect null causality traces. The tool is going to be used at debugging time by developers. It takes as input the program under debug and a main routine that triggers the null dereference. It then outputs the causality trace of the null dereference. It first instruments the program under debug by replacing `null`s with "ghosts" that track causal information during execution. We have named our tool CASPER, since it injects "friendly" ghosts into buggy programs. To instrument a program, CASPER applies a set of 11 source code transformations tailored for building causal connections. For instance, o = externalCall() is transformed into o = NullDetector.check(externalCall()) , where the method check stores causality elements in a null ghost (Section II-B) and assigns it to o if externalCall returns `null`. Section II-C details these tranformations.

We evaluate our contribution CASPER by providing and assessing the causality traces of 14 real null dereference bugs collected over six large, popular open-source projects. We collected these bugs from these project's bug reports, retaining those we were able to reproduce. CASPER constructs the complete causality trace for 13 of these 14 bugs. For 11 out of these 13 bugs, the causality trace contains the location of the actual fix made by the developer.

Furthermore, we check that our 11 source code transformations do not change the semantics of the program relative to the program's test suite, by running the program against that test suite after transformation and confirming that it still passes. The limitations of our approach are discussed in Section II-E and its overhead in paragraph III-C4c.

To sum up, our contributions are:

- The definition of causality traces for null dereference errors
- A set of source code transformations designed and tailored for collecting the causality traces.
- CASPER, an implementation in Java of our technique.
- An evaluation of our technique on real null dereference bugs collected over 6 large open-source projects.

CASPER and our dataset can be downloaded from https://github.com/Spirals-Team/casper.

II. CASPER'S NULL DEBUGGING APPROACH

CASPER tracks the propagation of `null`s used during application execution in a causality trace. A *null dereference causality trace* is the sequence of language constructs traversed during execution from the source of the `null` to its erroneous dereference. By building this trace, CASPER generalizes dynamic taint analysis to answer not only whether a null can reach a dereference, but *how* a dereference is reached[2] These traces speed the localization of the root cause of null dereferences errors.

Our idea is to replace `null`s with objects whose behavior, from the application's point of view, is same as a `null`, except that they store a causality trace, defined in Section II-A. We called these objects *null ghosts* and detail them in Section II-B. CASPER rewrites the program under debug to use null ghosts and to store a `null`'s causality trace in those null ghosts, Section II-C. An additional set of semantic preserving transformation are required (Section II-D). Finally, we discuss CASPER's realization in Section II-E. We instantiated CASPER in Java and therefore tailored our presentation in this section to Java.

A. Null Dereference Causality Trace

To debug a complex null dereference, the developer has to understand the history of a guilty null from its creation to its problematic dereference. She has to know the details of the `null`'s propagation, i.e. why and when each variable became null at a particular location. We call this history the "null causality trace" of the null dereference.

Definition 2.1: A *null dereference causality trace* is the temporal sequence of language constructs traversed by a dereferenced `null`.

Developers read and write source code. Thus, source code is the natural medium in which developers reason about programs for debugging. In particular, a `null` propagates through moves or copies, which are realized via constructs like assignments and parameter binding. This is why CASPER defines causal links in a null causality trace in terms of traversed language constructs. Table I depicts language constructs through which `null`s originate, propagate, and trigger null pointer exceptions.

[2]We take the source of a null to be a given, and do not concern ourselves with its cause. Under this assumption, Casper's traces, which answer the question of how a dereference is reached, are causal.

Mnemonic	Description	Examples

L	null literal	x = null;
		Object x; //Implicit null
		return null;
		foo(null);

P_e	null at entry	void foo(int x) { ... }
P_i	null at invocation	foo(e)

R	null return	x = foo()//foo returns null
		foo().bar()
		bar(foo())

U	unboxed `null`	Integer x = e; int y = x
A	null assignment	x = e;
D	**null dereference**	x.foo() x.field
X	**external call**	lib.foo(e)

TABLE I

NULL-PROPAGATING LANGUAGE CONSTRUCTS. WE USE e TO DENOTE AN ARBITRARY EXPRESSION. IN ALL CASES BUT X, WHERE e APPEARS, A NULL PROPAGATES ONLY IF e EVALUATES TO NULL. A IS GENERIC ASSIGNMENT; IT IS THE LEAST SPECIFIC NULL-PROPAGATING LANGUAGE CONSTRUCT.

Therefore, `nulls` can originate in hard-coded null literals (L), and in external library return (P_i) or callbacks (P_e). In our causality abstraction, these links are the root causes of a null dereference. Recall that Java forbids pointer arithmetic, so we do not consider this case.

A `null` propagates through method parameters, returns (R), and unboxing (U). With the least specificity, a `null` can propagate through source level assignment (A). **D** denotes the erroneous dereference of a `null`.

When `nulls` are passed as parameter, they can be detected at parameter binding bound at a call site (P_i) and method entry (P_e). The reasons are twofold. First, we don't make any assumption on the presence, the observability and the manipulability of library code, so even in the presence of external libraries, we can trace that nulls are sent to them (P_i). Second, it enables to decipher polymorphism in the traces, we trace the actual method that has been called.

Let us consider the snippet " x = foo(bar()); ... x.field" and assume that x.field throws an NPE. The resulting null causality trace is R-R-A-**D** (return return assignment dereference). Here, the root cause is the return of the method bar[3]. The trace of Listing 2, discussed in introduction, is L-A-P_e-P_i-**D**. L and A are redundant in this case, since a single assignment statement directly assigns a `null` to the field f2; P_e and P_i are also redundant since no library call is involved. Post-processing removes such redundancy in a causality trace before CASPER presents the trace to a user.

CASPER decorates the L, A, P_i, and U "links" in a null dereference causality trace with the target variable name and the signature of the expression assigned to the target For each causal link, CASPER also collects the location of the language

[3]The root cause is not somewhere above the return in bar's body or the causality trace would necessarily be longer.

```
// original type
public MyClass{
    private Object o;
    public String sampleMethod(){
        ...
} }

// corresponding generated type
public MyGhostClass extends MyClass{
    public String sampleMethod(){
        // enriches the causality trace to log
        // that this null ghosts was dereferenced
        computeNullUsage();
        throw new CasperNullPointerException();
    }
    ... // for all methods incl. inherited ones
}
```

Listing 3. For each class of the program under debug, CASPER generates a null ghost class to replace `nulls`.

constructs (file, line) as well as the name and the stack of the current thread. Consequently, a causality trace contains a temporally ordered set of information and not just the stack at the point in time of the null dereference. In other words, a causality trace contains a `null`'s root cause and not only the stack trace of the symptom.

A causality trace is any chain of these causal links. A trace *a*) starts with a L or, in the presence of an external library, with R or P_e; *b*) may not end with a dereference (if the null pointer exception is caught, the null can continue to propagate); and *c*) may contain a return, not preceded by a method parameter link, when a void external method returns `null`. A causality trace can be arbitrarily long.

B. Null Ghosts

The special value `null` is the unique bottom element of Java's nonprimitive type lattice. Redefining `null` to trace the propagation of `nulls` during a program's execution, while elegant, is infeasible, since it would require the definition and implementation of a new language, with all the deployment and breakage of legacy applications that entails.

For sake of applicability, we leave our host language, here Java, untouched and we use un vanilla unmodified Java virtual machine. We use rewriting to create a *null ghost* to "haunt" each class defined in a codebase. A null ghost is an object that 1) contains a null causality trace and 2) has the same observable behavior as a null value. To this end, a ghost class contains a queue and overrides all methods of the class it haunts to throw null pointer exceptions.

Listing 3 illustrates this idea. CASPER creates the ghost class MyGhostClass that extends the application class MyClass. All methods defined in the application type are overridden in the new type (e.g., `sampleMethod`) as well as all other methods from the old class (See Section II-E). The new methods completely replace the normal behavior and have the same new behavior. First, the call to `computeNullUsage` enriches the causality trace with a causal element of type **D** by stating that this null ghosts was dereferenced. Then, it acts as if one has dereferenced a null value: it throws a CasperNullPointerException (a special version of the Java error thrown when one calls a method on a null value called NullPointerException), which is discussed next. Also, a null ghost is an instance of

Method	Explanation
nullAssign(x, position)	logs whether x is null at this assignment, returns x if x is a valid object or a ghost, or a new ghost if x is null
nullParam(x, position)	logs whether x is null when passed as parameter, returns x if x is a valid object or a ghost, or a new ghost if x is null
nullPassed(x, position)	logs whether x is null when received as parameter at this position, returns void
nullReturn(x, position)	logs whether x is null when returned at this position, returns x if x is a valid object or a ghost, or a new ghost if x is null
exorcise(x, position)	logs whether x is null when passed as parameter to a library call at this position, returns "null" if x is a ghost, or x
nullUnbox(x, position)	throws a null pointer exception enriched with a causality trace if a null ghost is unboxed
nullDeref(x, position)	throws a null pointer exception enriched with a causality trace if a field access is made on a null ghost

TABLE II
EXPLANATIONS OF THE METHODS INJECTED WITH CODE TRANSFORMATION.

the marker interface NullGhost. This marker interface will be used later to keep the same execution semantics between real null and null ghosts.

C. CASPER's Transformations

CASPER's transformations instrument the program under debug to detect `nulls` and construct null dereference causality traces dynamically, while preserving its semantics.

1) Overview: Our idea is to inject a number of method calls in the program under debug. They are listed in Table II. For instance, `o = foo()` is transformed into `o = nullAssign(foo(), "o, line 24")` where method "nullAssign" logs whether x is null at this assignment, returns x if x is a valid object or a ghost, or a new ghost if x is null. There are a number of other methods and transformations that we now explain.

Figure 1 and Figure 2 define CASPER's transformations. We have separated these rules into these two figures for clarity; in practice, the rules in these two figures are applied simultaneously to a program. Figure 1 contains rewriting rules for expressions, with the exception of method calls; Figure 2 contains rules for statements and includes method calls, which are both expressions and statements. In the figures, e and e_n are Java expressions, and s is a statement. For brevity, Figure 2 introduces $\langle method_decl \rangle$ for a method declaration and $\langle method_body \rangle$ for its body. Since version 5.0, Java automatically boxes and unboxes primitives to and from object wrappers, $unbox(e_1)$ denotes this operation. Equations 1a–1c, in Figure 1, define our semantics-preserving expression transformations. Section II-D discusses them.

The equations inject the following functions into the program under debug: nullDeref, nullParam, nullPassed, nullAssign, and nullReturn. These functions all check their argument for nullity. If their argument is `null`, they create a null ghost and add the causality link built into their name, i.e. nullAssign adds A. If their argument is a null ghost, they all append the appropriate causality link to the causality trace.

2) Detection of `nulls`: To provide the origin and causality of null dereferences, one has to detect null values *before* they become harmful, i.e. before they are dereferenced. This section describes how CASPER's transformations inject helper methods that detect and capture `nulls`.

In Java, as with all languages that prevent pointer arithmetic, `nulls` originate in explicit or implicit literals within the application under debug or from an external library. L in Table I lists four examples of the former case. CASPER statically detects `nulls` appearing in each of these contexts. For explicit `null` assignment, as in `x = null`, it applies its Equation 2a; for implicit, `Object o;`, it applies Equation 2b. Both of these rewritings inject nullAssign, which instantiates a null ghost and starts its causality trace with L–A. Equation 2e injects nullReturn to handle `return null`, collecting R.

Not all `nulls` can be statically detected: a library can produce them. An application may be infected by a `null` from an external library in four cases: 1) assignments whose right hand side involves an external call; 2) method invocations one of whose parameter expressions involves an external call; 3) boolean or arithmetic expressions involving external calls; and 4) callbacks from an external library into the program under debug.

Equation 2a handles the assignment case, injecting the nullAssign method to collect the causality links R, A. Equation 1c wraps the parameters of internal method calls with nullParam, which handles external calls in a parameter list and adds the R and P_e links to a null ghost. Equation 1d handles boolean or arithmetic expressions. It injects the nullUnbox method to check nullity and create a null ghost or update an existing one's trace with R and U. Finally, Equation 2d handles library callbacks. A library call back happens when the application under debug provides an object to the library and the library invokes a method of this object, as in the "Listener" design pattern. In this case, the library can bind `null` to one of the method's parameters. Because we cannot know which method may be involved in a callback, Equation 2d inserts a check for each argument at the beginning of every method call, potentially adding P_e to a ghost's causality trace.

Rewriting Java in the presence of its `final` keyword is challenging. Listing 4 shows an example application of Equation 2d. The first method is the application method and the second one is the method after instruction by CASPER. The use of `final` variables, which can only be assigned once in Java, requires us to duplicate the parameter as a local variable. Renaming the parameters (b to b_dup), then creating a local variable with the same name as the original parameter, allows CASPER to avoid modifying the body of the method.

D. Semantics Preservation

Using null ghosts instead of `nulls` must not modify program execution. CASPER therefore defines the three transformations in Equations 1a–1c, whose aim is to preserve semantics. We evaluate the degree to which our transformations preserve application semantics in Section III.

a) Comparison Operators: Consider "`o == null`". When `o` is `null`, `==` evaluates to true. If, however, `o` points to a null

$$T(e) = \begin{cases} e_1 \texttt{ == null || } e_1 \texttt{ instanceof NullGhost} & \text{if } e = e_1 \texttt{ == null} \quad (1a) \\ e_1 \texttt{ instanceof MyClass \&\& !(}e_1 \texttt{ instanceof NullGhost)} & \text{if } e = e_1 \texttt{ instanceof MyClass} \quad (1b) \\ lib.m\texttt{(exorcise(}e_1\texttt{),}\cdots\texttt{)} & \text{if } e = lib.m\texttt{(}e_1, \cdots, e_k\texttt{)} \quad (1c) \\ \texttt{nullUnbox(}unbox(e_1)\texttt{)} & \text{if } e = unbox(e_1) \quad (1d) \\ \texttt{nullDeref(}e_1\texttt{).}f & \text{if } e = e_1.f \quad (1e) \\ e & \text{otherwise} \end{cases}$$

Fig. 1. CASPER's expression transformations (excepting method calls) : rules 1a–1c preserve the semantics of the program under debug (Section II-D); rules 1d–2c inject calls to collect the U, **D** and P_i causality links (Section II-A); in Equation 1e, f denotes either a function or a field. These rules are applied simultaneously to those in Figure 2.

$$T(s) = \begin{cases} o \leftarrow \text{nullAssign}(e); & \text{if } s = o \leftarrow e \quad (2a) \\ o \leftarrow \text{nullAssign(null);} & \text{if } s = o \quad (2b) \\ m(\text{nullParam}(p_1, \cdots)) & \text{if } s = m(p_1, \cdots, p_n) \quad (2c) \\ \text{m}(p_1, \ldots, p_n)\ \{ \\ \forall p_i,\ p_i \leftarrow \text{nullPassed}(p_i); \\ \langle method_body \rangle\ \} & \text{if } s = \langle method_decl \rangle \quad (2d) \\ \text{return nullReturn}(e); & \text{if } s = \text{return } e; \quad (2e) \\ s & \text{otherwise} \end{cases}$$

Fig. 2. CASPER's statement transformations (including method calls) : these rules inject calls into statements to collect the L, A, P_e, and R causality links (Section II-A); s denotes a statement; $\langle method_decl \rangle$ denotes a method declaration and $\langle method_body \rangle$ its body; and p_i binds to a function's formals in a declaration and actuals in a call. These rules are applied simultaneously to those in Figure 1.

```
//initial method
void method(Object a, final Object b){
    //method body
}

//is transformed to
void method(Object a, final Object b_dup){
    a = NullDetector.nullPassed(a);
    b = NullDetector.nullPassed(b_dup);
    //method body
}
```

Listing 4. Illustration of the Source Code Transformation for Causality Connection "Null Method Parameter" (P_e).

ghost, the expression evaluates to false. Equation 1a preserves the original behavior by rewriting expressions, to include the conjunct " !o instanceof NullGhost". Our example " o == null" becomes the expression " o == null && !o instanceof NullGhost". Here, NullGhost is a marker interface that all null ghosts implement. The rewritten expression is equivalent to the original, over all operands, notably including null ghosts.

Java developers can write " o instanceof MyClass" to check the compatibility of a variable and a type. Under Java's semantics, if o is null, no error is thrown and the expression returns false. When o is a null ghost, however, the expression returns true. Equation 1b solves this problem. To preserve behavior, it rewrites appearances of the instanceof operator, e.g. replacing " o instanceof MyClass" with " o instanceof MyClass && !o instanceof NullGhost".

b) Usage of Libraries: During the execution of a program that uses libraries, one may pass `null` as a parameter to a library call. For instance, o could be `null` when lib.m(o) executes. After CASPER's transformation, o may be bound to a null ghost. In this case, if the library checks whether its parameters are null, using x == null or x instanceof SomeClass , a null ghost could change the behavior of the library and consequently of the program. Thus, for any method whose source we lack, we modify its calls to "unbox the null ghost", using Equation 1c. In our example, lib.m(o) becomes lib.m(exorcise (o)). When passed a null ghost, the method exorcise returns the `null` that the ghost wraps.

c) Emulating Null Dereferences: When dereferencing a `null`, Java throws an exception object NullPointerException. When dereferencing a null ghost, the execution must also results in throwing the same exception. In Listing 3, a null ghost throws the exception CasperNullPointerException, which extends Java's exception NullPointerException. The CASPER's specific exception contains the dereferenced null ghost and overrides the usual exception reporting methods, namely the getCause, toString, and printStackTrace methods, to display the ghost's causality trace.

Java throws a NullPointerException in three cases: *a*) a method call on `null`; *b*) a field access on `null`; or *c*) unboxing a `null` from a primitive type's object wrapper. CASPER trivially emulates method calls on a `null`: it defines each method in a ghost to throw CasperNullPointerException, as Listing 3 shows. Java does not provide a listener that monitors field accesses. Equation 1e overcomes this problem; it wraps expressions involved in a field access in nullDeref, which checks for a null ghost, prior to the field access. For instance, CASPER transforms x.f into nullDeref(e).f. Since version 5.0, Java has supported autoboxing and unboxing to facilitate working with its primitive types. A primitive type's object wrapper may contain a `null`; if so, unboxing it triggers a null value triggers a null dereference error. For example, Integer a = null; int b = a, a

+ 3 or a * 3 all throw NullPointerException.

E. Implementation

CASPER requires, as input, the source code of the program under debug, together with the binaries of the dependencies. Its transformations are automatic and produce an instrumented version of the program under debug. We stack source code transformation at compile time and dynamic binary code transformation at load time. The reasons are low-level details that are specific to the Java platform as explained above.

a) Source Code Transformations: We perform our source code transformations using Spoon [11]. This is done at compile time, just before the compilation to bytecode. Spoon performs all modifications on a model representing the AST of the program under debug. Afterwards, Spoon generate new Java files that contain the program corresponding to the AST after application of the transformations of Figure 1 and Figure 2.

b) Binary Code Transformations: We create null ghosts with binary code generation using ASM[4]. The reason is the Java final keyword. This keyword can be applied to both types and methods and prevents further extension. Unfortunately, we must be able override any arbitrary class and all methods to create null ghost classes. To overcome this protection at runtime, CASPER uses its our own classloader, which ignores the final keyword in method signatures when the class is loaded. For example, when MyClass must be "haunted", the class loader generates MyClassGhost.class on the fly.

c) Limitations: CASPER cannot identify the root cause of a null pointer dereference in two cases. The first is when the root cause is in external library code that we cannot rewrite. This is the price we pay to avoid assuming a closed world, where all classes are known and manipulatable. The second is specific to the fact that we implemented CASPER in Java: our classloader technique for overriding final classes and methods does not work for JDK classes, because most of these classes are loaded before the invocation of application-specific class loaders. One consequence is that our implementation of CASPER cannot provide causality traces for Java strings.

III. EMPIRICAL EVALUATION

We now evaluate the capability of our approach to build correct causality traces of real errors from large-scale open-source projects. The evaluation answers the following research questions:

$RQ1$: Does our approach provide the correct causality trace?
$RQ2$: Do the code transformations preserve the semantics of the application?
$RQ3$: Is the approach useful with respect to the fixing process?

RQ1 and RQ2 concern correctness. In the context of null deference analysis, RQ1 focuses on one kind of correctness defined as the capability to provide the root cause of the null dereference. In other words, the causality trace has to connect the error to its root cause. RQ2 assesses that the behavior of the application under study does not vary after applying our code transformations. RQ3 studies the extent to which causality traces help a developer to fix null dereference bugs.

A. Dataset

We built a dataset of real life null dereference bugs. There are two inclusion criteria. First, the bug must be a real bug reported on a publicly-available forum (e.g. a bug repository). Second, the bug must be reproducible.

The reproducibility is challenging. Since our approach is dynamic, we must be able to compile and run the software in its faulty version. First, we need the source code of the software at the corresponding buggy version. Second, we must be able to compile the software. Third, we need to be able to run the buggy case. In general, it is really hard to reproduce real bugs in general and null dereferences in particular. Often, the actual input data or input sequence triggering the null dereference is not given, or the exact buggy version is not specified, or the buggy version can no longer be compiled and executed.

We formed our data set in two ways. First, we tried to replicate results over a published data set [2] as described below. Second, we selected a set of popular projects. For each project, we used a bag of words over their bug repository (e.g. bugzilla of jira) to identify an under approximate set of NPEs. We then faced the difficult challenge of reproducing these bugs, as bug reports rarely specify the bug-triggering inputs. Our final data set is therefore conditioned on reproducibility. We do not, however, have any reason to believe that any bias that may exist in our data set would impact Casper general applicability.

Under these constraints, we want to assess how our approach compares to the closest related work [2]. Their dataset dates back to 2008. In terms of bug reproduction 6 years later, this dataset is hard to replicate. For 3 of the 12 bugs in this work, we cannot find any description or bug report. For 4 of the remaining 9, we cannot build the software because the versions of the libraries are not given or no longer available. 3 of the remaining 5 do not give the error-triggering inputs, or they do not produce an error. Consequently, we were only able to reproduce 3 null dereference bugs from Bond et al.'s dataset.

We collected 7 other bugs. The collection methodology follows. First, we look for bugs in the Apache Commons set of libraries (e.g. Apache Commons Lang). The reasons are the following. First, it is a well-known and well-used set of libraries. Second, Apache commons bug repositories are public, easy to access and search. Finally, thanks to the strong software engineering discipline of the Apache foundation, a failing test case is often provided in the bug report.

To select the real bugs to be added to our dataset we proceed as follows. We took all the bugs from the Apache bug repository[5]. We then select 3 projects that are well used and well known (Collections, Lang and Math). We add the condition that those bug reports must have "NullPointerException" (or "NPE") in their title. Then we filter them to

[4] http://asm.ow2.org/

[5] https://issues.apache.org/jira/issues

# Bug Id	Problem summary	Fix summary
McKoi	new JDBCDatabaseInterface with null param -> field -> deref	Not fixed (Artificial bug by [2])
Freemarker #107	circular initialization makes a field null in WrappingTemplate-Model -> deref	not manually fixed. could be fixed manually by adding hard-code value. no longer a problem with java 7.
JFreeChart #687	no axis given while creating a plot	can no longer create a plot without axis modifying constructor for fast failure with error message
collection #331	no error message set in a thrown NPE	add a check not null before the throw + manual throwing of NPE
math #290	NPE instead of a domain exception when a null List provided	normalize the list to use empty list instead of null
math #305	bad type usage (int instead of double). Math.sqrt() call on an negative int -> return null. should be a positive double	change the type
math #1117	Object created with too small values, after multiple iterations of a call on this object, it returns null	create a default object to replace the wrong valued one
7 other bugs		add a nullity check

TABLE III

A DATASET OF 14 REAL NULL DEREFERENCE ERRORS FROM LARGE SCALE OPEN-SOURCE PROJECTS. THE DATASET IS MADE PUBLICLY AVAILABLE FOR FUTURE REPLICATION AND RESEARCH ON THIS PROBLEM.

Bug ID	#LOC	#classes
McKoi	48k	275
Freemarker #107	37k	235
JFreeChart #687	70k	476
coll-331	21k	256
lang-304	17k	77
lang-587	17k	80
lang-703	19k	99
math-290	38k	388
math-305	39k	393
math-369	41k	414
math-988a	82k	781
math-988b	82k	781
math-1115	90k	885
math-1117	90k	885

TABLE IV

DESCRIPTIVE SUMMARY OF THE DATASET OF NULL DEREFERENCES

keep only those which have been fixed and which are closed (our experimentation needs the patch). After filters 19 bug reports remain[6]. Sadly, on those 19 bug reports, 8 are not relevant for our experiment: 3 are too olds and no commit is attached (COLL-4, LANG-42 and Lang-144), 2 concern Javadoc (COLL-516 and MATH-466), 2 of them are not bugs at all (LANG-87 and MATH-467), 1 concerns a VM problem. Finally, we add the 11 remaining cases to our dataset.

Consequently, the dataset contains the 3 cases from [2] (Mckoi, freemarker and jfreechart) and 11 cases from Apache Commons (1 from collections, 3 from lang and 7 from math). In total, the bugs come from 6 different projects, which is good for assessing the external validity of our evaluation. This makes a total of 14 real life null dereferences bugs in the dataset.

Table III shows the name of the applications, the number of the bug Id (if existing), a summary of the NPE cause and a summary of the chosen fix. We put only one line for 7 of them because they use the same simple fix (i.e. adding a check not null before the faulty line). The application coverage of the test suites under study are greater than 90% for the 3 Apache common projects (11 out of the 14 cases). For the 3 cases from [2] (Mckoi, freemarker and jfreechart), we do not have access to the full test suites. Table IV gives the main descriptive statistics. For instance, the bug in McKoi is an application of 48000+ lines of code spread over 275 classes.

This dataset only contains real null dereference bugs and no artificial or toy bugs. To reassure the reader about cherry-picking, we have considered all null dereferenced bugs of the 3 selected projects. We have not rejected a single null dereference that CASPER fails to handle.

B. Methodology

1) Correctness: RQ1: Does our approach provide the correct causality trace? To assess whether the provided element is responsible for a null dereference, we manually analyze each case. We manually compare the result provided by our technique with those coming from a manual debugging process that is performed using the debug mode of Eclipse.

RQ2: Do the code transformations preserve the semantics of the application? To assert that our approach does not modify the behavior of the application, we use two different strategies.

First, we require that the origin program and the transformed program both pass and fail the same tests in the test suite (when it exists). Having the same behavior according to a test suite does not prove semantic equivalence, it is only an indication the program has not been broken. However, this is the only possible technique since current research on formal equivalence is not capable of proving semantic equivalence in a programming language as complex as Java.

This test suite test only addresses the correctness of externally observable behavior of the program under debug. To assess that our approach does not modify the internal behavior, we compare the execution traces of the original program (prior to code transformation) and the program after transformation. Here, an "execution trace" is the ordered list of all method calls and of all returned values, executing over the entire test suite. This trace is obtained by logging method entry (injecting package.class\#method(arg.toString \ldots)) and logging return (injecting package.class\#method():returnedValue.toString). We filter out all calls to the Casper framework, then align the two traces. They must be identical. As for the test suite, execution trace equivalence is only a proxy to complete equivalence.

2) Effectiveness: RQ3: Is the approach useful with respect to the fixing process? To assert that our additional data is useful, we look at whether the location of the real fix is given in the causality trace. If the location of the actual fix

[6]The link to automatically set those filters is given in https://github.com/Spirals-Team/casper

is provided in the causality trace, it would have helped the developer by reducing the search space of possible solutions. Note that in 7/14 cases of our dataset, the fix location already appears in the original stack trace. Those are the 7 simple cases where a check not null is sufficient to prevent the error. Those cases are valuable in the context of our evaluation to check that: 1) the variable name is given (as opposed to only the line number of a standard stack trace), 2) the causality trace is correct (although the fix location appears in the original stack trace, it does not prevent a real causality trace with several causal connections).

C. Results

1) RQ1: After verification by manual debugging, in all the cases under study, the element identified by our approach is the one responsible for the error. This result can be replicated since our dataset and our prototype software are made publicly available.

2) RQ2: All the test suites have the same external behavior with and without our modifications according to our two behavior preservation criteria. First, the test suite after transformation still passes. Second, for each run of the test suite, the order of method calls is the same and the return values are the same. In short, our massive code transformations do not modify the behavior of the program under study and provide the actual causality relationships.

3) RQ3: We now perform two comparisons. First, we look at whether the fix locations appear in the standard stack traces. Second, we compare the standard stack trace and causality trace to see whether the additional information corresponds to the fix.

Table V presents the fix locations (class and line number) (second column) and whether: this location is provided in the basic stack trace (third column); 2) the location is provided by previous work [2] (fourth column); 3) it is in the causality trace (last column). The first column, "# Bug Id", gives the id of the bug in the bug tracker of the project (same as Table III).

In 7 out of 14 cases (the 7 simple cases), the fix location is in the original stack trace. For those 7 cases, the causality trace is correct and also points to the fix location. In comparison to the original stack trace, it provides the name of the root cause variable.

In the remaining 7 cases, the fix location is not in the original stack trace. This means that in 50% of our cases, there is indeed a cause/effect chasm, that is hard to debug [4], because no root cause information is provided to the developer by the error message. We now explain in more details those 7 interesting cases.

The related work [2] would provide the root cause in only 1 out of those 7 cases (according to an analysis, since their implementation is not executable). In comparison, our approach provides the root cause in 4 out of those 7 cases. This supports the claim that our approach is able to better help the developers in pinpointing the root cause compared to the basic stack trace or the related work.

4) Detailed Analysis:

```
1  private static Cluster<T> getNearestCluster(final Collection<Cluster> clusters, final
       T point) {
2      double minDistance = Double.MAX_VALUE;
3      Cluster<T> minCluster = null; //initialisation
4      for (final Cluster<T> c : clusters) {
5          final double distance = point.distanceFrom(c.getCenter()); //return NaN
6          if (distance < minDistance) { //failing condition
7              minDistance = distance;
8              minCluster = c;
9          }
10     }
11     return minCluster; //return null
12 }
13
```

Listing 5. An excerpt of Math #305 where the causality trace does not contain the fix location.

a) Case Studies: There are two different reasons why our approach does not provide the fix location: First, for one case, our approach is not able to provide a causality trace. Second, for two cases, the root cause of the null dereference is not the root cause of the bug.

In the case of Math #290, our approach is not able to provide the causality trace. This happens because the null value is stored in an Integer, which is a final type coming from the jdk. Indeed, java.lang.Integer is a native Java type and our approach cannot modify them (see paragraph II-E0c).

In the case of Math #305, the root cause of the null dereference is not the root cause of the bug. The root cause of this null dereference is shown in Listing 5. The null responsible of the null dereference is initialized on line 4, the method call `distanceFrom` on line 6 return `NaN`, due to this NaN, the condition on line 7 fails, and the null value is returned (line 9). Here, the cause of the dereference is that a null value is returned by this method. However, this is the root cause of the `null` but this is not the root cause of the bug. The root cause of the bug is the root cause of the `NaN`. Indeed, according to the explanation and the fix given by the developer the call `point.distanceFrom(c.getCenter())` should not return `NaN`. Hence, the fix of this bug is in the `distanceFrom` method, which does not appear in our causality chain because no `null` is involved.

In the case of Math #1117, the root cause of the null dereference is not the root cause of the bug. The root cause of this null dereference is shown in Listing 6. The null responsible of the dereference is the one passed as second parameter of the constructor call on line 10. This null value is stored in the field minus of this SplitSubHyperplane. Here, the cause of the dereference is that a null value is set in a field of the object returned by this method. Once again, this is the root cause of the `null` but this is not the root cause of the bug. The root cause of the bug is the root cause of the failing condition $\text{global} < -1.0e-10$. Indeed, according to the explanation and the fix given by the developer the Hyperplane passed as method parameter should not exist if its two lines are too close from each other. Here, this Hyperplane comes from a field of a PolygonSet. On the constructor of this PolygonSet they pass a null value as a parameter instead of this "irregular" object. To do that, they add a condition based on a previously existing parameter called `tolerance`, if the distance of the two lines are lower than this tolerance, it returns a null value.

# Bug Id	Fix location	Fix location in the standard stack trace	Addressed by [2]	Fix location in CASPER's causality trace	Causality Trace	Exec. time instrumented (ms)
McKoi	Not fixed	No	No	Yes	L-A-R-A-**D**	382
Freemarker #107	Not fixed	No	Yes	Yes	L-A-**D**	691
JFreeChart #687	FastScatterPlot 178	No	No	Yes	L-P_e-P_i-**D**	222
collection #331	CollatingIterator 350	No	No	Yes	L-A-**D**	81
math #290	SimplexTableau 106/125/197	No	No	No	**D**	107
math #305	MathUtils 1624	No	No	No	L-R-A-R-**D**	68
math #1117	PolygonSet 230	No	No	No	L-A-R-A-**D**	191
7 simple cases		Yes	Yes	Yes	L-A-**D** (x6) L-A-U	147 (average)
Total		7 / 14	8/14	11/14		

TABLE V

EVALUATION OF CASPER: IN 13/14 CASES, A CAUSALITY TRACE IS GIVEN, IN 11/13 THE CAUSALITY TRACE CONTAINS THE LOCATION WHERE THE ACTUAL FIX WAS MADE.

```
1
2   public SplitSubHyperplane split(Hyperplane hyperplane) {
3       Line thisLine = (Line) getHyperplane();
4       Line otherLine = (Line) hyperplane;
5       Vector2D crossing = thisLine.intersection(otherLine);
6     if (crossing == null) { // the lines are parallel
7         double global = otherLine.getOffset(thisLine);
8         return (global < −1.0e−10) ?
9                 new SplitSubHyperplane(null, this) :
10                new SplitSubHyperplane(this, null); // initialisation
11    }
12    ...
13  }
```

Listing 6. An excerpt of Math #1117 where the causality trace does not contain the fix location.

(It is interesting that the fix of a null dereference is to return a null value elsewhere.)

b) Size of the Traces: There are mainly two kind of traces encountered in our experiment. First, the one of size 3 and of kind L-A-**D** type. The 7 obvious cases (were the fix location is in the stack trace) contains 6 traces of this kind. In all those cases encountered in our experiment, the null literal has been assigned to a field. This means that a field has not been initialized (or initialized to null) during the instance creation, hence, this field is dereferenced latter. This kind of trace is pretty short so one may think that this case is obvious. However, all of those fields are initialized long ago the dereference. In other words, when the dereference occurs, the stack has changed and no longer contains the information of the initialization location.

Second, the one of size ≤ 4 where the null is stored in a variable then passed as argument in one or multiple methods. In all those case, the null value is either returned by a method at least once or passed as a parameter.

c) Execution Time: To debug a null dereference error, CASPER requires to instrument the code and to run the instrumented version. In all the cases, the instrumentation time is less 30 seconds. At runtime, CASPER finds the causility trace of the failing input in less than 1 second (last column of Table V). This seems reasonable from the developer viewpoint: she obtain the causality trace in less than 30 seconds. We have also measured the overhead with respect the original test case trigerring the error: it's a 7x increase.

IV. DISCUSSION

A. Causality Trace and Patch

Once the causality trace of a null dereference is known, the developer can fix the dereference. There are two basic dimensions for fixing null references based on the causality trace.

First, the developer has to select in the causal elements, the location where the fix should be applied: it is often at the root cause, i.e. the first element in the causality trace. It may be more appropriate to patch the code elsewhere, in the middle of the propagation between the first occurrence of the null and the dereference error.

Second, the developer has to decide whether there should be an object instead of null or whether the code should be able to gracefully handle the null value. In the first case, the developer fixes the bug by providing an appropriate object instead of the null value. In the second case, she adds a null check in the program to allow a null value.

Sometimes, the fix for a null dereference will be an insertion of a missing statement or the correction of an existing conditional. In this case, the CASPER causality trace does not contain the exact location of the fix. However, as in the case of bug Math #290 that we have discussed, it is likely that those modifications will be made in the methods involved in the causality trace. According to our experience, those cases are uncommon but future work is required to validate this assumption.

B. Use in Production

As shown in Section III-C4c, the overhead of the current implementation is too large to be used in production. We are confident that advanced optimization can reduce this overhead. This is left to future work.

C. Threats to Validity

The internal validity of our approach and implementation has been assessed through RQ1 and RQ2: the causality trace of the 14 analyzed errors is correct after manual analysis. The threat to the external validity lies in the dataset composition:

does the dataset reflect the complexity of null dereference errors in the field? To address this threat, we took a special care in designing the methodology to build the dataset. It ensures that the considered bugs apply to large scale software and are annoying enough to be reported and commented in a bug repository. The generalizability of our results to null dereference errors in other runtime environments (e.g. .NET; Python) is an open question to be addressed by future work.

V. Related Work

There are several static techniques to find possible null dereference bugs. Hovemeyer et al. [6] use byte-code analysis to provide possible locations where null dereference may happen. Sinha et al. [14] use source code path finding to find the locations where a bug may happen and apply the technique to localize Java null pointer exceptions symptom location. Spoto [15] devises an abstract interpretation dedicated to null dereferences [15]. Ayewah and Pugh [1] discussed the problems of null dereference warnings that are false positives. Compared to these works, our approach is dynamic and instead of predicting potential future bugs that may never happen in production, it gives the root cause of actual ones for which the developer has to find a fix.

Dobolyi and Weimer [3] present a technique to tolerate null dereferences based on the insertion of well-typed default values to replace the null value which is going to be dereferenced. Kent [7] goes further and, proposes two other ways to tolerate a null dereference: skipping the failing instruction or returning a well-typed object to the caller of the method. In the opposite, our work is not on tolerating runtime null dereference but on giving advanced debugging information to the developer to find a patch.

The idea of identifying the root cause in a cause effect chain has been explored by Zeller [19]. In this paper, he compares the memory graph from the execution of two versions of a same program (one faulty and one not faulty) to extract the instructions and the memory values which differ and presumably had lead to the error. This idea has bee further extended by Sumner and colleagues [16], [17]. Our problem statement is different, those approaches takes as input two different versions of the program or two different runs and compare them. On the contrary, we build the causality trace from a single execution.

The Linux kernel employs special values, called poison pointers, to transform certain latent null errors into fail-fast errors [13]. They share with null ghosts the idea of injecting special values into the execution stream to aid debugging and, by failing fast, to reduce the width of the cause/effect chasm. However, poison values only provide fail-fast behavior and do not provide causality traces or even a causal relationship as we do.

Romano et al. [12] find possible locations of null dereferences by running a genetic algorithm to exercise the software. If one is found, a test case that demonstrate the null dereference is provided. Their technique does not ensure that the null dereferences found are realistic and represent production problem. On the contrary, we tackle null dereferences for which the programmer has to find a fix.

Wang et al. [18] describe an approach to debug memory errors in C code. What they call "value propagation chain" corresponds to our causality traces. They don't provide a taxonomy of causal elements as we do in Table I and they take a great care of pointer arithmetic, which is irrelevant in our case.. Their transformations are at the level of x86 code using dynamic instrumentation, while we work on Java source code. This makes a major difference: all the transformations we have described are novel, and cannot be inferred or derived from Wang et al.'s work.

Bond et al. [2] present an approach to dynamically provide information about the root cause of a null dereference (i.e. the line of the first null assignment). The key difference is that we provide the complete causality trace of the error and not only the first element of the causality trace. As discussed in the evaluation (Section III), the actual fix of many null dereference bugs is not necessary done at the root cause, but somewhere up in the causality trace.

Like null ghosts, the null object pattern replaces nulls with objects whose interface matches that of the null-bound variable's type. Unlike null ghosts, the methods of an instance of the null object pattern are empty. Essentially, the null object pattern turns method NPEs into NOPs. To this extent, the refactoring proposed by [5], does not help to debug null dereferences but avoids some of them. In contrast, null ghosts collect null dereference causality traces that allow a developer to localize and resolve an NPE.

VI. Conclusion

In this paper, we have presented Casper, a novel approach for debugging null dereference errors. The key idea of our technique is to inject special values, called "null ghosts" into the execution stream to aid debugging. The null ghosts collect the history of the null value propagation between its first detection and the problematic dereference, we call this history the 'causality trace". We define 11 code transformations responsible for 1) detecting null values at runtime, 2) collect causal relations and enrich the causality traces; 3) preserve the execution semantics when null ghosts flow during program execution. The evaluation of our technique on 14 real-world null dereference bugs from large-scale open-source projects shows that Casper is able to provide a valuable causality trace. Our future work consists in further exploring the idea of "ghost" for debugging other kinds of runtime errors such as arithmetic overflows.

References

[1] N. Ayewah and W. Pugh. Null dereference analysis in practice. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 65–72. ACM, 2010.

[2] 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. *ACM SIGPLAN Notices*, 42(10):405–422, 2007.

[3] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 47–56. IEEE, 2008.

[4] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997.

[5] M. A. G. Gaitani, V. E. Zafeiris, N. Diamantidis, and E. Giakoumakis. Automated refactoring to the null object design pattern. *Information and Software Technology*, 59:33–52, 2015.

[6] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 13–19. ACM, 2005.

[7] S. W. Kent and K. Mckinley. Dynamic error remediation: A case study with null pointer exceptions. 2008.

[8] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Does return null matter? In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 244–253, 2014.

[9] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[10] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[11] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon v2: Large scale source code analysis and transformation for java. Technical Report hal-01078532, INRIA, 2006.

[12] D. Romano, M. Di Penta, and G. Antoniol. An approach for search based testing of null pointer exceptions. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 160–169. IEEE, 2011.

[13] A. Rubini and J. Corbet. *Linux device drivers*. O'Reilly Media, Inc., 2001.

[14] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 153–164. ACM, 2009.

[15] F. Spoto. Precise null-pointer analysis. *Software & Systems Modeling*, 10(2):219–252, 2011.

[16] W. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *Fundamental Approaches to Software Engineering*, pages 355–369, 2009.

[17] W. Sumner and X. Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the International Conference on Software Engineering*, pages 272–281, 2013.

[18] Y. Wang, I. Neamtiu, and R. Gupta. Generating sound and effective memory debuggers. In *ACM SIGPLAN Notices*, volume 48, pages 51–62. ACM, 2013.

[19] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.