Casper: Automatic Tracking of Null Dereferences to Inception with Causality Traces

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

Casper: Automatic Tracking of Null Dereferences to Inception with Causality Traces

Benoit Cornu*, Earl T. Barr†, Lionel Seinturier*, Martin Monperrus*
*University of Lille & INRIA
†University College London

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 `null`s with special objects, 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.

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

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

```
1  Exception in thread "main" java.lang.NullPointerException
2  Dereferenced parameter "f" // symptom
3      at [..].BisectionSolver.solve(88)
4      at [..].BisectionSolver.solve(66)
5      at ...
6  Parameter f bound to field "f2"
7      at [..].BisectionSolver.solve(66)
8  Field "f2" set to null
9      at [..].UnivariateRealSolverImpl.<init>(55) //
           cause
```

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

1 Introduction

Null pointer dereferences are frequent errors that cause segmentation faults or uncaught exceptions. Li et al. found that 37.2% of all memory errors in Mozilla and Apache are null dereferences [15]. Kimura et al. [14] found that there are between one and four null checks per 100 lines of code on average. Problematic null dereferences are daily reported in bug repositories, such as bug MATH#305[1]. A null dereference occurs at runtime when a program tries to read memory using a field, parameter, or variable that points to "`null`", i.e. nothing. The terminology changes depending on the language, in this paper, we concentrate on the Java programming language, where a null dereference triggers an exception called NullPointerException, often called "NPE".

Just like any bug, fixing null dereferences requires understanding their root cause, a process that we call *null causality analysis*. At its core, this analysis is the process of connecting a null dereference, where the fault is activated and whose symptom is a null pointer exception, to its root cause, usually the initial assignment of a null value, by means of a *causality trace* — the execution path the null took through the code from its inception to its dereference.

The literature offers different families of techniques to compute the root cause of bugs, mainly program slicing, dataflow analysis, or spectrum-based fault-localization. However, they have been little studied and evaluated in the context of identifying the root cause of null dereferences [11, 4, 24]. Those techniques are limited in applicability ([11] is an intra-procedural technique) or in accuracy (program slicing results in large sets of instructions [3]). The fundamental problem not addressed in the literature is that the causality trace from null inception to the null symptom is missing. This is the problem that we address in this paper. We propose a causality analysis technique that uncovers the inception of null variable bindings that lead to errors along with the causal explanation of how null flowed from inception to failure during the execution. While our analysis may not report the root cause, it identifies the null inception point with certainty, further localizing the root cause and speeding debugging in practice.

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

Let us consider a concrete example. Listing 1 shows a null dereference stack trace which shows that the null pointer exception happens at line 88 of BisectionSolver. Let's assume that a perfect fault localization tool suggests that this fault is located at line 55 of UnivariateRealSolverImpl (which is where the actual fault lies). However, the developer is left clueless with respect to the relation between line 55 of UnivariateRealSolverImpl and line 88 of BisectionSolver where the null dereference happens.

What we propose is a *causality trace*, as shown in Listing 2. 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 variable that holds null[2]. Second, it explains the inception of the null binding to 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 causal links, of which Listing 2 shows only three: the name of the wrongly dereferenced variable, the flow of a null binding through parameter bindings, and null assignment. Section 2.2 presents the concept of null causality trace.

We present CASPER, a tool that transforms Java programs to capture causality traces and facilitate the fixing of null deferences[3]. CASPER takes as input the program under debug and a main routine that triggers the null dereference. It first instruments the program under debug by replacing `null` with "ghosts" that are shadow instances responsible for tracking causal information during execution. To instrument a program, CASPER applies a set of 11 source code transformations tailored for building causal connections. For instance, x = y is transformed into o = assign(y) , where method assign stores an assignment causal links in a null ghost (Section 2.3). Section 2.4 details these transformations.

Compared to the related work, CASPER is novel along three dimensions. First, it collects the complete causality trace from the inception of a null binding to its dereference. Second, it identifies the inception of a null binding with *certainty*. Third, CASPER is lightweight and easily deployable, resting on transformation rather than replacing the Java virtual machine, a la Bond et al. [4]. The first two properties strongly differentiate CASPER from the related work [20, 4], which tentatively labels root causes with suspiciousness values and does not collect causality traces nor identifies null inception points with certainty.

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.

To sum up, our contributions are:

- The definition of causality traces for null dereference errors from null inception to its dereference and the concept of "ghost" classes, which replace `null`, collect causality traces, while being otherwise indistinguishable from `null`.
- A set of code transformations that inject null ghosts and collect causality traces of null dereferences.
- CASPER, an Java implementation of our technique.
- An evaluation of our technique on 14 real null dereference bugs collected over 6 large open-source projects.

The remainder of this paper is structured as follows. Section 2 presents our technical contribution. Section 3 gives the results of our empirical evaluation. Section 4 discusses the limitations of our approach. Sections 5 and 6 respectively discusses the related work and concludes. CASPER and our benchmark can be downloaded from `https://github.com/Spirals-Team/casper`.

2 Debugging Nulls with CASPER

CASPER tracks the propagation of a `null` binding during application execution in a causality trace. A *null dereference causality trace* is a sequence of program elements (AST nodes) traversed during execution from the source of the `null` to its erroneous dereference.

2.1 Overview

We replace `null`s with objects whose behavior, from the application's point of view, is same as `null`, except that they store a causality trace, defined in Section 2.2. We call these objects *null ghosts* and detail them in Section 2.3. CASPER rewrites the program under debug to use null ghosts and to store a `null`'s causality trace in those null ghosts, (see Section 2.4).

[2] if the dereference is the result of a method invocation, we give the code expression that evaluates to null

[3] We have named our tool CASPER, since it injects "friendly" ghosts into buggy programs.

We instantiated CASPER's concepts in Java and therefore tailored our presentation in this section to Java (Section 2.5).

CASPER makes minimal assumptions on the application under debug, in particular, it does not assume a closed world where all libraries are known and manipulable. Hence, a number of techniques used in CASPER comes from this complication.

2.2 Null Dereference Causality Trace

To debug a complex null dereference, the developer has to understand the history of a null binding from its inception to its problematic dereference. When a variable is set to `null`, we say that a null binding is created. When clear from context, we drop "binding" and say only that a null is created. This is a conceptual view that abstracts over the programming language and the implementation of the virtual machine. In Java, there is a single `null` value to which variables are bound without creating a new `null` values.

To debug a null dereference, the developer 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. 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 assignments and method return values. This is why CASPER defines causal links in a null causality trace in terms of traversed program elements and their actual location in code, defined as follows and presented in Table 1.

Definition 2.1 *A* null dereference causality trace *is the temporal sequence of program elements (AST nodes) traversed by a dereferenced* `null`*.*

2.2.1 Inception

A `null` binding can obviously originate in hard-coded null literals (L). In our causality abstraction, these inception points form the first element (i.e. the presumed root cause) of a null dereference causality trace. Recall that Java forbids pointer arithmetic, so we do not consider this case. Also, when a parameter is bound to `null`, this binding can be detected at method entry (P). This enables CASPER to detect null values as soon they come from libraries that use callbacks within the application under debug (i.e. when the stack is of the form $app \rightarrow lib \rightarrow app$).

Mnemonic	Description	Examples
Inception		
L	null literal	x = null; Object x = null return null;
P	null at method entry	void foo(Object x) { ... }
Propagation		

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

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

Table 1: Causality trace elements, their mnemonic and the language constructs they are associated with. 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`.

2.2.2 Propagation

During execution, CASPER traces the propagation of `null` bindings. First, a `null` naturally propagates through source level assignment (A). Second, a null can be propagated through returned objects. Hence, CASPER detects `null`s at the "return site" (R). In foo ().bar() there are two R links if something follows such as in foo().bar().baz(). If this is followed by an assignment x=foo().bar(), this is one dereference case and one assignment case (explained in Section 2.2.2).

2.2.3 Causation

The cause of a null pointer exception is an erroneous dereference of a `null`. According to the Java Language Specification ("15.6. Normal and Abrupt Completion of Evaluation") [10], this can happen during method calls, field accesses or array accesses. All cases are represented as by causality link D in Table 1. Another case is unboxing, which can also trigger null pointer exceptions (U).

Let us consider the snippet " x = foo(); ... x.field" and assume that x.field throws an NPE. The resulting

```
// 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 ghost 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 `null`s.

null causality trace is R-A-D (return / assignment / dereference). Here, the root cause is the return of the method foo.

2.2.4 Traceability

Casper decorates the links in a null dereference causality trace with the corresponding source code expression. For each causal link, Casper also collects the location of the program elements (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 starts with a L or, in the presence of an external library, with R (direct call) or *P* (callback); A causality trace can be arbitrarily long (yet it cannot be longer than the whole program trace).

2.3 Null Ghosts

The special value `null` is the unique bottom element of Java's nonprimitive type lattice. Redefining `null` to track the propagation of a `null` binding during a program's execution would require changing the Java virtual machine. To make Casper practical and useful for practicing developers, we do not change Java's type lattice and leverage the insights that we can emulate null values at the language level as follows.

To define `null` values that track causality traces without changing Java, we create null ghost classes and instantiate them as objects. 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 of causal links and overrides all methods of the class it haunts to throw null pointer exceptions. We workaround the Java keyword final to the maximum possible extent as explained later in Section 2.5, so as to create a null ghost class for all classes used at runtime.

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 2.5). 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 ghost was dereferenced. Then, it acts as if one has dereferenced a null value: it throws a CasperNullPointerException (a special version of the Java exception NullPointerException, and subtype of it.). Also, a null ghost is an instance of the marker interface NullGhost. This marker interface will be used later to keep the same execution semantics between real null and null ghosts.

2.4 Code Transformations

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

Casper uses code transformation to inject the method calls listed in Table 2 into the program under debug. The argument x is an object, a null, or a ghost; the pos argument gives the position in the original source file of the program element being transformed, for the sake of clearer traces. 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 append the appropriate causality link to the causality trace. For instance, o = e is transformed into

o = nullAssign(e, "o, line 24")

where method nullAssign logs whether the expression *e* (a variable, a method call, a field access) is `null` at this assignment, returns x if x is a valid object or a ghost, or a new ghost if x is `null`.

Figure 1 defines Casper's transformations: α is a program element, e is a Java expression, and f denotes either a function or a field. Since version 5.0, Java automatically boxes and unboxes primitives to

Method	Explanation
nullAssign(x, pos)	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
nullPassed(x, pos)	logs whether x is null when received as parameter at this position, returns void
nullReturn(x, pos)	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, pos)	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, pos)	throws a null pointer exception enriched with a causality trace if a null ghost is unboxed
nullDeref(x, pos)	throws a null pointer exception enriched with a causality trace if a field access or array access is made on a null ghost

Table 2: Explanations of the methods injected under CASPER's code transformation.

$$T(\alpha) = \begin{cases} e \texttt{ == null || } e \texttt{ instanceof NullGhost} & \text{if } \alpha = e \texttt{ == null} \quad (1) \\ e \texttt{ != null \&\& !(}e \texttt{ instanceof NullGhost)} & \text{if } \alpha = e \texttt{ != null} \quad (2) \\ e \texttt{ instanceof MyClass \&\& !(}e \texttt{ instanceof NullGhost)} & \text{if } \alpha = e \texttt{ instanceof MyClass} \quad (3) \\ \mathit{unbox}(\texttt{nullUnbox}(e)) & \text{if } \alpha = \mathit{unbox}(e) \quad (4) \\ \mathit{lib}.m\texttt{(exorcise}(p_1)\texttt{,}\cdots\texttt{)} & \text{if } \alpha = \mathit{lib}.m\texttt{(}p_1\texttt{,}\cdots\texttt{,}p_n\texttt{)} \quad (5) \\ \texttt{nullDeref(}e\texttt{).}f & \text{if } \alpha = e.f \quad (6) \\ o \texttt{ <- nullAssign(}e\texttt{);} & \text{if } \alpha = o \texttt{ <- } e \text{ (assignment)} \quad (7) \\ \texttt{return nullReturn(}e\texttt{);} & \text{if } \alpha = \texttt{return } e\texttt{;} \quad (8) \\ \texttt{Ret m(}\vec{p}\texttt{, final }\vec{Q}\texttt{) \{} \\ \quad p_i \texttt{ <- nullPassed(}p_i\texttt{);}, \forall p_i \in \vec{p} \\ \quad \texttt{final } q_j \texttt{ <- nullPassed(}q'_j\texttt{);}, \forall q'_j \in \vec{Q} \\ \quad \langle \mathit{mbody} \rangle \texttt{ \}} & \text{if } \alpha = \texttt{Ret m(}\vec{p}\texttt{, final }\vec{q}\texttt{) \{} \langle \mathit{mbody} \rangle \texttt{ \}} \quad (9) \\ \alpha & \text{otherwise} \end{cases}$$

Figure 1: CASPER's transformations: α is program element; if α matches the element on the right, it is replaced with the element on the left; unmatched program elements are unchanged; the notation is defined in the text.

and from object wrappers, $\mathit{unbox}(e_1)$ denotes Java's implicit unboxing operation. The notation used in Rule 9 is discussed in Section 2.4.1 next.

Rules 1–5 preserve the semantics of the program under debug (Section 2.4.2). For instance, naïvely replacing `null` values with ghosts breaks conditionals that explicitly check for `null`: rules 1–3 handle this case. Rules 6–9 inject calls to collect U and D causality links (Section 2.2) into assignments, internal and external method calls, returns, and method declarations.

2.4.1 Detection of `null` Bindings

To provide the inception and causality trace of a null dereference, one must detect null values *before* they are dereferenced. This section describes how CASPER's transformations inject helper methods that detect and capture `null` bindings. In Java, as with all languages that prevent pointer arithmetic, `null` bindings originate in `null` literals within the application under debug or when an external library call returns `null`.

CASPER statically detects `null` literals, such as `Object o = null` (or `Object o`, which is an equivalent default initialization). For `null` assignment, as in `x = null`, CASPER applies rule Rule 7. Field initializations are handled in the same way, both for explicit and default initialization. The corresponding rewritings inject nullAssign, which instantiates a null ghost and starts its causality trace with L–A.

Not all `null`s can be statically detected: a library can produce them. An application may be infected by a `null` from an external library in the following cases:

1. assignments whose right hand side involves an external call; 2. method invocations one of whose parameter expressions involves an external call; 3. callbacks from an external library into the program under debug.

Rule 7 handles the assignment case, injecting the nullAssign method to collect the causality links R, A. Rule 4 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.

Rule 9 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 a callback may invoke, Rule 9 inserts a check for each argument at the beginning of every method call, potentially adding P to a ghost's causality trace. Rule 8 injects nullReturn to handle return null, collecting R.

Rewriting Java in the presence of its final keyword is challenging. 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 to fresh names (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.

In Rule 9, $\langle mbody \rangle$ matches the body of the declared method; $\vec{p}$ denotes the method's nonfinal parameters; $\vec{q}$ are the method's final parameters. In the replacement on the left, we abuse notation and use $\in$ on the parameter vector to denote selecting each parameter in each parameter vector. Under this interpretation, the forall operators generate an assignment that calls nullPassed on each parameter. Q is the vector of fresh names for each final parameter; q' is one of them, and q is the original name. Listing 4 shows the result of the following application of Rule 9:

T(void method(Object a, final Object b){ <mbody> })

The first method is the input and the second is the method after instrumentation by Casper.

2.4.2 Semantics Preservation

Casper uses additional transformations in order not to modify program execution. Consider " o == null". When o is `null`, == evaluates to true. If, however, o points to a null ghost, the expression evaluates to false. Casper defines other transformations shown in rules 1–5, whose aim is to preserve semantics with the presence of null ghosts.

```
// 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);
    final Object b = NullDetector.nullPassed(b_dup);
    // method body
}
```

Listing 4: Illustration of the transformation for the "Null Method Parameter" (P) causality connection.

Proving that a code transformation, like Casper's, preserves semantics is a hard problem: proving that simple refactorings (such as those built into popular IDEs) are semantics-preserving has been proposed as a verification challenge [19]. Even one of the simplest refactoring — renaming variables — cannot be shown as semantics-preserving because of reflection and dynamic code evaluation [19]. Thus, we validate (Section 3.2.1), rather than verify, the correctness of Casper.

Comparison Operators Rule 1 and Rule 2 preserve the original behavior by rewriting expressions, to include a disjunction !(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. Rule 3 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)".

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 Rule 5. 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 replaces.

Emulating Null Dereferences When dereferencing a `null`, Java throws an exception object NullPointerException. When dereferencing a null ghost, the execution must also result in throwing the same exception. In Listing 3, a null ghost throws the exception CasperNullPointerException, which extends Java's exception NullPointerException. 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 the following cases: *a*) a method call on `null`; *b*) a field access on `null`; or *c*) an array access on `null`; or *d*) 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. Rule 6 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(x).f. Since version 5.0, Java supports autoboxing and unboxing to facilitate working with its primitive types. A primitive type's object wrapper may contain a `null`; if so, unboxing a null value triggers a null dereference error. For example, Integer a = null; int b = a, a + 3 or a ∗ 3 all throw NullPointerException.

2.5 Implementation

Casper's transformations can be done either fully at the source code level or fully at the binary code level. This is a trade-off in terms of engineering and quality of information. With respect to engineering, according to our experience, the transformations are easier to write and debug at the source code level. In Casper, we choose a mix of both.

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.

Source Code Transformations We perform our source code transformations using Spoon [16]. 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 generates new Java files that contain the program corresponding to the AST after application of the transformations of Figure 1.

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 3: Descriptive summary of the benchmark of null dereferences.

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 to override all methods to create null ghost classes. To overcome this protection at runtime, Casper uses its 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.

3 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:

*RQ*1: Does our approach provide the correct causality trace?
*RQ*2: Do the code transformations preserve the semantics of the application?
*RQ*3: Is the approach useful with respect to the

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

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.

3.1 Benchmark

We built a benchmark 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 and reproducing null dereferences is no exception. 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 benchmark in two ways. First, we tried to replicate results over a published benchmark [4] 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 or 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 benchmark is therefore conditioned on reproducibility. We do not, however, have any reason to believe that any bias that may exist in our benchmark would impact Casper general applicability.

Under these constraints, we want to assess how our approach compares to the closest related work [4]. Their benchmark dates back to 2007. In terms of bug reproduction several years later, this benchmark 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 are 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 benchmark.

We collected 7 other bugs. The collection methodology follows. 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 benchmark 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 keep only those which have been fixed and which are closed (our experimentation needs the patch). After filtering, 19 bug reports remain[6]. Sadly, on those 19 bug reports, 8 are not relevant for our experiment: 3 are too old 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 benchmark.

Consequently, the benchmark contains the 3 cases from [4] (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 benchmark.

Table 4 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 [4] (Mckoi, freemarker and jfreechart), we do not have access to the full test suites. Table 3 gives the main descriptive statistics. For instance, the bug in McKoi is an application of 48000+ lines of code spread over 275 classes.

This benchmark only contains real null dereference bugs and no artificial or toy bugs. To reas-

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

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

# Bug Id	Problem summary	Fix summary
McKoi	new JDBCDatabaseInterface with null param -> field -> deref	Not fixed (bug by [4])
Freemarker #107	circular initialization makes a field null in WrappingTemplateModel -> 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
COLL-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 4: Our benchmark of 14 real null dereference errors from large scale open-source projects; this benchmark is publicly available to facilitate replication.

sure the reader about cherry-picking, we have considered all null dereference bugs of the selected projects. We have not rejected a single null dereference that Casper fails to handle.

3.2 Methodology

3.2.1 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.

We require that the original program and the transformed program both pass and fail the same tests in the test suite (when it exists). 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 and logging method return. We filter out all calls to Casper's 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.

3.2.2 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 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 benchmark, 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).

3.3 Results

3.3.1 RQ1

In all the cases under study, manual debugging showed that the trace identified by our approach is correct and the root cause of the trace is the one responsible for the error. This result can be replicated since our benchmark and our prototype software are publicly available.

3.3.2 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 code transformations do not modify the behavior of the program under study and provide the actual causality relationships.

3.3.3 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 5 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 [4] (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 4).

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 [7], 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 [4] 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.

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

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

3.3.4 Detailed Analysis

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 final Java type from the SDK and our approach cannot modify them (see Section 4.1).

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 returns `NaN`, due to this NaN, the condition on line 7 fails, and the null value is returned (line 12). 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

# Bug Id	Fix location	Fix location in standard stack trace	Addressed by Bond *et al.* [4]	Fix location in Casper's causality trace	Causality Trace
McKoi	Not fixed	No	No	Yes	L-A-R-A-D
Freemarker #107	Not fixed	No	Yes	Yes	L-A-D
JFreeChart #687	FastScatterPlot 178	No	No	Yes	L-*P*-D
COLL-331	CollatingIterator 350	No	No	Yes	L-A-D
MATH-290	SimplexTableau 106/125/197	No	No	No	D
MATH-305	MathUtils 1624	No	No	No	L-R-A-R-D
MATH-1117	PolygonSet 230	No	No	No	L-A-R-A-D
7 simple cases		Yes	Yes	Yes	L-A-D (x6) L-A-U
Total		7 / 14	8/14	11/14	

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

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

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

Size of Traces We now discuss the size of traces encountered in our experiment. First, the one of size 3 and of kind L-A-D type. The 7 obvious cases (where 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, and this field is dereferenced latter. This kind of trace is pretty short, so one may think that this case is obvious. However, all these fields are initialized long before 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 (JFreeChart #687) where the null is stored in a variable then passed as argument in one or multiple methods. In this case, the null value is either returned by a method at least once, or passed as a parameter.

Third, there are three cases where the causality traces are composed of 5 causal links. For McKoi, this long trace enables to identify the root cause of the error, which is provided neither by the standard stack trace, nor by the related work [4]. For MATH-305 and MATH-1117, the trace indeed points to the root cause, but according to the ground truth of the manual patch, the fix location is at a different place.

Execution Time Casper is primarily meant as a debugging tool, when a developer has reproduced a null pointer exception on her machine and she wants to understand it. To this extent, the time requirements are those of a developer working on a task: it's

acceptable for the developer to wait for some minutes before getting the causality trace of a null dereference. There are two kinds of overhead due to using CASPER: 1) instrumenting the code is an additional step and compiling the instrumented code takes longer; 3) running the instrumented code takes longer because of the additional checks and data collection. In all the cases of our benchmark, the first phase (instrumentation plus compilation overhead) takes less than 30 seconds. In average for all cases, CASPER computes the causality trace of the failing input in 147 ms, with a maximum of 1s. To sum up, the instrumentation and execution time is acceptable for using CASPER as a debugging tool.

4 Discussion

4.1 Limitations

There are a number of technical limitations in our current prototype implementation: 1. when the root cause is in external library code that we cannot rewrite; 2. when a ghost class has to be created for a JDK class that is loaded; before the invocation of our specific class loader (hence our implementation of CASPER cannot provide causality traces for Java strings); 3. when the dereference happens when throwing a null value, locking null, switching on null (this requires more engineering effort). 4. when an array object is dereferenced because there is no way in Java to create a ghost class for arrays. This is not a conceptual issue, creating ghost arrays would be trivial in dynamic languages such as Python. 5. external libraries might modify fields and set them to null. This is not handled, yet would mean that encapsulation best practices would be violated. A meta-object protocol as present in dynamic languages would also overcome this technical limitation specific to Java.

4.2 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 links, 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.

4.3 Use in Production

As shown in Section 3.3.4, 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.

4.4 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 benchmark composition: does the benchmark reflect the complexity of null dereference errors in the field? To address this threat, we took care to design the methodology to build the benchmark. 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.

5 Related Work

One way to eradicate null dereference errors upfront is to use a programming language with no null value by default such as Cobra [1], or checkers of the absence of possible null dereferences for certain annotated types or variables [8, 5]. We note that allowing certain primitively typed variables to be nullable such as in C# has the opposite effects, it increases the risk of possible null dereferences.

There are several static techniques to find possible null dereference bugs. Hovemeyer et al. [12] use byte-code analysis to provide possible locations where null dereference may happen. Sinha et al. [20] 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 [21] devises an abstract interpretation dedicated to null dereferences. Ayewah and Pugh [2] 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 [6] 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 [13] 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 [25]. In this paper, he compares the memory graph from the execution of two versions of the 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 been further extended by Sumner and colleagues [22, 23]. Our problem statement is different, since those approaches take 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 [18]. 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. [17] find possible locations of null dereferences by running a genetic algorithm to exercise the software. If one is found, a test case that demonstrates 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. [24] 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 1 and they consider pointer arithmetic, which is irrelevant in our case (no pointer arithmetic in Java). Their transformations are at the level of x86 code using dynamic instrumentation, while we work on Java code (mostly 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. [4] 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 3), 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 [9] 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.

6 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] The Cobra Programming Language. http://cobra-language.com/, 2016.

[2] 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.

[3] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[4] 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.

[5] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and Using Pluggable Type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[6] 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.

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

[8] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. *ACM SIGPLAN Notices*, 38(11), 2003.

[9] 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.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12), 2004.

[12] 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.

[13] S. W. Kent. Dynamic error remediation: A case study with null pointer exceptions. Master's thesis, University of Texas, 2008.

[14] 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.

[15] 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.

[16] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 2015.

[17] 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.

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

[19] M. Schäfer, T. Ekman, and O. de Moor. Challenge proposal: Verification of refactorings. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 67–72, New York, NY, USA, 2008. ACM.

[20] 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.

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

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

[23] 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.

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

[25] 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.