Enhancing Field Tracking and Interprocedural Analysis to Find More Null Pointer Exceptions

Dongfang Xie, Bihuan Chen, Kaifeng Huang, Yu Wang, Linghao Pan, Zhicheng Chen, Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

***Abstract*—Null pointer dereference raises Null Pointer Exceptions (NPEs). There are two groups of approaches to detect NPEs. Type-based approaches carry out strict type-based null safety checking. They heavily rely on annotations, and thus produce many false positives. Dataflow-based approaches leverage static forward and/or backward dataflow analysis. They mostly have a limited capability in tracking fields and interprocedural analysis, and introduce false positives and false negatives.**

To address these drawbacks, we propose WHEELJACK to detect NPEs for Java. It does not rely on annotations, and hence can work effectively under a lack of annotations. It leverages our novel abstraction of nullness status to enhance field tracking, and our novel invocation analysis (capturing change to return value and side effect of an invocation) to enhance interprocedural analysis. Our evaluation on 28 Java projects has demonstrated that WHEELJACK can mostly outperform the four state-of-the-art NPE detectors in recall without sacrificing precision. 5 and 2 new NPEs have been confirmed and fixed by developers after we submit 8 issues.

***Index Terms*—null pointer exception, program analysis**

I. INTRODUCTION

Null pointer dereference (a.k.a. CWE-476 [20]) is a common and serious type of bug in all application domains from mobile apps to web systems. It is ranked 11th in the "2022 CWE Top 25 Most Dangerous Software Weaknesses list" [19]. A null pointer dereference occurs when the program dereferences a pointer or an object that is not initialized or is explicitly set to a null value, thereby a Null Pointer Exception (NPE) is raised. Hereafter, we use null pointer dereferences and NPEs interchangeably. NPEs can result in serious consequences such as undefined behaviors.

While there exist many studies to empirically evaluate the effectiveness of static bug detectors [10, 24, 26] and developers' perception about static bug detectors [6, 14], only some studies have been conducted to investigate NPE detectors [1, 28]. As classified by a recent study [28], there are two groups of approaches to detect NPEs, i.e., type-based approaches [2, 8, 22] and dataflow-based approaches [3, 4, 12, 17, 18, 21].

In particular, type-based approaches verify null safety with a type system by ensuring that a `@Nullable` variable is never assigned to a `@NonNull` variable and a `@Nullable` variable is never dereferenced. These approaches suffer from two main drawbacks. First, they [8, 22] carry out strict type-based null safety checking for field initialization, argument passing and return value, and consequently generate a large number of false positive warnings which overwhelm users. Second, they [2, 8, 22] heavily rely on annotations, and hence produce many false positives when users do not use or provide annotations.

Dataflow-based approaches leverage static forward and/or backward dataflow analysis to find NPEs. One difference from type-based approaches is that they can work without the need of user-provided annotations. These approaches usually have different favors over scalability and effectiveness. They mostly have a limited capability in tracking fields and interprocedural analysis; e.g., complicated structures are not supported, and side effects of invocations are not precisely tracked. Consequently, they introduce false positives and false negatives.

To address the drawbacks, we propose WHEELJACK, a novel approach for detecting NPEs for Java. WHEELJACK employs no type-checking policy, which enables it to work effectively under a lack of annotations. WHEELJACK introduces instance status and constant status to abstract the nullness status of a variable in a way similar to the reference type variable and primitive type variable in Java. For two variables that share one particular instance status, the change to the instance status can be reflected on both variables. This mechanism not only binds the nullness status of two variables together, but also tracks the nullness status of fields. It enables us to track field initialization in a different way and reduces the number of false positive warnings.

Apart from that, we enhance the way of handling invocation, which focuses on not only the return value but also the side effect (including the change to arguments and receiver object) of an invocation. This mechanism brings benefit in recalling NPEs related to an invocation and reducing false positives. In addition, our appropriate usage of call graph information assists WHEELJACK in tracking the nullness status of arguments passed among methods, which further decreases false positives.

We conduct large-scale experiments to evaluate WHEELJACK. To evaluate the recall of WHEELJACK, we compare it with four state-of-the-art tools, i.e., CFNULLNESS [22], NULLAWAY [2], INFER-ERADICATE [4] and SPOTBUGS [12], on 57 NPEs from the DEFECTS4J [15] and BUGSWARM [27] datasets. WHEELJACK achieves a recall of 22.8%, which is 8.8% higher than the best of the state-of-the-art. To evaluate the precision of WHEELJACK, we manually inspect 50 NPE warnings produced by each tool. WHEELJACK has a 8% lower precision than the best of the state-of-the-art, with many more NPE warnings generated. Further, we measure the time overhead of WHEELJACK, and observe an approximately linear growth with the increase of project size, which is acceptable. Moreover, we submit 8 NPE issues for 8 open-source projects, and 5 and 2 of them have been confirmed and fixed by developers.

In summary, this paper makes the following contributions.

- We propose WHEELJACK to detect NPEs, with a novel abstraction of nullness status and a novel invocation analysis to enhance field tracking and interprocedural analysis.

- We evaluate WHEELJACK on 28 projects to demonstrate its recall, precision and efficiency in detecting NPEs. 5 and 2 new NPEs have been confirmed and fixed.

II. RELATED WORK AND MOTIVATION

NPE detection approaches for Java can be basically classified into two groups, i.e., type-based approaches and dataflow-based approaches. We first introduce these approaches (Sec. II-A and II-B), then compare their capabilities with respect to seven program analysis properties (Sec II-C), and finally use motivating examples to demonstrate their limitations (Sec. II-D).

A. Type-Based NPE Detector

The `@NonNull` and `@Nullable` annotations are the corner stones of type-based NPE detectors. In the type system constructed by these annotations, `@NonNull T` is a subtype of `@Nullable T` for any Java type `T`. With this type system, type-based NPE detectors [2, 8, 22] ensure null safety via verifying two type rules: (1) any value of `@Nullable` type must not be assigned to a variable qualified by `@NonNull`; and (2) any value of `@Nullable` type should never be dereferenced.

Papi et al. [22] introduced a nullness checker CFNULLNESS, implemented with the Checker Framework [7], to detect NPEs by finding violations of the two type rules. The key components of CFNULLNESS includes: (1) an intraprocedural flow-sensitive type checker to verify the safety of pointer dereferences, (2) a strict field initialization checker to ensure every field is initialized legitimately, and (3) a checker to analyze iterations over possibly null collections and arrays.

Facebook developed a type checker ERADICATE [8] as part of the INFER static analysis suite [3, 4]. It checks `@Nullable` annotations via an intraprocedural flow-sensitive analysis to propagate nullability through assignments and calls. It reports errors for accesses to nullable values that could lead to NPEs.

Banerjee et al. [2] proposed a null safety checker NULLAWAY with low time overhead and low annotation burden, based on the Error Prone framework [9] and the Checker Framework [7]. To reduce time overhead, it performs an intraprocedural analysis over the AST of each source file in one single pass. To reduce annotation burden and false positives, it assumes that methods are pure and methods in unchecked codes can always handle null parameters and will always return a non-null value.

Type-based NPE detectors are built on the belief that users could provide enough annotations and use them properly. When encountering a lack of annotations, they assume that unannotated method parameters, return values and class fields are considered as `@NonNull` types, while local variables are treated as `@Nullable` types. This assumption can cause false positives related to argument passing, return value and field initialization.

B. Dataflow-Based NPE Detector

Dataflow-based NPE detectors [3, 4, 12, 17, 18, 21] rely on static dataflow analysis to find NPEs. Different from type-based NPE detectors, they can work without annotations. They mainly differ on the tradeoff between scalability and effectiveness.

Hovemeyer and Pugh proposed SPOTBUGS, the successor of FINDBUGS [11, 13], to detect various bugs (e.g., infinite loops and null pointer dereferences) by pattern matching and dataflow analysis [12]. Specifically, it uses forward dataflow analysis to find NPEs by identifying contradictory beliefs (e.g., a check of a variable against null indicates the belief that the variable may be null, but a dereference of that variable outside the scope of the check indicates the belief that the variable may not be null). In addition, it leverages backward dataflow analysis to compute parameters that are unconditionally dereferenced, and reports an NPE when a nullable argument is passed to such a parameter.

INFER is a static analyzer, which was first aimed at C code and developed by Monoidics [3] and later extended to Java code after the acquisition of Monoidics by Facebook [4]. It uses separation logic and bi-abduction analysis [5] to detect null pointer dereferences and resource and memory leaks. As introduced in Sec. II-A, ERADICATE is part of INFER, which can be enabled by adding the option `--eradicate` to the checkers mode.

Besides, to verify the safety of pointer dereferences, Loginov et al. [17] proposed a sound analysis, named SALSA, based on abstract interpretation. It first uses an intraprocedural dataflow analysis to prove the safety of certain dereferences, and for the remaining unverified dereferences, it then gradually expands the scope (i.e., the depth of call chains) of interprocedural analysis. Nanda and Sinha [21] introduced an interprocedural backward dataflow analysis, named XYLEM, to identify NPEs. While it is context-sensitive and path-sensitive to reduce false positives, it bounds the paths that are explored for a cost-accuracy tradeoff. Further, Romano et al. [23] used genetic algorithm to generate test input data in order to trigger the NPEs detected by XYLEM. Madhavan and Komondoor [18] developed a sound, interprocedural context-sensitive backward dataflow analysis to verify the safety of dereferences. Compared with SALSA and XYLEM, it explicitly models aliasing relationships, soundly handles recursion, and explores more and longer paths. However, all these above approaches [17, 18, 21, 23] are not publicly available.

Dataflow-based NPE detectors mostly have limited capability in tracking fields and interprocedural analysis; e.g., the side effects of invocations are not precisely tracked. The unsoundness of their capability results in false positives and false negatives.

C. Approach Capability Comparison

We compare the capabilities of those approaches in Sec. II-A and II-B with respect to seven program analysis properties that are used in a recent study on NPE detectors [28]. The differences from the previous study [28] are that (1) we include three more existing NPE detectors [17, 18, 21] and (2) we refine and confirm all approaches' capabilities by inspecting paper or documentation, and writing test programs if the approaches are publicly available. The properties are (1) intraprocedural, (2) interprocedural, (3) field-sensitive, (4) object-sensitive, (5) context-sensitive, (6) flow-sensitive, and (7) path-sensitive. Specifically, an intraprocedural analysis is performed within the scope of an individual method, and an interprocedural analysis is performed across the boundaries of individual methods. Field-sensitivity distinguishes different fields of the same object, whereas object-sensitivity distinguishes different "host" objects for the same field of a class. Context-sensitivity considers calling contexts of

TABLE I: Approach Capabilities (✓= complete capabilities, × = no capabilities, ○ = limited capabilities, – = N/A)

Category	Approach	Intraproc.	Interproc.	Field-Sens.	Object-Sens.	Context-Sens.	Flow-Sens.	Path-Sens.
Type	CFNULLNESS [22]	✓	○	○	×	×	✓	○
	NULLAWAY [2]	✓	○	○	×	×	✓	○
Combined	INFER-ERADICATE [3, 4, 8]	✓	○	○	✓	×	✓	○
Dataflow	SPOTBUGS [11, 12, 13]	✓	○	○	✓	×	✓	○
	SALSA [17]	✓	○	–	–	✓	✓	○
	XYLEM [21]	✓	○	–	–	✓	✓	○
	M&K [18]	✓	○	–	–	✓	✓	○
	WHEELJACK	✓	○	○	✓	○	✓	○

the same method, flow-sensitivity considers the order of statements, and path-sensitivity considers the feasibility of paths. As ERADICATE is part of INFER, we use INFER-ERADICATE to represent their combination, which can be considered as a representative approach of combining type and dataflow analysis.

We present the capability comparison in Table I. First, these approaches support intraprocedural analysis, but perform a limited interprocedural analysis. Type-based approaches emphasize on passing arguments and returning values legitimately according to the type rules in Sec. II-A. SPOTBUGS only ensures that a nullable argument would never be passed to an unconditionally-dereferenced unannotated parameter. While the other dataflow-based approaches conduct a stronger interprocedural analysis than SPOTBUGS, they often limit it to improve scalability.

Second, type-based approaches have limited field-sensitivity because they only focus on legitimate field initialization (i.e., any `@NonNull` field should be initialized at its declaration, in constructors or initialization blocks). Instead, dataflow-based approaches provide limited tracking of fields, and hence they can find NPEs caused by dereferences of uninitialized fields. Besides, type-based approaches pay more attention on type verification rather than object-sensitivity, whereas dataflow-based approaches are often able to distinguish different objects.

Third, all approaches are flow-sensitive to reduce false positives. Due to the concern of scalability, all approaches except for [17, 18, 21] are context-insensitive. To be practically useful, all approaches only provide limited path-sensitivity by handling null check conditions. An exception is that INFER-ERADICATE can also handle numerical conditions.

D. Motivating Examples

To illustrate the limitations of previous approaches, we identify six sources of unsoundness with respect to three properties (i.e., field-sensitive, object-sensitive and interprocedural) with motivating examples. Here we mainly focus on the three properties because the previous approaches differ more significantly in them than in the other four properties. Besides, we do not include SALSA [17], XYLEM [21] and M&K [18] as they are not publicly available for running our test programs. We report the sources of unsoundness for each approach in Table II.

1) Field Initialization: Initializing fields properly is believed to be a good way to prevent NPEs, and initialization checking is a crucial part of type-based approaches [2, 8, 22]. In the example in Fig. 1, fields `f1` and `f2` are not initialized in the constructor. `f1` is dereferenced in method `toString()`, and `f2` is dereferenced only when it is not null. Type-based approaches

```
1 public class FieldInitializationCase {
2     private Object f1 = null, f2 = null;
3     public FieldInitializationCase() {}
4     @Override
5     public String toString() {
6       return "{field1:"+ f1.toString()+",field2:" // NPE
7             +(f2==null? "N/A":f2.toString())+'}';
8     }
9 }
```

Fig. 1: Example of Field Initialization

```
1  public class ObjectAliasingCase {
2      public void entry() {
3          Inited a = new Inited();
4          Inited b = a;
5          a.field = null;
6          b.field.toString(); // NPE
7      }
8      public static class Inited {
9          public Object field = new Object();
10         public Inited() {}
11     }
12 }
```

Fig. 2: Example of Object Aliasing

CFNULLNESS, NULLAWAY and INFER-ERADICATE warn that `f1` and `f2` should be annotated `@Nullable` or initialized in the constructor. However, the warning for `f2` is a false positive because it would never cause an NPE. Differently, SPOTBUGS reports that only the dereference of uninitialized `f1` at line 6 is risky. Similarly, WHEELJACK tracks fields and thus reports the NPE for `f1` at line 6 while not emitting a warning for `f2`.

2) Object Aliasing: Effective analysis of object aliasing is essential in nearly all non-trivial program analyses [25]. In the example in Fig. 2, `b` is the alias of `a`, and `a.field` is initialized when `a` is created by the constructor at line 3 and then assigned null at line 5. As `field` is not annotated, it is considered as a `@NonNull` type in type-based approaches. Hence, CFNULLNESS, NULLAWAY and INFER-ERADICATE report a warning at line 5 because a null value is assigned to a `@NonNull` field. INFER-ERADICATE and SPOTBUGS correctly detect the NPE at line 6 caused by the dereference of `b.field`, which indicates their capability in handling aliases. WHEELJACK also supports aliasing and warns the NPE at line 6.

3) Complicated Structure: Field- and object-sensitivity can be challenged by complicated structures. Fig. 3 presents a recursive data structure `NotInited`. It contains `field` that is also an object of class `NotInited`. `obj1` is assigned to `obj2.field` at line 5. As `obj1.field`, which is also `obj2.field.field`, is not initialized, the dereference at line 6 triggers an NPE. CFNULLNESS, NULLAWAY and INFER-ERADICATE warn that `field` should be intialized or annotated `@Nullable`. Except

TABLE II: Sources of Unsoundness (✓= complete capabilities, × = no capabilities, ○ = limited capabilities)

Capability		CFNULLNESS	NULLAWAY	INFER-ERADICATE	SPOTBUGS	WHEELJACK
Field- and Object-Sensitive	Field Initialization	○	○	○	✓	✓
	Object Aliasing	×	×	✓	✓	✓
	Complicated Structure	×	×	×	×	✓
Interprocedural	Return Value	○	○	✓	×	✓
	Argument Passing	○	○	✓	○	✓
	Side-Effect of Invocation	×	×	○	×	✓

```
public class ComplicatedStructureCase {
    public void entry() {
        NotInited obj1 = new NotInited();
        NotInited obj2 = new NotInited();
        obj2.field = obj1;
        obj2.field.field.toString(); // NPE
    }
    public static class NotInited {
        public NotInited field = null;
        public NotInited() {}
    }
}
```

Fig. 3: Example of Complicated Structure

```
public class ReturnValueCase {
    public void entry() {
        Object obj1 = m1();
        obj1.toString(); // NPE
        Object obj2 = m2();
        if (obj2 != null) obj2.toString();
    }
    public Object m1() { return null; }
    public Object m2() { return null; }
}
```

Fig. 4: Example of Return Value

for that, no approach reports the NPE at line 6. Instead, WHEELJACK, by enhancing field tracking, detects the NPE at line 6.

4) Return Value: Analyzing return value is a crucial part of interprocedural analysis. In the example in Fig. 4, methods `m1` and `m2` return a null value, but only `m1`'s return value causes an NPE at line 4. Type-based approaches CFNULLNESS, NULLAWAY and INFER-ERADICATE report warnings at line 8 and 9 as they assume that without a `@Nullable` annotation, `m1` and `m2` should not return a null value. However, the warning for `m2` at line 9 is a false positive because it would not cause an NPE. Besides, INFER-ERADICATE also reports the NPE at line 4, but SPOTBUGS does not provide any NPE warning. WHEELJACK determines the return values of these invocations through our invocation analysis and only reports the NPE warning at line 4.

5) Argument Passing: Analyzing argument passing is also a crucial part of interprocedural analysis. In the example in Fig. 5, an NPE is triggered at line 13 because method `entry` passes a null value to parameter `p1` of method `m2` at line 4 and `p1` is directly dereferenced at line 13. The null value passed to `p2` and `p3` of `m2` does not cause an NPE because `p2` is not used and the dereference of `p3` is guarded by a null check. Besides, as a non-null value is passed to `m1` at line 3, it does not cause any NPE. CFNULLNESS, NULLAWAY and INFER-ERADICATE warn that the three null values should not be passed to `m2` at line 4 because the three parameters of `m2` are not annotated `@Nullable`. Besides, INFER-ERADICATE also reports the NPE at line 13. SPOTBUGS not only reports the NPE at line 13, but also falsely reports an NPE warning at line 10 because it is confused by the

```
public class ArgumentPassingCase {
    public void entry() {
        m1(new Object());
        m2(null, null, null);
    }
    private void m1(Object obj) {
        if (obj == null) {
            System.out.println();
        }
        obj.toString();
    }
    private void m2(Object p1, Object p2, Object p3) {
        p1.toString(); // NPE
        if (p3 == null) return;
        p3.toString();
    }
}
```

Fig. 5: Example of Argument Passing

null check at line 7. WHEELJACK leverages argument passing information properly and identifies the NPE at line 13.

6) Side Effect of Invocation: These approaches handle the side effect of invocation differently. CFNULLNESS marks those methods annotated `@Pure` as side-effect-free and deterministic [7]. NULLAWAY makes a purity assumption that all methods are side-effect-free and deterministic to reduce false positives [2]. SPOTBUGS assumes that any invocation can modify any field of any object passed to the invocation. INFER-ERADICATE tracks some side effects of methods. For example, if a method contains `this.field = null`, the side effect will be tracked at the invocation. In the example in Fig. 6, `obj.field` is not initialized at first, and thus an NPE is introduced at line 4. `obj.field` is then set to a non-null value by the invocation of `setField` at line 5. Then, `obj.field` is set to a null value by the invocation of `setField` at line 7, and hence an NPE is caused at line 8. CFNULLNESS, NULLAWAY and INFER-ERADICATE warn that `field` should be initialized or annotated `@Nullable`, and a null value should not be passed to `setField` at line 7. SPOTBUGS reports no NPE warning. WHEELJACK makes no purity assumption and is able to handle side effects by our invocation analysis, and thus it reports two NPE warnings at line 4 and 8.

III. METHODOLOGY

Based on the insights from Sec. II-C and II-D, we propose an automated approach, named WHEELJACK, to identify NPEs for Java programs. WHEELJACK leverages our novel abstraction of nullness status to track nullness information (Sec. III-A). Generally, WHEELJACK takes as inputs entry methods and jar files of a Java project, and returns a set of detected NPE warnings. If no entry method is provided by users, WHEELJACK uses all public and protected non-deprecated methods as the entry methods.

Given the entry methods, WHEELJACK uses Soot [29] to generate a call graph. We enable SPARK [16] in Soot for a more

```java
public class SideEffectCase {
    public void method1() {
        NotInited obj = new NotInited();
        obj.field.toString(); // NPE
        obj.setField(new Object());
        obj.field.toString(); // Safe
        obj.setField(null);
        obj.field.toString(); // NPE
    }
    public static class NotInited {
        public Object field = null;
        public NotInited() {}
        public void setField(Object field) {
            this.field = field;
        }
    }
}
```

Fig. 6: Example of Side Effect of Invocation

$$\begin{aligned} \mathit{Status} &::= \mathit{ConstantStatus} \mid \mathit{InstanceStatus} \\ \mathit{ConstantStatus} &::= \mathit{Null} \mid \mathit{NonNull} \mid \mathit{Unknown} \\ \mathit{InstanceStatus} &::= \mathit{NonNullInstance} \mid \mathit{UnknownInstance} \\ \mathit{NonNullInstance} &::= \{f : \mathit{Status} \mid f \ \mathit{is\ a\ field\ of\ an\ object}\} \end{aligned}$$

Fig. 7: Nullness Status Abstraction

precise call graph. Then, WHEELJACK performs interprocedural analysis to visit the methods in the call graph in a topological order and propagate calling context information from callers to their callees (Sec. III-C). During this procedure, each method is visited via performing intraprocedural analysis (Sec. III-B), which consists of a nullness analysis and a reachability analysis to generate nullness information, reachability information, and calling context information. The nullness information and reachability information are used to produce NPE warnings, while the calling context information is used in interprocedural analysis. To balance scalability and accuracy, we bound our analysis with two configurable parameters (Sec. III-D).

A. Nullness Status Abstraction

Fig. 7 presents our abstraction of nullness status Status. Inspired by the philosophy of primitive variable and reference variable in Java, we distinguish Status between constant status $\mathit{ConstantStatus}$ and instance status $\mathit{InstanceStatus}$. In a typical nullness analysis, nullness status has three types, `Null`, `NonNull` or `Unknown`. Hence, we model $\mathit{ConstantStatus}$ as Null, $\mathit{NonNull}$ or $\mathit{Unknown}$, which can be seen as constants of the three corresponding types. Like primitive variables, if two variables have the same constant status, they do not share the same nullness status, i.e., the nullness status change of one variable will not change the nullness status of the other variable. Besides, we model $\mathit{InstanceStatus}$ as $\mathit{NonNullInstance}$ or $\mathit{UnknownInstance}$, which can be considered as instances of the `NonNull` and `Unknown` type. Like reference variables, if two variables have the same instance status, they share the same nullness status, i.e., the nullness status change of one variable (if the change is not caused by an assignment to the variable) will cause the same nullness status change of the other variable. However, if the change is caused by an assignment, the nullness status of the other variable will not be affected. Naturally there is no $\mathit{NullInstance}$ because null by itself is a constant. We define $\mathit{NonNullInstance}$ as a map whose key is a field and whose value is the nullness status of the field. We abstract nullness status in such a way to achieve field-sensitivity and object-sensitivity for our NPE detection.

B. Intraprocedural Analysis

In general, our intraprocedural analysis is forward and conducted on the control flow graph of a method. It consists of two analyses: *Nullness Analysis* and *Reachability Analysis*.

1) Preliminary Settings: As will be introduced in Sec. III-C, before our intraprocedural analysis is performed on a method, our interprocedural analysis has visited all callers of the method, and thus has already collected and merged all the calling context information for the method. In other words, it has obtained the nullness status of the method's parameters and `this` variable. If the method has no caller, we assign $\mathit{NonNullInstance}$ to the status of `this`, and conservatively assume $\mathit{Unknown}$ to the status of the parameters and `this`'s fields.

2) Nullness Analysis: Our nullness analysis aims to track the nullness information of variables in the method. We first introduce the transfer function. For each statement of the method, we maintain the nullness status that flows in the statement (i.e., in-status) and the nullness status that flows out of the statement (i.e., out-status) after applying the transfer functions in Table III. Therefore, our approach achieves flow-sensitivity.

i) Assignment. For an assignment statement `x = y`, if the in-status of `y` is Null, the out-status of `x` becomes Null. If the in-status of `y` is $\mathit{NonNull}$ or $\mathit{Unknown}$, the out-status of `x` and `y` becomes a $\mathit{NonNullInstance}$ or $\mathit{UnknownInstance}$. If the in-status of `y` is a $\mathit{NonNullInstance}$ or $\mathit{UnknownInstance}$, this status is assigned to the out-status of `x`. In the latter two cases, we create a binding between `x` and `y` by making them sharing an instance status. As a result, any change to the nullness status of the fields in one variable (e.g., `x`) is simultaneously reflected in the other variable (e.g., `y`). Only when another assignment to `x` or `y` happens, this binding no longer holds.

This transfer function, together with our abstraction of nullness status, empowers WHEELJACK to track nullness information passed among variables and fields as well as the binding among them. It is also worth mentioning that the key difference of our approach from points-to analysis [16] is that even when the in-status of `y` is unknown, we still create an instance status to bind `x` and `y`. In essence, we focus on the status of a variable rather than a memory object.

ii) Field Load and Store. WHEELJACK is naturally enabled to track both field load and field store, using our abstraction of $\mathit{NonNullInstance}$. For a load statement `x = obj.f`, if the in-status of `obj` is a constant status, its out-status becomes a $\mathit{NonNullInstance}$. Notice that if the in-status of `obj` is Null, an NPE warning will be reported, but its out-status still becomes a $\mathit{NonNullInstance}$ because our analysis continues to successor statements whose execution implies no exception for the load statement. In this way, we can avoid generating redundant NPE warnings due to the same root cause (i.e., the dereference of the null `obj` in successor statements). Besides, the out-status of the field `obj.f` and the out-status of `x` share an instance

TABLE III: Transfer Functions ($status_0$, $status_1$, $status_2$, $status_3$, $status_4$, $status_5$ $\in$ $Status$, $status_1$, $status_2$, $status_3$ $\neq$ $Null$, $status_2$, $status_3$ $\notin$ $NonNullInstance$; $status_{c1}$ $\in$ $ConstantStatus$; $status_{i1}$ $\in$ $InstanceStatus$; $status_{ni1}$, $status_{ni2}$, $status_{ni3}$ $\in$ $NonNullInstance$; $status'_{ni2}$ and $status'_{ni3}$ denote that the nullness status of fields in $status_{ni2}$ and $status_{ni3}$ are updated; $status_{ui1}$ $\in$ $UnknownInstance$)

Kind	Statement	In-Status	Out-Status
Assignment	x = y	x : $status_0$, y : $Null$ x : $status_0$, y : $status_1$	x : $Null$, y : $Null$ x: $status_{i1}$, y : $status_{i1}$
Field Load	x = obj.f	x : $status_0$, obj : $status_{c1}$ x : $status_0$, obj : {f : $status_1$}	x : $status_{i1}$, obj : {f : $status_{i1}$} x : $status_{i1}$, obj : {f : $status_{i1}$}
Field Store	obj.f = x	x : $status_1$, obj : $status_{c1}$ x : $status_1$, obj : {f : $status_0$}	x : $status_{i1}$, obj : {f : $status_{i1}$} x : $status_{i1}$, obj : {f : $status_{i1}$}
New Assignment	x = new Type()	x : $status_0$	x : $status_{i1}$
Invocation	x = obj.m(arg)	x : $status_0$, obj : $status_2$, arg : $status_3$ x : $status_0$, obj : $status_{ni2}$, arg : $status_{ni3}$	x : $status_4$, obj : $status_{ni1}$, arg : $status_5$ x : $status_4$, obj : $status'_{ni2}$, arg : $status'_{ni3}$
Null Check	if (x == null)	x : $status_{ui1}$	x : $Null$ (True Branch) x : $status_{ni1}$ (False Branch)
instanceof Check	if (x instanceof Type)	x : $status_{ui1}$	x : $status_{ni1}$ (True Branch) x : $status_{ui1}$ (False Branch)
Array Assignment	x = a[i]	x : $status_0$, a : $status_2$	x : $Unknown$, a : $NonNull$

status in the same way in **Assignment**. The same principle is applied for a store statement, and hence we omit the details.

iii) New Assignment. For a new assignment statement `x = new Type()`, no matter what the in-status of `x` is, we set the out-status of `x` to a $NonNullInstance$.

iv) Invocation. Our invocation analysis is context-sensitive, which aims to analyze the return value and side-effect of an invocation. For an invocation statement `x = obj.m(arg)`, the return value of `m` is assigned to `x`, which changes the nullness status of `x`. The side-effect includes the nullness status change of the arguments (e.g., `arg`) passed to the callee (e.g., `m`) as well as the receiver (e.g., `obj`) on which the callee is called.

As an invocation can contain other invocations, we conduct our invocation analysis iteratively until reaching an invocation depth for the sake of scalability (Sec. III-D). In that sense, our intraprocedural analysis also involves limited interprocedural analysis. To start an invocation analysis, we pass the calling context information to the callee. The transfer function in our invocation analysis is the same as in our intraprocedural analysis. The difference of invocation analysis from intraprocedural analysis is that we conclude the status of return value and the status change of arguments and receiver only when there is a reachable `return` statement (Sec. III-B3). When there is no reachable `return` statement, this invocation is marked as *dead*.

When our invocation analysis finishes, we apply the nullness status change to arguments and receiver and assign the nullness status of the return value as long as the invocation is not dead. Specifically, for the arguments and receiver, if their in-status is not a $NonNullInstance$ and not $Null$, their out-status is changed to the ones obtained from our invocation analysis. Even if the in-status of receiver is $Null$, the out-status of receiver is a $NonNullInstance$ due to the same reason in **Field Load and Store**. If their in-status is already a $NonNullInstance$, we update its fields' status. We currently do not support invocations to third-party methods whose code is not available.

v) Null and instanceof Check. Null check is a common mechanism to implement defensive programming and prevent NPEs. For a null check statement `if (x == null)`, if the in-status of `x` is an $UnknownInstance$, its out-status is set to $Null$ in the true branch, and a $NonNullInstance$ in the false branch. For any variable sharing the same $UnknownInstance$ with `x`, its out-status is also changed to $Null$ in the true branch, and the same $NonNullInstance$ in the false branch. Besides, if the in-status of `x` is $NonNull$ or a $NonNullInstance$, the true branch is marked as unreachable (Sec. III-B3), and its out-status is not changed in the false branch. If the in-status of `x` is $Null$, the false branch is marked as unreachable (Sec. III-B3), and its out-status is not changed in the true branch. Note that WHEELJACK is also able to evaluate negation of null checks.

Besides, instanceof check is used to cast an object safely. For an instanceof check statement `if (x instanceof Type)`, if the in-status of `x` is an $UnknownInstance$, its out-status is set to a $NonNullInstance$ in the true branch, and is not changed in the false branch because we do not store type information. Any variable that shares the same $UnknownInstance$ with `x` still shares the status with `x` in both branches. Besides, if the in-status of `x` is a $NonNullInstance$ or $NonNull$, its out-status is not changed. If the in-status of `x` is $Null$, the true branch is marked as unreachable (Sec. III-B3), and its out-status is not changed in the false branch.

Based on our null check and instanceof check analysis, our approach achieves limited path-sensitivity.

vi) Array Assignment. An array is also a kind of object, but our way to handle it is different from other reference type objects. For an array assignment statement `x = a[i]`, we only infer that `a` is $NonNull$ and assume `x` to be $Unknown$. Since array is frequently used with loop iterations and mutable indexes, it could be costly to track array element status accurately.

In summary, there are four situations to generate $Null$ status. (1) *NullFromAssign*: the assignment statement, e.g., `a = null`, makes the status of `a` become $Null$. (2) *NullFromReturn*: the statement returning a null value, e.g., `return null`, makes the

status of the return value become $Null$. (3) *NullFromAnnotation*: when invocation analysis cannot determine the status of return value and arguments and an explicit `@Nullable` is declared, the annotated return value or argument is assigned $Null$. (4) *NullFromNullCheck*: a variable can be inferred as $Null$ by a null check. These four situations have a decreasing confidence of being $Null$. As these four situations of how and where $Null$ is generated can be helpful for developers to debug NPEs, we record them in our null analysis while also recording the class name and line number that generates $Null$.

Apart from how and where $Null$ is generated, we also record where $Null$ is passed and returned (i.e., the class name and line number that $Null$ is passed to or returned from a method) until it becomes $NonNull$ or $NonNullInstance$ or it is dereferenced. Such a trace can help developers to debug NPEs.

Then, we introduce how control flows are joined. Specifically, when two $Null$ statuses are merged, the one with a high confidence of being $Null$ is preserved preferentially. For unknown statuses including $Unknown$ and $UnknownIstance$, they are designed to be carriers for $Null$. Concerning space issue, these unknown statuses carry one or no $Null$ status. In detail, an unknown status that carries no $Null$ status starts to carry a $Null$ status when it is merged with a $Null$ status. When two unknown statuses that both carry a $Null$ status are merged, the two $Null$ statuses are merged, and an unknown status with the merged $Null$ status is generated. When a $Null$ status is merged with an unknown status that carries a $Null$ status, an unknown status is generated with a $Null$ status merged by the two $Null$ statuses. Moreover, for a non-null status including $NonNull$ and $NonNullInstance$, when it is merged with a $Null$ status, an unknown status that carries the $Null$ status is generated. When it is merged with an unknown status, the unknown status is preserved. When it is merged with a non-null status, a non-null status is generated through merging the status of the fields if the non-null status is a $NonNullInstance$.

3) Reachability Analysis: Our reachability analysis and nullness analysis interact and interweave with each other. Reachability analysis focuses on the reachability of a statement in order to ensure that data flow in our nullness analysis is not affected by an unreachable statement. Reachability analysis leverages the nullness information from nullness analysis to identify unreachable statements. For a null check, if a variable in the check is already known as null or non-null, one of the two branches will be marked as unreachable. Similarly, for an instanceof check, if a variable in the check has a null status, one of the two branches will be marked as unreachable. Besides, for dead invocations identified in our invocation analysis, as they will never return unexceptionally, the successor statements of the dead invocations will be marked as unreachable.

One statement can have several predecessor statements, and as long as one predecessor marks it as reachable, it is assumed reachable; otherwise, it is unreachable. In our nullness analysis, the out-status will be marked as *dead* if the statement is a dead invocation or is unreachable. When a dead status is merged with another status, it makes no change (i.e., the another status is preserved). This protects the data flow from being polluted by unreachable statements. In addition, those invocations in unreachable statements are excluded in both intraprocedural analysis and interprocedural analysis.

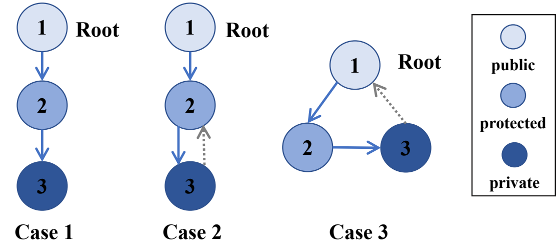


Fig. 8: Cases of Call Graph

4) NPE Warning Generation: With the nullness information and reachability information, we detect possible risks related to variable dereferences. In particular, when a variable with a $Null$ status or an unknown status that carries a $Null$ status is dereferenced, we generate an NPE warning if the statement is reachable. Other unknown status is less risky because no evidence indicates that it could be null.

The warning information contains the dereference location (i.e., the class, method and line number), the dereferenced variable, and the method or field it is dereferenced for. Moreover, the warning information also includes how and where the $Null$ status is generated, and where it is passed and returned.

C. Interprocedural Analysis

Our interprocedural analysis aims to visit the methods in the call graph in such an order that all the calling context information is collected when visiting each method. To this end, we visit the methods in the call graph in a topological order according to their invocation relationship. However, when the call graph contains loops, this strategy fails. Therefore, we first break the loops before we apply topological visit.

To better illustrate how we break loops, we show three cases of call graph in Fig. 8, where each node denotes a method and each edge denotes an invocation relationship from a caller to its callee. We define those methods without any caller as roots. For example, method 1 in case 1 and 2 in Fig. 8 is the root method. Then, for such cases, we conduct a depth-first visit from root methods, and break a loop at the first method that enters the loop (which is also defined as the entry method of the loop). For example, method 2 in case 2 is the entry method of a loop. For breaking a loop, we remove the invocation to the entry method of the loop. For example, the invocation from method 3 to method 2 in case 2 is removed for breaking the loop. For a cyclic call graph without any root, e.g, case 3 in Fig. 8, we select a public method (or a protected method when there is no public method) as the root, and break a loop with the above strategy.

We conservatively set the calling context of the entry method of loops in our intraprocedural analysis to unknown; i.e., the nullness status of its parameters and `this` variable is assumed $Unknown$. This is because the number of loops are limited and setting the calling context to unknown could only cause a limited number of false negatives. Compared to a fixed-point strategy that stops when the calling context remains stable, our strategy is more practical and less expensive.

After breaking loops, we visit each method in the call graph in a topological order. For each method, we first merge all the calling context information from its callers by the same strategy in joining control flows (Sec. III-B). In other words, we collect the nullness status of each parameter by merging the nullness status of the argument passed from all callers, and the nullness status of `this` variable by merging the nullness status of the receiver of all callers. Then, we run our intraprocedural analysis to obtain its nullness information, reachability information, and calling context information for its callees. The nullness information and reachability information are used to generate NPE warnings (Sec. III-B4). The calling context information is passed to its callees for their intraprocedural analysis and for the invocation analysis in the current intraprocedural analysis.

D. Balancing Scalability and Accuracy

To balance scalability and accuracy, we bound our analysis with two configurable parameters.

1) Field Depth: $NonNullInstance$ is designed to record the nullness status of each field of an object (Sec. III-A), and each field can be an object and its nullness status can be a $NonNullInstance$ that records the nullness status of fields as well. To avoid large memory consumption, we introduce a configurable parameter, *field depth*, to bound the depth of tracking fields. When reaching the field depth, we set the nullness status of a field to $Unknown$, which may hurt the accuracy. In that sense, our approach achieves limited field-sensitivity.

2) Invocation Depth: Our invocation analysis in nullness analysis (Sec. III-B2) leverages an depth-first strategy to analyze each invocation it encounters. As the invocation chain can be deep, we introduce a configurable parameter, *invocation depth*, to bound the depth of our invocation analysis for the sake of scalability. When reaching the invocation depth, we do not explore any further and conservatively assume that the invocation is not dead, the nullness status of return value and fields of the receiver is $Unknown$, and the nullness status of arguments' fields is $Unknown$ if the nullness status of arguments is a $NonNullInstance$, which may sacrifice the accuracy. In that sense, our approach achieves limited context-sensitivity.

IV. Evaluation

We have implemented and tested a prototype of WHEELJACK in 8.17K lines of non-test Java code and 6.25K lines of test Java code. We have also released all the source code at our website https://wheeljack23.github.io/ with our experimental data.

A. Evaluation Setup

To evaluate the effectiveness and efficiency of WHEELJACK, we designed our evaluation to answer four research questions.

- **RQ1 Recall Evaluation:** How is the recall of WHEELJACK in detecting NPEs, compared to the state-of-the-art tools?
- **RQ2 Precision Evaluation:** How is the precision of WHEELJACK, compared to the state-of-the-art tools?
- **RQ3 Efficiency Evaluation:** How is the time overhead of WHEELJACK in detecting NPEs for each project?
- **RQ4 Sensitivity Analysis:** How is the sensitivity of each parameter to the effectiveness and efficiency of WHEELJACK?

TABLE IV: Tool Recall ("Avg Warn." denotes the average number of NPE warnings produced by a tool for each project; "Found" denotes the number of benchmark NPEs found by a tool; "Recall" denotes the recall of a tool; "Unique" denotes the benchmark NPEs that could only be found by a tool)

Category	Tool	Avg Warn.	Found	Recall	Unique
Type	CFNULLNESS	1,299	7	12.3%	2
	NULLAWAY	86	2	3.5%	1
Combined	INFER-ERADICATE	1,354	8	14.0%	2
Dataflow	SPOTBUGSHT	4	3	5.3%	0
	SPOTBUGSLT	38	7	12.3%	0
	WHEELJACK	66	13	22.8%	8

State-of-the-Art Tools. For **RQ1** and **RQ2**, we selected four state-of-the-art publicly available tools, i.e., CFNULLNESS [22], NULLAWAY [2], INFER-ERADICATE [4] and SPOTBUGS [12], which are already introduced in Sec. II. We configured the tools by following the recent empirical study on NPE detection [28]. Specifically, SPOTBUGS was configured in two settings, i.e., high and low confidence threshold setting, respectively denoted as SPOTBUGSHT and SPOTBUGSLT.

Data Set. For **RQ1**, we started with the 102 benchmark NPEs across 42 projects from the recent empirical study on NPE detection [28]. However, we further excluded 45 NPEs for three reasons: 19 NPEs cannot be reproduced when running project tests; 17 NPEs are not caused by production code but by test code; and 9 NPEs are from five projects that cannot be built successfully. We collected a data set of 57 benchmark NPEs across 28 projects. Among these NPEs, 23 are from the DEFECTS4J dataset [15] and 34 are from the BUGSWARM dataset [27]. For **RQ2**, **RQ3** and **RQ4**, we also used these 28 projects.

B. Recall Evaluation (RQ1)

We ran tests of each project to collect the stack traces of the 57 benchmark NPEs. As the recent study [28] provided detailed NPE warning reports of the four state-of-the-art tools, we directly reused their warning reports for the 28 projects. Then, we manually investigated the stack traces of benchmark NPEs and the warning reports of each tool to decide whether benchmark NPEs could be found by each tool. Table IV reports the recall of each tool as well as the average number of NPE warnings generated by each tool for each project.

WHEELJACK generates 66 NPE warnings per project on average, and successfully detects 13 of the 57 benchmark NPEs, achieving the highest recall of 22.8%. However, CFNULLNESS and INFER-ERADICATE generate 18 times more NPE warnings than WHEELJACK, but only achieve a recall of up to 14.0%. In fact, CFNULLNESS and INFER-ERADICATE report a considerable amount of NPE warnings related to passing a nullable variable to an unannotated parameter, or an unannotated method that returns a null value. Thus, facing a lack of annotations, they could overwhelm users and have a poor usability. NULLAWAY holds a different assumption, and generates much fewer NPE warnings than CFNULLNESS and INFER-ERADICATE, but has

TABLE V: Tool Precision ("Avg Warn." denotes the average number of NPE warnings produced by a tool for each project; "Sampled" denotes the number of NPE warnings randomly sampled for our manual investigation; "precision" denotes the sampled precision of a tool)

Category	Tool	Avg Warn.	Sampled	Precision
Type	CFNULLNESS	1,299	50	18%
	NULLAWAY	86	50	32%
Combined	INFER-ERADICATE	1,354	50	8%
Dataflow	SPOTBUGSHT	4	50	60%
	SPOTBUGSLT	38	50	46%
	WHEELJACK	66	50	52%

the lowest recall of 3.5%. SPOTBUGSHT and SPOTBUGSLT generate the fewest NPE warnings, but achieve a recall of 5.3% and 12.3% which are both lower than WHEELJACK.

In addition, we investigate the overlap among the benchmark NPEs found by different tools. In total, these tools find 22 of the 57 benchmark NPEs, and most of the tools can find unique benchmark NPEs that cannot be found by other tools. Specifically, WHEELJACK finds 8 unique benchmark NPEs, owing to our enhanced field tracking and invocation analysis capability. Besides, WHEELJACK can respectively find 3 of the 7, 0 of the 2, 2 of the 8, 2 of the 3, and 4 of the 7 benchmark NPEs found by CFNULLNESS, NULLAWAY, INFER-ERADICATE, SPOTBUGSHT, and SPOTBUGSLT. These results indicate that these tools have complementary capabilities with each other.

Summary. WHEELJACK outperforms the best of the state-of-the-art tools in recalling NPEs by 8.8%, while generating a moderate amount of NPE warnings on average.

C. Precision Evaluation (RQ2)

To measure the precision of each tool, we randomly sampled for each tool 50 NPE warnings from all generated NPE warnings across all projects, and manually investigated each NPE warning by digging deep into the code to determine whether it is indeed an NPE. We report the precision results in Table V.

WHEELJACK achieves the second highest precision of 52% with an acceptable average number of NPE warnings. For tools that use type analysis, CFNULLNESS and INFER-ERADICATE achieve the lowest precision but generate the largest amount of NPE warnings. As a result, the majority of their NPE warnings are false positives. NULLAWAY has the highest precision among type-based tools with the fewest NPE warnings generated, but its precision is still 20% lower than WHEELJACK. The precision of SPOTBUGSHT is 60%, which is the highest. However, it only generates a very small number of NPE warnings, and thus it may miss NPEs (as revealed by the low recall in Sec. IV-B).

We summarized three reasons of imprecision of the tools that use type analysis. First, many unannotated fields that are not initialized in the constructor could be initialized latter or used within a defensive null check. However, type-based tools would report the unannotated fields. Second, an invocation that returns a null value is usually followed by a null check before dereference or the return value is not dereferenced, which will not cause any NPE consequently. If the invoked method is not annotated, however, type-based tools would report it. Third,

```
1 public void m() {
2     if (getField() != null) {
3         getField().toString();
4     }
5 }
6
7 public Object getField() {
8     return field;
9 }
```

Fig. 9: Example of Getter

```
1 public void m(Object arg) {
2     requireNonNull(arg);
3     arg.toString();
4 }
5
6 public static void requireNonNull(Object arg) {
7     if (arg == null) throw new RuntimeException();
8 }
```

Fig. 10: Example of Argument after Invocation

```
1  public void m() {
2      Object obj = null;
3      // getOrientation only returns two kinds of values
4      Orientation ori = getOrientation();
5      if (ori == HORIZONTAL) {
6          obj = new Object();
7      }
8      else if (ori == VERTICAL) {
9          obj = new Object();
10     }
11     obj.toString();
12 }
```

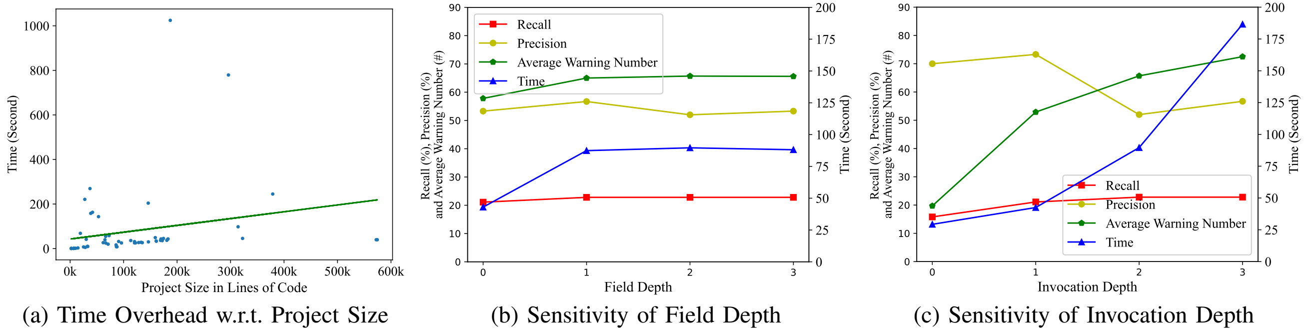
Fig. 11: Example of Limited Kinds of Values

when a nullable argument is passed to an unannotated parameter of a method, type-based tools would report an NPE warning. However, the nullable argument could be used safely inside the method, e.g., within a null check.

SPOTBUGS's imprecision is mainly caused by its limited capability in analyzing invocations. It is a quite common practice to use getters to access an object's fields. For example, at line 3 of Fig. 9, the return value of `getField` would not be null because of the null check at line 2. SPOTBUGS would report the dereference as risky, but WHEELJACK could figure out that the variable `field` is non-null after the null check. Another typical case is that, after an invocation, an argument could become non-null. For example, at line 3 of Fig. 10, `arg` is not null because of the invocation of `requireNonNull` at line 2. SPOTBUGS could not leverage the information brought by the invocation, whereas WHEELJACK could distinguish the variable `arg` at line 3 as non-null. Besides, SPOTBUGS also introduces imprecision in argument passing as type-based tools do, and it does not have the capability to analyze instanceof checks.

WHEELJACK could misjudge a dereference as risky in two major situations. First, when a variable has limited kinds of values, but WHEELJACK does not know. For example, at line 11 of Fig. 11, the variable `obj` is indeed initialized because `ori` only has two kinds of values, but WHEELJACK would falsely report an NPE warning. Second, conditional checks are not fully supported in WHEELJACK, which leads to a limited support in path-sensitivity, and causes wrong NPE warnings.

Furthermore, we sampled 20 NPE warnings for submitting issues. However, 6 of them have already been fixed in the latest



(a) Time Overhead w.r.t. Project Size (b) Sensitivity of Field Depth (c) Sensitivity of Invocation Depth

Fig. 12: Results of Time Overhead and Parameter Sensitivity Analysis

project version, and 6 of them have been deleted or deprecated in the latest project version. Therefore, we submitted 8 NPE issues for 8 open-source projects, and 5 and 2 of them have been confirmed and fixed by developers. The other 3 issues are still waiting for confirmation. These results demonstrate the usefulness of WHEELJACK in practice.

Summary. WHEELJACK achieves a 8% lower precision in detecting NPEs than the best of the state-of-the-art tools, while generating much more NPE warnings.

D. Efficiency Evaluation (RQ3)

We measured the lines of Java code of each project as well as the time spent for each project by running WHEELJACK. Notice that due to the way that benchmark NPEs were constructed [28], some projects had multiple versions, and we had 57 project versions for 28 projects. The result is shown in Fig. 12a. Overall, we observe an approximately linear growth of time with the increase of project size. We see two outliers that are caused by complicated data structures or invocation relationships in two project versions. Numerically, WHEELJACK takes less than 270 seconds for 55 of the 57 project versions with an average of 49 seconds. Besides, WHEELJACK takes 40 seconds for the largest project version which has 574K lines of Java code.

Summary. The time overhead of WHEELJACK has an approximately linear growth with the increase of project size. It scales to large projects with 574K lines of Java code.

E. Sensitivity Analysis (RQ4)

Two parameters, i.e., field depth and invocation depth, are configurable in WHEELJACK (Sec. III-D). They are set to 2 and 2 by default, which is used for **RQ1**, **RQ2** and **RQ3**. To evaluate their sensitivity to the effectiveness and efficiency of WHEELJACK, we re-configured one parameter and fixed the other one, and re-ran WHEELJACK. Field depth and invocation depth were configured from 0 to 3 by a step of 1. For each configuration, we randomly sampled 30 NPE warnings to measure precision.

1) Field Depth: Fig. 12b presents the result of field depth. As the field depth increased from 0 to 1, the nullness status of fields became tracked. Hence, the average number of NPE warnings greatly increased by 12.5%. Meanwhile, both precision and recall had a slight improvement, but the time overhead was nearly doubled. As the field depth increased from 1 to 3, all the four metrics were relatively stable, potentially because most projects' data structures are not very complicated. Therefore, we suggest a positive value for the field depth, and use 2 by default.

2) Invocation Depth: Fig. 12c shows the result of invocation depth. When the invocation depth was set to 0, WHEELJACK did not conduct invocation analysis, and conservatively handled the return value, receiver and arguments (Sec. III-D2). Hence, the average number of NPE warnings was small, but the precision was high. As the invocation depth increased, the average number of NPE warnings first increased fast and then grew slowly. At the same time, the time overhead grew rapidly (e.g., from 89.6 seconds at depth 2 to 186.6 seconds at depth 3). The recall increased by 5.3% from depth 0 to 1, then by 1.7% from depth 1 to 2 and remained stable from depth 2 to 3. The recall increased because invocation analysis assisted in detecting more NPEs related with invocations. The precision suffered a decrease from depth 1 to 2 because more NPE warnings were generated and imprecision accumulated as the depth increased. Therefore, we suggest 1 or 2 for the invocation depth, and use 2 by default.

Summary. Field depth and invocation depth can significantly affect the number of NPE warnings. A configuration with a reasonable tradeoff between scalability and accuracy can be setting 2 to both field depth and invocation depth.

F. Threats to Validity

One major threat to our evaluation is the size of benchmark NPEs used in our recall evaluation. We believe they are good representatives of NPEs as they are from a diverse set of 28 projects from two popular bug datasets. Another main threat is our manual investigation to determine the recall and precision of each tool in **RQ1**, **RQ2** and **RQ4**. To mitigate the threat, four of the authors conducted an independent analysis, while a fifth author was involved to resolve disagreement.

V. CONCLUSIONS

We have proposed WHEELJACK, a practical tool to detect NPEs for Java. WHEELJACK provides a new perspective of nullness status abstraction as well as a new way for handling invocations in order to reduce false positives and false negatives. Our evaluation on 28 Java projects has demonstrated the effectiveness and efficiency of WHEELJACK. We have released the source code of WHEELJACK and our experimental data at our website https://wheeljack23.github.io/.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114). Bihuan Chen is the corresponding author of this work.

REFERENCES

[1] N. Ayewah and W. Pugh, "Null dereference analysis in practice," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2010, p. 65–72.

[2] S. Banerjee, L. Clapp, and M. Sridharan, "Nullaway: Practical type-based null safety for java," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 740–750.

[3] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *Proceedings of the NASA Formal Methods Symposium*, 2011, pp. 459–465.

[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*, 2015, pp. 3–11.

[5] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 289–300.

[6] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 332–343.

[7] 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*, 2011, pp. 681–690.

[8] Facebook, "Infer: Eradicate," https://fbinfer.com/docs/next/checker-eradicate/, 2022, accessed: 2022-05-05.

[9] Google, "Error Prone," http://errorprone.info/, 2022, accessed: 2022-04-27.

[10] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018, pp. 317–328.

[11] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[12] ——, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 9–14.

[13] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005, pp. 13–19.

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 672–681.

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

[16] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, 2003, p. 153–169.

[17] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 213–224.

[18] R. Madhavan and R. Komondoor, "Null dereference verification via over-approximated weakest pre-conditions analysis," *ACM Sigplan Notices*, vol. 46, no. 10, pp. 1033–1052, 2011.

[19] MITRE, "2022 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022, accessed: 2022-08-12.

[20] ——, "CWE-476: NULL Pointer Dereference," https://cwe.mitre.org/data/definitions/476.html, 2022, accessed: 2022-08-12.

[21] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *Proceedings of the IEEE 31st International Conference on Software Engineering*, 2009, pp. 133–143.

[22] M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst, "Practical pluggable types for java," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2008, pp. 201–212.

[23] D. Romano, M. Di Penta, and G. Antoniol, "An approach for search based testing of null pointer exceptions," in *Proceedings of the Fourth IEEE international conference on software testing, verification and validation*, 2011, pp. 160–169.

[24] N. Rutar, C. Almazan, and J. Foster, "A comparison of bug finding tools for java," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004, pp. 245–256.

[25] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*, 2013, pp. 196–232.

[26] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, p. 50–59.

[27] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 339–349.

[28] D. A. Tomassi and C. Rubio-González, "On the real-world effectiveness of static bug detectors at finding null pointer exceptions," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021, pp. 292–303.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.