Team Members:
Mengxing Chen (chenm32)
Anny Kong (yk57)
Yuqi Huang (yh73)
Xinrong Zhao (zhaox29)

Nullness_Lite: An unsound option of the Nullness Checker with fewer false positives

https://github.com/weifanjiang/Nullness_Lite

1. Abstract

In this paper, we are going to focus on the Nullness Checker part of the Checker Framework. We create another version of Nullness Checker, Nullness_Lite, as a option users can choose before using the complete Nullness Checker. Nullness_Lite is more user-friendly since it requires fewer annotations than the complete Nullness Checker, but it is unsound while the Nullness Checker is. We will focus on four features of the current Nullness Checker, and modify/disable each of them in order to reduce the number of false positives and the number of annotations needed.

We compare our Nullness_Lite with other checkers which can be used to find NullPointerException. Specifically, we choose a unit testing framework, Junit4, which is widely used in industry and unannotated by any of the nullness bug detector or type system. We run each checker on Junit4, and record the number of true positives, the number of false positives, the number of true positives the checker does not reveal, and the number of annotations needed to run this checker on Junit4. We treat the complete Nullness Checker as the sample in order to record the number of true positives each checker does not reveal since the complete Nullness Checker is sound.

2. Problems & Motivation

Null is so significant to software development that almost all the programming languages allow programmers to write null. But null has been the cause of countless troubles in the history of software development. Tony Hoare, the inventor of null, calls it “the billion-dollar mistake” [10]. More than that, since null is the main reason for the infamous null pointer error, there is hardly a programmer who is not troubled by null. According to research, the null pointer exceptions (NPE) are reported as the most frequent bug in Java programs [31]. Null pointer dereferences are not only frequent [32], but also catastrophic [33]. Though Java already provides an infrastructure for exceptions, the current state of the language is just a partial solution. Professor John Sargeant from Manchester School of Computer Science also said “of the things which can go wrong at runtime in Java programs, null pointer exceptions are by far the most common” [3].

As developers seek a world where no NPEs are raised, some nullness bug detectors, such as FindBugs [11], NullAway [24], and the Nullness Checker of the Checker Framework [4], are emerging as a supplement of Java’s weak type system that does not support compile-time nullness checking. While there is a correlation between popularity and star counts on GitHub [34], we look at stars and notice there are about 2.1k stars for NullAway [24], 0.5k stars for FindBugs [23], and 361 stars for the Checker Framework [25]. As these numbers suggest, on the large open-source platform for developers, there might be still not a large number of developers on Github using the Nullness Checker of the Checker

Framework, the one with the sound analysis, we have an idea: add an option with fewer false positives and make it easier to use. As an initial step and evaluation of this idea, we proposed to build a new Nullness_Lite option on the Nullness Checker of the Checker Framework, the one that is sound but complicated to use. It is be unsound but still effective in the static analysis of nullness errors. And it can provide a faster and easier option for Java developers who would like to get a compile-time nullness analysis on their source code, but hesitate to spend time running full verification tools like the Nullness Checker.

3. Recent Solutions & Related Work

NullAway is an annotation-based nullness bug detector which is mainly built to detect bugs in Android projects, its command-line based version also works on non-Android programs. It uses its own type system to detect nullness bugs [30]. It is fast: "the build-time overhead of running NullAway is usually less than 10%" [24]. Although it is built as a plugin of ErrorProne [17], it actually benefits most of the users since ErrorProne is widely used in industry. However, NullAway has its limitations. For example, it cannot check code using generics [16] and null assertions [18].

IDEs like IntelliJ [14] and Eclipse [15] also provide annotation-based null analysis. And they have their own libraries for annotations. IntelliJ supports two annotations, `@Nullable` and `@NotNull`. Eclipse support three annotations, `@NonNull`, `@Nullable`, and `@NonNullByDefault`. Also, IntelliJ has a functionality called "Infer Nullity" which automatically add `@Nullable` and `@NotNull` in the project. Also, they provide dataflow analysis which runs in the background so that when the user types in the program, they statically check for (possible) null-related errors.

FindBugs [27] is another nullness bug detector. It analyzes nullness bugs using heuristic based pattern matching which is a code idiom that has a high probability of being an error [41]. Also, FindBugs is powerful in that it can directly analyze the bytecode of a program [29], so the users do not even need the source code of their program in order to use FindBugs.

The Nullness Checker of Checker Framework is a sound, pluggable type checker [9], which, unlike those bug detectors introduced above, aims at detecting all nullness bugs. It requires users to add annotations into their code as machine-checked documentation that its type-based dataflow analysis, "a technique to statically derive information about the dynamic behavior of a program" [19], can take advantage of to give a precise error report. An experiment of checking the Lookup program [35] showed that the Nullness Checker has the stronger analysis than nullness bug detectors FindBugs, Jlint [20] and PMD [21], by successfully detecting all 9 true positives of nullness bugs while others missed all [4].

Although the Nullness Checker introduces 11 annotations while other nullness bugs detectors above introduce fewer, customers can still find it easy to use. The Nullness Checker has local type inference [26], so that if customers do not specify the annotations for a variable then it will automatically infer annotations for the variable and issue errors if no annotations allow the program to type-check. There are also some type inference tools customers can use as an extra step before checking their programs with the Nullness Checker. Since the unannotated types are analyzed by the Nullness Checker as `@NonNull` by default [26], customers can use the AnnotateNullable tool [36] of Daikon [37] to add `@Nullable`, although sometimes customers need to write additional `@Nullable`, it reduces the burden of customers to add annotations for using the Nullness Checker.

However, since "the Checker Framework is designed to value soundness over limiting false warnings" [4], some assumptions for nullness bugs analysis in the Nullness Checker are so conservative that overwhelms the users by raising too many false positives. One false positive example raised is the BaseTestRunner class [39] from JUnit4 [38]. The Nullness Checker issues a field uninitialized error at the declaration of the class variable `fPreferences`. Yet leaving `fPreferences` uninitialized will not cause NPEs at runtime, because it is a private static field that is never dereferenced inside the BaseTestRunner class and the only accessor method, `getPreferences()`, will never return the uninitialized `fPreferences` directly. We will examine more about why the Nullness Checker raise this error and also other kinds of false positives in next section. After all, the false positives raised by the Nullness Checker require extra effort from users to verify the them manually.

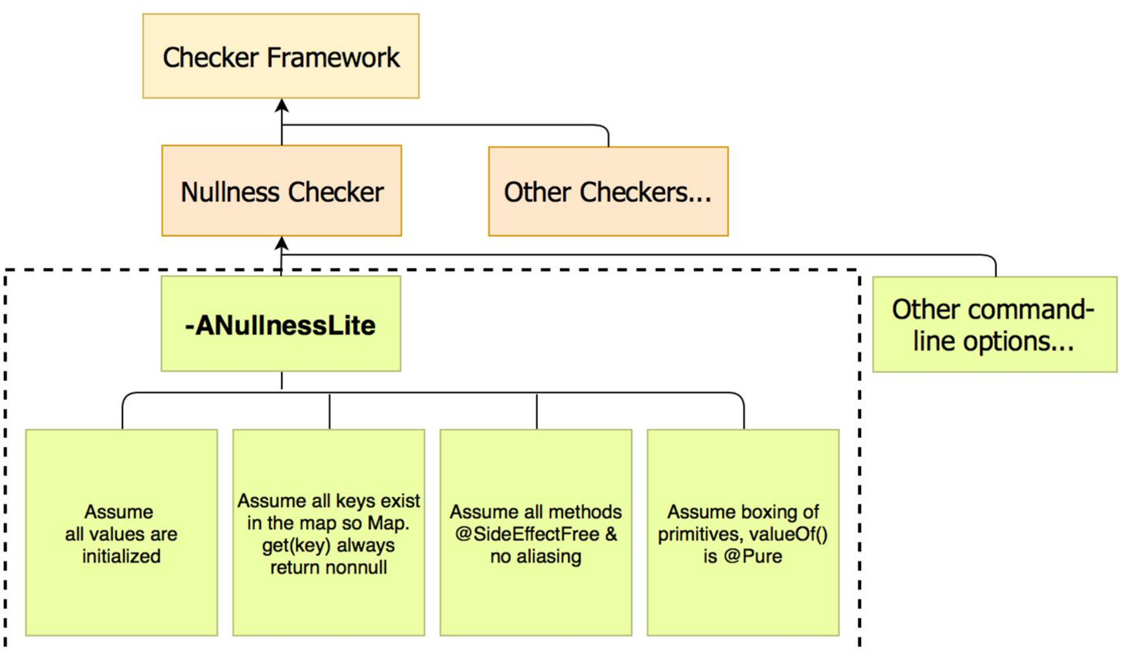
```java
public abstract class BaseTestRunner implements TestListener {
    public static final String SUITE_METHODNAME = "suite";

    private static Properties fPreferences;
    protected static Properties getPreferences() {
        if (fPreferences == null) {
            fPreferences = new Properties();
            fPreferences.put("loading", "true");
            fPreferences.put("filterstack", "true");
            readPreferences();
        }
        return fPreferences;
    }
```

(Figure 1. False positive example of Uninitialized Error)

4. Approaches

4.1 Hypothesis

Considering the suggestions given by Michael Ernst, one of the Checker Framework lead maintainers, and listed in the Nullness bug detector section of GSoC ideas 2018 [22], we attempted to build on the current Nullness Checker and add a new option ---- Nullness_Lite, as shown in figure 2.



(figure 2. Nullness_Lite Outline)

As an extension to the original Nullness Checker, Nullness_Lite is a fast, but incomplete and unsound nullness checker. We keep the original sound checker, and provide a new possibility for users to start small and advance to full verification in the future, rather than having to start out doing full verification.

The Nullness_Lite option can be enabled via providing a command line argument "`-ANullnessLite`". And it has four differences from the current Nullness Checker: disable the Initialization Checker and the Map Key Checker that the nullness analysis based on; modify some assumptions of the dataflow analysis and the behaviors of boxing primitives.

Our goal is to reduce the number of false positives. When testing with an open source program, Google Collections, the original one was found to report 362 false positives while only reporting 9 true errors [9], which is obviously a significant trouble for bug fixing. The following four features were suggested as an implementation of this idea. After trying with each feature, we see the potential possibility for these features to produce a large number of false warnings, an annoying thing for programmers to deal with. They have to look into each warning and suppress each by a manual proof that it is not a true positive. In the following section, we give examples for some possible existing problems with false positives, which may have time costly effects, in the Nullness Checker's internal implementation:

- **Assume all values are initialized:**

  For `@NonNull` values, the Nullness Checker assumes that all instance variables are initialized literally inside the constructors and all static class variables are initialized at their declarations. If violating either of the assumptions above, the Nullness Checker simply issue fields uninitialized errors no matter whether these fields actually raise NPEs or not.

  The conservative assumptions are good to eliminate the bugs caused by uninitialized fields, but sometimes are impractical. Back to the BaseTestRunner example in JUnit4 shown in figure 1, we realized that developers sometimes decide not to initialize static fields at declarations in the real world projects. Although we analyzed that the uninitialized field fPreferences cannot raise NPEs at runtime, the Nullness Checker still issues the false positive that confuses the developers when debugging.

  Therefore, we have the hypothesis that the unsound Nullness_Lite option can reduce the false positives by suppressing the uninitialized errors.

- **`Map.get` returns `@NonNull` result:**

  The Map Key Checker which the Nullness Checker depends on tracks which values are keys for which maps. The Nullness Checker uses its `@KeyFor` annotation to determine if a value is a key for a given map — that is, to indicate whether `map.containsKey(value)` would evaluate to `true`. If variable v has type `@KeyFor("m")...`, then the value of v is a key in Map m. That is, the expression `m.containsKey(v)` evaluates to true [27].

  The `Map.get` method is annotated with `@Nullable` in the annotated jdk files, therefore assumed to be returning nullable values except some cases. The `@KeyFor` annotation is checked by a Map Key Checker that the Nullness Checker invokes. This annotation enables the Nullness Checker to treat calls to `Map.get` more precisely by refining its result to `@NonNull` if the following two conditions are satisfied[27]: 1) `mymap`'s values are all non-null; that is, mymap was declared as `Map<`*`KeyType`*`, @NonNull `*`ValueType`*`>`. Since `@NonNull` is the default type, it need not be written explicitly. 2) `mykey` is a key in `mymap`; that is, `mymap.containsKey(mykey)` returns `true`. This fact could be expressed to the Nullness Checker by declaring `mykey` as `@KeyFor("mymap")` *`KeyType`* `mykey`.

However, the Map Key analysis is not strong enough to cover many cases when `Map.get` also returns a non null result. In particular, it is unable to analyze codes outside the scope of method where the variable `key` is declared. As `Map.get` method is assumed to be `@Nullable`, key-value pairs added elsewhere, say added in other methods, are not recognized by the Nullness Checker and will cause it to produce a false warning. For example, figure 3 is an example of such false positives in Junit4. While we run the Nullness checker on the code, it gives a warning, while it should not since the key `validateWithAnnotation` is in the map `VALIDATORS_FOR_ANNOTATION_TYPES`. The variable `validateWithAnnotation` is added to the map `VALIDATORS_FOR_ANNOTATION_TYPES` when it is absent in method

```
public AnnotationValidator createAnnotationValidator(ValidateWith validateWithAnnotation) {
    ...
    AnnotationValidator annotationValidator = validateWithAnnotation.value().newInstance();
    VALIDATORS_FOR_ANNOTATION_TYPES.putIfAbsent(validateWithAnnotation, annotationValidator);
    return VALIDATORS_FOR_ANNOTATION_TYPES.get(validateWithAnnotation); // OK but a warning
    ...
}
```

(Figure 3. False positive example of the Map Key checker)

`putIfAbsent()` right before `get()` being called. Additionally, the corresponding value `annotationValidator` is not null which is ensured by `newInstance()`. Since the method `putIfAbsent()` is not within the scope of the function `createAnnotationValidator()`, Map Key Checker, the subchecker of the Nullness Checker, could not recognize its existence and assumes that it is not in the map `VALIDATORS_FOR_ANNOTATION_TYPES`, while it is actually in the map at the time when `get(validateWithAnnotation)` is called. As a result, it gives a false warning.

As demonstrated in the example above in practice, the Map Key Checker's analysis cannot always infer a true warning for `Map.get`, and there is a potential for more keys being added outside the current method than within the current method, therefore false warnings may reduce if we disable the Map Key Checker. We will make Nullness_Lite to assume for every call to `Map.get(key)`, the given key exists in the map and always returns a `@NonNull` result.

- **Assume all methods `@SideEffectFree` and no aliasing:**

The Nullness Checker considers two situations that potentially invalidate the dataflow facts. One is method invocation. The other is updating one field which potentially causes a change of its aliasing.

The Nullness Checker analyzes the former using the side effect analysis [26]. When a `@Nullable` variable is refined to `@NonNull`, the Nullness Checker will invalidate the `@NonNull` fact if any method having access to this variable is called. Yet the assumption is conservative because methods are considered "dangerous" even if they do not actually modify the fields. Thus, the Nullness Checker sometimes raise false positives for the real world projects because it is not uncommon for developers to write code that can cause this kind of false warnings.

One example is the ComparisonCompactor class [40] also from JUnit4, partial code showed in figure 4. The Nullness Checker issues an error at line 29, that the instance variable `fExpected` is a `@Nullable` string but the method `compactString(String source)` requires the parameter to be `@NonNull`. However, this error is a false positive of invalidation of dataflow. First of all, `fExpected` is refined to be `@NonNull` after the if statement at line 23, where we exit the whole method if `fExpected` is null. Although having access to `fExpected`, the methods called at line 27 & 28 do not reassign `fExpected` to be null (code is not shown in figure 4). Thus, we draw the conclusion that `fExpected` is `@NonNull` at line 29. Yet the Nullness Checker invalidates the `@NonNull` fact after the method call at line 27 simply because it has the access to `fExpected`.

```
21      @SuppressWarnings("deprecation")
22      public String compact(String message) {
23          if (fExpected == null || fActual == null || areStringsEqual()) {
24              return Assert.format(message, fExpected, fActual);
25          }
26
27          findCommonPrefix();
28          findCommonSuffix();
29          String expected = compactString(fExpected);
30          String actual = compactString(fActual);
31          return Assert.format(message, expected, actual);
32      }
```

(Figure 4. False positive example of Method Side Effect Error)

```
import org.checkerframework.checker.nullness.qual.Nullable;

public class FWExample {
    public static class Node {
        public @Nullable Node next;
        public int val;

        public Node (int val) {
            this.val = val;
            this.next = null;
        }
    }

    public void foo(Node a) {
        Node b = new Node(0);
        if (a.next != null) {
            b.next = null;
            a.next.toString(); // False Positive Warning
        }
    }
}
```

```
zhaox29@ubuntu:~/403$ javac -processor nullness FWExample.java
FWExample.java:18: error: [dereference.of.nullable] dereference of
 possibly-null reference a.next
            a.next.toString(); // False Positive Warning
             ^
1 error
```

(figure 5. A False Positive Example of Aliasing)

For the latter, when assigning null to some field of a variable, the Nullness Checker checks whether other variables under the same scope can be its aliases and invalidate the `@NonNull` fact of their fields as well. However, it conservatively assumes that any two objects can be aliases of each other if one is the same type or the subtype of the other. In this way, although revealing all aliasing bugs, the Nullness Checker sometimes raises false positives for variables of the same type but not aliases of each other. We did not find the real world example for this kind of false warnings so far, but one "naive" example in shown in figure 5.

Due to reasons above, we predict that although making the Nullness Checker unsound, the Nullness_Lite option can reduce the false positives by both assuming all methods are `@SideEffectFree` and no aliasing allowed.

- **Boxing of primitives to be not `@Pure`:**

The Nullness Checker has most boxed classes' valueOf(primitiveType) method such as `Integer.valueOf(int)`, `Character.valueOf(char)`, `Short.valueOf(short)` being annotated as `@SideEffectFree` rather than `@Pure` in the annotated jdk for its soundness. The `valueOf` methods in wrapper classes always (`Boolean`, `Byte`) or sometimes (`Character`, `Integer`, `Short`) return an interned result as stated in JLS 5.1.7 because it does not guarantee that the boxed primitive is always the same object [28].

```
void showIntegerValueOfFalseWarnings() {
    if (foo(Integer.valueOf(127)) != null) {
        foo(Integer.valueOf(127)).toString(); // OK but a warning
    }
}

@Pure
@Nullable String foo(int b) {
    return "notnull";
}
```

(Figure 6. False positive example of the boxing of primitives)

However, for type `int` between -128 and 127 inclusive, it will generate false warnings. JLS states "when the value `p` being boxed is an integer literal of type `int` between -128 and 127 inclusive, and let `a` and `b` be the results of any two boxing conversions of `p`, it is always the case that `a == b`" [28]. But since the Nullness Checker assumes it to be only equals by `equals()` for being sound, it produces a false warning while passing an `int` between -128 and 127. As we have not encountered a real-world example so far, in the contrived example shown in figure 6, `foo()` is expected to return the same string since it is annotated with `@Pure`. However, the Nullness Checker gives a false warning since it assumes two calls to `Integer.valueOf(127)` to return two different strings.

The false positives also happen when the boxing `Integer.valueOf(128)` is passed in as a parameter to some method, say `foo()`, then in this case, the method `foo()` will only return null if it distinguishes `Integer.valueOf(128)`. Hence, as long as the call of foo with the boxing of 128 returns nonnull, another call to `foo()` with the same boxing of 128 will also be nonnull if the method does not care about inputs or does not distinguish `Integer.valueOf(128)`. Then we will propose to assume that the JVM is always interning integers. Hence Nullness_Lite will assume the boxing of primitives always returns the same object on every call by replacing the original annotation `@SideEffectFree` with `@Pure`.

Although we are deliberately giving up soundness instead of purely improving the Nullness Checker, we provide users with an additional faster and easier-to-use option. With our unsound option, users can reach another point in the design space where fewer annotations are required and fewer false positives are produced. Our hypothesis states with the trade-off of soundness, the Nullness Checker with the Nullness_Lite option enabled should produce fewer false warnings and require fewer annotations, therefore faster and more usable for programmers.

4.2 Command-line User Interface

Nullness_Lite has a command line interface. To use Nullness_Lite checker, users will add a command line argument "`-ANullnessLite`" when using Nullness Checker. Figure 7 describes an example running Nullness_Lite through javac:

```
zhaox29@ubuntu:~/jsr308/checker-framework$ javac -processor nullness -ANullnessLite <MyFile.java>
```

(figure 7. Nullness_Lite's Command Line Usage)

Note that "`-ANullnessLite`" option will only work for the Nullness Checker. The behavior is undefined if the option is passed to other checkers. All original commands in Checker Framework will still work.

5. Evaluation & Experiments

This section shows our evaluation work, the comparison of the nullness-related errors reported on a real world project by the Nullness_Lite option, the Nullness Checker, NullAway, FindBugs, IntelliJ, and Eclipse. We evaluated our project on the open-source Java Program, JUnit4[38], which is a unit testing framework which is widely used in industry and unannotated by any of the nullness bug detector or type system. The code segment we analyzed excludes the test code in JUnit4, because it does not reflect the way developers would implement in real world projects.

Checkers/Features	Bugs Revealed	Bugs Not Revealed	False Positives	# Annotations Added
The Nullness_Lite Option	20	0	80	318
- All variables initialized	20	0	84	319
- `Map.get()` returns `@NonNull`	20	0	90	318
- No aliasing + all methods `@SideEffectsFree`	20	0	94	319
- `BoxedClass.valueOf()` are `@Pure`	20	0	95	319
The Nullness Checker	20	0	95	319
NullAway (using IntelliJ's Infer Nullity)	3	17	1	1285 (added by Infer Nullity)
NullAway (using annotations required for the Nullness Checker)	3	17	0	319
NullAway (using annotations required for the Nullness_Lite)	3	17	0	318
IntelliJ	0	20	1	0
IntelliJ (Infer Nullity)	4	16	76	16 changed
FindBugs	0	20	0	0
Eclipse	0	20	3	0

(figure 8. Measurement Table)

5.1 Evaluation for the Nullness_Lite option and the Nullness Checker

We analyzed the Nullness_Lite option, each feature of the Nullness_Lite option and the Nullness Checker by the following steps:

1. Compile all source files with the checker we evaluate, with annotated JDK provided by the Checker Framework.
2. If some false positives can be reduced by simply adding annotations provided by the Nullness Checker, then leave brief comment why we added annotations here. Then add the annotations and

repeat step 1 (because adding annotations sometimes results in new errors). If no false positives can be reduced by that, then proceed to step 3.

3. Analyzed the rest errors and leave a comment, which consists of the type of the error, either it is true or false positive, and a brief reasoning. An error is analyzed as an true positive (bug revealed) if the latest JUnit4 API[42] provides users the needed information and functions to raise NPEs or the internal implementation does not prevent NPEs. We provided a code example in the comment for each true positive. An error is analyzed as an false positive if the all situations to raise NPEs are impossible according to the JUnit4 documentation and implementation.

The evaluation result is shown in figure 8. In the result, the Nullness_Lite option revealed all 20 bugs reported by the Nullness Checker, which is good for JUnit4. However, the Nullness_Lite option is still unsound by disabling features and can suppress true positives reported by the Nullness Checker in other projects.

Compared to the Nullness Checker, the Nullness_Lite reduced 15 false positives related to the features being disabled. Among these false positives, we found 11 reduced by "All variables initialized" feature, 3 reduced by "`Map.get()` returns `@NonNull`" feature, and 1 reduced by "all methods `@SideEffectsFree`" feature. Although we didn't find any errors related to features "No aliasing" and "`BoxedClass.valueOf()` are `@Pure`" in JUnit4, we reported it on our table as a reference for developers who want to extend our project in the future.

5.2 Evaluation for NullAway

We used IntelliJ's infer nullity to add `@Nullable` and `@NotNull` automatically, and there are 1160 annotations added by IntelliJ. NullAway reported 4 errors, where 3 of them were true positives. However, since we used annotations generated by Intellij, we might miss some annotations at current stage (since NullAway can check packages and files which are not fully annotated). In that case, since we already analyzed annotations needed to run Nullness Checker and NullnessLite, we created another two branches to record the result of running NullAway with annotations required by Nullness Checker and NullnessLite, respectively. Nullness Checker used 319 annotations, and NullAway found 3 errors (all of them are true positives). NullnessLite used 318 annotations, and NullAway found the same 3 true positives.

5.3 Evaluation for IntelliJ

We evaluate the nullness checker of IntelliJ in 2 ways: 1. Run the nullness checker on JUnit4 which does not have any annotations. 2. Run the nullness checker on JUnit4 which has annotations added by Infer Nullity, a functionality that automatically examines the project and then adds `@Nullable` and `@NotNull` in the source code of the project.

We found only 1 false positive in the non-annotation version of JUnit4. And we cannot reduce this false positive by adding annotations in the source code. Detailed analysis is included in comments in the source code.

Infer Nullity introduced 1116 annotations into JUnit4. And then we ran IntelliJ's checker on this version of JUnit4. We examined each error found by IntelliJ's checker, and tried to eliminate some of them by changing some annotations added by Infer Nullity. It turns out that after changing 16 annotations, the number of errors are reduced to the largest extent: 76 errors are false positives and 4 are true positives. And we've included detailed analysis in comments in the source code.

It is noticeable that Infer Nullity adds this many annotations to JUnit4. Although the number of annotations added by Infer Nullity is much more than that added by either Nullness Checker or Nullness_Lite, Infer Nullity is effective in that it reduces around 15 false positives, compared to Nullness Checker and Nullness_Lite. Also, as shown in figure 8, IntelliJ finds 4 false positives in the annotated version of JUnit4, which indicates that Infer Nullity is not powerful enough because Nullness Checker and Nullness_Lite can find more true positives with fewer annotations added, and true positives are what the users most care about.

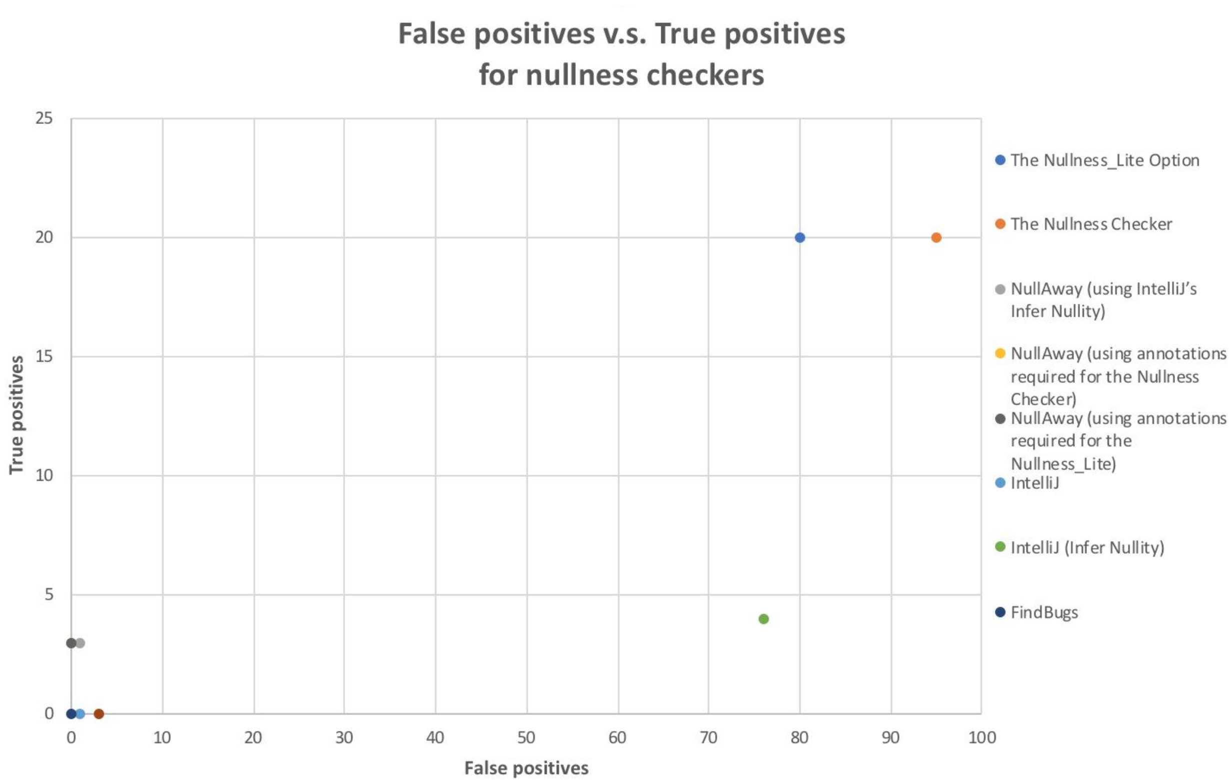
5.4 Evaluation for FindBugs

We used findbugs-3.0.1 version. It does not need any change of its configuration since errors related to NPEs are placed in the same directory after fully analysing the project. Since FindBugs reported 8 null-related errors in JUnit4 test files and 0 errors in source files, we had the conclusion that FindBugs report 0 warnings and the results shown in the table above.

5.5 Evaluation for Eclipse

In order to run Eclipse's nullness analysis, we changed the compiler preference by checking "Null Pointer Access" and "Possible Null Pointer Access". Then, by rebuilding the project, Eclipse found 3 null-related errors. We tried to eliminate as many errors as possible by adding annotations into the source code, but it turns out that all the 3 errors found by Eclipse cannot be eliminated in this way. So the number of annotations added for Eclipse is zero. Also, we've classified that all 3 errors are false positives and then attached detailed analysis in the source code in order to justify our classification.

5.6 Evaluation Summary



(Figure 9. False positives v.s. True positives for nullness checkers)

Figure 9 shows the results according to the true and false positives. However, our evaluation result does not imply that one checker is definitely better than others. Every checker has its own strengths and weaknesses, which results in different design spaces. As a result, users should choose the tool that fit their situations best. For example, the Nullness Checker is a good choice when users value a good verification over the time consumed. The other bug detectors are good in the reversed situation. The NullAway is good for users working for android projects; the IDEs like IntelliJ and Eclipse are good for users who want instant check while typing the code; FindBugs automatically classifies errors by the type so that users can find the same type of bugs together; the Nullness_Lite option is in the middle ground of a good verification and fair time consumed. Depending on the project, the Nullness_Lite option can potentially reveal more true positives than other nullness bug detectors, but users should still be aware that it can also reveal more false positives.

5.7 Reproducibility

In the repository of the Nullness_Lite option, we provide users with the instructions to reproduce the results we got in figure 8 and our analysis details in branches of our forked JUnit 4 repository (https://github.com/NullnessLiteGroup/junit4) in figure 10.

The Checker for Evaluation	**Branch Name**
The Nullness_Lite Option	annos_nl_all
--All variables initialized	annos_nl_init
--No aliasing + all methods `@SideEffectsFree`	annos_nl_inva
--`Map.get()` returns `@NonNull`	annos_nl_mapk
--`BoxedClass.valueOf()` are `@Pure`	annos_nl_boxp
The Nullness Checker	annos_nc_all
IntelliJ	intellij1
IntelliJ with Infer Nullity	intellij2
Eclipse	eclipse
FindBugs	findbugs
NullAway with annotations added by Infer Nullity	Nullaway_Intellij
NullAway with annotations required for the Nullness Checker	Nullaway_nc
NullAway with annotations required for NullnessLite	Nullaway_nl

(figure 10. The Evaluation Branch Table)

6. Risks & Challenges

Our evaluation is helpful for developers but still has some limitations. As for positives, our evaluation is of value in that it includes almost all popular checkers which detect nullness bugs. By showing and comparing the data we've got from evaluating the checkers on JUnit4, we come to the conclusions of the relative advantages/disadvantages of these checkers compared to each other. Thus, our evaluation provides useful information for developers when they need to choose a proper checker in order to detect potential nullness bugs in their code.

However, there are limitations within our evaluation. One is that our experimental subject, JUnit4, is not the latest version of Java unit testing framework which might not be well maintained, because it is not the latest version of Java unit testing framework. But it is still a popular choice among developers based on star counts on Github. Therefore, our evaluation provides the developers of JUnit4 with useful information about the potential nullness bugs in Junit4. Another limitation of our evaluation is the limit size of evaluation object, that we chose only 1 project for evaluation so that we didn't show the case Nullnes_Lite option can suppress true positives and the error reduced by some features in the Nullness_Lite option. With more projects, the result could be more generalizable and thus more convincing.

7. Bibliography

1. "Interesting Facts about Null in Java." GeeksforGeeks, 14 Sept. 2017, www.geeksforgeeks.org/interesting-facts-about-null-in-java/.
2. Neumanns, Christian. "Why We Should Love 'Null'." *Why We Should Love 'Null' - CodeProject*, 18 Nov. 2014, www.codeproject.com/Articles/787668/Why-We-Should-Love-null.
3. Sargeant, John. "Null Pointer Exceptions." *Null Pointer Exceptions*, www.cs.man.ac.uk/~johns/npe.html.
4. Dietl, Werner, and Michael Ernst. " Preventing Errors Before They Happen The Checker Framework." *Preventing Errors Before They Happen The Checker Framework*, www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&cad=rja&uact=8&ved=0ahUKEwiup6_V5bPaAhXKrFQKHW-gCxYQFghaMAY&url=https%3A%2F%2Fstatic.rainfocus.com%2Foracle%2Foow17%2Fsess%2F1492901668615001brln%2FPF%2F2017-10-02%2520CF%2520%40%2520JavaOne_1507012791774001WJ2t.pdf&usg=AOvVaw3mAtzExTzYm6gr3sCn0cXb.
5. Osman, Haidar, et al. "Tracking Null Checks in Open-Source Java Systems." *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, doi:10.1109/saner.2016.57.
6. Admin. "Java NullPointerException - Reasons for Exception and How to Fix?" *The Java Programmer*, 5 June 2017, www.thejavaprogrammer.com/java-nullpointerexception/.
7. Dobolyi, Kinga. "Changing Java's Semantics for Handling Null Pointer Exceptions."
8. Maciej Cielecki, Je¸drzej Fulara, Krzysztof Jakubczyk, and Jancewicz. Propagation of JML non-null annotations in java programs. In *Principles and practice of programming in Java*, pages 135–140, 2006.
9. Dietl, Werner, et al. "Building and Using Pluggable Type-Checkers." Proceeding of the 33rd International Conference on Software Engineering - ICSE '11, 2011, doi:10.1145/1985793.1985889
10. Hoare, Tony. "Null References: The Billion Dollar Mistake." *InfoQ*, Tony Hoare, 25 Aug. 2009, www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare.
11. Ayewah, Nathaniel, et al. "Using Static Analysis to Find Bugs." *IEEE Software*, vol. 25, no. 5, 2008, pp. 22–29., doi:10.1109/ms.2008.130.
12. Sridharan, Manu. "Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android." *Uber Engineering Blog*, 15 Feb. 2018, eng.uber.com/nullaway/.
13. Hovemeyer, David, et al. "Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs." *The 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering - PASTE '05*, 2005, doi:10.1145/1108792.1108798.
14. "Infer Nullity." *JetBrains*, www.jetbrains.com/help/idea/inferring-nullity.html.

15. *Help - Eclipse Platform*, help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-using_null_annotations.htm.
16. Sridharan, Manu. “Support Models of Generic Types · Issue #54.” *GitHub, Uber/NullAway*, 6 Nov. 2017, github.com/uber/NullAway/issues/54.
17. Google. “Error Prone.” *Error Prone*, 2017, errorprone.info/.
18. kevinzetterstrom. “Support for null assertions · Issue #122 · Uber/NullAway.” GitHub, github.com/uber/NullAway/issues/122.
19. Abel, Andreas, et al. “Dataflow Framework for Checker Framework.” *Dataflow Framework for Checker Framework*, courses.cs.washington.edu/courses/cse501/10au/JavaDFF.pdf.
20. Knizhnik, Konstantin, and Cyrille Artho. “About Jlint.” *Jlint - Find Bugs in Java Programs*, jlint.sourceforge.net/.
21. Pmd. “PMD.” *PMD*, 29 Apr. 2018, pmd.github.io/.
22. “GSoC Ideas 2018.” *Checker Framework Organization: GSoC Ideas 2018*, rawgit.com/typetools/checker-framework/master/docs/developer/gsoc-ideas.html.
23. Findbugsproject. “Findbugsproject/Findbugs.” *GitHub*, 23 Sept. 2017, github.com/findbugsproject/findbugs.
24. Uber. “Uber/NullAway.” *GitHub*, github.com/uber/NullAway.
25. Typetools. “Typetools/Checker-Framework.” *GitHub*, github.com/typetools/checker-framework.
26. *The Checker Framework Manual: Custom Pluggable Types for Java*, checkerframework.org/manual/#map-key-checker.
27. “FindBugs™ - Find Bugs in Java Programs.” *FindBugs™ - Find Bugs in Java Programs*, findbugs.sourceforge.net/.
28. “Chapter 5. Conversions and Contexts.” *Lesson: All About Sockets (The Java™ Tutorials > Custom Networking)*, docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.7.
29. “FindBugs™ Fact Sheet.” *FindBugs™ - Find Bugs in Java Programs*, findbugs.sourceforge.net/factSheet.html.
30. “Engineering NullAway, Uber's Open Source Tool for Detecting NullPointerExceptions on Android.” *Uber Engineering Blog*, 15 Feb. 2018, eng.uber.com/nullaway.
31. Maciej Cielecki, Je¸drzej Fulara, Krzysztof Jakubczyk, and Jancewicz. Propagation of JML non-null annotations in java programs. In *Principles and practice of programming in Java*, pages 135–140, 2006.
32. Arthur van Hoff. The case for java as a programming language. *IEEE Internet Computing*, 1(1):51–56, 1997.
33. V Vipindegep and Pankaj Jalote. List of common bugs and programming practices to avoid them. Technical report, Indian Institute of Technology, Kanpur, March 2005.
34. Papamichail, Michail, et al. “User-Perceived Source Code Quality Estimation Based on Static Ana*lysis Metrics.”* 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, doi:10.1109/qrs.2016.22.
35. *Lookup*, 10 May 2018, types.cs.washington.edu/plume-lib/api/plume/Lookup.html.
36. The Daikon Invariant Detector User Manual, plse.cs.washington.edu/daikon/download/doc/daikon.html#AnnotateNullable.
37. Ernst, Michael D., et al. “The Daikon System for Dynamic Detection of Likely Invariants.” *Science of Computer Programming*, vol. 69, no. 1-3, 2007, pp. 35–45., doi:10.1016/j.scico.2007.01.015.
38. junit-team. “Junit-Team/junit4.” *GitHub*, github.com/junit-team/junit4.
39. junit-team. “Junit-Team/junit4.” *GitHub*, github.com/junit-team/junit4/blob/master/src/main/java/junit/runner/BaseTestRunner.java.
40. junit-team. “Junit-Team/junit4.” *GitHub*, github.com/junit-team/junit4/blob/master/src/main/java/junit/framework/ComparisonCompactor.java.
41. “Analysis Tool Report.” http://www.cs.cmu.edu/~aldrich/courses/654/tools/simulacrum-findbugs-06.pdf

42. “JUnit API.” *JUnit 4*, junit.org/junit4/javadoc/latest/.

Total Hours Spent: 12 + 6 + 10 + 16 + 10 * 4 * 4 * 4