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. 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 strong correlation between popularity and star counts on GitHub [34], we notice there are about 2.1k stars for NullAway [24], 0.5k stars for FindBugs [23], and only 361 stars for the Checker Framework [25]. As we realize there are still not as many people 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 propose 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 will be unsound but still effective in the static analysis of nullness errors. And it will 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.

2. 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.

```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;
    }
```

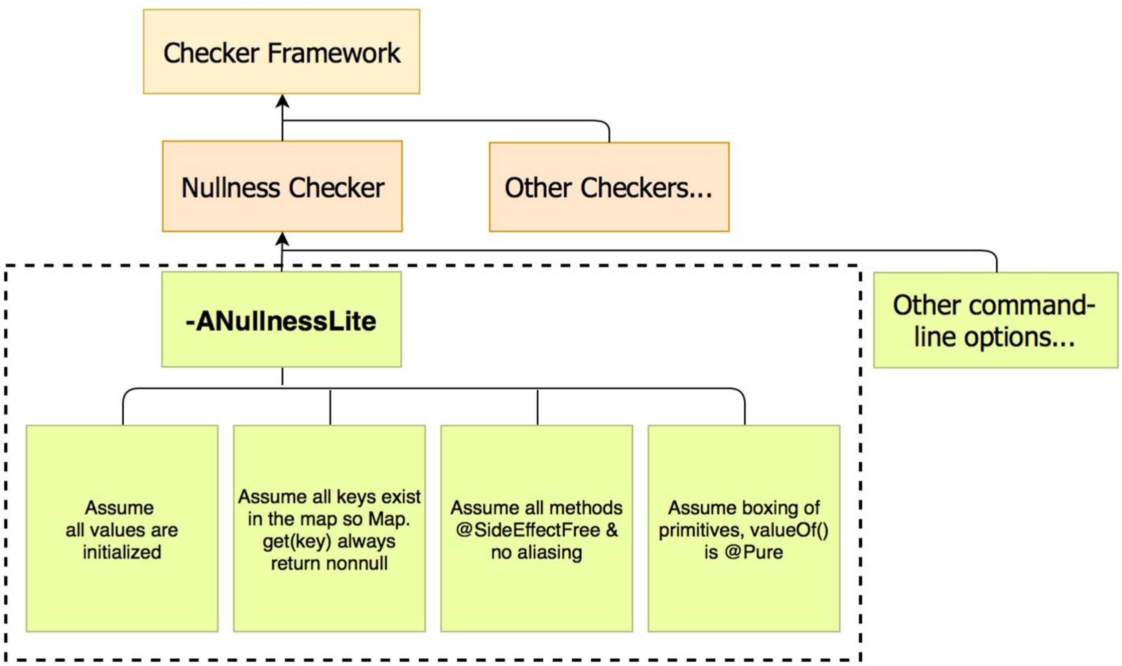
(Figure 1. False positive example of Uninitialized Error)

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]. We only show the segment of its code revealing the error reported by the Nullness Checker in figure 1, otherwise it will be too long to fit in pages. 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.

3. Approaches

3.1 Hypothesis

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



(figure 2. Nullness_Lite Outline)

It is basically an extension to the original Nullness Checker, a fast, but incomplete and unsound, nullness checker. We will 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 could be enabled via providing a command line argument "`-ANullnessLite`". And it will have four differences from the current Nullness Checker: disable part features of 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. The details of each feature will be discussed as follows.

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 state the 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, 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`.

```
void showFalseWarnings() {
    Map<String, String> m = new HashMap<String, String>();
    String in = "in";
    foo(m, in);

    m.get(in).toString(); // Ok but a warning
}

void foo(Map<String, String> m, String in) {
    m.put(in, in);
    return;
}
```

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

  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, when we run the code in figure 3, the Nullness checker gives a warning, while it should not since in is in the map m. The variable in is added to the map m in method foo(), which is not within the scope of the function showFalseWarnings(). Then the Map Key Checker, the subchecker of the Nullness Checker, could not recognize its existence and assumes that it is not in the map m, while it is actually in the map m at the time when m.get(in) is called. As a result, it gives a false warning.

As demonstrated in the example, 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)

```java
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].

```java
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 will say this is a worthy trade-off to make. 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.

Besides, we need to provide more effective evidence for features we disabled or modified. These features are all included in Nullness Checker, but how do we know they are good choices? We have to answer it with evaluation. Therefore, our plan is to first evaluate each feature independently. We will test to see how many true warnings are lost and false warnings are reduced. If the evaluation shows a positive result for each one, we have a reason for including it in Nullness_Lite.

Further, to prove our hypothesis, we need to evaluate our option with other recent nullness bug detectors, and the way how we determine whether or not an evaluation result is positive will be discussed in section 4.

3.2 Architecture & Implementation Plan

The brief implementation plan is to remove functions related to these features in the Nullness Checker under the fork of Checker Framework. Features are disabled separately for evaluation. After evaluation, we choose the features we want to keep and merge the implementations for different features. Then we evaluate Nullness_Lite again with these features included, to see if Nullness_Lite will meet the expectation to be a competitive one among all other nullness bug detectors.

Then we will make Nullness_Lite an extended option for Nullness Checker. We will keep the Nullness Checker architecture but add more sub-functions to realize the behaviors of Nullness_Lite. We will also change the control flow of the Nullness Checker. Thus, when Nullness_Lite is enabled, its corresponding behaviors will be invoked. For instance, Nullness Checker has three components: Nullness Checker proper, Initialization Checker and MapKey Checker, which are completely independent. We may add the new behaviors of Initialization and MapKey parts for Nullness_Lite. Then, when Nullness_Lite is used, the functionality of Initialization Checker and Map Key Checker will be turned off, because we will invoke the new behaviors instead of these checkers' original behaviors used for Nullness Checker. By possibly adding new behaviors and control flows, we will not change the behaviors of Nullness Checker when Nullness_Lite is not used.

3.3 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.

3.4 Current Progress of Implementation

We have finished the implementation, only left one pull request still in reviewing. The last thing to do is to have the developer manual completed.

4. Evaluation Plan & Experiments

Our purpose is to build a fast and user-friendly checker that can be an alternative choice for Java developers to detect nullness bugs at compile time. And once we have built it, it is crucial for us to fully analyze our new checker by comparing it with the existing checkers in the market.

We choose NullAway, FindBugs, IntelliJ, and Eclipse we have discussed in section 2 to be experimental subjects that we compare the Nullness_Lite with. The standard of our measurement is shown below (note that our measurement focuses only on nullness bugs):

Checkers/Features	Bugs Revealed	Bugs Not Revealed	False Positives	# Annotations Added	Avg Time to Check Programs
The Nullness Checker with Nullness_Lite Option	30		64	320	
- All variables initialized					
- `Map.get()` returns `@NonNull`					
- No aliasing + all methods `@SideEffectsFree`					
- `BoxedClass.valueOf()` are `@Pure`					
NullAway					
FindBugs	0		0	0	
IntelliJ	0		1	0	

IntelliJ (Infer Nullity)	4 (in progress)		18 (in progress)	1160 (by Infer Nullity)	
Eclipse	0		0	0	
The Nullness Checker	63(need review)	0	65(need review)	468(need review)	

(figure 8. Measurement Table)

4.1 Target Program for Evaluation

We evaluate the above checkers by focusing on an open-source Java Program, JUnit4 (link to our fork: https://github.com/junit-team/junit4/). JUnit4 is a unit testing framework which is widely used in industry. It contains about 15k lines pure Java code, which is enough for our evaluation. Also, we don't find any annotations which are needed for specific verification tools, such as `@Nullable`. In that case, our measurement of the number of annotations needed for each verification tool would be fair. Furthermore, we are going to analyze NullPointerExceptions in the source code of JUnit4. Because JUnit4 itself is a unit test framework, we think that its test files are well written and thus are unnecessary to measure.

Also, if time allows, we will measure the checkers above using another open-source Java Program in order to make our evaluation results more generalizable.

4.2 Evaluation of Annotations

The attribute, "Number of Annotations Added", is the number of annotations that we need to add to the JUnit4 in order to eliminate as many errors found by the checkers as possible. Since different checkers have different measurement, "Number of Annotations Added" for JUnit4 to pass each checker should vary and thus it should be an attribute when we evaluate each checker.

4.3 Evaluation of Running Time

We consider running time can be a valuable attribute to evaluate, because of the fact that unsound nullness bug detectors are usually faster than a sound type system, but our Nullness_Lite option is in between of the two categories. Although checkers will have different running time depend on the hardware, we will evaluate these checkers on the CSE lab machine, so our clients can have a reference of these checkers' relative time and how the Nullness_Lite option may or may not improve the running time of the Nullness Checker.

4.4 Evaluation of False Positives

We measure the number of false positives generated by each checker by manually reasoning about each error found by each checker. For one specific Java program, if a checker generates more false positives and another checker generates fewer, the former checker is more difficult to use while the latter one is more flexible and user-friendly.

4.5 Evaluation of Bugs Revealed and Bugs Not Revealed

We use the result produced by Nullness Checker as the ground truth when evaluating true positives (i.e. bugs revealed), because Nullness Checker has the "strongest" soundness among all checkers, which means its specification tests will detect all the nullness bugs in the program that other checkers may not be able to detect. So, if a checker fails to detect a bug which is detected by the Nullness Checker, this bug will be counted as a “not revealed” bug for this checker. We run JUnit4 on each checkers and record the number of bugs revealed and not revealed in the measurement table.

4.6 Determine the Features to Include in Nullness_Lite

This part answers the question that whether a specific feature of Nullness Checker is good to be disabled. The goal of Nullness_Lite is to let the users add fewer annotations and get fewer false positives, which will be more convenient for them to use. Accordingly, it is good if disabling a feature results in the desirable goal above. Also, we will consider the unsoundness caused by disabling some specific features when we evaluate Nullness_Lite.

Since we care if disabling a feature can eliminate as many false positives as possible, based on results, we will prioritize the features with an increasing order of false positives generated and then choose appropriate features for Nullness_Lite. After including these features, we will evaluate Nullness_Lite by experiments 4.2~4.7

4.7 Determine the Quality of Nullness_Lite

For Nullness_Lite, the trade-off of its user-friendliness is the “relative” unsoundness. Therefore, Nullness_Lite is never perfect. The evaluation table shows the pros and cons of the checkers listed above, including Nullness_Lite, and let the clients decide whether they should use Nullness_Lite as a checker when they are programming.

4.8 Current Progress and Reproducibility

The clients can see our current progress of evaluation of each checker in our fork of JUnit4 (https://github.com/NullnessLiteGroup/junit4/), and we also provided the instructions in our github repository (https://github.com/weifanjiang/Nullness_Lite) to reproduce the results we got in the figure 8. Figure 9 contains the branches for some checkers we finished and currently working on.

The Checker for Evaluation	**Complete Status**	**Branch Name**	**Script provided for reproducing evaluation results?**
The Nullness_Lite Option	Close to the end (< 10 un-analyzed errors)	annos_nl_all_xz	YES
The Nullness Checker	Need review	analysis_3_nc_yk_xz	YES
IntelliJ	Finished	intellij1	NO
IntelliJ with Infer Nullity	In progress (~ 40 errors)	intellij2	NO
Eclipse	Finished	eclipse	NO
FindBugs	Finished	findbugs	NO

(figure 9. The Evaluation Branch Table)

4.9 Analysis of the Result In Progress

We claim the results we got in the table, at the beginning of the section 4, are reasonable and reliable for the following reasons:

- For the initial result of the Nullness Checker and the Nullness_Lite Option:

  Here is the methodology for evaluation of the Nullness Checker and the Nullness_Lite Option. For evaluation of the number of annotations, the annotations we count are `@Nullable`, `@UnderInitialiation` and `@UnknownInitialization`. These annotations are added in order to reduce false positives as many as possible. We add annotations according to the both the documentation and the actual implementation of JUnit4, and leave the comments about why annotations is necessary to be added.

  After no false positives can be reduced by repeating the process above, we began to analyze the errors left. We only used `@SuppressWarnings("nullness")` to suppress true and false positives. The true positives include the errors from the JUnit4 API that users can write code to raise NPEs and errors from the internal implementation of JUnit4 where developers write some code which can raise NPEs. The false positives are ones that can never happen under any circumstance. On each error, we left the comment consisting of the type of the error, either it is true or false positive, and a brief reasoning. Since we leave comments for any changes we made in the source files, the reviewer can check the correctness of our analysis easily.

  We claim the errors reported by the Nullness_Lite options are the subset of errors reported by the Nullness Checker, because a checker with some dataflow analysis features disabled cannot report errors it never reports before. As a result, as we finished the initial analysis of the Nullness Checker on 5/11 and got familiar with JUnit4 implementation, we decided to analyze the Nullness_Lite from beginning (instead of according to the results of the Nullness Checker) based on the consideration of time emergence and the truth that the Nullness Checker lacked reviewing.

  Shown in our current result, the number of annotations added in the JUnit4 and the number of true positives are largely decreased when evaluating the Nullness_Lite. As we finished the Nullness_Lite and reviewed, we will start to revise the result of the Nullness Checker.

- Eclipse

  In order to run null analysis, we changed the compiler preference by checking "Null Pointer Access" and "Possible Null Pointer Access". And then by rebuilding the project, Eclipse find 3 null-related errors. In the source code, we've attached detailed analysis in the comment blocks near each error. However, since the 3 null-related errors found by Eclipse are in JUnit4's test files rather than source files, we did not include them in the measurement table.

- 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.

- IntelliJ

We have two versions of results for evaluating IntelliJ. One version we finished is IntelliJ without Infer Nullity, which means we use IntelliJ to check JUnit4 with no annotations added. We only found 1 false positive in this version. The other version we haven't finished is Intellij with Infer Nullity. And we currently found 4 true positives and 18 false positives by 1160 `@Nullable` and `@NotNull` annotations added by Infer Nullity.

5. Risks & Challenges

Since we finished the implementation and were able to use all the checkers we evaluate, our current challenges are analyzing the errors for evaluation. In specific, the current evaluation program JUnit4 has inconsistent documentation (some source files have more specification documented but some don't). As a result, the analysis process takes more time than our estimate and it is difficult to get everything correct. To shrink the time elapsed for each error analyzed, we mainly focused on analyzing the errors from components exposed to clients in JUnit4 API (https://junit.org/junit4/javadoc/latest/), because usually it is easy to decide true positives in these components by thinking of some examples. For the internal errors, we trace the call hierarchy in the project to see whether the developers write the wrong code somewhere without any documentation. For correctness, our solution is to leave comments on every annotations and analysis where errors are issued, and to assign peer review for these annotations and analysis.

6. Week-By-Week Schedule

We have finished the "midterm": implementation of the Nullness_Lite. The "final" remain to be finished: complete the evaluation results on JUnit4 and other checkers if time allows.

	Current Schedule
Week 2 (4/4 - 4/11)	Setups - Start reading Checker Framework Manual(Especially Nullness checker chapter) - Finish BUILD checker framework - Finish Eclipse/IntelliJ setup
Week 3 (4/11 - 4/18)	Preparation - Finish reading Checker Framework Manual - Begin understanding Checker Framework source code - Setup roles in the group - Start implementing Nullness_Lite
Week 4 (4/18 - 4/25)	Implementation of Nullness_Lite - Begin disabling annotations - Begin making it independent of other type checkers-light - Evaluation Preparation: become familiar with other checkers and find tests
Week 5 (4/25 - 5/2)	Implementation of Nullness_Lite - Finish implementation of the option - Finish the specification tests of the option - Finish peer review between developers - Start evaluation of other current nullness tools (NullAway, FindBugs, Eclipse and IntelliJ)
Week 6 (5/2 - 5/9)	Evaluation - (JUnit4) Add annotations and evaluate Nullness Checker & NullnessLite (zhaox29 & yk57) - (JUnit4) Add annotations and evaluate other checkers (chenm32 & yh73) - Document -ANullnessLite in Checker Framework manual

	- Give the initial result (5/10)
Week 7 (5/9 - 5/16)	Evaluation - Continue the evaluation of JUnit4 - If time allows, choose an additional open-source project to evaluate - Write a guide to reproduce the evaluation results - Complete the user manual of Nullness_Lite - Tests, manual & report for improvement
Week 8 (5/16 - 5/23)	Evaluation - Continue the evaluation of the current project for separate features of NullnessLite - If time allows, choose an additional project to evaluate - Summarize the current evaluation results - Analyze current results and how our results compare to the hypothesis - Tests, manual & report for improvement
Week 9 (5/23 - 5/30)	Evaluation - Address the final report - Continue evaluation of the current project - Enhancement according to the user feedback

Week 10 (5/30 - 6/6)	Evaluation and Comparison
	- Finish final reports
	    - Finish annotations analysis
	    - Finish evaluation of results
	- Prepare presentation of our project

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 Vipindeep 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

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