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/979216944/checker-framework

1. Problems & Motivation

Null is so significant to the 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 (called NullPointerException in Java), there is hardly a programmer who is not troubled by null. According to research, the null pointer error is reported as the most frequent bug. Professor John Sargeant from manchester school of computer science says that "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 for a world where no NPEs (NullPointerException) 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. On github, 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 Works

Although 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].

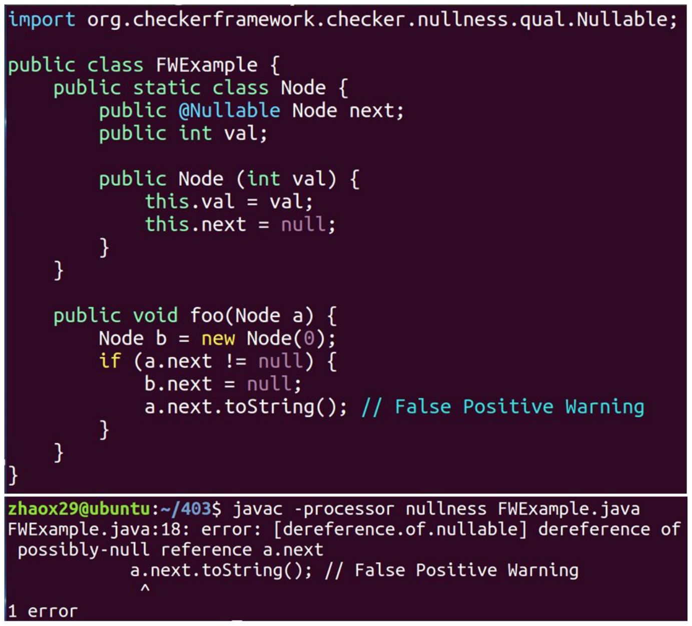
IDEs like IntelliJ[14] and Eclipse[15] also provide null analysis. Both of them support a few annotations to refine analysis, thus very easy to use. IntelliJ supports two annotations `@Nullable` and `@NotNull`. Eclipse support three annotations `@NonNull`, `@Nullable`, and `@NonNullByDefault`. They are also convenient for developers because they will statically check the code while developers type it out.

Yet they are also unsound. For example, they cannot detect nullness bugs caused by assigning null to fields of aliased variables.

FindBugs[27] is another nullness bug detector. It uses static analysis to find the bugs in a program, which means that it can find bugs by simply looking at the program's source code without actually executing it. 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.

Also, it is worth mentioning that FindBugs has various forms for users to choose: command-line based version, plugins for IDEs, and its own GUI version. As to our experience, since the users can use FindBugs using its own user interface (and it can be used as plugins for IDEs), it is visually more intuitive, compared to other command-line based checkers such as Nullness Checker and our Nullness_Lite. Also, users do not need to remember a bunch of commands or follow what the manual exactly says in order to use FindBugs; all they need to do is clicking the button on the user interface of FindBugs and then import their programs to it. However, there are some disadvantages for the users: FindBugs tells the user where the bugs are, but it does not allow users to directly open the target files from its user interface. Therefore, it is inconvenient when the users want to directly change their files according to the bugs FindBugs shows.

The Nullness Checker of Checker Framework is a sound, pluggable type checker[9], which, unlike bug detectors introduced above, aims to detect 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 (which no longer exists) 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].



```
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 1. A False Positive Example)

Although the Nullness Checker introduces 11 annotations while other nullness bugs detectors above introduce fewer, customers can still find it easy to use, because the Nullness Checker has the automatic type refinement and there are type inference tools to automatically add annotations to the programs.

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. The figure 1 is one example of a false warning reported by the Nullness Checker. It is caused by a method that determines whether the null assignment invalidates the dataflow facts of other non-null aliasings. However, the Nullness Checker assumes that any two objects can be aliasings 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 requires extra effort from users to verify the false warnings manually.

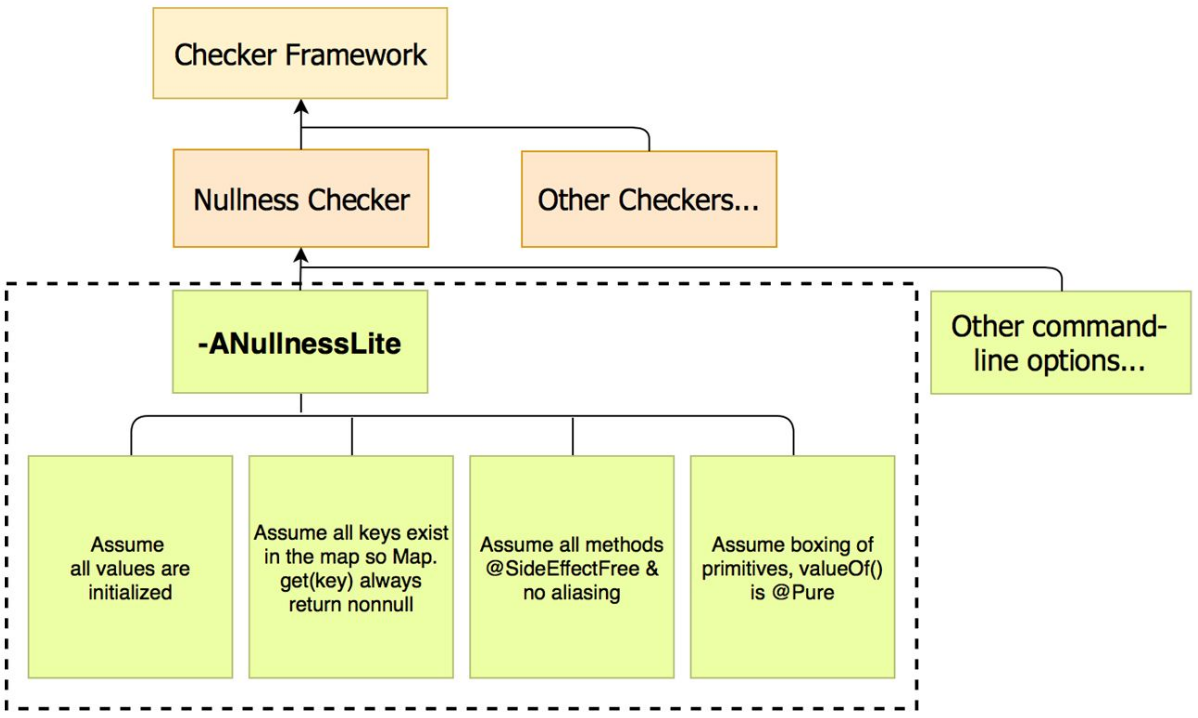
3. Approaches

3.1 Hypothesis

Considering the suggestions given by the original author of the Checker Framework, 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.

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 different functionality 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 to the dataflow analysis and the behaviors of boxing primitives.


Checker Framework
Nullness Checker
Other Checkers...
-ANullnessLite
Other command-line options...
Assume all values are initialized
Assume all keys exist in the map so Map.get(key) always return nonnull
Assume all methods @SideEffectFree & no aliasing
Assume boxing of primitives, valueOf() is @Pure

(figure 2. Nullness_Lite Outline)

Our goal is to reduce number of false positives. The original one was found to report 4 false positives while finding 9 true errors [4], which is about a half. As listed in GSoC ideas 2018, the following four features were suggested as an implementation of this idea [22]. After trying with these features, 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 manually 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 issues fields uninitialized errors no matter these fields actually raise NPEs or not.

  The conservative assumptions are good to eliminate the bugs caused by uninitialized fields, but sometimes are impractical. In the real world projects, developers sometimes decide to initialize fields by calling some helper methods from a constructor. Thus, they may be confused by the false positives of the Nullness Checker not recognizing their initialization in helper methods. Besides that, most NPEs caused by dereferencing fields can still be caught if developers use annotations correctly.

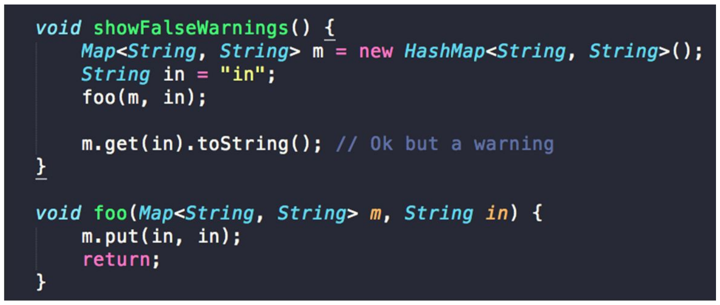
  Therefore, we have the hypothesis that by suppressing the uninitialized errors, the unsound NullnessLite option can reduce the false positives and still keep a fair amount of true positives.

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

  The Map Key Checker, 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 other words, 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



```
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) 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 sub checker 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 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 the 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. Plus, it is not uncommon for developers in the real world projects to write code that can cause this kind of false warnings. Dereferencing a `@Nullable` static field before a static method call can be a simple example.

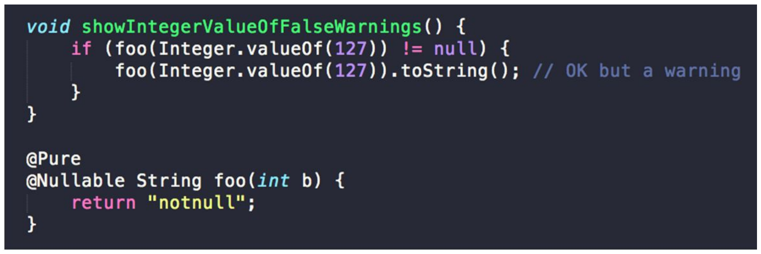
  We described the latter in section 2 and gave an example of the false positives we want to avoid.

  Due to reasons above, we predict that although making the Nullness Checker unsound, the NullnessLite 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 `BoxedClass.valueOf(primitiveType)` 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].

  However, for type `int` between -128 and 127 inclusive, it will generate false warnings. 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



```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 4. False positive example of the boxing of primitives) be only equals by `equals()` for being sound, it produces a false warning while passing an `int` between -128 and 127. In the example shown in figure 4, `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 many other nullness bug detectors, and the way how we determine whether or not an evaluation result is positive will be discussed in section 4.

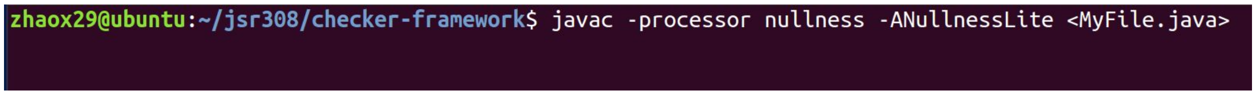
3.2 Architecture & Implementation Plan

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 5 describes an example running Nullness_Lite through javac:


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

(figure 5. Nullness_Lite's Command Line Usage)

Note that "`-ANullnessLite`" option will only work for 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

For now, we have finished the implementation of NullnessLite option, features, and peer-reviews between developers. We will continue adding tests necessary to test the functionality of NullnessLite, and begin to modify the user manual documented in our forked Checker Framework repo.

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
Nullness_Lite					
- All variables initialized					
- Map.get() returns `@NonNull`					
- No aliasing+all methods `@SideEffectsFree`					
- BoxedClass.valueOf() are `@Pure`					
NullAway					
FindBugs	0		8	0	
IntelliJ					
Eclipse	0		3	0	

Nullness Checker		0			
… (if time allows)					

(figure 6. 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/). This program is an "industrial" one rather than an "toy" one and has around 30k lines of real code. Also, it does not write by following the rules of any specific checkers listed above. Otherwise, our measurement is unfair. Furthermore, we are going to analyze NullPointerExceptions in both the source code and test files of JUnit4. Because JUnit4 itself is a unit test framework, we think that its test files are valuable to measure.

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 evaluate each checker above by looking at its running time on JUnit4. Since the running time of different checkers should vary, the running time will be an appropriate attribute for the users when they choose their checkers. However, the running time is not as important as other attributes listed in the table above, since it depends on the machine that runs the checker and it does not vary too much for different checkers as we've observed (all of them finish in 1 minute).

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. However, some checkers, including Nullness Checker, unavoidably generate false positives, which is a trade-off of their soundness.

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 Nullness Checkers, 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

Till now, we've finished evaluating Eclipse and FindBugs, and are still working on Nullness Checker and IntelliJ. 2 of the team member (zhaox29 & yk57) are responsible for evaluating the checker except Nullness Checker and Nullness_Lite, and the other 2 team members (chenm32 & yh73) are working on evaluating Nullness Checker. Since we deem Nullness Checker as the "ground truth", we will first measure all other attributes except "Bugs Not Revealed", and will fill that column out after we finish evaluating Nullness Checker.

We've created our own fork of JUnit4 (https://github.com/NullnessLiteGroup/junit4/) and will have one branch of each checker. The client should focus on the respective branch when he/she is examining one specific checker. Currently, we've created three branches, "eclipse", "findbugs" and "CFanno_yk_xz". The first two branches show our analysis for each true/false positives found by the respective checkers and the client can find it by looking at the latest commits for these branches. The branch "CFanno_yk_xz" is for the evaluation of Nullness Checker. Currently, our work is adding annotations on that branch in order to eliminate as many errors found by Nullness Checker as possible.

We will provide the files which helps the client reproduce our results. For each checker that is command-line based, we will provide a script file that shows the number of annotations added and our analysis for true/false positives. For other checkers that have user interfaces (including Eclipse, IntelliJ and FindBugs), we will provide the script files that show the number of annotations added as well as the config files in order to ensure that the client has the same settings when he/she reproduces our results. Then, if needed, we will provide manuals which include which parts of JUnit4 the client should look at in order to see our analysis (for each true/false positive).

5. Risks & Challenges

For implementation, one challenge is to extend the Checker Framework, which is a big project (more than 200,000 lines of Java codes) with complex dependencies. It would be tricky for us to check whether we have made the correct changes. Our solution is to start with specific questions. For example, we want to add an option to Nullness Checker, so we start with searching the keywords through the files under the Checker folder, see how relevant features are implemented and trace the extra information from other parts of Checker Framework if necessary. Then we assign peer review among implementers to double check we are making the correct changes.

For evaluation, building and using other checkers (such as NullAway, FindBugs, IntelliJ, etc.) can be challenging, because these checkers can run in different environments and have distinct user manuals that we have to be familiar with in order to give a reliable evaluation report. Since some of them also use annotations to support verification, it is tricky to add annotations in the program for evaluation. In this case, our solution is to also peer review among evaluators to check whether we are adding the annotations correctly.

6. Week-By-Week Schedule

The "midterm" will be finishing up the implementation of Nullness_Lite. The "final" will be completing a systematic report on the evaluation of Nullness_Lite and the comparison between Nullness_Lite and other checkers. Here is our week-by-week 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, ...)
Week 6 (5/2 - 5/9)	Evaluation - (JUnit4) Finish adding annotations and evaluations of the Nullness Checker & NullnessLite (zhaox29 & yk57) - (JUnit4) Finish adding annotations and evaluations of other checkers (chenm32 & yh73) - Document -ANullnessLite in Checker Framework manual - Give the initial result (5/10)
Week 7 (5/9 - 5/16)	Evaluation - Continue evaluation for JUnit4 or choose additional real world project to evaluate - Record the reproducible steps for evaluations - Complete the user manual on repo - User tests & report for improvement
Week 8 (5/16 - 5/23)	Evaluation - Continue evaluation for the current project or choose additional project to evaluate - Record the reproducible steps for evaluations - User tests & report for improvement

Week 9 (5/23 - 5/30)	Evaluation - Address the final report - checker(#annotations/100 lines, time to set up, time to check, easiness to use, ...)
Week 10 (5/30 - 6/6)	Evaluation and Comparison - Finish final reports - Presentation for the final product

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.

Total Hours Spent: 12 + 6 + 10 + 16