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

Nullness_Lite: A lite option of the Nullness Checker

https://github.com/weifanjiang/Nullness_Lite

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 are introduced, such as FindBugs[11], NullAway[12], and the Checker Framework [4], as a supplement of Java's weak type system that does not support compile-time nullness checking. As we realized there are still not as many people around us using these checkers, we have an idea: make it lighter and 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

We previously mentioned several other nullness bug detectors for Java as choices for developers to verify their source code. Some of them are popular and widely used by developers, but each of them has its own pros and cons.

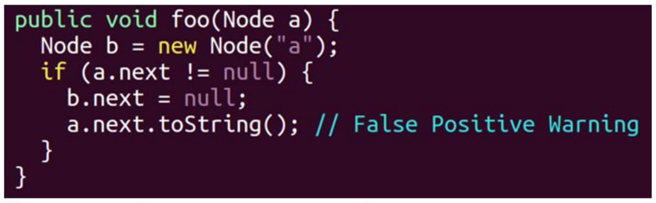
NullAway is an annotation-based nullness bug detector mainly built to detect bugs in Android projects though, it can also detect non-Android programs by running from command line.. It is very fast that "the build-time overhead of running NullAway is usually less than 10%"[18]. However, NullAway is built as a plugin to Error Prone[17], thus requiring developers to build their code with Error Prone. Besides, NullAway is unsound that, for example, it cannot check code using generics [17].

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 @NotNulla. 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 is another nullness bug detector. It can be run from command line or as plugins to several IDEs, such as IntelliJ and Eclipse. Thus, it is convenient for developers to use. It also supports 13 annotations. Although developers have to learn more than IntelliJ and Eclipse, they can detect more bugs with it, including the one mentioned in the last paragraph.

The Nullness Checker of Checker Framework was found to be especially successful in finding null pointer errors when an experiment of checking the Lookup program was carried out. The Nullness Checker of Checker Framework detects all 9 true positives of nullness bugs while most other nullness tools missed all [4].

As a full verification tool, however, the Nullness Checker also has a problem in raising many false positive warnings which can overwhelm the developers. For example, although Nullness Checker can guarantee the absence of errors, it as well reports 4 false warnings and requires 35 extra annotations written [4]. Here is one concrete example of false positives.



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

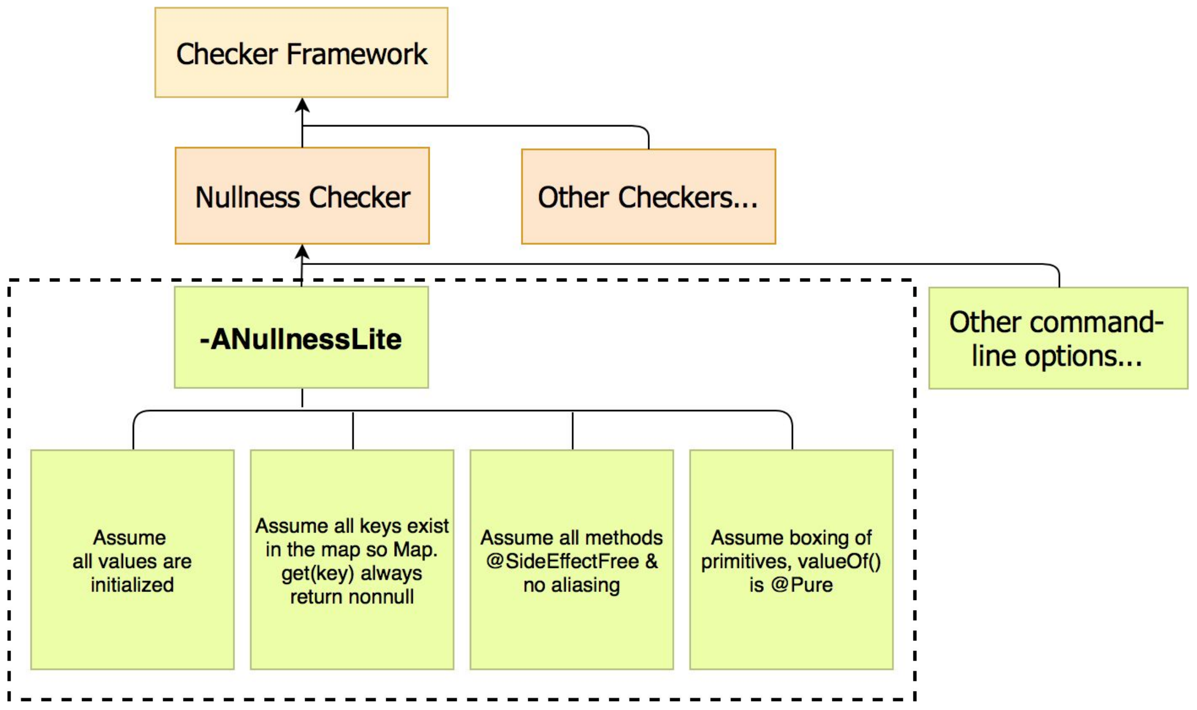
(figure 1: False Positive Warning)

Nullness Checker will issue a null dereference warning for code in figure 1, because it analyzes that b can be an aliasing of a. Thus, modifying b.next will make a.next unsafe (become @Nullable). However, we can see from the code that a and b are not aliasing of each other, thus a false positive.

3. Approaches

3.1 Our Approach

We attempt to build on the Nullness Checker and add a new option ---- Nullness_Lite. When Nullness_Lite is enabled via providing a command line argument, it will disable a set of features and make some assumptions for the Nullness Checker shown below:


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)

We were suggested to modify these features, and we have tried with some examples to see the effects:

- **Initialization Checker:** By assuming all variables initialized, Nullness_Lite issues fewer warnings for flexible programs. For example, developers sometimes purposely leave some fields uninitialized for a class in large programs.
- **MapKey Checker:** As Map.get() method is assumed to be @Nullable, key-value pairs added elsewhere, say other methods, are not recognized by the Nullness Checker and will always produce a false warning. Therefore Nullness_Lite assumes given key exist in the map and every call to Map.get(key) always return @NonNull.
- **Invalidation of Dataflow:** Current aliasing analysis is so strict that cause many false positives like figure 2. Although disabling this feature will be unsound, we expect to reduce many false positives. With Nullness_Lite, assuming methods to be @SideEffectFree reduces false positives.
- **Boxing of primitives to be not @Pure:** If the boxing `Integer.valueOf(128)` is passed in as a parameter to some method, say `foo()`, then 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 be very less likely to be null. Hence Nullness_Lite will assume 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 to some extent 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 losing 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.

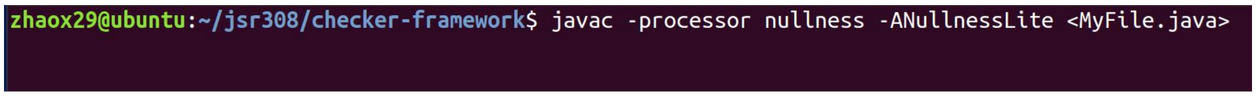
3.2 Architecture & Implementation Plan

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. Here is an example running Nullness_Lite through javac:



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

(figure 3. 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

The way how we modify the features is addressed in section 3.2.

For now, we have finished most implementation of features but the last one, assume boxing of primitives is @Pure. We are still seeking for a way to add it by making the annotation @Pure only being applied to

BoxedClass.valueOf() only if the Nullness_Lite option is enabled instead of changing the annotated JDK source files of the original Nullness Checker.

In addition to features, we have made the Nullness_Lite a command line option for users to simply add "-ANullnessLite" and we have added specification tests and some examples to Nullness Checker.

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 Warnings	# Annotation Used	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					
IntelliJ					
Eclipse					
Nullness Checker					
… (if time allows)					

(figure 3. Measurement Table)

4.1 Target Program for Evaluation

We are going to evaluate the above checkers by focusing on one specific open-source Java Program. This program should be a "industrial" rather than an "toy" one and have around 50k lines of real code. Also, it should not write by following the rules of any specific checkers listed above. Otherwise, our measurement is unfair.

4.2 Evaluation of Annotations

The attribute, “Number of Annotations Used”, is the number of annotations that we need to add to the Java program for evaluation. Since different checkers have different rules for annotations, “Number of Annotations Used” for the Java program to pass each checker should vary.

4.3 Evaluation of Running Time

We evaluate each checker above by looking at its running time on the Java program we choose. 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.

4.4 Evaluation of False Positives

We measure the number of false positives generated by each checker. We calculate the number of false positives by subtracting the number of true positives from the number of @SuppressWarnings annotations in the Java program.

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 unavoidably generate false positives, which is a trade-off of their soundness.

4.5 Evaluation of True Positives

We use Nullness Checker as the standard for evaluating true positives because Nullness Checker has stronger soundness than other checkers, which means its specification tests will cover more bugs that other nullness checkers may not be able to detect. We run the Java program we choose on each checkers and record the number of bugs revealed and unrevealed 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 run faster, use fewer annotations, and generate fewer false positives, which will be more convenient for clients 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 a feature can reduce as many false positive warnings as possible, based on results, we will prioritize the features with an increasing order of false positive warnings and then choose appropriate features to be included in Nullness_Lite. After including this features, we will evaluate Nullness_Lite by experiments 6.2~6.5.

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.

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 disabling annotations - Finish making it independent of other type checkers-light - Start evaluation of other current nullness tools(NullAway, FindBugs, ...)
Week 6 (5/2 - 5/9)	Testing - Correctness Test - Code coverage Test for better correctness tests
Week 7 (5/9 - 5/16)	Testing - Improvements according to correctness tests - User tests - User tests report for improvement - Finish evaluation of other current nullness tools(NullAway, FindBugs, ...)

Week 8 (5/16 - 5/23)	Evaluation and Comparison - Improvements according to user tests - Write code examples - Evaluate on Nullness_Lite and original Nullness checker
Week 9 (5/23 - 5/30)	Evaluation and Comparison - More code examples - More comparison between Nullness_Lite and original Nullness checker(#annotations/100 lines, time to set up, time to check, easiness to use, ...) - Evaluation of Nullness_Lite and other current nullness tools(NullAway, FindBugs, ...)
Week 10 (5/30 - 6/6)	Evaluation and Comparison - More improvements - Finish final reports

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. IntelliJ IDEA Infer Nullity: https://www.jetbrains.com/help/idea/inferring-nullity.html
15. Eclipse user guide: using null annotations. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-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. Error Prone : http://errorprone.info/
18. NullAway Repository & User Manual: https://github.com/uber/NullAway

Total Hours Spent: 12 + 6 + 10 + 16