On Leveraging Tests to Infer Nullable Annotations

Anonymous Author(s)
anonymous organisation(s)

Abstract

Issues related to the dereferencing of null pointers are a pervasive and widely studied problem, and numerous static analyses have been proposed for this purpose. These are typically based on dataflow analysis, and take advantage of annotations indicating whether a type is nullable or not. The presence of such annotations can significantly improve the accuracy of null checkers. However, most code found in the wild is not annotated, and tools must fall back on default assumptions, leading to both false positives and false negatives. Manually annotating code is a laborious task and requires deep knowledge of how a program interacts with clients and components.

We propose to infer nullable annotations from an analysis of existing test cases. For this purpose, we execute instrumented tests and capture nullable API interactions. Those recorded interactions are then refined (santitised and propagated) in order to improve their precision and recall. We evaluate our approach on seven projects from the spring ecosystems and two google projects which have been extensively manually annotated with thousands of `@Nullable` annotations. We find that our approach has a high precision, and can find around half of the existing `@Nullable` annotations. This suggests that the method proposed is useful to mechanise a significant part of the very labour-intensive annotation task.

2012 ACM Subject Classification Software reliability; Software defect analysis; Dynamic analysis

Keywords and phrases null analysis, null safety, testing, program analysis

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Null-pointer related issues are one of the most common sources of program crashes. Much research has focused on this issue, including: eliminating the problems of `null` in new language designs [56, 49, 52, 12, 59]; mitigating the impact of `null` in existing programs [23, 67, 5, 19]; and, developing alternatives for languages stuck with `null` [20, 29, 68].

More recently, several industrial-strength static analyses have been developed to operate at scale, such as *infer / nullsafe* [1, 19] and *nullaway* [5]. Such tools employ some form of dataflow analysis and take advantage of an extended type system that distinguishes in some way between nullable and nonnull types [23]. Here, a nonnull type is considered a subtype of a nullable type, and this relationship enables checkers to identify illegal assignments pointing to potential runtime issues. In Java, the standard annotation mechanism can be used to define such custom *pluggable* types [8]. For instance, using an annotation defined in JSR305 (i.e., the `javax.annotation` namespace), we can distinguish between the two types `@Nullable String` and `@Nonnull String`, with `@Nonnull String` being a subtype of `@Nullable String`. In a perfect world, developers would annotate all methods and fields, allowing static checkers to perform analyses with high recall and precision. Not surprisingly, this hasn't happened. Annotating code is generally a complex problem [13], and recent developer discussions reflect this. For instance, for *commons-lang* the issue LANG-1598 has been open since 14 August 20.[1] In a comment on this issue one developer commented *"Agreed this idea, but it is a HUGE work if we want to add NotNull and Nullable to all public*

---

[1] Open as of 20 October 22, see `https://issues.apache.org/jira/browse/LANG-1598`

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

functions in commons-lang." A similar comment can be found in a discussion on adding null-safety annotations to *spring boot* (*"it may well be a lot of work"*).[2]

Null-related annotations form part of a contract between the provider and consumer of an API. For instance, consider a library that provides some class `Foo` with a method `String foo()`. Adding an annotation may change this to `@Nullable String foo()`. This alters the contract with downstream clients which may have assumed the return was not nullable. Technically this change weakens the postcondition, thus violating Liskov's Substitution Principle (LSP) [42].[3] This may therefore cause breaking changes, forcing clients to refactor, for instance, by guarding call sites to protect against null pointer exceptions. Such a change may imply the downstream client was using the API incorrectly (i.e. by assuming `null` could not be returned). As such, one might argue the downstream client is simply at fault here and this change helps expose this. But, such situations arise commonly and oftentimes for legitimate reasons: perhaps the downstream client uses the API in such a way that, in fact, `null` can never be returned; or, the method in question only returns `null` in very rare circumstances which weren't triggered despite extensive testing by the downstream client. Regardless, developers must gauge the impact of such decisions carefully when modifying APIs. This illustrates the complexity of the task, and suggests that it is laborious and therefore expensive to add nullability-related annotations to projects.

Null checkers deal with missing annotations by using defaults to fill in the blanks. Those assumptions have a direct impact on recall and precision. The question arises whether suitable annotations can be inferred by other means.[4] Indeed, some simple analyses could be used here in principle, such as harvesting existing runtime contract checks. Using such checks is increasingly common as programmers opt to implement defensive APIs in order to reduce maintenance costs [17]. This includes the use of contract APIs such as *guava's* `Preconditions` [5], *commons-lang3's* `Validate` [6], *spring's* `Assert` [7] and the standard library `Objects::requireNonNull` protocol which all include non-null checks. Such an analysis could boost the accuracy of static null checkers that integrate with the compiler, as those contract APIs are defined in libraries that are usually outside the scope of the analysis performed by static checkers. However, exploiting the call sites of such methods is of limited benefit as those checks would only establish that a reference *must not be null.*

It is much more beneficial for static checkers to annotate code indicating that a reference *may be null* (i.e., '*"is nullable"*). The reason is that many static checkers use the *non-null-by-default* assumption that was suggested by Chalin and James after studying real-world systems and finding the vast majority of reference type declarations are not null, making this a sensible choice to reduce the annotation burden for developers [14]. They also point out that this is consistent with default choices in some other languages. The *checkerframework* and *infer* nullness checkers are based on this assumption, whilst some other null checkers such as the one embedded in the Eclipse IDE can be configured as such. Sometime, this is formalised.

---

[2] `https://github.com/spring-projects/spring-boot/issues/10712`

[3] LSP was formulated for safe subtyping, but can be applied in this context if we consider evolution as replacement

[4] Other here means not using the same technique used by static checkers. One could argue that if a static dataflow analysis was used to infer annotations, then that should be integrated into the checker in the first place

[5] `https://guava.dev/releases/21.0/api/docs/com/google/common/base/Preconditions.html`

[6] `https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html`

[7] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html`

For instance, the *spring framework* makes the use of the *non-null-by-default* assumption explicit by defining and using two package annotations [8] `@NonNullApi` and `@NonNullFields` in `org.springframework.lang`, with the following semantics (`@NonNullApi`, similar for `@NonNullFields` for fields): *"A common Spring annotation to declare that parameters and return values are to be considered as non-nullable by default for a given package"*. [9]

Using dynamic techniques is a suitable approach to observe nullability, and can be combined with static analyses to improve accuracy. Such hybrid techniques consisting of a dynamic pre-analysis feeding into a static analysis have been used very successfully in other areas of program analysis [6, 31]. A common reason to use those approaches is to boost recall [66].

In this paper, we explore this idea of inferring nullable annotations from test executions. This is based on the assumption that tests are a good (although imperfect) representation of the intended semantics of a program. We then refine those annotations by means of various static analyses in order to reduce the number of both false positives and false negatives.

This paper makes the following contributions: **1. a dynamic analysis** to capture nullable API interactions representing potential `@Nullable` annotations ("nullability issues") from program executions, **2. a set of static analyses ("sanitisation")** to identify false positives **3. a static analysis ("propagation")** to infer additional nullability issues from existing issues **4. a method to mechanically add the annotations inferred** into projects by manipulating the respective abstract syntax trees (ASTs) **5. an experiment** evaluating how the annotations we infer compare to existing `@Nullable` annotations of seven projects in the spring framework ecosystem and two additional google projects, containing some of the most widely used components in the Java ecosystem **6. an open source implementation** of the methods and algorithms proposed. These contributions directly relate to concrete research questions which we study in the context of evaluation experiments in Section 7.

2 Approach

Our approach consists of the following steps and the construction of a respective processing pipeline:

1. **Capture:** The execution of an instrumented program and the recording of *nullability issues*, i.e. uses of `null` in method parameters, returns and fields.
2. **Refinement:** The refinement of nullability issues captured using several light-weight static analyses.
    a. **Sanitisation:** The identification and removal of nullability issues captured that may not be suitable to infer `@Nullable` annotations to be added to the program, therefore eliminating potential false positives.
    b. **LSP Propagation:** The inference of additional nullability issues to comply with Liskov's Substitution Principle [42], therefore addressing potential false negatives.
3. **Annotation:** the mechanical injection of captured and inferred annotations into projects.

These steps are described in detail in the following sections.

---

[8] I.e., annotation used in `package-info.java`

[9] `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/NonNullApi.html`

3 Capture

3.1 Driver Selection

A dynamic analysis can be used to observe an executing program, and to record when `null` is used in APIs that can then be annotated. The question arises which driver to use to exercise the program. One option is to use existing tests, assuming they are representative of the expected and intended program behaviour.

If libraries are analysed there is another option – to use the tests of *downstream clients*. This approach has been shown to be promising recently to identify breaking changes in evolving libraries [47]. The advantage is that clients can be identified mechanically using an analysis of dependency graphs exposed by package managers and the respective repositories.[10] However, this raises the question which clients to use. Using an open world assumption to include all visible clients (i.e., excluding clients not in public repositories) is practically impossible given the high number of projects using commodity libraries like the ones we have in our dataset. There is no established criteria of how to select *representative* clients.

In principle, synthesised tests [54, 28] could also be used. However, they expose *possible*, but not necessarily *intended* program behaviour. Using synthesised tests would therefore likely result in too many `@Nullable` annotations being inferred. We note that some manually written tests may have the same issue. We will address this in Section 4.2.

In the approach presented here we opted to use only a project's own tests for generating actual annotations.

3.2 Instrumentation

In order to instrument tests, Java agents were implemented to record uses of `null` in APIs during the execution of tests. These agents can be deployed by modifying the (Maven or Gradle) build script of the project under analysis. The agents intercept code executions using the following six rules which check for occurrence of null references during program execution, and record those occurrences:

ARG	at method entries, parameter values are checked for `null`
RET	at method exits, return values are checked for `null`
FL1	at constructor (`<init>`) exits, reflection is used to check non-static fields for `null`
FL2	at non-static field writes (i.e. the `putfield` bytecode instruction), the value to be set is checked for `null`
SFL1	at class initialiser (`<clinit>`) exits, reflection is used to check static fields for `null`
SFL2	at static field writes (i.e. the `putstatic` bytecode instruction), the value to be set is checked for `null`

We have implemented agents implementing those rules using a combination of *ASM* [10] and *bytebuddy* [70]. If `null` is encountered, a *nullability issue* is created and made persistent.

Instrumentation can be restricted to certain (project-specific) packages, a system variable is used to set a package prefix for this purpose. This is to filter out relevant issues early as the amount of data collected is significant (see results in Table 2, column 3).

---

[10] Note that this requires the analysis of incoming dependencies, which is not as straightforward as the analysis of outgoing dependencies (which can simply use the maven dependency plugin) and requires some manual analysis, web site scraping or use of third-party repository snapshots such as *libraries.io*

3.3 Capturing Context

A nullability issue is identified by the position of the nullable API element (return type or argument index), and the coordinates (class name, method name, descriptor) of the respective method or field. We are also interested to capture and record the execution context for several reasons: **1.** to record sufficient information providing provenance about the execution, sufficient for an engineer who has to decide whether to add a `@Nullable` annotation or not **2.** related to the previous item, the number of contexts in which a nullable issue has been observed may itself serve as a quality indicator for the issue – more observed contexts provide some support for this being an issue (instead of a single tests triggering “unintended” program behaviour) **3.** to distinguish issues detected by running a project’s own tests from issues detected by running client tests **4.** to facilitate the sanitisation of issues, with some sanitisation techniques analysing the execution context.

In order to achieve this, we record the stack during capture. From the stack, we can then infer the *trigger*, i.e. the test method leading to the issue. The following algorithm is used to remove noise from the captured stack and identify the trigger:

1. the invocation of `java.lang.Thread::getStackTrace` triggering the stacktrace capture is removed from the stacktrace
2. all elements related to the instrumentation are removed
3. elements related to test processing (*surefire*, *junit*), reflection and other JDK-internal functionality are removed based on the package names of the respective classes owning those methods [11]
4. the last element in the stacktrace is set to be the trigger

3.4 Example

Listing 1 shows an issue captured running a test in *spring-core* and serialized using JSON. The test (trigger) is `ConcurrentReferenceHashMapTests::shouldGetSize`, it uses the `Map::put` API implemented in `ConcurrentReferenceHashMap`, which leads to `put` returning `null`.

```
{
"className":"$s.ConcurrentReferenceHashMap",
"methodName":"put",
"descriptor":"(Ljava/lang/Object;Ljava/lang/Object;Z)Ljava/lang/Object;",
"kind":"RETURN_VALUE",
"argsIndex":-1,
"stacktrace":[
  "$s.ConcurrentReferenceHashMap::put:282",
  "$s.ConcurrentReferenceHashMap::put:271",
  "$s.ConcurrentReferenceHashMapTests::shouldGetSize:331"
  ]
}
```

Listing 1 A serialised null issue captured in *spring-core* (for better readability `org.springframework.util` is replaced by `$s`)

3.5 Deduplication

When issues are captured, it is common that several versions of the same issue are being reported. For instance, there might be two nullability issues reported for the return type of the same method in the same class, but triggered by different tests, and therefore with

---

[11] More specifically, we consider methods in packages starting with the following prefixes as noise: `java.lang.reflect.`, `org.apache.maven.surefire`, `org.junit.`, `junit.`, `jdk.internal.`

different stack traces. Throughout the paper, only deduplicated (aggregated) issue counts are reported unless mentioned otherwise. The raw issues might still be of interest as they differ with respect to their provenance, which might be important for a developer reviewing issues.

3.6 Limitations

Our approach does not support generic types. For instance, consider a method returning `List<String>`. In order to establish that the list may contain `@Nullable` strings the analysis would need to traverse the object graph of the list object using reflection or some similar method, in order to check that some elements of the list are (or in general some referenced objects associated with the type parameters) are nullable. This is generally not scalable.

Secondly, there are dynamic programming techniques that may bypass the instrumentation. This is in particular the case if reflective field access is used, either directly using reflection, or via deserialisation. This is a known problem, however, reflective field access is rare in practice [66].

4 Sanitisation

4.1 Scope Sanitisation

When exercising code using instrumented tests, potential issues are captured and recorded for all classes including classes defined in dependencies, system and project classes. By setting project-specific namespace (package) prefixes, the analysis can be restricted to project-defined classes only as discussed in Section 3.2. However, this still does not distinguish between classes used at runtime (in Maven and Gradle, this is referred to as the *main* scope), and classes only to be used during testing (the *test* scope). Engineers may not see the need to annotate test code, and a static null checker would usually be configured to ignore test code as its purpose if to predict runtime behaviour such as potential null dereferences resulting in runtime exception.

The analysis to filter out classes not defined in main scope is straightforward: scopes are encoded in the project structure if build systems like Maven and Gradle are used. Those build systems and the associated project structures are the defacto-standards used in Java projects [2]. For instance, *spring* uses Gradle, and the compiled classes in main scope can be found in `build/classes/java/main`. The main scope sanitiser simply removes issues in classes not found in this folder.

4.2 Negative Test Sanitisation

The code in Listing 2 from the *spring-core* project is an example of a defensive API practice in `org.springframework.util.Assert`. A runtime exception is used to signal a violated pre-condition, a `null` parameter in this case. The exception (`IllegalArgumentException`) is thrown in the `Assert::notNull` utility method. While a null pointer exception is also a runtime exception, throwing an `IllegalArgumentException` here is more meaningful as this is (expected to be) thrown by the application, not by the JVM, and clearly communicates to clients that this is a problem caused by how an API is used, as opposed to an exception caused by a bug within the library.

```
public static void isInstanceOf(Class<?> type, @Nullable Object obj, String message) {
  notNull(type, "Type to check against must not be null");
  ..
}
```

Listing 2 A defensive API in *spring-core*, `org.springframework.util.Assert::isInstanceOf`

This contract is then tested in `org.springframework.util.AssertTests::isInstanceOfWithNullType`, shown in Listing 3.

```
@Test void isInstanceOfWithNullType() {
  assertThatIllegalArgumentException().isThrownBy(
    () -> Assert.isInstanceOf(null, "foo", "enigma")
  ).withMessageContaining(..);
}
```

Listing 3 Testing a defensive API in *spring-core* with JUnit5

We refer to such tests as *negative tests* – i.e. tests that exercise abnormal and unintended but possible behaviour, and use an exception or error as the test oracle for this purpose. Features often used to implement such tests are the `assertThrows` method in JUnit5, and the `expected` attribute of the `@Test` annotation in JUnit4.

Including such tests (as drivers) is likely to result in false positives – nulls are passed to the test to trigger defense mechanisms, such as runtime checks. We therefore excluded issues triggered by such tests. This is done by a lightweight ASM-based static analysis that checks for the annotations and call sites indicating the presence of an exception oracle and produces a list of negative tests, and a second analysis that cross-references the context information captured while recording issues against this list, and removes issues triggered by negative tests.

The analysis checks for the above-mentioned negative test patterns in JUnit4 and JUnit5, and a similar pattern in the popular *assertj* library. Finally, the analysis looks for call sites of methods in `com.google.common.testing.NullPointerTester`. This is a utility that uses reflection to call method with null for parameters not marked as nullable, expecting a NPE or an `UnsupportedOperationException` being thrown. This may be considered as over-fitting as *guava* is also part of our data set used for evaluation. However, like JUnit, *guava* is a widely used utility library, which warrants supporting this features in a generic tool.

4.3 Shaded Dependency Sanitisation

The return type of `org.springframework.asm.ClassVisitor::visitMethod` is not annotated as nullable. The problem here is that *spring-core* also defines several subclasses of this class overriding this method (including `SimpleAnnotationMetadataReadingVisitor`, package name omitted for brevity), which mark the return type as nullable. Reading this as pluggable types with the non-null by default assumption, with `@Nullable MethodVisitor` being a subtype of `MethodVisitor`, this violates Liskov's substitution principle [42] as the postcondition of a non-null return value is weakened in the overriding method.

The reason that engineers wont add the annotation is that this class originates from a shaded dependency.[12] Shading is a common practice were library classes and often entire package or even libraries are inlined, i.e. copied into the project and relocated into new name spaces. A common use case is to avoid classpath conflicts when multiple versions of the same class are (expected to be) present in a project. This is usually not done manually, but automated using build plugins such as *maven-shade-plugin*. The respective section of the Gradle build script for *spring-core* is shown in Listing 4.

---

[12] See pull request (URL tba after double-blind review)

```
task cglibRepackJar(type: ShadowJar) {
 archiveBaseName.set('spring-cglib-repack')
 archiveVersion.set(cglibVersion)
 configurations = [project.configurations.cglib]
 relocate 'net.sf.cglib', 'org.springframework.cglib'
 relocate 'org.objectweb.asm', 'org.springframework.asm'
}
```

Listing 4 Shading spec in *spring-core*`spring-core.gradle`

This makes adding `@Nullable` annotations for those classes almost useless, and the developer effort to add them is wasted as the source code is replaced during each build. A possible solution would be to add annotations during code generation at build time, but to the best of our knowledge, there are no suitable tools or meta programming techniques readily available to engineers that could be used for this purpose.

A sanitiser to take this into account takes a list of packages corresponding to shaded classes as input, and removes issues detected within those classes.

4.4 Deprecation Sanitisation

The final sanitiser removes issues collected from deprecated (i.e., annotated with `@java.lang.Deprecated`) fields, methods or classes. The rationale is that given the significant cost of annotating code, engineers might be reluctant to add annotations to code scheduled for removal, and will consider the inference of such annotations less useful. Such a sanitiser can be implemented with a straightforward byte code analysis as `@Deprecated` annotations are retained in byte code. We used ASM for this purpose in our proof-of-concept implementation.

4.5 Discussion: Sanitisation by Package-wide Default Nullability Assumption

There are various other possible sanitisers we have considered. A particular interesting scenario is the use of package-wide annotations setting defaults. As briefly discussed in Section 1, the *spring framework* uses package annotations to declare the non-null-by-default assumption for entire packages. Interestingly, those annotations are not used for all packages.

This raises the question whether nullability issues discovered in packages not annotated with those annotations should be sanitised. It is however not clear what the rationale of not having those annotations is, and what should replace non-null-by-default. What is more, this is an issue specific to the *spring* project, using special annotations defined *within* in *spring*. For some of the relatively few unannotated packages in *spring*, not having those annotations merely states that they are not applicable.

For instance, *spring-core* is the module in the data set used in the evaluation with the highest number of unannotated packages. It consists of 35 packages, 7 of those (20%) do not use the `@NonNullApi` and `@NonNullFields` package annotations. Of those, `org.springframework.lang` only defines annotation types without methods or fields that could be annotated with `@Nullable`, and 5 more packages (`org.springframework.asm`, `org.springframework.cglib.*`) are the result of shading, as discussed in Section 4.3 and therefore, potential false positives are being removed by the shaded dependency sanitisation. This only leaves one non-annotated package `org.springframework.objenesis`, and this package only contains a single class `SpringObjenesis`. This class does define methods (constructors) and some of the API elements appear to be nullable. It is not clear why this package has not been annotated.

For this reason, we believe that there is no sufficient justification to sanitise by (the lack of) package-wide nullability annotations.

5 Propagation

Annotating an API with `@Nullable` annotations changes the expectations and guarantees of the API contract with clients. In terms of Liskov's Substitution principle (LSP), adding `@Nullable` to the method (i.e., to the type it returns) weakens its postconditions if we consider `@NonNull` to be the baseline. To preserve LSP, the same annotation should therefore be applied to the overridden method.

For nullable arguments, the direction changes: while overriding a method making arguments nullable complies to LSP as expectations (for callers) are weakened, nullable arguments should not be made non-null in overridden methods. If we assume `@NonNull` to be the default, this implies that `@Nullable` should also applied to the arguments of the overriding method. However, the standard Java language semantics only supports covariant return types (e.g., methods can be overridden using a more specific return type), while for argument types invariance is used. Different null checkers and other languages use a variety of approaches here [13] and it is not completely clear what the canonical approach should be. Therefore, in our proof-of-concept implementation, LSP propagation can be customised to propagate nullability for arguments, or not, with propagation being the default strategy.

Listing 5 illustrates our approach. Assume we have annotated `B::foo` using observations from instrumented test runs. Then we also have to add `@Nullable` to the return type of the overridden method `A::foo` , and to the sole argument of the overriding method `C::foo`.

```
public class A {
   public @Nullable Object foo (Object arg) ;
}
public class B extends A {
   public @Nullable Object foo (@Nullable Object arg) ;
}
public class C extends B {
   public Object foo (@Nullable Object arg) ;
}
```

Listing 5 Propagation of `@Nullable` to Sub- and Supertypes

LSP propagation is implemented using a lightweight ASM-based analysis that extracts overrides relationships from compiled classes, and cross-references with with captured issues, creating new issues. For provenance, references to the original parent issues leading to inferred issues are captured as well and stored alongside the (JSON-serialised) inferred issues as a *parent* attribute.

5.1 Limitations

There is a limitation to hierarchy-based propagation — subtype relationships may extend across libraries, and we may infer nullable annotations for classes that are not in the scope of the analysis, and cannot be refactored. While project owners know super types (and can use methods like opening issues or creating pull requests for projects we don't control), they are not in control of subtypes in an open world, and rely that downstream projects would eventually pick up those annotations through notifications from some static analyses tools checking for those issues.

5.2 Sanitisation vs Propagation Fixpoint

Sanitisation and propagation have opposite effects. Preferably, an algorithm used to refine the initially collected nullability issues would reach a unique fix point where the future application of sanitisation and propagation would not change the set of refined nullability issues. However, such a fixpoint does not exist. Consider for instance a scenario where a shaded class has a method that is overridden and has a nullable return type in the overriding method. Then LSP propagation suggests to also add this to the return of the overridden method in the super class (to avoid weakening the post conditions), while sanitisation suggests not to refactor the shaded class. This is the issue we have observed in *spring-core* and discussed in Section 4.3.

6 Annotation Injection

We implemented a tool to inject the inferred annotations into projects, using the following steps:

1. compilation units are parsed into ASTs using the *javaparser* API [63]
2. for each nullable issue, the respective method arguments, returns or fields are annotated by adding nodes representing the `@Nullable` annotation to the respective AST
3. after the AST for a compilation unit is processed, it is written out as a Java source code file
4. if necessary, the respective import for the nullable annotation type used is added to the `pom.xml` project file

The tool has been evaluated using standard JUnit unit tests, and by round-tripping (removing and then reinserting existing annotations) the spring projects studied.

6.1 Annotation Abstraction

There are different annotation libraries available defining nullable annotations, and static checkers often support multiple such annotations. For this reason, the annotator tool supports pluggable annotations. This abstraction is implemented as a `NullableAnnotationProvider` service, implementations provide the nullable type and package names, and the coordinates of an Maven artifact providing the respective annotation. The default implementation is based on JSR305. Alternative providers can be deployed using the standard Java service loader mechanism.

7 Evaluation

Our evaluation is based on a study of some of the popular real-word projects which have been manually null-annotated by project members. We compare those existing annotations with the annotations captured and inferred by our method, and check those two sets for consistency. This is done by measuring *precision* and *recall*. Informally, those measures represent the ratio of inferred annotations to existing annotations, and the percentage of existing annotations our method is able to infer. More precisely, given a set of existing nullable annotations $Existing$ and a set of annotations inferred using our method $Inferred$, we define the following metrics:

$TP := Existing \cap Inferred$
$FP := Inferred \setminus Existing$
$FN := Existing \setminus Inferred$
$precision := |TP|/(|TP| + |FP|)$
$recall := |TP|/(|TP| + |FN|)$

Those are standard definitions, however, they need to be used with caution here. The concepts suggest that the existing annotations are *the ground truth.* This hinges on two assumptions: **1.** The existing annotations are complete. **2.** The project test cases provide enough coverage to exercise all possible nullable behaviour.

The first assumption means that all exiting nullable annotations our method fails to infer are in fact false positives. This might not be true as the annotations may not be complete, and we explore this issue further in Section 7.8. Therefore, the precision reported needs to be understood as the *lower precision bound (lpb)* in the sense of false positive detection. The second assumption means that all existing issues our tool cannot detect are false negatives. While this is correct in some sense, it does not necessarily indicate a weakness of our method as such, rather than an issue of the quality of input data, i.e. the quality of tests.

Existing annotations are extracted by using a simple byte code analysis (noting that common nullable annotation use runtime retention), we are looking for `@Nullable` annotations in any package to account for the multiple annotation providers. We also support two semantically closely related annotations defined in widely used utility libraries or tools, *guava's* `@ParametricNullness` and *findbug's* `@CheckForNull`.

7.1 Dataset

The data set we use in our study consists of seven projects (modules) from the *spring framework* ecosystem, plus two additional google projects. Those projects were located by searching the Maven repository for projects using libraries providing `@Nullable` annotations, and the selecting projects that actually use a significant number of those annotations. The reason that we chose this method was that we wanted to use existing annotations as (an approximation) of ground truth to evaluate the inferred annotations. We were particularly looking for projects backed by large engineering teams and well-resourced organisations, assuming that this would result in high-quality annotations.

Spring is the dominating framework for enterprise computing in Java [69], it is supported by a large developer community, is almost 20 years old and keeps on maintaining and innovating its code base. What makes those projects particularly suitable for evaluation is the fact that they have been manually annotated with `@Nullable` annotations. Spring defines its own annotation for this purpose in *spring-core* [13]. The amount of annotations found in those projects is extensive, see Section 7.4 for details.

Spring is organised in modules, projects with their own build scripts producing independent deployable binaries. We selected seven projects with different characteristics in particular with respect to how APIs are provided or consumed: *core*, *beans* and *context* are foundational projects for the spring framework overall, with few dependencies. *orm* and *oxm* are middleware components for applications to interact with XML data and relational databases, and integrate with existing frameworks for this purpose like *hibernate*, *jpa* and *jaxb*. Finally, *web* is a utility

---

[13] Defined in in `org.springframework.lang`

		main			test			
program	version	java	kotlin	groovy	java	kotlin	groovy	coverage
s.-beans	5.3.22	301	2	1	126	4	0	60%
s.-context	5.3.22	640	5	0	483	7	2	63%
s.-core	5.3.22	499	1	0	214	14	0	66%
s.-orm	5.3.22	72	0	0	32	0	0	39%
s.-oxm	5.3.22	31	0	0	19	0	0	58%
s.-web	5.3.22	653	1	0	268	5	0	18%
s.-webmvc	5.3.22	368	3	0	225	5	0	39%
guava	31.1	619	0	0	502	0	0	70%
error-prone	2.18.0	745	0	0	1,222	0	0	73%

Table 1 project summary, reporting the number of Java, Kotlin and Groovy source code files for both main and test scope, and branch coverage

library for web programming (including an HTTP client), and *webmvc* is a comprehensive application framework based on the model-view-controller design pattern [30].

We also include two additional non-spring programs to demonstrate the generality of the method proposed, and avoid over-fitting for spring. Those are *guava* and *error-prone*, both by google. *Guava* is a very popular utility library, whereas *error-prone* is a code analysis utility, similar to *findbugs*. Those two projects also use Maven as build system, and have a modular structure, with some modules only containing tests, test tools or annotations. We analysed nullability for the *errorprone/core* and *guava/guava* modules, respectively.

Table 1 provides an overview of the data set used together with some metrics, broken down by scope as discussed in Section 4.1. While those projects predominately contain Java classes, they also contain a smaller amount of Kotlin and Groovy code. Most of this are tests, and as the capture is based on bytecode instrumentation, those tests are still being used as drivers for the dynamic analysis. The table also contains some coverage data.[14] This provides some indication that the projects detected are well tested, and provide reasonable drivers for a dynamic analysis. The coverage data compares favourably to the coverage observed for typical Java programs [18].

7.2 Capture

For the dynamic analysis, we used the agents described in Section 3. With those agents deployed in the build scripts, ground truth extraction is a matter of running the projects builds using the test targets. The agents collect large amounts of data. For instance, the raw uncompressed size of the nullability issue file collected is 19.96 GB for *spring-context*, 4.11 GB for *guava* and 3.57 GB for *error-prone* (see also Table 2). To avoid memory leaks caused by instrumentation, agents dump data frequently, and after test execution using a shutdown hook.

Not unexpectedly, the presence of the agents significantly prolongs the build times – to around one hour for *spring* and 16 hours for *guava* [15]. We argue that this is acceptable as this is an one-off effort, i.e. this is not designed to be integrated into standard builds.

---

[14] Branch coverage is reported, calculated using the *jacoco* coverage tool integrated into the IntelliJ IDEA 2022.2 (Ultimate Edition) IDE, and reporting the values aggregated by IntelliJ for the respective packages

[15] Builds were run on a MacBook Pro (16-inch, 2021) with Apple M1 Pro, and OpenJDK 11.0.11

program	ex	obs	agg	agg/obs	r,lpb
s.-beans	1,290	321,851	1,320	0.0041	0.54,0.52
s.-context	1,435	6,872,413	5,945	0.0009	0.49,0.12
s.-core	1,510	175,725	1,171	0.0067	0.52,0.67
s.-orm	377	3,443	279	0.0810	0.47,0.63
s.-oxm	84	501	64	0.1277	0.54,0.70
s.-web	2,025	127,882	1,656	0.0129	0.45,0.55
s.-webmvc	1,437	192,800	2,392	0.0124	0.69,0.41
guava	3,993	2,708,816	4,923	0.0018	0.48,0.39
error-prone	507	1,095,752	1,736	0.0016	0.39,0.11

Table 2 RQ1 - existing (ex) vs observed (obs) issues, also reported are the aggregation of observed issues (agg), aggregation ratios (agg/obs) and recall / lower precision bound (r,lpb)

7.3 Research Questions

We break down the evaluation into a number of research questions. RQ1 compares the possible nullable annotations collected from instrumented test runs with existing annotations. RQ2 and RQ3 assess the utility of the refinements (sanitisation and propagation) performed on the nullability issues collected to improve recall and precision. Finally, in RQ4 we assess the interaction between sanitisation and propagation.

RQ1 How does nullability observed during test execution compare to existing `@Nullable` annotations?

RQ2 Can sanitisation techniques improve the precision of `@Nullable` annotation inference?

RQ3 Can propagation improve the recall of `@Nullable` annotation inference?

RQ4 Does the repeated application of sanitisation and propagation reach a fixpoint?

7.4 How does nullability observed during test execution compare to existing @Nullable annotations ? [RQ1]

The data to answer this RQ are presented in Table 2. Column 2 (ex) contains the number of `@Nullable` annotations found in the respective program (existing `@Nullable` annotations are extracted and also represented as *extracted issues* to facilitate comparison), column 3 (obs) shows the number of `@Nullable` issues observed during the execution of instrumented tests, corresponding to inferred `@Nullable` annotations. The number of observed issues is surprisingly large, but often, multiple nullability issues are reported for the same field, method parameter or method return. To take this into account, we also report the aggregated issues resulting from deduplication as discussed in Section 3.5 in column 4 (agg), and the aggregation ratio (agg/obs) in column 5. This demonstrates that deduplication is very effective. I.e., nullability reported for a given field, method return or parameter is usually supported by different tests, resulting in different contexts. We see this as a strength of our methods as each context provides independent support for the nullability that is being detected. Finally, we report recall and lower precision bound (r,lpb) in column 6. Both are around 50% with two notable exceptions – the significantly lower recall for *spring-core*, and the significantly lower precision for *spring-context* and *error-prone*.

These results suggests that inferring nullability issues dynamically by only observing tests is not sufficient, and further refinement of those results by means of additional analyses is needed.

program	base	san(D)	san(M)	san(N)	san(S)	san(all)
s.-beans	1,320	1,298	763	1,247	1,320	687
s.-context	5,945	5,922	788	5,662	5,682	718
s.-core	1,171	1,140	999	1,024	1,124	780
s.-orm	279	279	192	270	279	184
s.-oxm	64	64	49	64	64	49
s.-web	1,656	1,606	1,076	1,544	1,656	941
s.-webmvc	2,392	2,374	1,076	2,327	2,392	1,048
guava	4,923	4,813	4,008	3,384	4,923	2,464
error-prone	1,736	1,736	1,337	1,736	1,736	1,337

Table 3 RQ2a – observed issues after applying sanitisers (base – no sanitisation applied, D - deprecation, M - main scope, N - negative tests, S - shading)

program	r,lpb(D)	r,lpb(M)	r,lpb(N)	r,lpb(S)	r,lpb(all)
s.-beans	0.52,0.52	0.54,0.91	0.52,0.53	0.54,0.52	0.50,0.95
s.-context	0.48,0.12	0.49,0.90	0.48,0.12	0.49,0.12	0.47,0.94
s.-core	0.50,0.67	0.52,0.78	0.49,0.72	0.52,0.70	0.47,0.92
s.-orm	0.47,0.63	0.47,0.92	0.45,0.63	0.47,0.63	0.45,0.93
s.-oxm	0.54,0.70	0.54,0.92	0.54,0.70	0.54,0.70	0.54,0.92
s.-web	0.43,0.54	0.45,0.85	0.44,0.57	0.45,0.55	0.42,0.90
s.-webmvc	0.68,0.41	0.69,0.92	0.68,0.42	0.69,0.41	0.67,0.92
guava	0.48,0.40	0.48,0.48	0.48,0.56	0.48,0.39	0.48,0.77
error-prone	0.39,0.11	0.39,0.15	0.39,0.11	0.39,0.11	0.39,0.15

Table 4 RQ2b – recall and lower precision bound (r,lpb) w.r.t. existing annotations after applying sanitisers (D - deprecation, M - main scope, N - negative tests, S - shading)

7.5 Can sanitisation techniques improve the precision of @Nullable annotation inference ? [RQ2]

The various sanitisation techniques discussed in Section 4 address potential false positives. To evaluate their impact, we applied the sanitisers to the observed nullability issues for each program in the data set, and report the number of aggregated inferred nullability issues after santitisation. We also report the results of applying all sanitisers. The absolute numbers are reported in Table 3, the recall / precision metrics are reported in Table 4.

The results suggest that most sanitisers have only a minor impact on precision and, sometimes, those improvements come at the price of slight drops in recall. However, one sanitiser stands out: by focusing on classes in the main scope, the precision can be improved dramatically. This suggests that our instrumented tests pick up a lot of nullability in test classes or other test-scoped classes supporting tests.

After applying all santisation techniques, we observe a very high lower precision bound of 0.9 or better for all *spring* programs, with some minor drops in recall. The lower precision boun for *guava* is still fairly high, but surprisingly low for *error-prone*, to be discussed below. Balancing precision and recall is a common issue when designing program analyses, but we believe that the focus should be on precision as developers have little tolerance for false alerts. For instance, it has been reported that "Google developers have a strong bias to ignore static analysis, and any false positives or poor reporting give them a justification for inaction." [60].

To investigate the low lower precision bound we observed for *error-prone* further, we conducted an additional experiment where we calculated the *annotation ratio*. For this purpose, we counted the existing `@Nullable` annotations, and the number of program elements that can be annotated, i.e. fields, method parameters and return types for non-synthetic methods and fields whose type is not a primitive type. The results are displayed in Table 5. This show that the annotation ratio for *error-prone* is by on order of a magnitude lower than for the other programs. Therefore, many of the potential false positives are likely to be true positives, and the existing annotations are not suitable to act as a ground truth

program	annotated	annotatable	annotation ratio	`Void` usage
s.-beans	1,290	5,230	0.25	0
s.-context	1,435	8,849	0.16	0
s.-core	1,510	10,628	0.14	0
s.-orm	377	1,676	0.22	0
s.-oxm	84	467	0.18	0
s.-web	2,025	13,658	0.15	6
s.-webmvc	1,437	8,317	0.17	1
guava	3,964	25,472	0.16	2
error-prone	507	22,669	0.02	958

Table 5 Annotated vs annotatable program elements, in the last column the number of annotatable elements of type `java.lang.Void` is reported

program	all	2	3	4	5	6	7	8	9	10	>10
s.-beans	687	167	109	64	58	46	35	24	22	39	123
s.-context	718	197	122	76	58	52	25	26	22	11	129
s.-core	780	266	165	105	63	37	32	23	21	10	58
s.-orm	184	23	28	20	18	14	24	2	3	3	49
s.-oxm	49	35	4	1	7	0	0	0	0	0	2
s.-web	941	305	258	149	77	52	10	8	2	9	71
s.-webmvc	1,048	329	195	212	117	50	32	12	9	10	82
guava	2,464	972	606	399	163	122	37	20	13	11	121
error-prone	1,337	8	23	56	4	26	4	8	6	2	1,200

Table 6 Observed and sanitised issues by context depths

here. To investigate the matter further, we looked for patterns amongst the potential false positives detected. One pattern stands out – the frequent use of `java.lang.Void` as method parameter and return type. The respective numbers are shown in Table 5, column 5. The use of `Void` in *error-prone* is unusually high. `Void` has an interesting semantics – this class cannot be instantiated, i.e. *it must be null*. However, in *error-prone*, the respective method returns and parameters are not annotated as `@Nullable`. Interestingly, this is in violation of one of *error-prone's* own rule *VoidMissingNullable (‘The type Void is not annotated @Nullable‘”)* [16]. I.e., *error-prone* is not *dog-fooding* [32] here. *Error-prone* has recently opened an issue to address this [17]. We also note that the *nullaway* checker treats `Void` as nullable [18], and the *checkerframework* declares `@Nullable` as default for `Void` using a meta annotation [19].

We rerun the recall and precision calculation against a ground truth that interprets `Void` as nullable, and for *error-prone* as expected the result change significantly to a recall of 0.72 and a lower precision bound of 0.79.

After performing sanitisation, we also investigated the context depth, i.e. the size of the stack traces recorded. Without sanitisation this data would be distorted by issues discovered in testing scope, leading to very low context depth. For each aggregated issue equivalence class modulo the deduplication relationship (see Section 3.5), we computed the lowest context depth for all issues in the respective equivalence class, and then counted aggregated issues by this depth. The results are reported in Table 6.

The results suggest that there are some issues revealed by trivial tests (e.g., tests directly invoking functions with `null` parameters). However, a significant number of issues is revealed

---

[16] `https://errorprone.info/bugpattern/VoidMissingNullable`

[17] `https://github.com/google/error-prone/issues/3792`

[18] `https://github.com/uber/NullAway/blob/master/nullaway/src/main/java/com/uber/nullaway/NullAway.java`, commit `https://github.com/uber/NullAway/commit/1548c69a27e9e3dd1cb185d04b2e870f3b11a771`

[19] `https://checkerframework.org/api/org/checkerframework/checker/nullness/qual/Nullable.html`

program	s	sp	r,sps	r,lpb(s)	r,lpb(sp)	r,lpb(sps)
s.-beans	687	693	693	0.50,0.95	0.51,0.95	0.51,0.95
s.-context	718	736	736	0.47,0.94	0.48,0.94	0.48,0.94
s.-core	780	791	788	0.47,0.92	0.48,0.91	0.48,0.92
s.-orm	184	184	184	0.45,0.93	0.45,0.93	0.45,0.93
s.-oxm	49	49	49	0.54,0.92	0.54,0.92	0.54,0.92
s.-web	941	949	949	0.42,0.90	0.42,0.90	0.42,0.90
s.-webmvc	1,048	1,059	1,059	0.67,0.92	0.68,0.92	0.68,0.92
guava	2,464	2,503	2,503	0.48,0.77	0.49,0.77	0.49,0.77
error-prone	1,337	1,361	1,361	0.39,0.15	0.43,0.16	0.43,0.16

Table 7 RQ3a – effect of propagation, aggergated issue counts and recall / lower precision bound for santitised issues (s), santitised and then propagated issues (sp) and santitised, propagated and resanitised issues (sps)

by more complex behaviour with deep calling contexts. We consider this to be a strengths of the analysis being presented. Note that the context depths are not inflated by boiler-plate code as the stack traces are cleaned during capture (see Section 3.3).

7.6 Can propagation improve the recall of @Nullable annotation inference ? [RQ3]

Next, we applied propagation to the sanitised nullability issues (using all sanitisers). This can discover additional nullability issues not observable during testing, and therefore improve recall. The results are reported in Table 7. Those results suggests that propagation does not significantly change the quality of the analysis. We observe minor improvements in recall for only four programs in our dataset.

As already discussed in Section 7.5 , the results for *error-prone* are heavily impacted by the fact that `Void` is not annotated as nullable. If we consider it as implicitly annotated as nullable, and extend the ground truth used to compare the inferred annotations accordingly, the results change to a recall of 0.73 and a lower precision bound of 0.79. We therefore observe a small increase of the recall for error-prone as the result of propagation.

7.7 Does the repeated application of sanitisation and propagation reach a fixpoint ? [RQ4]

Propagation can introduce new annotations which would otherwise be sanitised, and the process generally does not converge against a fix point. An example was already discussed in Section 5.2. However, it is still relevant question to study to quantify whether we come close to a fixpoint, and whether it is common for programs that there is no fixpoint. Therefore, we investigated whether this is a significant observable effect by applying sanitisation to the propagated inferred annotations. This had almost no effect, with only a very few issues in *spring-core* being re-sanitised, the respective data is reported in the columns labelled *sps (sanitised-propagated-sanitised)* in Table 7.

Since propagation is the last step of our inference pipeline (capture-sanitise-propagate), we also report a breakdown of nullability issues by program element annotated, as shown in Table 8. What stands out is that for fields both recall and precision of inferring nullability is better than average.

7.8 False False Positives

Despite the generally high precision our approach achieves, it is not perfect. The question arises whether this is caused by false positives. This relates to the fact that our baseline – the

program	prop(F)	prop(P)	prop(R)	r,lpb(F)	r,lpb(P)	r,lpb(R)
s.-beans	205	279	209	0.81,1.00	0.41,0.90	0.47,0.97
s.-context	308	220	208	0.80,0.98	0.34,0.91	0.41,0.90
s.-core	125	422	241	0.80,1.00	0.43,0.86	0.46,0.97
s.-orm	111	38	35	0.90,1.00	0.21,0.76	0.26,0.89
s.-oxm	35	12	2	0.70,1.00	0.45,0.83	0.00,0.00
s.-web	308	438	203	0.72,0.94	0.36,0.87	0.33,0.91
s.-webmvc	373	319	367	0.95,1.00	0.52,0.87	0.63,0.88
guava	353	1,474	676	0.88,0.98	0.42,0.68	0.48,0.87
error-prone	77	700	584	0.80,0.10	0.47,0.11	0.40,0.23

Table 8 RQ3b - number of propagated issues and recall / lower precision bound of propagated issues by type (F - field, P - method parameters, R - method return types)

existing `@Nullable` annotations, only (under-)approximates the ground truth. In particular, it is unclear whether it is complete. If it was not, some of the false positives our analysis produces would actually be true positives. Sometimes additional analyses can reveal patterns where developers missed annotations that should have been added by some heuristics, an example is the `Void` analysis for *error-prone* discussed in Section 7.5. If no such pattern can be identified, there is another way to find out – add additional annotations inferred by our tool to the respective project(s) via pull requests.

The number of annotations to be added is still relatively large, and given the importance spring has in the developer ecosystem, it can be expected that project owners are generally reluctant to accept pull requests from newcomers. Pull requests have also experienced some amount of inflation recently (partially caused by bots creating pull requests), and therefore processing is delayed.[20]

We have submitted two pull requests with different outcomes: PR1 [21] has resulted in a `@Nullable` annotation inferred being added [22]. PR2 [23] was rejected, but the developers refined the test the inference is based on [24] .

While PR1 and PR2 have resulted in different outcomes, they both have revealed issues in *spring*, and after rerunning the analysis after the action taking by developers in response to the PRs, precision would increase in both cases. Adding an inferred annotation clearly shows that some false positives are actually true positive. Refining the tests has a similar effect – the semantics of tests is sometimes at odds with what is considered intended behaviour, and our tools exposes this. After the test is fixed, the false positive disappears as the tool can no longer infer it.

7.9 Comparison with Purely Static Inference

Houdini [25] infers annotations using the Esc/Java checker. The platform has been deprecated and replaced by other tools, and there is no implementation available. Houdini still uses “pseudo-annotation“ using special markup. This approach is also highly unscalable. The authors report that *“the running time on the 36,000-line Cobalt program was 62 hours”*. For comparison, the version of *spring-core* used in the evaluation experiments alone contains over 146,000 lines of Java code, and checkers rarely scale linearly. For comparison, our analysis

---

[20] There were 164 open pull requests on 20 October 2022, `https://github.com/spring-projects/spring-framework/pulls?q=is%3Aopen`

[21] `https://github.com/spring-projects/spring-framework/pull/29150`

[22] `https://github.com/spring-projects/spring-framework/commit/35d379f9d3882a02f0368f928b2cecb975404334`

[23] `https://github.com/spring-projects/spring-framework/pull/29242`

[24] `https://github.com/spring-projects/spring-framework/commit/c14cbd07f449d845269c99faa29241e7e2d0dfc1`

Table 9 Comparing our approach with JastAddJ NonNull inference.

program	annotatable	@Nonnull	@Nullable	Intersection
commons-lang-3.0	4,647	1,480	1,041	633
commons-cli-3.1	2,724	1,179	65	17
commons-io-2.5	2,241	1,012	326	184
commons-math-3.0	9,404	3,208	270	50

generally scales. The bottleneck of our method is the capture, and while this is expensive it generally scales as discussed in Section 7.2.

We contacted the authors of several tools [21, 36, 35] and succeeded in using *jasaddj-nonnullinference* [21] to analyse some programs, and compare results.[25] The tool has been maintained until 2015, and based on advice by the authors, we selected some older programs buildable with Java 1.7. The builds had to be heavily customised in order to deal with broken dependencies, details are described in the artefact. The comparison is not straightforward as *jasaddj* infers `@Nonnull` annotations, whereas our method infers `@Nullable`.

The results are shown in Table 9. The *annotatable* column shows the total number of fields, method return and parameters with nullable types. The `@Nonnull` column show the number of annotations inferred by *jasaddj*, and the `@Nullable` columns shows the number of annotations our approach infers. We also report the intersection between both sets in the last column. Both approaches annotate less than half of all annotatable elements. It is not clear how to interpret the set complement for both tools. If we interpret everything not `@Nonnull` annotated by *jasaddj* as `@Nullable`, then *jasaddj* has a low precision. The intersection column suggests that there are a significant number of cases where the tools produce inconsistent results. Given the low number of false positive we observe with our tool, it is likely that *jasaddj* produces false positives here.

However, this is not really surprising given that tools like *jasaddj* have been designed to analyse program (as opposed to libraries), where all method calls and field access is known. Our method however is designed for an open world where API interactions from unknown clients have to be considered, and test cases act as proxies for those clients.

8 Related Work

Much work exists on the problem of eliminating null dereferences, of which the vast majority focuses on static checking. Nevertheless, a number of empirical studies exist which are relevant here. The early work of Chalin *et al.* empirically studied the ratio of parameter, return and field declarations which are intended to be non-null, concluding 2/3 are [13, 14]. Another early work was that of Li *et al.* who sampled hundreds of real-world bugs from two large open source projects [41]. They found (amongst other things) null dereferences are the most prevalent of memory-related bugs.

Kimura *et al.* argued that "*it is generally felt that a method returning null is costly to maintain*" [38]. Their study of several open source projects examined whether statements returning `null` or checks against `null` were modified more frequently than others and they observed a difference for the former (but not the latter). Furthermore, they found occurrences of developers replacing statements returning `null` with alternatives (e.g. Null Objects [29] or exceptions) suggesting a desire to move away from using `null` like this. Osman *et al.* also investigated null checks across a large number of open source programs [53]. They found the

---

[25] `https://bitbucket.org/jastadd/jastaddj-nonnullinference`

most common reason developers insert null checks is for method returns and, furthermore, that this is most often to signal errors. The follow-up work of Leuenberger *et al.* investigated the nullability of method returns in Apache Lucene (a widely-used search library) [40]. For each method call site (either internally within Lucene or externally across clients), they identified whether the method return was checked against `null` before being dereferenced (i.e. as this indicates whether the caller expected it could return `null` or not). They confirmed that most methods are expected to return non-null values. However, they also found that external clients were more likely to check a method against `null`, suggesting clients employ defensive behaviour (e.g. when documentation is missing, etc).

8.1 Migration

Dietrich et al. harvested lightweight contracts, such as `@NonNull` and `@Nullable` annotations, from real-world code bases [17]. Unfortunately, they found such annotations are rarely used in practice and that, instead, throwing `IllegalArgumentExceptions` and (to a lesser extent) use of Java `assert` remain predominant. This suggests a key problem faced by all tools for checking non-null annotations (such as those above) is that of annotating existing code bases.

Brotherston *et al.* aimed to simplify migration of existing code bases to use non-null annotations [9]. Their goal is to enable incremental migration of existing code bases to use non-null annotations. Here, developers begin by annotating the most important parts of their system and then slowly widen the net until, eventually, everything is covered. Their approach follows gradual typing [62] and divides programs into the *checked* and *unchecked* portions, such that null dereferences cannot occur in the former. To achieve this, runtime checks are added to unchecked code to prevent exceptions occurring within checked code (i.e. by forcing exceptions at the boundary between them). Such an approach is complementary to our work, and the two could be used together. For example, one might start by inferring annotations using our technique and, subsequently, shift to a gradual typing approach to manage parts where inferred annotations were insufficiently strong, or otherwise require manual intervention. Estep *et al.* further apply ideas of gradual typing to static analysis, using null-pointer analysis as an example [22]. They argue gradual null-pointer analysis hits a "sweet spot" by mixing static and dynamic analysis as needed. A key question they consider is *"why it is better to fail at runtime when passing a null value as a non-null annotated argument, instead of just relying on the upcoming null-pointer exception"*. In essence, they provide two answers: (1) for languages such as C, null dereferences lead to undefined behaviour and, hence, catching them in a controlled fashion is critical; (2) for others, such as Java, it is generally better practice to catch errors as early as possible. Neito *et al.* also take inspiration from gradual typing by considering *blame* across language interop boundaries [51]. In particular, when null-safe languages (e.g. Scala or Kotlin) interact with unsafe languages (e.g. Java), problems can arise.

Houdini statically infers a range of annotations (including non-null) for Java programs [25]. The tool works by generating a large number of candidate annotations and using an existing (modular) checker to eliminate spurious ones. Ekman *et al.* also developed a tool for inferring non-null annotations which could identify roughly 70% of dereferences as safe [21]. Hubert *et al.* formalised an inference tool for non-null annotations based on pointer analysis [36, 35], whilst Spoto developed a similar system arguing it is faster and more precise in practice [64]. XYLEM employs a backwards analysis to find null dereferences [50]. Whilst it doesn't (strictly speaking) infer annotations, it could be modified to do so. Bouaziz *et al.* also propose a backwards analysis to infer *necessary field conditions* on objects (e.g. that a field is non-null) [7]. This approach is *demand driven* in the sense that fields are marked non-null only if this is necessary to prohibit a null dereference being reported elsewhere.

Finally, inference tools have been developed for pluggable type systems [26, 27, 15, 16]. However, such tools typically cannot account for null checks in conditionals making them relatively imprecise in this context.

8.2 Static Checking

Many tools for statically checking non-null annotations have been proposed. Typically, they differ from traditional type checkers by operating *flow-sensitively* to account for conditional null checks. They also assume non-null annotations have already been added to programs. NullAway provides a nice example here, since it was developed by Uber for static non-null checking at scale [5]. The key requirement was that it could run on all builds, rather than just at code review time (as for a previous tool they used). Their tool is flow-sensitive, but often takes an "optimistic" view (i.e. is unsound). Their reasoning is that sound (i.e. pessimistic) tools produce too many false positives. NullAway does not soundly handle initialisation (see below); likewise, for external (unannotated) code it assumes all interactions are safe. Despite this, they found no cases where unsoundness lead to actual bugs across a 30-day period of usage on a real-world code base. Indeed, this corroborates the earlier findings of Ayewah and Pugh who argued many null dereferences reported by tools do not actually materialise as bugs in practice [4]. As another example, *Eradicate* is part of Facebook Infer [1, 19, 11] and, in many ways, is similar to NullAway.

A number of other tools have been developed which can be used for static `@NonNull` checking, such as FindBugs [34, 33], ESC/Java [24], JastAdd [21], JACK [46] and more [57, 45]. Almost all of these employ flow-sensitive analysis, and many are unsound in various ways (e.g. support for initialisation). Indeed, the initialisation problem has proved so challenging that a large number of works are devoted almost exclusively to its solution [23, 37, 58, 67, 65, 61, 43, 44, 39]. Roughly speaking, the issue is that fields of reference type are assigned a default value of `null` and, thus, every `@NonNull` field initially holds `null` (and this is observable [67]). In our approach we check nullability at the end of object construction. This method is unsound only if super constructors allow access to fields defined in subclasses. We think that this is a rare programming pattern, and note that our approach while aiming for high recall, does not guarantee soundness anyway as it is based on a dynamic analysis.

Finally, so-called "pluggable type systems" [8] allow optional type systems to be layered on existing languages, thus allowing them to evolve independently [26, 27, 15, 3, 16, 48]. The *checkers framework* provides a prominent example which heavily influenced JSR308 (included in Java 8) [55]. A key advantage of this tool over others is the ability to support for flow-sensitive type systems (a.k.a. *flow typing* [56]). Indeed, without this feature checking non-null types is largely impractical [3].

9 Conclusion

We have presented a hybrid analysis pipeline that can be used to capture and refine nullability issues and mechanically inject inferred `@Nullable` annotations into Java programs. Our experiments on some of the most widely used Java commodity libraries demonstrates that this approach is suitable for real-world programs, and that the inferred annotations are consistent with annotations manually added by engineers. In particular, our approach has high precision, and there is evidence from pull requests we have initiated that this precision is potentially higher as our analysis is able to discover missing annotations in the already nullable-annotated programs we have used for evaluation.

Mechanising this process addresses a major issues in real-world projects: the lack of null annotations. Such annotations are part of the program semantics, and generally the annotation process requires deep understanding by project owners and contributers. However, the workload of adding such annotations is significant, and the lack of annotations compromises the utility of static checkers. We have argued that the semantics of which types are nullable and not is already at least partially encoded in existing test cases, and our pipeline exploits this idea of leveraging tests.

The tool has been open sourced and is available at < URL tbc after double-blind review>.

10 Acknowledgments

Removed for double-blind review

References

1 A tool to detect bugs in java and c/c++/objective-c code before it ships. https://fbinfer.com/.

2 JetBrain Developer Ecosystem Survey 2021, 2021. `https://www.jetbrains.com/lp/devecosystem-2021/java/`.

3 C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, pages 57–74, 2006.

4 Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *Proc. PASTE*, pages 65–72. ACM Press, 2010.

5 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proc. ESEC/FSE'19*, pages 740–750. ACM, 2019.

6 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE'11*, pages 241–250. IEEE, 2011.

7 Mehdi Bouaziz, Francesco Logozzo, and Manuel Fähndrich. Inference of necessary field conditions with abstract interpretation. In *Proc. APLAS*, pages 173–189. Springer-Verlag, 2012.

8 Gilad Bracha. Pluggable type systems. In *Proc. Workshop on Revival of Dynamic Languages*, 2004.

9 Dan Brotherston, Werner Dietl, and Ondrej Lhoták. Granullar: gradual nullable types for java. In *Proc. CC*, pages 87–97. ACM Press, 2017.

10 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

11 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *Proc. NFM*, pages 3–11. Springer-Verlag, 2015.

12 Giuseppe Castagna. Programming with union, intersection, and negation types. abs/2111.03354, 2021.

13 Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.

14 Patrice Chalin, Perry R James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software*, 2(6):515–531, 2008.

15 Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.

16 Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.

17 Jens Dietrich, David J Pearce, Kamil Jezek, and Premek Brada. Contracts in the wild: A study of java programs. In *Proc. ECOOP'17*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

18 Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. Xcorpus – an executable corpus of java programs. *Journal of Object Technology*, 16(4):1:1–24, August 2017. URL: `http://www.jot.fm/contents/issue_2017_04/article1.html`, `doi:10.5381/jot.2017.16.4.a1`.

19 Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *CACM*, 62(8):62–70, 2019.

20 Kinga Dobolyi and Westley Weimer. Changing Java's semantics for handling null pointer exceptions. In *Proc. ISSRE*, pages 47–56. IEEE Computer Society, 2008.

21 T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

22 Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. Gradual program analysis for null pointers. pages 3:1–3:25, 2021.

23 Manuel Fähndrich and K Rustan M Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312, 2003.

24 C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

25 Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.

26 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203, 1999.

27 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12, 2002.

28 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. FSE'11*, pages 416–419, 2011.

29 Maria Anna G. Gaitani, Vassilis Zafeiris, N. A. Diamantidis, and Emmanouel A. Giakoumakis. Automated refactoring to the null object design pattern. *Inf. Softw. Technol*, 59, 2015.

30 Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

31 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. In *Proc. OOPSLA'17*, pages 1–27. ACM, 2017.

32 Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006.

33 Martin Hofmann and Mariela Pavlova. Elimination of ghost variables in program logics. volume 4912 of *LNCS*, pages 1–20. sv, 2007.

34 David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proc. PASTE*, pages 13–19. ACM Press, 2005.

35 Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 36–42, 2008.

36 Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the International conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 132–149. Springer-Verlag, 2008.

37 Laurent Hubert and David Pichardie. Soundly handling static fields: Issues, semantics and analysis. *ENTCS*, 253(5):15–30, 2009.

38 Shuhei Kimura, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Does return null matter? In *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 244–253. IEEE, 2014.

39 Alexander Kogtenkov. Practical void safety. In *Proc. VSTTE*, pages 132–151. Springer-Verlag, 2017.

40 Manuel Leuenberger, Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. Harvesting the wisdom of the crowd to infer method nullness in java. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–80. IEEE, 2017.

41 Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33. ACM Press, 2006.

42 Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

43 Fengyun Liu, Ondrej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. A type-and-effect system for object initialization. In *Proc. OOPSLA*, pages 175:1–175:28, 2020.

44 Fengyun Liu, Ondrej Lhoták, Enze Xing, and Nguyen Cao Pham. Safe object initialization, abstractly. In *Proceedings of the Symposium on Scala*, pages 33–43. ACM Press, 2021.

45 Magnus Madsen and Jaco van de Pol. Relational nullable types with boolean unification. pages 1–28, 2021.

46 C. Male, D.J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

47 Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node. js libraries. In *proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

48 Ana Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *Proc. ESEC/FSE*, page 26. ACM Press, 2012.

49 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *PACMPL*, 2(OOPSLA):112:1–112:29, 2018.

50 Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *2009 IEEE 31st International Conference on Software Engineering*, pages 133–143. IEEE, 2009.

51 Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondrej Lhoták. Blame for null. In *Proc. ECOOP*, pages 3:1–3:28, 2020.

52 Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In *Proc. ECOOP*, volume 166, pages 25:1–25:26, 2020.

53 Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. Tracking null checks in open-source java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 304–313. IEEE, 2016.

54 Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Proc. OOPSLA'07*, pages 815–816. ACM, 2007.

55 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proc. ISSTA'08*, pages 201–212. ACM, 2008.

56 D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.

57 P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. CC*, pages 334–554, 2001.

58 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proc. POPL*, page 53–65. ACM Press, 2009.

59 Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. Union Types with Disjoint Switches. In *Proc. ECOOP*, volume 222, pages 25:1–25:31, 2022.

60 Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

61 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In *Proc. ECOOP*, volume 7920, pages 205–229. Springer-Verlag, 2013.
62 Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proc. ECOOP*, pages 151–175. Springer-Verlag, 2007.
63 Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub, oct. de*, 2017.
64 Fausto Spoto. Nullness analysis in boolean form. In *Proc. SEFM*, pages 21–30. IEEE, 2008.
65 Fausto Spoto and Michael D. Ernst. Inference of field initialization. In *Proc. ICSE*, pages 231–240. ACM Press, 2011.
66 Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proc. ICSE'20*, pages 1049–1060. IEEE, 2020.
67 Alexander J. Summers and Peter Mueller. Freedom before commitment: A lightweight type system for object initialisation. In *Proc. OOPSLA*, page 1013–1032, 2011.
68 Timothy A. V. Teatro, J. Mikael Eklund, and Ruth Milman. Maybe and either monads in plain c++ 17. In *Proc. CCECE*, pages 1–4. IEEE, 2018.
69 Brian Vermeer. Spring dominates the java ecosystem with 60% using it for their main applications, 2020. `https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/`.
70 Rafael Winterhalter. Byte buddy – a code generation and manipulation library for creating and modifying java classes during the runtime, 2014. `https://bytebuddy.net/`.