On Leveraging Tests to Infer Nullable Annotations

Jens Dietrich
Victoria University of Wellington, New Zealand

David J. Pearce
ConsenSys, Wellington, New Zealand

Mahin Chandramohan
Oracle Labs, Brisbane, Australia

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 (sanitized 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 and its engineering → Software defect analysis; Software and its engineering → Software reliability; Software and its engineering → Dynamic analysis

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

Funding Jens Dietrich: The first author was supported by Oracle Labs, Australia.

Acknowledgements The authors would like to thank Chris Povirk for his feedback on using our tool on guava, and Görel Hedin for assisting us to set up the experiment reported in Section 7.9.

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 [55, 48, 51, 58]; mitigating the impact of null in existing programs [23, 66, 5, 19]; and, developing alternatives for languages stuck with null [20, 29, 67].

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 [9]. 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 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) [41].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*.

---

¹ Open as of 20 October 22, see https://issues.apache.org/jira/browse/LANG-1598
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/google/common/base/Preconditions.html
⁷ https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html
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. 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 [7, 31]. A common reason to use those approaches is to boost recall [65].

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.

Our analysis is sound in the sense that it will not infer or add `@NonNull` to a method or field where it may become inconsistent with the runtime behaviour of the program. It is conservative in the sense that it will never retract `@Nullable` annotations added by developers.

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, available from [https://github.com/jensdietrich/null-annotation-inference](https://github.com/jensdietrich/null-annotation-inference)

These contributions directly relate to concrete research questions which we study in the context of evaluation experiments in Section 7.

Our approach meets the expectations of engineers for tools with high precision [6, 59], and has clear economic benefits – it can partially automate the expensive task of manually annotating code. At the time of writing, some of the results produced by our tool have already been adapted into spring and guava.

---

\(^8\) I.e., annotation used in `package-info.java`

\(^9\) [https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/NonNullApi.html](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/NonNullApi.html)
The paper is organised as follows. Starting with the introduction in this section, we provide a high-level overview of our approach in Section 2. This is followed by a more detailed discussion of the major steps of our method: the capture of potential nullability issues from the execution of instrumented tests (Section 3), the removal of likely false positives through sanitisation (Section 4), and inference of additional issues to address false negatives through propagation (Section 5). We then briefly discuss a utility to inject the annotations our tool infers back into programs in Section 6, and present evaluation experiments in Section 7. This includes the formulation of four research questions in Section 7.3. Finally, we discuss related work in Section 8, and finish with a brief conclusion in Section 9.

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 [41], 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.

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 [46]. The advantage is that clients can be identified mechanically using an analysis of dependency graphs exposed by package managers and the respective repositories. 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.

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
In principle, synthesised tests [53, 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 [11] and bytebuddy [70]. If null is encountered, nullability issues are created and made persistent (written to disk).

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).

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:
On Leveraging Tests to Infer Nullable Annotations

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
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`.

```json
```

### 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 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 [65].

---

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`. 
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 directory 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.

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

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

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

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

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

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
On Leveraging Tests to Infer Nullable Annotations

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. We also check for call sites for the JUnit4 and JUnit5 `fail` methods in try blocks, usually indicating that tests pass if they enter the corresponding catch block. Finally, the analysis looks for call sites of methods in `com.google.common.testing.NullPointerTester`. This is a utility that uses reflection to call methods 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

Shading is a common practice where 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 [69].

For instance, 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 the 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 [41] 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 This is usually not done manually, but automated using build plugins such as Maven’s `shade plugin` 13. The respective section of the Gradle build script for `spring-core` is shown in Listing 4.

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

1 task cglibRepackJar(type: ShadowJar) {
2     archiveBaseName.set('spring-cglib-repack')
3     archiveVersion.set(cglbVersion)
4     configurations += [project.configurations.cglib]
5     relocate 'net.sf.cglib', 'org.springframework.cglib'
6     relocate 'org.objectweb.asm', 'org.springframework.asm'
7 }
```

This makes adding `@Nullable` annotations for those classes unattractive – any developer effort to add them is wasted as the source code is replaced during each build, and modern projects are rebuilt often, sometimes multiple times a day. 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.

Another option would be to add the annotation to the respective method provided by the shaded library, however, engineers are usually not in a position to make such a change.

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.

---

12 See [https://github.com/spring-projects/spring-framework/pull/28852](https://github.com/spring-projects/spring-framework/pull/28852) for discussion

4.4 Deprecation Sanitisation

The final sanitiser removes issues collected from deprecated (i.e., annotated with `@java.lang.-Depreciated`) 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 `@Depreciated` annotations are retained in byte code. We used `asm` for this purpose in our implementation.

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 a method (i.e., to the type it returns) weakens its postconditions if we consider `@NonNull` to be the baseline. To preserve LSP, a non-null return type must not be made nullable in a overriding 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 must not be made non-null in overridden methods. If we assume `@NonNull` to be the default, this implies that `@Nullable` should also be 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 (if argument propagation is enabled) to the sole argument of the overriding method `C::foo`.

```java
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 ) ;
}
```

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 [62]
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.

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:
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) the 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 [68], 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 14. The amount of annotations found in those projects is extensive, see Section 7.4 for details. Spring uses gradle as build system.

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

---

14 Defined in in org.springframework.lang
Table 1 project summary, reporting the number of Java, Kotlin and Groovy source code files for both main and test scope, and branch coverage.

<table>
<thead>
<tr>
<th>program</th>
<th>version</th>
<th>main</th>
<th>test</th>
<th>coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>java</td>
<td>kotlin</td>
<td>groovy</td>
</tr>
<tr>
<td>s.-beans</td>
<td>5.3.22</td>
<td>301</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>s.-context</td>
<td>5.3.22</td>
<td>640</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>s.-core</td>
<td>5.3.22</td>
<td>499</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>s.-orm</td>
<td>5.3.22</td>
<td>72</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s.-oxm</td>
<td>5.3.22</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s.-web</td>
<td>5.3.22</td>
<td>653</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>s.-webmvc</td>
<td>5.3.22</td>
<td>368</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>guava</td>
<td>31.1</td>
<td>619</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>error-prone</td>
<td>2.18.0</td>
<td>745</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 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 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. 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. Profiling reveals that stack capture and IO are performance bottlenecks.

---

15 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.

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

### 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?

Results will be reported in Tables for each RQ, and we will summarise the distribution of recall and lower precision bound values across our dataset at the end of Section 7.7 in Figures 1 (for lpb) and 2 (for recall).

### 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 suggest 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.

### 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 sanitisation. 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.
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).

<table>
<thead>
<tr>
<th>program</th>
<th>ex</th>
<th>obs</th>
<th>agg</th>
<th>agg/obs</th>
<th>r,lpb</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.-beans</td>
<td>1,290</td>
<td>321,851</td>
<td>1,320</td>
<td>0.0041</td>
<td>0.54,0.52</td>
</tr>
<tr>
<td>s.-context</td>
<td>1,435</td>
<td>6,872,413</td>
<td>5,945</td>
<td>0.0009</td>
<td>0.49,0.12</td>
</tr>
<tr>
<td>s.-core</td>
<td>1,510</td>
<td>175,725</td>
<td>1,171</td>
<td>0.0067</td>
<td>0.52,0.67</td>
</tr>
<tr>
<td>s.-orm</td>
<td>377</td>
<td>3,443</td>
<td>279</td>
<td>0.0810</td>
<td>0.47,0.63</td>
</tr>
<tr>
<td>s.-oxm</td>
<td>64</td>
<td>501</td>
<td>64</td>
<td>0.1277</td>
<td>0.54,0.70</td>
</tr>
<tr>
<td>s.-web</td>
<td>2,025</td>
<td>127,882</td>
<td>1,656</td>
<td>0.0129</td>
<td>0.45,0.55</td>
</tr>
<tr>
<td>guava</td>
<td>3,993</td>
<td>2,708,816</td>
<td>1,656</td>
<td>0.0129</td>
<td>0.45,0.55</td>
</tr>
<tr>
<td>error-prone</td>
<td>507</td>
<td>1,095,752</td>
<td>1,736</td>
<td>0.0016</td>
<td>0.39,0.11</td>
</tr>
</tbody>
</table>

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

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

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).

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

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 sanitisation 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 bound 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.” [59].
Table 5 Annotated vs annotatable program elements, in the last column the number of annotatable elements of type `java.lang.Void` is reported.

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

To investigate the lower 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 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, and is therefore always nullable by definition. 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`”) \(^{17}\). I.e., `error-prone` is not dog-fooding [32] here. `Error-prone` has recently opened an issue to address this \(^{18}\). We also note that the `nullaway` checker treats `Void` as nullable \(^{19}\), and the `checkerframework` declares `@Nullable` as default for `Void` using a meta annotation \(^{20}\).

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.

---

\(^{17}\) https://errorprone.info/bugpattern/VoidMissingNullable

\(^{18}\) https://github.com/google/error-prone/issues/3792

\(^{19}\) https://github.com/uber/NullAway/blob/master/nullaway/src/main/java/com/uber/nullaway/NullAway.java, commit

\(^{20}\) https://checkerframework.org/api/org/checkerframework/checker/nullness/qual/Nullable.html
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 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.
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).

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

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 a relevant question to study and quantify whether we come close to a fix point, and whether it is common for programs not to reach such a fix point. 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.

Figures 1 and 2 show the distribution of lpb and recall values across the dataset after each step discussed.

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 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.21

21 There were 164 open pull requests on 20 October 2022, https://github.com/spring-projects/spring-framework/pulls?q=is%3Aopen
We have submitted two pull requests with different outcomes: PR1 has resulted in an `@Nullable` annotation inferred being added. PR2 was rejected, but the developers refined the test the inference is based on.

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.

Our tools has also led to the re-annotation of some classes in `guava`.

---

22 https://github.com/spring-projects/spring-framework/pull/29150
23 https://github.com/spring-projects/spring-framework/commit/35d379f9d3882a02f0368f928b2cecb975404334
24 https://github.com/spring-projects/spring-framework/pull/29242
25 https://github.com/spring-projects/spring-framework/commit/14c4bd07f44d845269c99f3a29241e7e2d0d4f1c
26 https://github.com/google/guava/commit/2b98d3c1e96b750dc997c29f283084aeb72f3b3cf, https://github.com/google/guava/pull/6490
Figure 2 Recall distribution across the dataset after each step: capture, applying the various sanitisers (san-*), propagation, propagation followed by re-sanitisation (sps), and special handling of Void in errorprone.

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 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, 35, 34] and succeeded in using jasaddjnonnullinference [21] to analyse some programs, and compare results. 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

27https://bitbucket.org/jastadd/jastaddj-nonnullinference
Table 9  Comparing our approach with JastAddJ NonNull inference.

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

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 [40]. 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” [37]. 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 [52]. They found the 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) [39]. 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 [10]. 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 [61] 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 [50]. 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 [35, 34], whilst Spoto developed a similar system arguing it is faster and more precise in practice [63]. XYLEM employs a backwards analysis to find null dereferences [49]. 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) [8]. 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, 12] and, in many ways, is similar to Nullable.

A number of other tools have been developed which can be used for static @NonNull checking, such as FindBugs [33], ESC/Java [24], JastAdd [21], JACK [45] and more [56, 44]. 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, 36, 57, 66, 64, 60, 42, 43, 38]. 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 [66]). 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” [9] allow optional type systems to be layered on existing languages, thus allowing them to evolve independently [26, 27, 15, 3, 16, 47]. The checkers framework provides a prominent example which heavily influenced JSR308 (included in Java 8) [54]. A key advantage of this tool over others is the ability to support for flow-sensitive type systems (a.k.a. flow typing [55]). 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 contributors. 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 from https://github.com/jensdietrich/null-annotation-inference.

References

1. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. URL: https://fbinfer.com/.
On Leveraging Tests to Infer Nullable Annotations


Rafael Winterhalter. Byte Buddy – A code generation and manipulation library for creating and modifying Java classes during the runtime, 2014. URL: https://bytebuddy.net/.