Harvesting the Wisdom of the Crowd to Infer Method Nullness in Java

Manuel Leuenberger, Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{*leuenberger, osman, ghafari, oscar*}@*inf.unibe.ch*

***Abstract*—Null pointer exceptions are common bugs in Java projects. Previous research has shown that dereferencing the results of method calls is the main source of these bugs, as developers do not anticipate that some methods return null. To make matters worse, we find that whether a method returns null or not (nullness), is rarely documented. We argue that method nullness is a vital piece of information that can help developers avoid this category of bugs. This is especially important for external APIs where developers may not even have access to the code.**

In this paper, we study the method nullness of Apache Lucene, the de facto standard library for text processing in Java. Particularly, we investigate how often the result of each Lucene method is checked against null in Lucene clients. We call this measure *method nullability*, which can serve as a proxy for method nullness. Analyzing Lucene internal and external usage, we find that most methods are never checked for null. External clients check more methods than Lucene checks internally. Manually inspecting our dataset reveals that some null checks are unnecessary. We present an IDE plugin that complements existing documentation and makes up for missing documentation regarding method nullness and generates nullness annotations, so that static analysis can pinpoint potentially missing or unnecessary null checks.

***Index Terms*—static analysis; API usage analysis; dependency management systems; null pointer exceptions;**

I. Introduction

Missing null checks are among the most frequent bug patterns in Java [1], leading to `NullPointerException`s (NPEs) and causing systems to crash. Methods that return null appear to be the main source of this problem [1] as 70% of the null-checked values are returned from method calls [2]. This indicates that developers are often unaware of the nullness of the invoked methods. Method nullness denotes whether a method might return null (nullable) or never returns null (non-null).

Although it might be fairly easy for developers to reason about the nullness of methods in their own projects, analyzing methods in external dependencies is not. When developers want to dereference the return value from methods in external APIs, they often face three scenarios:

1) They assume that the method does not return null, then they deal with an NPE later if it occurs. Although this technique can be used during development with NPEs being detected and fixed through testing, some null dereferences can make it into production code causing system crashes.
2) They defensively add a null check to eliminate the risk of getting an NPE. This technique clutters the code with excessive null checks hindering code comprehension and maintainability [3].
3) They check the source code and read the documentation of the external method and try to reason about its return value, then add a null check when necessary. However, as we show later, documentation is often missing and reverse engineering unknown methods can be complicated.

We argue that the method nullness is an important post-condition of methods for developers to be aware of. In this paper we devise an empirical approach to infer the nullness of library methods relying on the wisdom of the crowd. Given a particular library, we collect and analyze its clients and track how they handle the returned values from method calls, *i.e.*, how often they are checked to be non-null. The results are then aggregated per library method in a *nullability* measure defined as follows:

$$Nullability(Method) = \frac{CheckedDereferences(Method)}{Dereferences(Method)}$$

The nullability measure serves as a proxy for method nullness and expresses the confidence that a particular method returns null. A nullability of zero indicates that a method never returns null, *i.e.*, the returned value is always dereferenced in clients without a null check. The method's nullness is therefore non-null. Conversely, a non-zero nullability indicates that a method is nullable.

In this paper, we investigate the nullability of methods in Apache Lucene, which is the de facto standard library in Java for full-text searching. We collect 4 197 versions of 186 Lucene clients by exploiting the Maven dependency management system with KOWALSKI, a tool developed particularly for this purpose. We compute the nullability for the 42 092 detected methods belonging to 75 versions of Lucene. We formulate the following research questions and use them to guide our research:

RQ1: How is nullability distributed and what are the factors affecting it?

We partition the collected methods into methods that expose object state while keeping encapsulation intact (getters) and methods that compute their return value (processors). Then we calculate the nullability distribution for the methods when they are called within Lucene itself (internal usage) and from

external clients (external usage). The majority of methods are never checked against null, but those that are account for most of the usage. Getters do not document nullness at all, whereas there is some nullness documentation for nullable processors. While the shapes of the distributions in internal and external usages are similar, in external usage there are many methods checked that are not checked in internal usage, hinting at unnecessary null checks.

RQ2: How does nullability reflect method nullness?

We select 600 methods that range in nullability from 0 to 1, and try to reverse-engineer the nullness by manually inspecting the documentation and the source code of the method. For about half the methods, this context is not broad enough to decide whether a method returns null or not. Nonetheless, for some methods with a nullability of 0 we can deduce that they do not return null, whereas most methods with a non-zero nullability either document that they potentially return null or they contain a `return null;` statement.

RQ3: How can the nullability measure be used in practice?

We present an IDE plugin that adds nullability information to the method's documentation as a usage recommendation, and we can add a corresponding annotation to the return type of the method to assist null analysis tools. The nullability documentation enables developers to make a more informed decision about adding or removing a null check. The null analysis points developers at locations where a null check is unnecessary or potentially missing.

The rest of this paper is structured as follows: In section II we describe how we collect the bytecode of a library and its clients with KOWALSKI and how we measure nullability. We look at the nullability distribution in section III. In section IV we summarize the results of the manual inspection of Lucene source code and documentation. Section V shows how we can use nullability to assist developers with an IDE plugin. In section VI we note possible threats to validity imposed through imprecisions in the analysis and inaccuracies of the applied heuristics. Related work is discussed in section VII. In section VIII we conclude our work with a summary of the most important results, their implications and future work.

II. BYTECODE COLLECTION AND ANALYSIS

A. Bytecode Collection

We collect binaries of Lucene releases and Lucene clients, so that we can compare internal and external usage of Lucene methods. We find and download the Lucene related binaries by exploiting the *Maven* dependency management system. *Maven* projects declare their dependencies in a meta-data file and binary releases are published on a central package repository. Collecting binaries related to a specific library means that we need to extract a sub-graph of the dependency graph spanned by all projects, as shown in Figure 1. First, we need to find the libraries for which we want to collect clients, as in Figure 1(a). Second, we need to find clients of the matched libraries, as in Figure 1(b). Third, we need to download the binaries we want to analyze, as in Figure 1(c).

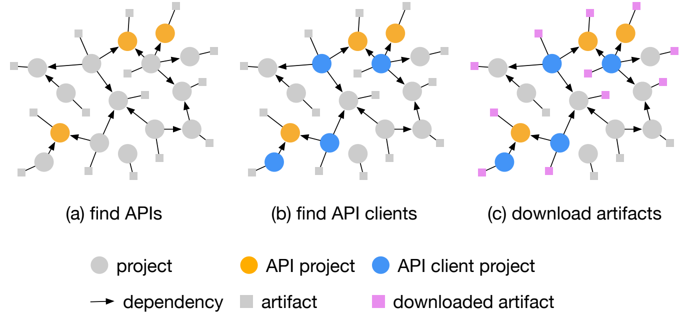

Fig. 1. Dependency sub-graph extraction steps with project nodes marked as API, API client and artifacts.

We implement this client collection process in a tool designed for this purpose called KOWALSKI. KOWALSKI takes a project name as an input, Lucene in our case, and finds all releases of Lucene by querying *Maven Central Search*.[1] Then it scrapes *mvnrepository* to find projects depending on Lucene.[2] In the third step it uses *Maven* to fetch the clients, including their dependencies, from the package repository. More information about KOWALSKI can be found in the dedicated paper [4].

KOWALSKI finds and collects 7 123 versions of 294 artifacts belonging to 174 groups related to Lucene. The whole process is highly parallelized and completes within two hours on a multi-core machine. The machine runs a 64 bit Ubuntu OS, has 32 cores at 1.4 GHz, and 128 GB of RAM.

B. Static Analysis

```
1 public void traverse(Node child) {
2    Node parent = child.getParent();
3    child.mark();
4    if (parent != null) {
5       parent.getSibling().mark();
6    }
7 }
```

Listing 1. Example method for null dereference analysis.

We analyze the collected binaries to detect which return values of Lucene methods are checked for null and which are not. The static analysis is based on SOOT's null analysis [5], [6], an intra-procedural and path-sensitive data-flow analysis that tracks the nullness of local variables. For each path in the control flow graph it tracks whether a local variable is known to be null, non-null, or unknown. The analysis supports variable aliasing and learns about their nullness from null checks, `instanceof` checks, assignments of newly instantiated objects, and dereferences in field accesses, array accesses, method invocations, *etc.* For instance, in Listing 1, SOOT finds the nullness of the variable `parent` to be non-null on line 5 as it was checked in the condition of the wrapping `if`. Our analysis extends the SOOT nullness analysis by extracting the following three features from all method implementations

[1] https://search.maven.org/#search%7Cga%7C1%7Cg%3A%22org.apache.lucene%22

[2] https://mvnrepository.com/artifact/org.apache.lucene/lucene-core/usages

in the collected clients of Lucene. First, we identify the *first* dereference of every value. Those are the locations where a null pointer exception could potentially happen. The method in Listing 1 contains three potential null dereferences: `child` on line 2, `parent` and `parent.getSibling()` on line 5. Note that the subsequent dereference of `child` on line 3 can never throw a null pointer exception, as the same value has been dereferenced on line 2 before. Second, we track where the dereferenced value originates from. For `child` this is the parameter, for `parent` this is `child.getParent()`, and for `parent.getSibling()` it is `parent.getSibling()` itself. Third, we note whether or not the dereferenced value is checked for null before, indicating whether its origin is nullable or not. For `child` the nullness of the parameter is unknown, for `parent` we know that `child.getParent()` is non-null as it was checked, and for `parent.getSibling()` it is unknown.

```
if (child.getParent() != null) {
   child.getParent().mark();
}
```

Listing 2. Null check of a method return value.

A conservative null analysis only works on local variables. However, not all null checks use a local variable that is checked and dereferenced later. Especially fields accessed through getters and values that are only used once may not be assigned to local variable, but checked for null using an expression. For example, Listing 1 uses a local variable, but Listing 2 uses a method call. A conservative analysis detects a potential null dereference of `child.getParent()` on line 2 in Listing 2, but for a human reader the same dereference looks safe, as the method call is checked before dereferencing it. Our analysis assumes lexically identical expressions to evaluate to the same value, thus classifying this dereference as safe. This assumption is unsound as methods can return different objects for each invocation, but it is a cheap heuristic compared to a more precise, but computationally much more expensive, object-sensitivity [7].

Different methods might have conceptually different intents that can reveal further insight into what exactly is checked for null, therefore we distinguish between getters and processors. We introduce a heuristic to classify a method as a getter if its name is `get` followed by the name of a field of the declaring class, otherwise we classify the method as a processor. Note that this classification is merely a tag on a method, it does not interfere with the analysis.

The analysis of the 7 123 artifacts takes approximately 12 hours on the same 32-core machine on which we collect them. Of all artifacts, 1 627 Lucene artifacts and 2 570 artifacts of external clients contain a dereference of Lucene methods. External clients include Solr, Elasticsearch, Neo4j, and OrientDB amongst others. Overall we find 292 871 dereferences of return values of 42 092 methods belonging to 75 different Lucene releases.

```
public void register(XMLReader parser) {
   try {
      // 1 unsafe, is unsafe
      parser.setFeature("...", true);
   } catch (SAXException e) {
      log.warn("...");
   }
   try {
      // 2 unsafe, but is safe
      parser.setFeature("...", false);
   } catch (SAXException e) {
      log.warn("...");
   }
   // 3 unsafe, but is safe
   parser.setContentHandler(this);
   // 4 safe, is safe
   ErrorHandler handler = parser.getErrorHandler();
}
```

Listing 3. Dereferences of `parser` with dereferences 2 and 3 falsely classified as potentially unsafe. String literals shortened and method trailer omitted from `AbstractTopicMapContentHandler` in `net.ontopia:ontopia-engine:5.3.0`.

C. Validation

We inspect the precision of our analysis and heuristics by manually inspecting the results and the related code.

1) Dereference Analysis: We check how precise our potentially unsafe dereference analysis is. Over half of all analyzed methods do not contain potentially unsafe dereferences. About a quarter of the methods contain a single dereference. About 96% of the methods contain no more than eight dereferences. We ignore the remaining methods with more dereferences in the inspection, as they are potentially complex and hard to reason about. For 98% of all dereferences there is only a single possible origin. This allows us to precisely reason about the intention of a null check, as it can only check a single value. We inspect 50 randomly selected methods in which no dereferences are detected and 50 randomly selected methods for which we detected at least one dereference. Our analysis reliably finds 109 potentially unsafe dereferences and their originating values. Only in two methods do we miss a dereference, which is mistakenly classified as potentially unsafe. For the originating values we found no errors. All detected originating values are true possible origins and all possible originating values are detected.

2) Nullness Analysis: We inspect the accuracy of the nullness analysis. We inspect another 100 randomly selected methods for which we detected at least one dereference. Of 117 dereferences we correctly classified 15 as non-null and 98 as unknown. One instance is misclassified as non-null, another one as unknown. Three dereferences are mistakenly detected as potentially unsafe dereferences. In one instance the dereferenced field is initialized with a new object immediately before the dereference. As we do not track assignments to fields, but only locals, we cannot detect this initialization. The other two misclassifications are the second and third dereference of the parameter `parser` on line 10 and line 15 in the method listed in Listing 3. They are caused by the way the control flow graph is constructed. A try block creates a

new branch in the control flow graph and we have to merge two branches after the try statement. In the first branch the try block succeeds, in which case `parser` is now safe to dereference. In the second branch the try block fails and it is assumed that no statement was executed, in which case we do not know if `parser` is safe to dereference on line 10 and line 15. Assuming that pushing the method parameters and receiver of the method `setFeature()` on the stack on line 4 always succeeds (since they are constants and a reference), the actual method invocation is the first operation that could fail. Correcting these misclassifications would require more complex control flow graphs that replicate the execution model of the virtual machine more precisely.

As we have a low rate of non-null instances, we inspect another 100 randomly selected potentially unsafe dereferences classified as non-null. In 95 cases we correctly classify it as non-null. In 5 cases the classification is wrong. They happen in complex loops, try-catch statements, and casting situations. Here we also find interesting patterns where values are checked for null. Many null checks for fields occur within `hashcode()` and `equals()` methods, which can be auto-generated by an IDE like Eclipse. The pattern to check a resource for null before closing it in a finally block is also prevalent. This can be refactored to use Java 7 try-with-resource blocks[3] to improve code readability. We find null checks for lazy initialization of fields, and setting defaults for null parameters. It is also common to throw an error if a value is null, or exit the method with a `return` or a `return null;`.

3) Getter Heuristic: We validate our getter heuristic, which classifies methods as getters if their name starts with `get` followed by the name of field of the class. Therefore, we randomly select 100 methods and inspect them, 50 of which classified as getters, 50 of which are not. Only one method classified as a getter does not return the field with the getter name, but instead computes the return value. Eight methods which are not classified as getters, are getters. They return the field with a name that is very similar to the method name, or do not use a preceding `get`. Most getters directly return the associated field. In some cases the field is lazily initialized. In case of a collection type a shallow copy is sometimes returned. The getter heuristic we use is an under-approximation, as we can precisely identify getters, but we miss some getters.

III. Nullability Distribution

To answer our first research question, *RQ1: How is nullability distributed and what are the factors affecting it?*, we measure the nullability for all 42 092 methods whose return values are dereferenced. In Figure 2 we observe that 79% of the methods are never checked for null, 12% are always checked, and 9% are sometimes checked and sometimes not. Note that the nullability peaks at $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, *etc.* are due to methods for which only a handful of dereferences are found.

We look at the dataset from two different perspectives. First, we contrast internal and external usage of Lucene regarding nullability to identify potential disagreement. Second, we distinguish between the usage of getters and processors, as they serve different intents. We filter the dataset to only include methods that are used internally by Lucene itself as well as by external clients. We reduce the selected methods belonging to 75 releases to the six major versions of Lucene by their signature (qualified class name, method name, argument types) to increase the support for each method. This reduction assumes that the nullness of a method remains unchanged for all different releases of a major version. In Table I we find the 568 methods spread across the major releases dereferenced 156 717 times. We partition the dereferences and methods by the getter/processor and external/internal dimensions. For each of the four partitions we find a median of around 30 dereferences per method. The full spectrum ranges from methods only dereferenced once to some dereferenced 5 071 times.

Figure 3 shows the distribution of the selected methods over the whole nullability range from 0 (returned value is never checked for null before dereferencing) to 1 (returned value is checked for null before every dereference). We find 86 getters and 482 processors. Across all four partitions most methods are never checked (external getter 90%, external

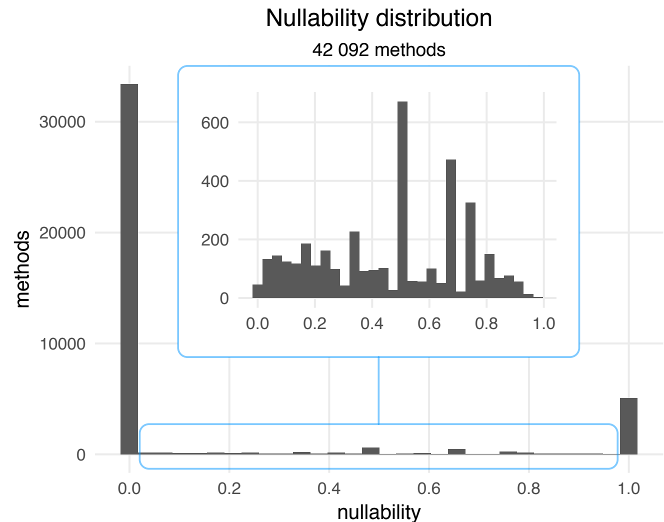

Fig. 2. The nullability distribution of the 42 092 Lucene methods that are dereferenced. The nested distribution zooms in to the nullability range excluding the extremes]0, 1[.

TABLE I
Methods used both internally by Lucene and external clients, reduced to the Lucene major version.

major version	methods	dereferences
1	6	35
2	57	12 038
3	105	34 128
4	209	64 502
5	117	31 992
6	74	14 022
	568	156 717

[3]https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

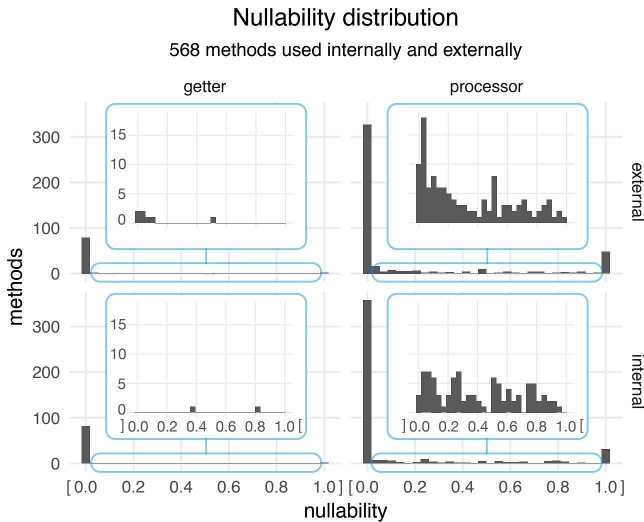

Fig. 3. The nullability distribution of the 568 Lucene methods that are dereferenced both internally and externally. The nested distribution zooms in to the nullability range excluding the extremes $]0, 1[$.

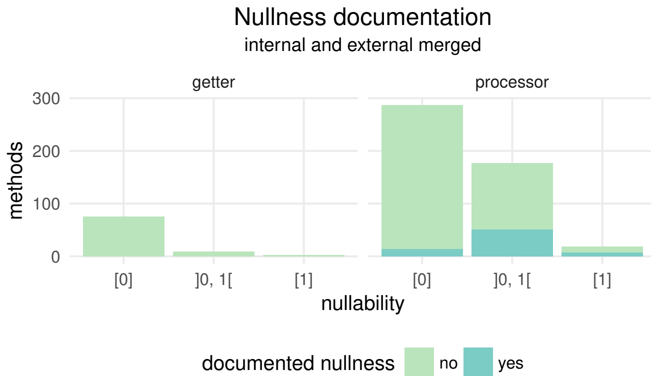

Fig. 4. Nullness documentation of methods, split into nullability classes. See Table II for exact counts.

processor 65%, internal getter 95%, internal processor 74%). Some processors are always checked while this category is not significant for getters. For getters the share of checked methods is much smaller than for processors. This could indicate that fields are rarely null. Evidence for this hypothesis is also reported by Chalin *et al.*, as they show that fields are eventually non-null [8]. In the validation of our getter heuristic we also find many cases of lazy initialization of fields, which supports this hypothesis.

A. Documentation

Documentation could be another factor that affects whether or not the return value of method is checked before dereferencing it. For this purpose we collect and process the JavaDoc for all selected methods. If either the word `null` occurs in the return section of the method documentation, or both words `null` and `return` occur in the general method description, we classify the method as having its nullness documented.

TABLE II
NULLNESS DOCUMENTATION OF METHODS, SPLIT INTO NULLABILITY CLASSES.

	nullability	class doc. not found	method doc. not found	nullness not mentioned	nullness mentioned
getter	[0]	4	11	60	0
	]0, 1[	0	4	5	0
	[1]	0	1	1	0
processor	[0]	14	63	196	14
	]0, 1[	1	47	78	51
	[1]	0	6	5	7

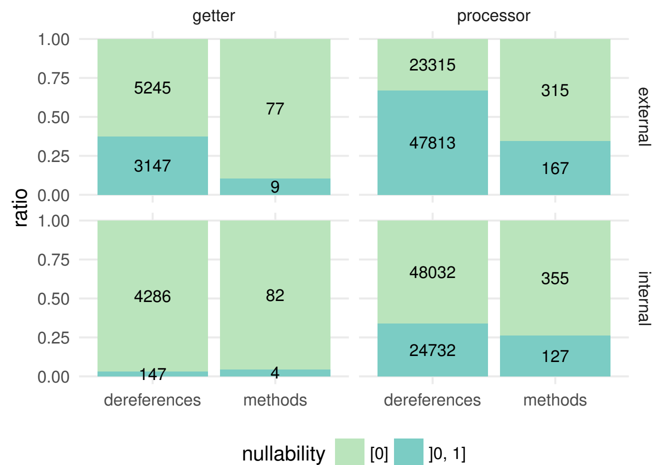

Fig. 5. Dereferences contrasted with the originating methods of the dereferenced values. The methods with a nullability of 0 and the associated dereferences are painted in red. The methods and dereferences with a non-zero nullability are painted in turquoise.

If the method documentation is found but the heuristic does not pass, we classify the method as not mentioning the nullness of the return value. The measure does not extract whether the documentation states that the method is non-null or nullable, but only if the nullness is mentioned at all. We do not differentiate between internal and external usage, as the documentation is an attribute of a method. Figure 4 shows that nullness is not documented at all for getters and often undocumented for processors. Table II contains the exact counts for each category, as we could not always find the documentation. Regarding nullability, nullness is rarely documented for processors when the nullability is 0. For processors with a nullability higher than zero, the method documentation often makes a statement about the nullness. Getters do not document nullness at all, which can again be seen as support for eventually non-null fields [8].

B. Disagreement between Internal and External Usage

The sets of checked and unchecked methods are only partially overlapping between internal and external usage.

TABLE III
CLASSIFICATION DIFFERENCES REGARDING NULLABILITY OF METHODS BETWEEN INTERNAL AND EXTERNAL USAGE.

	nullability (internal → external)	**getter**	**processor**
agree	$[0] \rightarrow [0]$	75	287
	$]0,1] \rightarrow]0,1]$	2	99
disagree	$[0] \rightarrow]0,1]$	7	68
	$]0,1] \rightarrow [0]$	2	28

TABLE IV
INSPECTED METHODS WITH CLASSIFICATION DIFFERENCES REGARDING NULLABILITY BETWEEN INTERNAL AND EXTERNAL USAGE.

	nullability	manual classification		
	(internal → external)	**unknown**	**non-null**	**nullable**
getter	$[0] \rightarrow]0,1]$	4	2	1
	$]0,1] \rightarrow [0]$	2	0	0
processor	$[0] \rightarrow]0,1]$	46	15	7
	$]0,1] \rightarrow [0]$	15	3	10

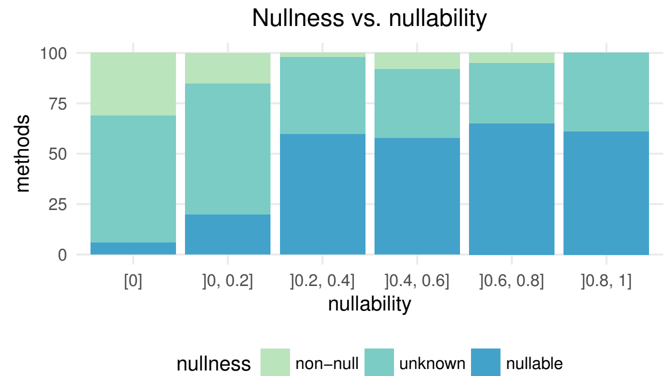

Fig. 6. Manual classification of sources and documentation of 600 Lucene methods, partitioned by their nullability into five intervals of equal length, except the [0] category.

In Figure 3 we see that in external usage there are many processors with a low nullability whereas the internal nullability distribution in $]0,1[$ is more balanced. The disagreement in usage is also visualized by contrasting the number of dereferences and associated methods in Figure 5. In external usage there are generally more methods checked and they are more used than unchecked ones. The checked processors are more often dereferenced, relative to all externally dereferenced processors as well as relative to the internal usage. Table III shows that there are 7 getters and 68 processors that are never checked internally but are checked externally. Only 2 getters and 28 processors are checked internally but not externally. We inspect the 105 methods where internal and external usage disagree. From the method source code and documentation we try to reverse-engineer the methods nullness. We classify a method as non-null if either the documentation states so or all return paths always return an object. We classify a method as nullable if either the documentation states so or at least one return path returns null. The results in Table IV show that for most of those methods our reverse-engineering approach cannot deduce the nullability of the majority of the methods. However, we find that the internal usage is more often in concordance with our classification than external usage. For example, we classify 15 processors that are never checked internally as non-null, whereas 7 are nullable.

We learn three things from the inspection of disagreement of internal and external usage. First, external usage is more defensive than internal usage. Second, reverse-engineering the nullness of a method is non-trivial, even with the source code at hand. Third, internal usage is a better indicator for method nullness than external usage, yet, it is not precise.

IV. Manual Inspection

We answer our second research question, *RQ2: How does nullability reflect method nullness?*, by manually inspecting Lucene methods across the whole nullability range and trying to reverse-engineer the nullness of the method. For this purpose we partition the nullability spectrum into six intervals and randomly select 100 methods with a nullability in each interval. The nullability intervals are $[0]$, $]0,0.2]$, $]0.2,0.4]$, $]0.4,0.6]$, $]0.6,0.8]$, and $]0.8,1]$. We select those methods from the 11 754 Lucene methods with a minimal support of five dereferences, so that their nullability is computed from at least a handful of dereferences. For each method we find their implementation and documentation, then we try to decide on the nullness of the return value. We classify a method as non-null if either the documentation states so or all return paths always return an object. We classify a method as nullable if either the documentation states so or at least one return path returns null. If none of the above conditions holds, we classify the method as unknown. The results of this inspection are shown in Figure 6. For about half of the methods the nullness is unknown using our procedure. It is quite rare that the documentation states that the method is non-null. Even rarer are methods for which we can deduce non-nullness from their implementation. In those cases the return value is mostly a collection. We can see that the inferred nullability correlates with the reverse-engineered nullness. The majority of methods with low nullability are non-null. Even more convincing is the situation at the other end of the spectrum. The vast majority with a high nullability are actually nullable. We observe non-null methods with a non-zero nullability, indicating unnecessary null checks, as well as nullable methods with nullability of zero, indicating potentially missing null checks. The direction of causality is unclear though: Do developers check return values because of the documentation or is the nullness documented because null pointers have been observed? Nonetheless, we see that nullability reflects the nullness quite well.

V. Transferring Nullability to the IDE

Inspecting the nullability distribution reveals that non-null and nullable methods can be partially separated. Manual inspection suggests that the nullness of methods is often undocumented or non-trivial to reverse-engineer. These findings lead us to the third research question, *RQ3: How can the nullability*

```
@Override
public Query rewrite(IndexReader reader) throws IOException {
    Query rewritten = super.rewrite(reader);
    if (rewritten != this) {
        return rewritten;
    }
    boolean hasPayloads = false;
    for (LeafReaderContext context : reader.leaves()) {
        final Terms terms = context.reader().terms(term.field());
        if (terms.hasPayloads()) {
            hasPayloads = true;
            break;
        }
    }
    if (hasPayloads == false) {
        return new TermQuery(term);
    }
    return this;
}

@Override
public Weight createWeight(IndexSearcher sear
    if (needsScores == false) {
        return new TermQuery(term).createWeig
    }
    final TermContext termStates = TermConte
    final CollectionStatistics collectionSt
    final TermStatistics termStats = searche
    final Similarity similarity = searcher.ge
    final SimWeight stats = similarity.comput
    return new Weight(this) {
```

```
@Nullable Terms
org.apache.lucene.index.LeafReader.terms(String field)
throws IOException

This may return null if the field does not exist.

Parameters:
    field
Throws:
    IOException

Returns:
    63% check the returned value before dereferencing it (22 out of 35 times)
```

Fig. 7. Augmented JavaDoc with nullability (1), nullness annotation (2) for a Lucene method with a high nullability giving a warning for a potential bug (3).

```
@Override
public Query rewrite(IndexReader reader) throws IOException {
    Query rewritten = super.rewrite(reader);
    if (rewritten != this) {
        return rewritten;
    }
    boolean fieldExists = false;
    boolean hasPayloads = false;
    for (LeafReaderContext context : reader.leaves()) {
        final Terms terms = context.reader().terms(term.field());
        if (terms != null) {
            fieldExists = true;
            if (terms.hasPayloads()) {
                hasPayloads = true;
                break;
            }
        }
    }
    if (fieldExists == false) {
        return new MatchNoDocsQuery();
    }
    if (hasPayloads == false) {
        return new TermQuery(term);
    }
    return this;
}

@Override
public Weight createWeight(IndexSearcher sear
    if (needsScores == false) {
        return new TermQuery(term).createWeig
    }
```

```
@Nullable Terms
org.apache.lucene.index.LeafReader.terms(String field)
throws IOException

This may return null if the field does not exist.

Parameters:
    field
Throws:
    IOException

Returns:
    63% check the returned value before dereferencing it (22 out of 35 times)
```

Fig. 8. Augmented JavaDoc with nullability (1), nullness annotation (2) for a Lucene method with a high nullability giving no warning for a potential bug (3) due to a missing null check.

measure be used in practice? As a showcase of how nullability information can be harvested, we have implemented an Eclipse plugin that integrates Lucene nullability information in the IDE. The goal of the plugin is to give hints to developers about potential null dereferences and unnecessary null checks. If we can detect potential null pointers, we can avoid bugs. If we can detect unnecessary null checks, we can reduce cyclomatic complexity [9]. The latter point is relevant as we find unnecessary null checks in the manual inspection and null checks account for 35% of all conditional statements [2].

First, the plugin adds the nullability information to the JavaDoc of a method, with both confidence and support. The nullability gives the developer the frequency of a null return value. Confidence and support are a proxy for the trust that can be put into the nullability. If the nullability is computed from only a handful of samples, it might not be trustworthy as not all usage scenarios of the method might be covered. Figure 7 shows our plugin applied on a code excerpt of Elasticsearch 2 for that a null pointer bug was reported.[4] The cause of the bug is the unconditional dereference of the return value of the Lucene method `LeafReader.terms(String)`. Our plugin adds nullability documentation with a blue background at position (1). Besides the nullability of 63% it also tells us that this is computed from 22 out of 35 dereferences that were checked for null. Documenting nullness is certainly important, but it still requires developers to read the documentation to detect the nullness. For example, in Figure 7 the nullness is described in the original documentation, but the developer nevertheless missed it. To raise awareness for potential nullness, the tools should give more obvious hints to developers about potential null pointers.

[4]https://github.com/elastic/elasticsearch/pull/12495/commits/d7491515b21fb4b3c94956c75bcb74b8a5c863ae

Second, the plugin generates external nullness annotations[5] from the nullability measured for Lucene methods. For methods with a nullability of zero we create a `@NonNull` annotation. For methods with a non-zero nullability we create a `@Nullable` annotation. These annotations are stored as Eclipse External Annotations that do not require bytecode manipulation of the Lucene binary, but they are stored alongside the library and linked through the Eclipse project configuration. The internal Eclipse JDT Null Analysis[6] considers these external annotations. In Figure 7 the annotation is visible in the documentation at position (2). The warning generated for the potential null pointer is at position (3). The bug in Elasticsearch was eventually fixed by guarding the dereference with a null check (see Figure 8). After the bug fix, the null analysis issues no warning anymore, as the dereference is now safe. Detecting potential bugs is not the only use case for the null analysis, as it can also detect dead code caused by unnecessary null checks.

Third, our plugin can be configured to only generate annotations for methods with nullability within a certain range. For example we can set the upper limit for the generation of non-null annotations to 0 and the lower limit for nullable annotations to 0.8. For all methods with a nullability in between 0 and 0.8 no annotations are generated and conversely the null analysis does not generate any warnings. This tuning can be used to reduce the number of warnings.

In the future, we plan to evaluate the IDE plugin with developers in the industry to assess its usefulness and further improve it. The plugin sources are available online,[7] including a sample project and sample nullability data.

[5]http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.user/tasks/task-using_external_null_annotations.htm
[6]https://wiki.eclipse.org/JDT_Core/Null_Analysis
[7]https://github.com/maenu/method-nullability-plugin

A. Evaluation

We inspect the warnings generated by our plugin on the sources of the Elasticsearch project at version 1.7.3 from October 15th, 2015. This is the newest version of Elasticsearch we analyzed and uses Lucene 4, for which we collect the most usage data. Our analysis identifies 764 potentially unsafe dereferences of Lucene methods. The plugin is configured to generate `@Nullable` annotations for methods with a nullability of at least 0.2, as our manual inspection reveals that there is a high agreement of nullability and nullness above this theshold. For methods with a nullability of at most 0.1, the plugin generates `@NonNull` annotations, so that a few unnecessary null checks detected in the analysis still lead to a generated annotation. When our plugin is activated, we find 72 problems, caused by 21 different Lucene methods. The inspection of those problems reveals that four warnings are duplicates, *i.e.*, they warn about the same variable. We classify the warnings according to the nullness of the causing method using the same procedure as in section IV. We get six warnings about dead code caused by null checks, of which two warnings are considered correct as they are caused by non-null methods. Two dead code warnings refer to methods that are nullable, two are unknown. The latter four warnings can be false positives. Ten warnings are about unnecessary null checks, of which only one warning is a true positive, as the respective method is documented to never return null. Eleven warnings hint at the illegal return of null in overriden methods for which a `@NonNull` annotation has been generated. The manual inspection reveals that ten of those cases are false-positives, as the two different, overriden methods are nullable. We notice that the false positives or potential false positives are caused by methods for which we have a relatively low support of at most 28 dereferences examined in our analysis, whereas the true positives have higher support of hundreds of analyzed dereferences. The remaining 41 warnings are about potential null dereferences. Among those we find one false positive where the dereferenced getter is checked for null before dereferencing it. While our own analysis accounts for this case, the Eclipse null analysis does not. Eight other false positives are not recognized as such as the null check is performed in a utility method that is called before the dereference and avoids the null dereference by returning from the method early. The remaining potential null dereference warning consider method with a nullability ranging from 0.23 to 0.87. We check if the potential null dereferences are changed in the latest version of Elasticsearch. For 18 cases we cannot find the affected code because the class or call no longer exists. In 14 cases we find the affected code and in all of them the code shows no changes regarding the handling of a potential null handling. This indicates that the potential null dereference warnings generated the generated annotations are false positives, although the manual inspection reveals that the methods can return null in some cases.

We conclude that our plugin cannot predict null dereferences of method return values, as the return value of nullable methods is context-sensitive, *i.e.*, depends on the state of callee and the parameters. The plugin shows some potential in detecting unnecessary null checks though. We see the main use for the plugin to provide developers with nullability information through the augmented documentation, so that they can quickly assess whether or not they should consider to add a null check. The inspection of the warnings confirms that the nullness inferred from the nullability is correct for methods with a high support. The warnings generated cannot be viewed as bugs, but rather as hints to method calls that should be inspected.

VI. Threats To Validity

A. Construct Validity

The results we gain are only as accurate as the analysis. We validate all parts of the analysis we suspect to be critical. We report some wrongly classified samples that hint at bugs in the analysis implementation. Nonetheless, most samples are correctly classified. Our confidence in the analysis is supported by the manual inspection which reveals that methods with a high nullability are mostly nullable, whereas many methods with a low nullability are non-null.

To inspect where internal and external usage disagree on nullability we group methods by their Lucene major version. This is an over-approximation, as Lucene does not strictly adhere to semantic versioning and introduces some breaking changes in minor releases. If a method changes from non-null to nullable without changing its signature between two minor versions of the same major release, this approximation mixes the nullability for two different methods.

Based on nullability we generate nullness annotations that lead to warnings. Those warnings can also be false positives in some cases. We find both checked methods that never return null according to the method's documentation and implementation, as well as unchecked methods that document a potential nullness. The nullability measure of a method hints at the frequency of null checked return values. Nullability cannot express in which context a method returns null or not. To reduce the number of false positive warnings, our plugin can be configured to generate annotations only for methods in a specific range.

B. Generalizability

The composition of our dataset influences the generalizability of our results. KOWALSKI collects only Lucene and other OSS projects that depend on Lucene and are published on Maven Central. First, this excludes all closed source projects. If given access to a company repository, KOWALSKI can also be used to collect a dataset of clients of a company-internal library. The static analysis can be reapplied as well. Second, all open source projects that are not published on Maven Central are excluded. There are other popular Maven repositories that may contain other Lucene clients, for example jcenter[8]

[8]https://bintray.com/bintray/jcenter

and clojars.[9] However, Maven Central is a large repository that serves 1 935 045 versions of 185 693 artifacts.[10] Some software projects are not published in a repository at all. We lack a measure to estimate how many Lucene clients are only distributed as sources, for example on GitHub. As our analysis is tailored to run on binaries, it would require a build of these projects. Package repositories are primarily used to distribute reusable libraries, therefore our dataset has a strong bias towards libraries as clients. Libraries may use Lucene differently than projects further down the dependency hierarchy. Third, we only analyze the Lucene ecosystem. The results may not be generalizable to other ecosystems.

VII. Related Work

A. Null Dereference Analysis

Hovemeyer *et al.* use intra-procedural data-flow analysis to find null dereferences in Java code and continuously improve their FindBugs tool [10], [11], [12]. With FindBugs we cannot find the possible null dereferences where we dereference a value returned from a method, as this is beyond the scope of an intra-procedural analysis, unless the analysis is aware that the called method is annotated as returning null. These situations arise often, as values returned from methods are the most often checked for null category [2]. The nullness annotations we generate are exactly those missing links between a project and its dependencies that allow an intra-procedural null analysis to reason about values received by calling a method.

There are approaches to verify dereference safety, reporting 10% [13] and 16% [14] unverifiable dereferences. Our approach is not a verification technique, but it focusses on the main cause of null pointer bugs, that is the dereference of a method return value [1]. With our generated annotations we can suggest the nullness for those dereferences to help developers make more informed decisions.

Nanda and Sinha present Xylem, an unsound inter-procedural, path- and context-sensitive data-flow analysis [15]. Contrary to Tomb et. al. [16] they report a 16 times higher number of possible null dereferences in inter-procedural than in intra-procedural analysis [10], yet they claim a low false positive rate. Ayewah and Pugh examine the warnings generated by Xylem and account only 60% of them to be plausible [17]. They argue that no warning should be issued for dereferences that are unchanged over many versions of stable software. If our analysis finds the same unconditionally dereferenced method return value in many versions, it decreases the originating method's nullability, which reflects that the method is rarely causing problems.

Many researchers propose to avoid null dereferences by extending the type system with `@Nullable` and `@NonNull` annotations [18], [19]. Method, field, variable and collection item declarations can be annotated, so that a static checker can verify that no nullable value is ever dereferenced and the nullness matches for parameters passed to methods. If annotations for library methods are missing, they need to be added manually. Flanagan *et al.* circumvent this problem by assuming that all library method never return null [20]. Our approach gets rid of the required manual work to add the nullness annotations to existing projects, as the annotations are automatically generated from usage.

B. API Usage Analysis

There are several datasets over large parts of GitHub. Google's BigQuery GitHub dataset[11] can be queried for contents of source files. It even runs static analysis remotely,[12] but type information is not provided and must be reconstructed. Dyer *et al.* provide ASTs that include partial type information from sources in GitHub projects in the Boa dataset [21]. The binaries collected by Kowalski provide more type information as we can track method invocations to the invoked method and library version without ambiguity. By choosing Maven repositories as our datasource we are restricted to a smaller set of OSS projects than GitHub, but we gain type precision.

Lämmel *et al.* check out 6 286 SourceForge projects and manage to build 1 476 of them with Ant to analyze them for API usage [22]. They manually search for missing dependencies to fix build errors in 15% of the built projects. Instead of building projects from source, we collect binary Maven artifacts with resolved dependencies. We use the Soot analysis framework that creates phantom references for unresolvable classes.

Sawant *et al.* build a typed dataset for five APIs and their usages in 20 263 GitHub projects using Maven [23]. They use partial compilation to work around unresolvable classes. By compiling from source, projects can be inspected for any revision of a source file in a version control system. Our dataset includes only built binary artifacts, yet they are versioned as well, therefore we can track the evolution of a project as well, although on a coarser level of releases.

VIII. Conclusions And Future Work

In this paper we harvest the wisdom of the crowd to infer the nullness of methods of Apache Lucene. The inference is based on nullability that measures the frequency of null checks of the method return value before it is dereferenced. We find that most methods are used as if they were non-null, *i.e.*, they never return null. The nullness of the non-null methods is rarely documented, nullable methods document the nullness more often. Getters are generally non-null and do not document nullness at all. We present an IDE plugin that utilizes method nullability to augment documentation and to integrate with static analysis tools. The plugin points to potential null dereferences and unnecessary null checks.

[9] https://clojars.org/

[10] https://search.maven.org/#stats, date of access May 3, 2017

[11] https://cloud.google.com/bigquery/

[12] https://medium.com/google-cloud/static%2Djavascript%2Dcode%2Danalysis%2Dwithin%2Dbigquery%2Ded0e3011732c

A. Future Work

We fail to generate external annotations and look up nullability for methods with generics, as binaries lack this information due to type erasure. The linked source code of the binary methods needs to be processed to reconstruct the type information. The nullability analysis should be repeated for other ecosystems to validate whether our results are generalizable. We need more nullability data and generate nullness annotations; for libraries other than Lucene, frameworks, and the JRE itself. The IDE plugin then needs to be evaluated in the context of an industrial project to prove its usefulness.

Currently, the plugin includes the external annotations and the nullability database as resources. The library-specific resources should be distributed through Maven as well, so that the plugin can automatically fetch them for the desired libraries. Once computed, nullability information can be reused by any user of a library, so that the burden of repeating the time-consuming analysis for the same library multiple times is removed.

Acknowledgments

The authors gratefully acknowledge the funding of the Swiss National Science Foundations for the project "Agile Software Analysis" (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018, http://p3.snf.ch/Project-162352).

References

[1] H. Osman, M. Lungu, and O. Nierstrasz, "Mining frequent bug-fix code changes," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb. 2014, pp. 343–347. [Online]. Available: http://scg.unibe.ch/archive/papers/Osma14aMiningBugFixChanges.pdf

[2] H. Osman, M. Leuenberger, M. Lungu, and O. Nierstrasz, "Tracking null checks in open-source Java systems," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Mar. 2016. [Online]. Available: http://scg.unibe.ch/archive/papers/Osma16a.pdf

[3] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Does return null matter?" in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb. 2014, pp. 244–253.

[4] M. Leuenberger, H. Osman, M. Ghafari, and O. Nierstrasz, "KOWALSKI: Collecting API clients in easy mode," in *2017 IEEE 33rd International Conference on Software Maintenance and Evolution (ICSME)*, [to appear].

[5] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot — a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[6] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing Java using attributes," in *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2000, p. 8.

[7] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM Press, 2002, pp. 1–11.

[8] P. Chalin, P. R. James, and F. Rioux, "Reducing the use of nullable types through non-null by default and monotonic non-null," *IET Software*, vol. 2, no. 6, pp. 515–531, December 2008. [Online]. Available: https://www.researchgate.net/profile/Perry_James/publication/220386928_Reducing_the_Use_of_Nul_lable_Types_through_Nonnull_by_Default_and_Monotonic_Non_null/links/54f84b3e0cf2ccffe9de0142.pdf

[9] T. J. McCabe, "A measure of complexity," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[10] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[11] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '05. New York, NY, USA: ACM, 2005, pp. 13–19. [Online]. Available: http://doi.acm.org/10.1145/1108792.1108798

[12] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251537

[13] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390657

[14] R. Madhavan and R. Komondoor, "Null dereference verification via over-approximated weakest pre-conditions analysis," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 1033–1052. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048144

[15] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for Java," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133–143. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070515

[16] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 97–107. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273478

[17] N. Ayewah and W. Pugh, "Null dereference analysis in practice," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '10. New York, NY, USA: ACM, 2010, pp. 65–72. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806686

[18] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390656

[19] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller, "Building and using pluggable type-checkers," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 681–690. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985889

[20] C. Flanagan and K. R. M. Leino, *Houdini, an Annotation Assistant for ESC/Java*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 500–517. [Online]. Available: http://dx.doi.org/10.1007/3-540-45251-6_29

[21] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431. [Online]. Available: http://design.cs.iastate.edu/papers/ICSE-13/icse13.pdf

[22] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982471

[23] A. A. Sawant and A. Bacchelli, "A dataset for API usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 506–509.