Nullable Method Detection

Inferring Method Nullability From API Usage

Master Thesis

Manuel Leuenberger
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

6. Februar 2017

Prof. Dr. Oscar Nierstrasz
Haidar Osman

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Null dereferences are the cause of many bugs in Java projects. Avoiding them is hard, as they are not detected by the compiler. Many of those bugs are caused by dereferencing values returned from methods. This finding implies that developers do not anticipate which methods possibly return null and which do not.

In this study we detect the *nullable methods* within Apache APIs by analyzing their usage in API clients. We compute the *nullability* of each invoked method, *i.e.,* the ratio between null-checked and all dereferenced method return values. To collect many API clients of Apache API, we perform a targeted API client collection. Our tool, COLUMBO, exploits the widespread use of the Maven dependency management to find clients of Apache APIs. COLUMBO is fast and scalable.

We collect and analyze 45'638 Apache API clients and measure 31.4% of conditional expressions to be null checks. We find 65.0% of dereferenced return values of Apache API methods are never checked for null, 33.5% are sometimes checked and 1.5% are always checked. A manual inspection of the methods rarely checked in client usage shows that about a third of them can never return null, hence checking the return value for null is superfluous and hinders code readability. In the Apache API clients we also analyze their usage of the JRE and we find a similar nullability distribution as in Apache usage. We consider method nullability an important part of a method contract, but we find it to be incompletely documented in the JRE API documentation. Most method documentations do not make a statement about their nullability. To bridge this gap, we integrate the nullability data in an IDE plugin that shows developers the measured nullability for each method, giving them an estimation of the potential null return.

Contents

1 Introduction

Null pointers are an evil Java developers are familiar with. The code passes tests, but exhibits defective behavior in production; it throws a `NullPointerException`. The exception is caused by a null dereference, *i.e.,* when a field is accessed or a method is called on an object reference that has not been initialized. With the stack trace of the exception at hand, it is possible to find the causing null dereference in the code, but it is hard to predict the bug at compile time. In Java, references are nullable by default and the compiler does not enforce checks on dereferences. As language tools do not detect null dereferences, we need other means to detect them.

1.1 An Empirical Approach To Find Nullable Methods

```
1 for (String field : fields) {
2    // [...]
3    Terms terms = fields.terms(field);
4    TermsEnum termsEnum = terms.iterator();
5    // [...]
6 }
```

Listing 1: Excerpt from `CompletionFieldsConsumer.write(Fields)` in Apache Lucene 6.3.0. It contains a null dereference bug on the second line that is fixed in subsequent version as in Listing 2

```
1 for (String field : fields) {
2    // [...]
3    Terms terms = fields.terms(field);
4    if (terms == null) {
5       continue;
6    }
7    TermsEnum termsEnum = terms.iterator();
8    // [...]
9 }
```

Listing 2: Excerpt from `CompletionFieldsConsumer.write(Fields)` in Apache Lucene 6.4.0. It fixes the null dereference bug in Listing 1 by guarding the dereference with a null check.

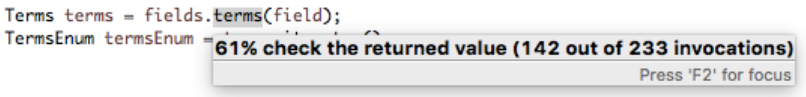

Figure 1.1: Tooltip showing how often the value returned from `Fields.terms(String)` is checked for null.

We need to understand why potential null dereferences remain undetected while writing code. Listing 1 shows a code excerpt from Apache Lucene 6.3.0. The code compiles and runs just fine in most situations, yet in some cases it throws a `NullPointerException` at the fourth line. In those cases `fields.terms(field)` on the third line returns `null`, which is assigned to `terms`. As `terms` is dereferenced at `terms.iterator()` on the fourth line, the exception is thrown. The bug was reported and fixed by guarding the dereference of `terms` with a null check,[1] as shown in Listing 2. To avoid the bug, the developer would have needed to anticipate that `fields.terms(field)` may return null in certain situations. Although the documentation of `Fields.terms(String)` notes that it might return null,[2] he did not check the returned value. Either he was not aware of this behavior, or he did not consider that the code might be executed in a context for which the method returns null. Null dereferences are common bugs in Java projects [33]. This example is representative for the majority of fixed null dereference bugs, as the majority of values checked for null are values returned from method invocations [32].

In this study we exploit a major cause of null dereferences, which are unchecked values returned from method invocations. We collect many clients of the same APIs and inspect the usage of each API method. We measure per API method how many times it is invoked by the clients and if the returned value is checked for null. We call a method whose returned value is checked for null a *nullable method*. Knowledge about method nullability gives the developer a heuristic measure that he can use to estimate the impact of a missed null check and identifies the invocations that need a second look. Figure 1.1 depicts the IDE plugin we built to visualize our experimental data for the invocation of `Fields.terms(String)` from Listing 1. The plugin shows a *nullability* of 61% for the hovered method, which can be interpreted as a warning. In 142 out of 233 usages of the method, the returned value was checked for null. A method with a nullability of 0% does not need to be checked, as it has not caused any problems before, a nullability of 100% implies that the returned value should always be checked. A nullability between the two extremes can be interpreted as a warning. The method nullability derived from the method's usage recommends the usage for a developer unfamiliar with the invoked method.

Mainstream approaches to detect null dereferences are based on static dataflow analysis of the project under test. To prove the absence of null dereferences, every dereference must be verified to be non-null on every feasible path in the control flow graph. So far, there is no sound and complete solution for this problem, which is unsurprising, regarding that language features like I/O, concurrency, reflection, and polymorphism complicate the problem by orders of magnitude. There are approaches that attempt reference verification [23], but most focus on bug detection [17, 24, 30, 44]. All of them report false positives, false negatives or undecidable instances, *e.g.,* FINDBUGS [16–18] does not warn about the bug in Listing 1, as it performs an intra-procedural analysis only. Static dataflow analysis seems to be no silver bullet to solve null dereferences.

The usage of an API in its clients can provide empirical evidence for the correct or incorrect usage of an API. This knowledge complements incomplete [4, 45], yet desired [14] API documentation with a part of the method's contract. Method nullability is a post-condition of the method contract.

Our approach to determine method nullability is based on the wisdom of the crowd. Many null

[1] `https://issues.apache.org/jira/browse/LUCENE-7562`

[2] `https://lucene.apache.org/core/6_3_0/core/org/apache/lucene/index/Fields.html#terms-java.lang.String-`

dereference bugs have been fixed and avoided in open source projects. We want to avoid repeating the mistakes others have already fixed. We analyze many API clients to collect empirical evidence for an API method's nullability by associating its incoming invocations and the null checks its returned value is used in. As API usage diversity is high [27], we collect clients of specific API instead of analyzing randomly selected projects. With this targeted dataset collection we reach a high usage count of the API's methods. The analysis marks all method invocations and associates them with null checks. In Listing 2 our analysis finds the value returned from `fields.terms(field)` to be checked often, but not the returned value from `terms.iterator()`. As analysis output, we get confidence and support measures of each method's nullability.

We present COLUMBO, a tool to collect clients of specified APIs to analyze the API's usage. COLUMBO navigates through the artifact dependency graph spanned by Maven projects to find the dependents of APIs. COLUMBO is scalable and configurable to collect the desired API clients. We collect clients of the Apache ecosystem APIs and compute the nullability of each method invoked in the clients. Based on the results of the analysis we answer the following research questions:

- **RQ1: How much API usage can be collected by dependency sub-graph extraction?**
  Within two days the analysis processes 45'638 clients of Apache APIs collected by COLUMBO. Apache client usage accounts for 7.7% of all invocations tracked to 10.3% of all invoked methods.
- **RQ2: How frequent are null checks?**
  Similar to Osman *et al.* [32], we find that 31.4% of 30'739'729 detected conditional expressions are null checks. Null usage is therefore a major driver of cyclomatic complexity [26].
- **RQ3: How many null checks are associated with method return values?**
  We find 45.5% of null checks to be associated with the return value of an invoked method. The manual inspection of samples shows that indeed most of the detected null checks are installed to check if the invoked method returned null, but we also detect instances where null is used as a temporary value to represent the state of an iterative procedure.
- **RQ4: How is method nullability distributed?**
  We compute the nullability of the invoked methods and find that 65.0% of them are never checked for null, 33.5% are sometimes checked and 1.5% are always checked. There are many methods that are rarely checked, and we find many superfluous null checks among them.
- **RQ5: How is method nullability documented?**
  In the JRE API documentation, method nullability is incompletely documented, although it is an import part of a method's contract. To fill this gap, we create an IDE plugin that shows the nullability of a hovered method.

1.2 Outline

The rest of the thesis is outlined as follows: Chapter 2 elaborates on how we collect many clients of specific APIs rapidly. In Chapter 3 we explain in detail the static analysis we apply on the collected API clients and how we associate method invocations and null checks to produce confidence and support measures for a method's nullability. Chapter 4 documents in detail the findings of our analysis of clients of Apache and JRE APIs and answers the research questions. The limitations of our approach are outlined in Chapter 5. In Chapter 6 we summarize our findings and lists ideas to improve and change our solution and how our results may be used in other contexts. We compare the requirements on our dataset collection with those of other API usage studies and compare our static analysis with those of null dereference detectors in Chapter 7.

2 API Client Collection

In this chapter we explain how we collect many clients of specific APIs rapidly, so that we can analyze the API's usage in their clients.

2.1 Requirements

We want to measure per method how often the returned values are checked for null across its clients or callers. For that we need a dataset containing many outgoing invocations to the same method to reach a relevant level of support in the results of the analysis. Mendez *et al.* [27] point out the high diversity in API usage. An API provides many callable methods, and different API clients call different subsets of those. The findings imply that analyzing randomly selected samples will not give us the required support. We want our dataset to consist of the clients of a specific API, so that our downstream analysis finds many usages of the same API methods. Besides requiring a high support, we also require the invoked methods to be precisely identifiable. For that we need precise type information about the invoked methods, namely the method signature and the declaring type. As APIs evolve over time, method contracts may change. Different versions of an API may co-exist. E.g. the method `org.apache.lucene.search.Weight.scorer()` never returned null in an early version of Lucene, but does so in later versions.[1] Hence a method invocation must be traceable to the method signature, the declaring class, the declaring API, and the API's version - a *universally unique method identifier*. We need a way to find the API clients and the universally unique identifier of the invoked method for each method invocation we detect.

2.2 Maven

```
<project>
   <groupId>org.elasticsearch</groupId>
   <artifactId>elasticsearch</artifactId>
   <version>5.0.2</version>
   <dependencies>
```

[1] `https://issues.apache.org/jira/browse/JCR-3481`

```
6       <dependency>
7          <groupId>org.apache.lucene</groupId>
8          <artifactId>lucene-core</artifactId>
9          <version>6.2.1</version>
10      </dependency>
11   </dependencies>
12 </project>
```

Listing 3: Simplified `pom.xml` of the Elasticsearch core project.

Luckily Maven[2] meets many of the aforementioned requirements. Maven is widely used in Java projects and serves as a build tool as well as a dependency management system. A Maven project is defined with a `pom.xml` file (project object model *POM*), in which it declares its dependencies. *Artifacts* are resources that are generated or used by a Maven project, *e.g.,* the JAR of a dependency. Listing 3 shows a simplified version of the POM of the Elasticsearch project for version 5.0.2. It declares a dependency to the Lucene API version 6.2.1. When the Elasticsearch project is built, Maven resolves the required dependency to Lucene to compile Elasticsearch itself. The compiled Elasticsearch classes are then packaged into a *JAR* (Java archive) artifact. The JAR and the POM artifacts can then be published on a *repository*. A repository is an index of projects and associated artifacts, *e.g.,* POMs and JARs. When Maven resolves the dependencies required to compile or run a project, it downloads the JARs containing the necessary classes from the configured repositories.[3]

As projects declare their dependencies, we can select the projects that depend on a specific API and analyze them. We can get the classes to analyze by downloading the JAR artifact of the project from a repository. The classes are compiled bytecode that contains precise type information, so that from a method invocation we can extract the method signature and the declaring class. If we download the dependencies of the analyzed artifact, we can resolve the class declaring the invoked method to the originating JAR. As artifacts are versioned, we can build a universally unique identifier for methods.

2.3 Dependency Sub-Graph Extraction

We need to find API clients with their dependencies to satisfy the requirements on our dataset. Figure 2.1 shows the dependency graph of some Maven projects and the steps to extract the sub-graph of APIs and their clients. We build our sub-graph starting from the APIs by navigating to their dependents and JARs. (Figure 2.1 a) shows the state of the graph before the procedure starts. In this section we explain in three steps how we extract the sub-graph by navigation through the graph: First, we find the APIs, the projects matching a query as marked in Figure 2.1 (b). Second, we find their respective clients, the dependent projects as marked in Figure 2.1 (c). Third, we resolve the JAR artifacts of the clients and their dependencies as marked in Figure 2.1 (d).

2.3.1 Matching Entry Points

We need to match the entry points of our dependency sub-graph, the API projects. In Figure 2.1 the APIs are tagged. Files in a Maven repository are indexed by their coordinates; the contents of the `groupId`, `artifactId`, and `version` tags, as in Listing 3. The index is designed for lookup only, but not for searching, *i.e.,* to get the POM of Apache Lucene, we lookup the descriptor `org.apache.lucene:lucene-core:6.2.1:pom`. Trying to find all Lucene files with a query like `org.apache.lucene:lucene-core` fails, as this query is not a valid index lookup. For our experiment in Chapter 4 we have no precise description of the APIs,

[2] `https://maven.apache.org/`

[3] Some widely used publicly accessible repositories are Maven Central (`https://search.maven.org/`), jcenter (`https://bintray.com/bintray/jcenter`) and clojars (`https://clojars.org/`)

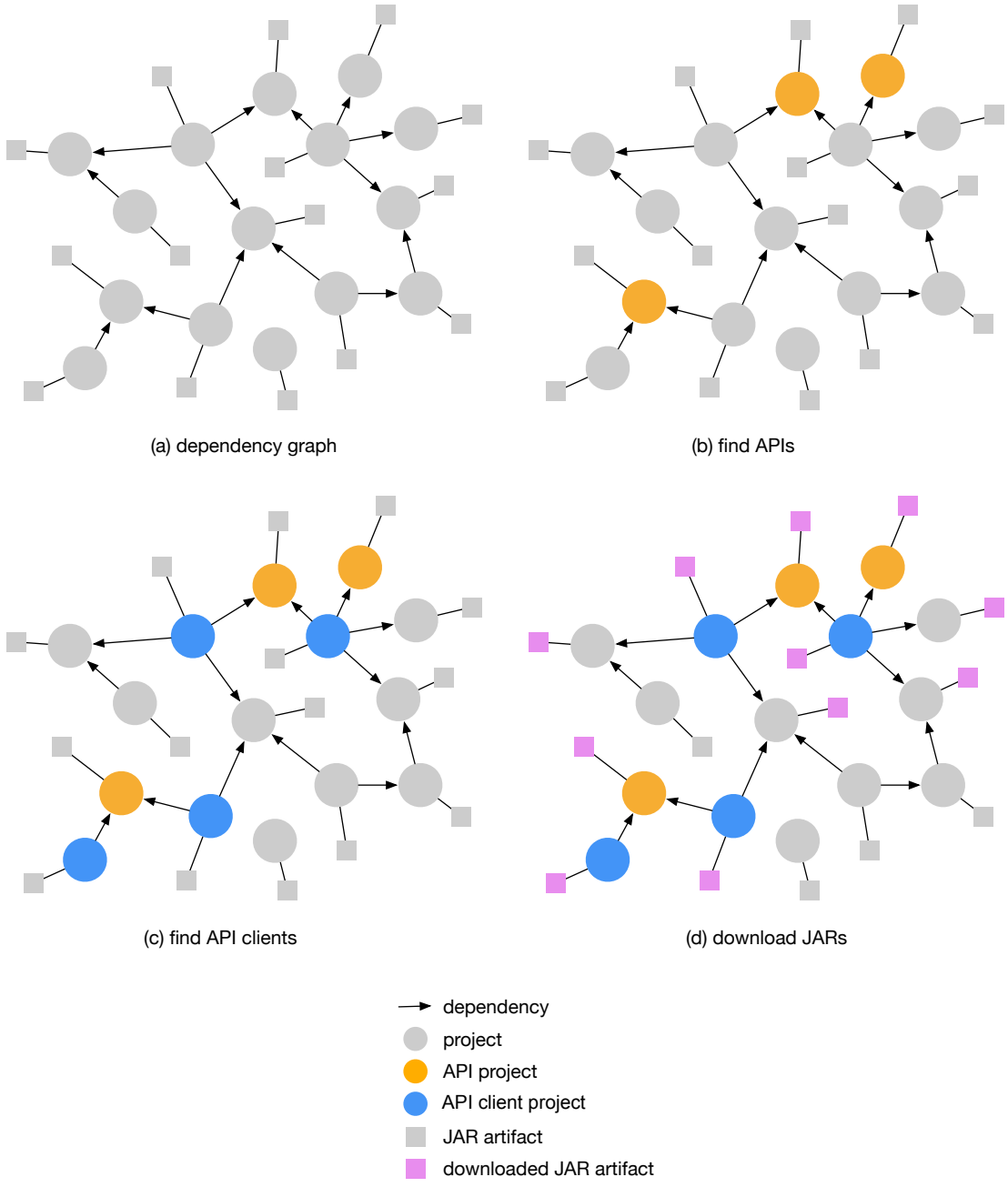

Figure 2.1: Dependency sub-graph extraction steps with project nodes marked as API, API client and JAR artifacts.

but only a vague description: The entry points of our dependency sub-graph are projects with `groupId` starting with `org.apache`. We use Maven Central search[4] to filter the artifacts that match the query to find the APIs.

2.3.2 Finding Dependents

Once we have identified the entry points, we need to find their dependents; the API clients. They are tagged in Figure 2.1 and reachable from the entry points by navigating backwards the incoming dependency edges. Projects declare their dependencies in their respective POM, as in Listing 3. But the dependencies between projects are not indexed in a Maven repository. We use mvnrepository[5] to find dependencies between artifacts to expand our sub-graph to include the dependents of the entry points.

2.3.3 Resolving Dependencies

Now that we know which artifacts are the clients of the APIs, we need to collect their JARs and their dependencies. They are tagged in Figure 2.1 as attachments to a project node and are easily navigated to from it. Maven is designed to resolve dependencies: Given an artifact descriptor like `org.elasticsearch:elasticsearch:5.0.2:jar` Maven collects the JARs of this artifact and all its dependencies by interpreting the POM. We end up with the class-path of the API client. Our dependency sub-graph is now complete.

2.4 COLUMBO

We want to analyze many artifacts. Therefore we need a big dataset that we want to collect rapidly. For this purpose we implement a tool called COLUMBO. COLUMBO is designed for high concurrency. In the previous section we described that COLUMBO must extract a sub-graph of the dependency graph. We split the extraction into three tasks: matching entry points, finding dependents, and resolving dependencies. COLUMBO wraps these tasks in such a way that they can be executed in parallel. In this section we outline COLUMBO's architecture.

2.4.1 Scenarios

COLUMBO takes a query as an input that matches some APIs. For each API, COLUMBO collects its clients and for each client it collects all required dependencies, including the API itself. *e.g.,* if the query matches all artifacts with a `groupId` starting with `org.apache`, COLUMBO finds Apache Lucene as a match. As Elasticsearch declares a dependency on Lucene, COLUMBO finds Elasticsearch as dependent. In the final step COLUMBO downloads the Elasticsearch JAR including its dependencies, so that the downstream analysis described in Chapter 3 can consume this list to analyze the usage of Lucene in Elasticsearch.

2.4.2 Logical View

Figure 2.2 shows the components of COLUMBO. The components doing the heavy lifting are the `Task`s, the `Job`s, and the `Collector`.

[4] `http://search.maven.org/`
[5] `https://mvnrepository.com/`

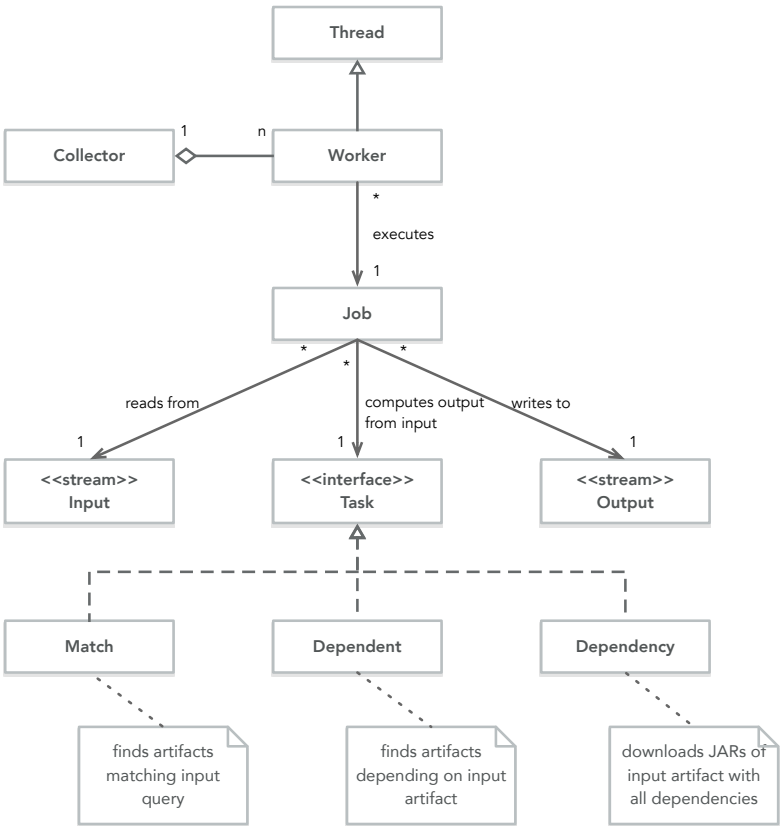

Figure 2.2: Class diagram depicting the main components COLUMBO is built upon.

2.4.2.1 Tasks

The tasks are decoupled from each other, so that they can be executed concurrently. Even multiple instances of the same task can be run in parallel. They only depend on their initial configuration and the input they are provided with. They should be viewed as the implementations of the mathematical functions for the navigations in the dependency graph.

The `Match` task takes a query in the form `g:org.apache*` as input. The task then collects all matching artifacts and outputs their descriptors, *e.g.,*`org.apache.lucene:lucene-core:6.2.1` for Lucene 6.2.1. There is no configuration necessary for the match task as the input query defines the task's behavior entirely.

The `Dependent` tasks accepts an artifact descriptor and collects their dependents. It outputs artifact descriptors again, *e.g.,*`org.elasticsearch:elasticsearch:5.0.2` for Elasticsearch 5.0.2. Again there is no configuration necessary for this task.

The `Dependency` task takes artifact descriptor of the dependent and collects the corresponding JAR and the JARs of its dependencies. This task can be configured using a traditional Maven `setting.xml`[6] to declare the repositories the JARs should be fetched from. It is also possible to configure the dependency scopes that should be used to resolve the necessary dependencies. Maven dependencies can be annotated with *compile*, *provided*, *system*, *runtime*, and *test* scopes.[7] For instance, a dependency to a unit testing framework is declared in the *test* scope. A test scoped dependency is not required to compile the project. If we were to analyze tests, we could configure the dependency task to include the test dependencies.

2.4.2.2 Jobs

The tasks are executed in jobs. Jobs are responsible for providing input for the wrapped tasks and deciding what to do with the task's output. The output stream of one job is the input stream of another job. The chaining of jobs through streams acts as the composition mechanism to build the pipeline required to produce the dataset for the downstream analysis. A job reads the task's input from an input stream, delegates the input to the task to process, and writes the task's output to an output stream. As tasks are concurrently executable and the access to input and output streams is synchronized, jobs can executed in parallel too. Input and output streams are named, so that a job can be configured by the name of the input stream, the type of the task it executes, and the name of the output stream.

2.4.2.3 Collector

The collector runs multiple workers which encapsulate the executed jobs. A worker is merely a thread running a job. All workers are executed in parallel by the collector. The collector can be configured by specifying how many workers for each job are instantiated. At startup, all jobs are created based on this configuration and executed with the number of workers desired.

2.4.3 Development View

Figure 2.3 shows the communication between the components of COLUMBO. The jobs are run in a batch processing framework. The streams are managed as named queues in an external JMS framework.[8] In this section we explain the communication between the batch jobs and the JMS queues used as streams, and how the tasks fetch and cache data.

[6] `https://maven.apache.org/settings.html`

[7] `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope`

[8] `http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html`

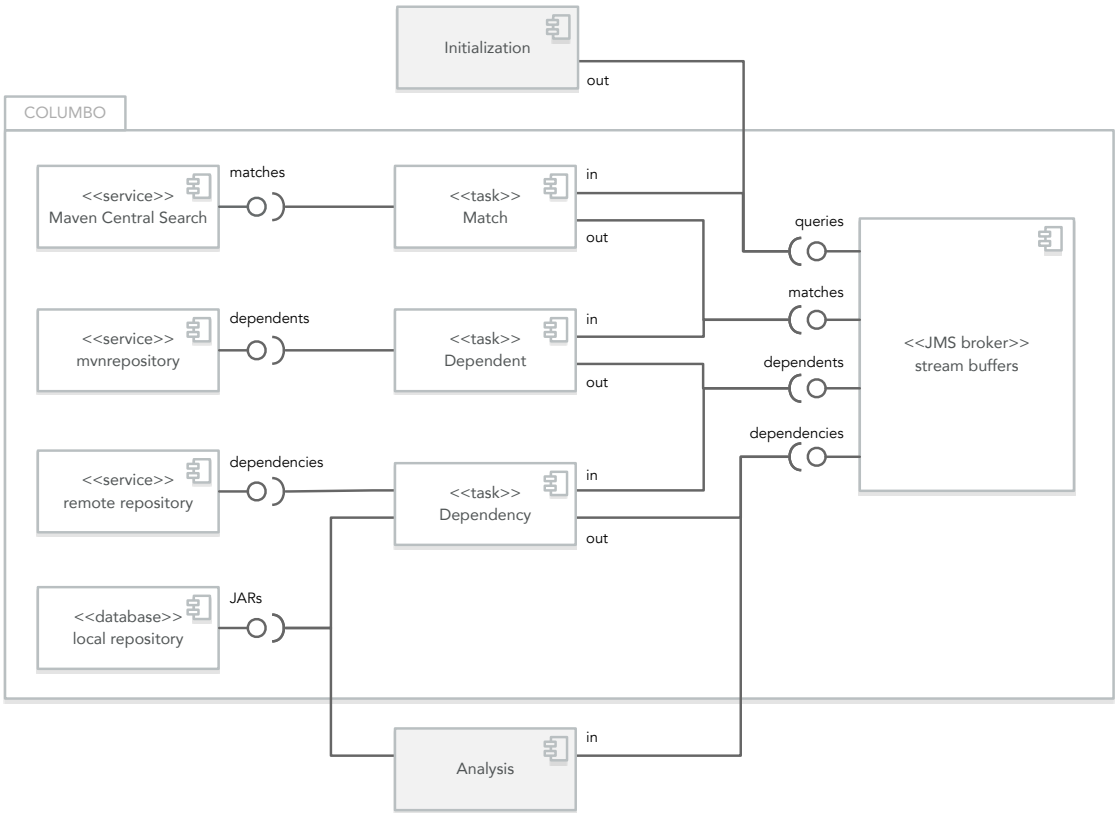

Figure 2.3: Component diagram depicting the communication between COLUMBO's components.

2.4.3.1 Streams

A batch job pulls messages from its JMS input queue. It sleeps until it fetches a message. The fetched message is deserialized into the input for the wrapped task. The task processes the input and the output is serialized into another message. The output message is pushed into the JMS output queue. JMS queues guarantee that each message can only be consumed once, hence no two jobs will process the same message. The job sleeps while there is no message in the input queue and the task is not processing an input.

2.4.3.2 Match Task

We use Maven Central Search[9] to search for APIs. Maven Central Search allows one to search for artifacts based on partial coordinates. Our query for APIs with a `groupId` starting with `org.apache` can be formulated as `g:org.apache*` and we get a list of matching artifacts as a result. The search is run over all artifacts in the Maven Central Repository. Artifacts in other repositories like jcenter or clojars are not covered, yet many widely used artifacts are published in the Central Repository. Therefore we will get most of the existing APIs.

2.4.3.3 Dependent Task

Mvnrepository[10] aggregates meta-data about many Maven Repositories, including an index over dependents. E.g. the dependents of Lucene can be found.[11] We use this index to find the dependents of the APIs.

2.4.3.4 Dependency Task

An artifact is often depended on by multiple other artifacts. We use a local repository to cache files downloaded from remote repositories. This way we can avoid downloading the same files twice, but it requires some disk space.

2.4.4 Process View

In this section we define the *DSL* (domain specific language) we use to configure COLUMBO and how it affects run-time behavior.

2.4.4.1 Configuration DSL

```
1 [<worker-pool>=<worker-pool-size>x <input-queue> -<task>-> <output-queue>]+
```

Listing 4: Grammar of the configuration DSL.

```
1 matchJobs=1x queries -match-> matches
2 dependentJobs=4x matches -dependent-> dependents
3 dependencyJobs=16x dependents -dependency-> dependencies
```

Listing 5: Configuration of COLUMBO to collect the JARs including dependencies of API clients.

[9] `http://search.maven.org/`
[10] `https://mvnrepository.com/`
[11] `https://mvnrepository.com/artifact/org.apache.lucene/lucene-core/usages`

Tasks, jobs, and workers can be composed to build the pipeline required for the dataset collection by our configuration DSL. The language grammar looks like Listing 4, where `<worker-pool>`, `<input-queue>`, `<task>` and `<output-queue>` are strings and `<worker-pool-size>` is an integer. Each configuration line declares a pool of workers that all execute the same job.

- `<worker-pool>` declares the name of the worker pool. It is used for logging.
- `<worker-pool-size>` declares the size of the worker pool. A value of 16 as in the example means that the pool will contain 16 workers of the same job.
- `<input-queue>` declares the name of the input stream. It identifies a JMS queue.
- `<task>` declares the name of the task to execute in the job. It must be `match`, `dependent` or `dependency`.
- `<output-queue>` declares the name of the output stream. It identifies a JMS queue.

The configuration in Listing 5 collects API clients with three worker pools of multiple workers for the `dependent` and the `dependency` task. The `match`, `dependent` and `dependency` tasks are chained in sequence, connected by the streams. A simpler configuration could be used if we wanted to analyze the API implementations instead.

2.4.4.2 Run-Time Behavior

Figure 2.4 shows the run-time behavior of COLUMBO configured with the configuration in Listing 5. Note that the three worker pools and the workers themselves all run in parallel. The synchronization happens through the stream connectors. The `match` and `dependent` tasks output multiple artifact descriptors per input. Those outputs can be individually processed in parallel by different workers of the subsequent task. The first result is available in the `dependencies` queue as soon as the first artifact descriptor for the initial query is received, the first dependent of the first matched API is received and its dependencies are resolved. This allows the downstream analysis to start consuming the `dependencies` queue very fast and we do not have to wait for the collection to terminate. In fact COLUMBO never terminates, as all jobs are constantly polling from their input queue.

2.5 Summary

In this chapter we explain how we design a API client collection by exploiting the wide-spread use of the Maven infrastructure for dependency management. As Maven projects declare their dependencies, we navigate through the dependency graph to extract a sub-graph containing the APIs and their clients, including all dependencies. We implemented the design in our tool called COLUMBO, which is scalable and configurable to collect the artifacts to analyze.

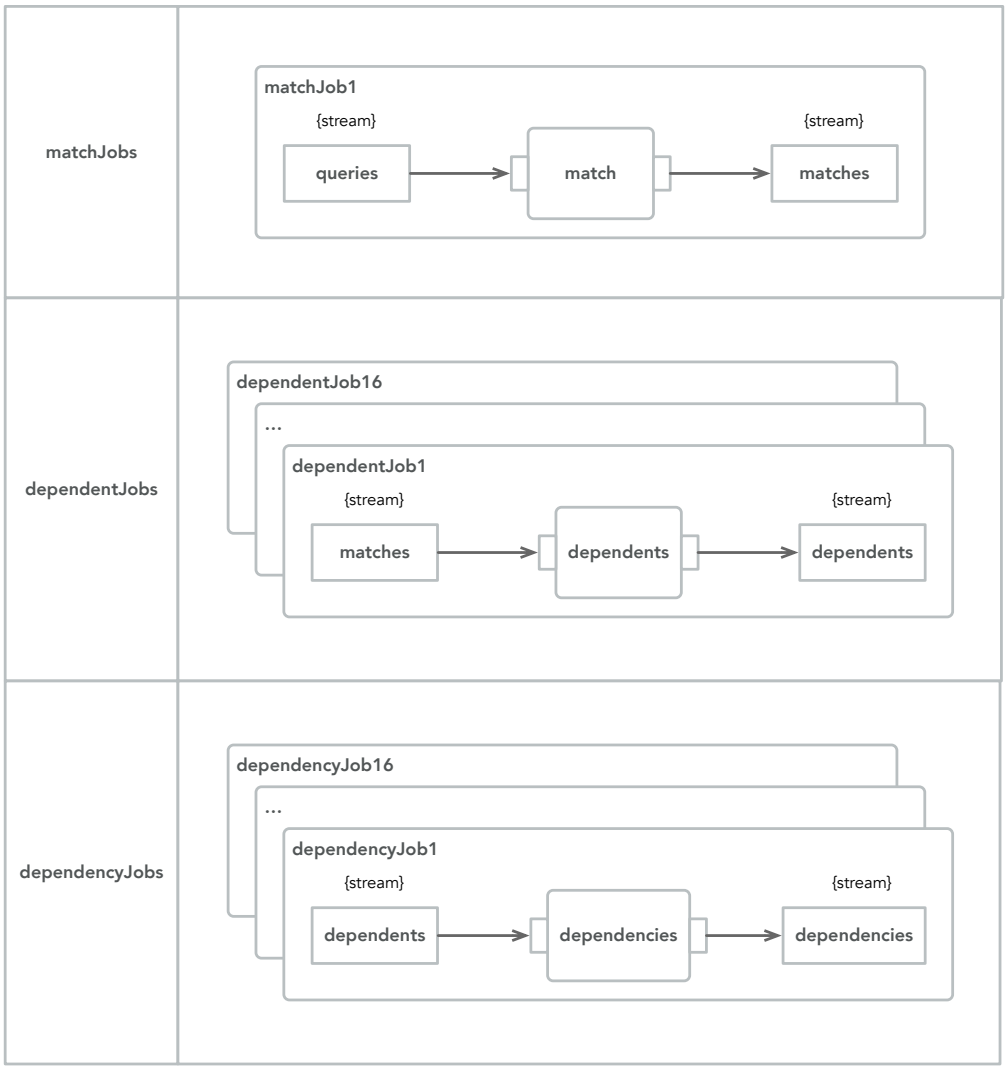

Figure 2.4: Activity diagram for COLUMBO's run-time behavior for the configuration in Listing 5.

3
Nullable Method Detection

We want to detect *nullable* methods in APIs as the values returned from methods are the biggest group of values checked for null [32]. The dataset collection outlined in Chapter 2 generates a dataset of API client JARs including their dependencies. In this chapter we associate the usage of API methods in their clients with the null checks the method's returned values are checked in. The results of the analysis tell us which methods are nullable and which are not, as their returned values are either always or never checked for null or somewhere in between.

We explain the analysis on a synthetic example of a binary tree traversal in Figure 3.1 and Listing 6. In Section 3.1 we present a procedure to extract invoked methods and their universally unique identifier. Section 3.2 describes how we extract the expressions checked for null. We explain how we associate method invocations with null checks in Section 3.3.

3.1 Invocation Analysis

The analysis processes all method bodies in the API clients and collects all outgoing method invocations. For each invoked method its universally unique identifier is extracted and we determine whether or not the returned value is dereferenced.

3.1.1 Extracting Universally Unique Method Identifiers

For each invocation we extract the universally unique method identifier to discriminate invocations to the same method over multiple analyzed projects. As the analysis runs over Java bytecode, we have precise type information about the invoked methods. We find the artifact declaring a method following these steps:

1. Determine the class of the object the method is called on.
2. Navigate the inheritance tree of the class up to the most general class declaring the invoked method.
3. Find the class in a JAR on the class-path.
4. Lookup the project corresponding to the JAR.

```
1    public void visit(Node n) {
2       MarkNode node = (MarkNode) n;
3       MarkNode left = (MarkNode) node.getLeft();
4       node.getMark().becomeGrey();
5       if (this.isRightToLeft(left)) {
6          node = (MarkNode) node.getRight();
7       } else {
8          node = left;
9       }
10      if (node != null) {
11         node.accept(this);
12      }
13   }
14
```

Listing 6: Implementation of `MarkVisitor.visit(Node)` as in Figure 3.1. A binary tree traversal with branching and a null check for the local variable `node` at the tenth line.

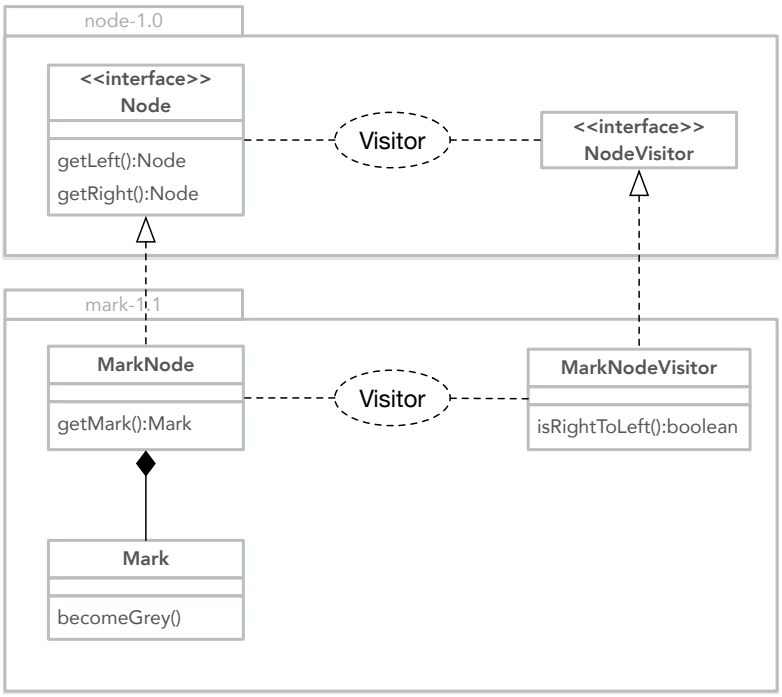

Figure 3.1: Class diagram for the visitor in Listing 6.

method	declared class	**defining class**	defining JAR	**defining project**
getLeft()	MarkNode	Node	node-1.0	nodes:node:1.0
getMark()	MarkNode	MarkNode	mark-1.1	marks:mark:1.0
becomeGrey()	Mark	Mark	node-1.0	nodes:node:1.0
isRightToLeft(MarkNode)	MarkNodeVisitor	MarkNodeVisitor	mark-1.1	marks:mark:1.0
getRight()	MarkNode	Node	node-1.0	nodes:node:1.0
accept(NodeVisitor)	MarkNodeVisitor	Node	node-1.0	nodes:node:1.0

Table 3.1: Construction of universally unique identifiers for methods invoked in Listing 6. The resulting identifier is the tuple `(method, defining class, defining project)`.

method	dereferenced
getLeft()	true
getMark()	true
becomeGrey()	false
isRightToLeft(MarkNode)	false
getRight()	true
accept(NodeVisitor)	false

Table 3.2: Dereferences of return value of methods invoked in Listing 6.

Table 3.1 lists the method invocations in Listing 6 and visualizes the construction of their corresponding universally unique identifiers. The universally unique method identifier is the tuple `(method, defining class, defining project)`. Note that we do not look up the possibly invoked method implementations, but we take the most general method definition to form the identifier. *e.g.,* `[getLeft(), getRight(), accept(NodeVisitor)]` are invoked on `MarkNode`s, but they are defined in the class `Node`. We apply Liskov's substitution principle [22], requiring covariance in return types: If a type declares a method not to return null, no overriding method in any subtype must return null. We ignore that the declared type of an invoked method might actually define a more restrictive contract: If a type declares a method to possibly return null, some overriding methods in some subtypes might never do so. *e.g.,*`Node.getLeft()` can return null by its specification, but the adapted specification of `MarkNode.getLeft()` might not allow it to return null. We lack a rationale to decide which declaration restricts a contract from possibly null to never null, as contracts are typically not formalized in Java. But if a method might return null, it must be that the contract of the most general declaration of that method allows returning null. Otherwise it would be a violation of the contract. Therefore we assume that the contract, defining whether a method might return null or not, is never restricted not to return null in a subtype.

3.1.2 Tracking Dereferences Of Values Returned From Methods

A `NullPointerException` is only thrown by the JVM when null is dereferenced. We extract whether or not an invocation's return value is dereferenced. The dereference condition ensures that a null check is required, if the value might be null. Without this condition, methods whose returned value is either not used at all, assigned to fields, passed as a parameter or returned would also be included. But these aforementioned situations do not require a null check if the value is null. This distinction is important, as an invocation for side effects only, *i.e.,*, its return value is not accessed, reveals no information about the nullability of the invoked method. For instance, the value returned from `mark.becomeGray()` is never dereferenced, hence it cannot count as evidence for or against the invoked method's nullability. But if `mark` is dereferenced as the value returned from `node.getMark()`, we can detect whether or not the dereferenced value is checked for null before it is dereferenced.

Table 3.2 denotes which invocations are dereferenced. The values returned from `becomeGrey()` and `accept(NodeVisitor)` are never referenced, the methods are just invoked for side effects. `isRightToLeft(MarkNode)` returns a primitive type, not a reference, thus the returned value cannot be checked for null. This leaves `getLeft()`, `getMark()`, and `getRight()` whose returned values are dereferenced. Following the definition-use chains of the corresponding assignments, we find that the reference returned from `getMark()` is dereferenced on the third line. The return values of `getLeft()` and `getRight()` are dereferenced at the tenth line.

3.2 Null Check Analysis

Now that we know which method return values are dereferenced, we learn which values are checked for null and which are not. In an earlier study we tracked null checks to find what kinds of expressions are checked for null [32]. We replicate parts of the prior analysis and extend it with additional type information about invoked methods. Our analysis is based on use-definition chains to determine if the returned value is checked for null. We account for variable aliasing. In this section we explain how our null check analysis works.

3.2.1 Identifying Null Checks

Definition 1 (Null Check)**.** Every comparison with the `null` constant is a *null check*.

Only object references can be checked for null, therefore the expression checked for null cannot be of a primitive type. Object references can only be checked for equality, using the == operator, or inequality, using the `!=` operator. We identify null checks as defined in Definition 1 as binary expressions involving the `null` constant. If the null check is not performed within the analyzed method, *e.g.,*, if `Objects.requireNonNull()` is used, we do not detect it as a null check, as our analysis is entirely intra-procedural.

3.2.2 Identifying Nullable Expressions

Definition 2 (Nullable Expression)**.** The expression that is checked for null is called the *nullable expression*.

We want to find what is checked for null. Listing 6 shows a null check of the `node` variable. We call `node` the nullable expression as defined in Definition 2. The null expression is the part of the null check that is not the null constant. A null expression may not be a variable, but can be any expression that evaluates to an object. E.g. a method return value as in `map.get(key) != null`.

3.2.3 Reducing Nullable Expressions

Definition 3 (Locally Scoped Variables)**.** *Locally scoped variables* live exclusively for an execution of a method. Their entry in the variable lookup map is created when a method starts the execution and destroyed when it terminates.

If a variable is checked for null, we want to know what they reference at the null check. As our analysis is intra-procedural, local variables and method parameters are the only variables whose referenced values we can accurately reason about. We call them *locally scoped variables*, as defined in Definition 3. Any field variable like `this.field` or `other.field` might be changed as a side effect of the execution of an outgoing method invocation or another thread. An intra-procedural analysis cannot account for those situations, so we do not track field variables.

Definition 4 (Locally Reduced Nullable Expressions)**.** The set of nullable expressions in which all locally scoped variables are replaced by their assigned, uncasted expressions is called *locally reduced nullable expressions*.

We construct the *locally reduced nullable expressions* for each null check as defined in Definition 4. This is achieved with a flow-sensitive analysis with use-definition chains. As an example, we want to construct the locally reduced nullable expressions for the null check at the tenth line in Listing 6. We identify `node` as the nullable expression, hence our initial locally nullable expressions are `[node]`. As `node` is a

method	invocations	null checks	nullability
getLeft()	1	1	1
getMark()	1	0	0
getRight()	1	1	1

Table 3.3: Method nullability for dereferenced method return values in Listing 6, as $null\,checks/invocations$.

locally scoped variable, we reduce it by replacing it with the expressions assigned to it. The use-definition chain for `node` at the null check give us the potentially assigned expressions `node.getRight()` and `left`, which form the new set `[(MarkableNode) node.getRight(), left]`. Note that there is no path to the null check where `node` could have the value of `(MarkNode) n` assigned at the first line, therefore `(MarkNode) n` must not be included in the reduction. The new set contains again a locally scoped variable, `left`, which we reduce again. `left` is assigned to `node` at the eighth line. At this point, the only definition of `left` is `MarkableNode left = (MarkableNode) node.getLeft();` on the second line. Now the set is `[(MarkableNode) node.getRight(), (MarkableNode) node.getLeft()]`. The cast of an expression is irrelevant for our purpose of deciding about the nullness of the expression, therefore we can ignore the cast itself and continue with the cast expression. The locally reduced nullable expressions for the null check in the example are therefore `[node.getRight(), node.getLeft()]`.

3.3 Method Nullability

The result of the analysis is a combination of the extracted null checks and method invocations. All dereferenced method return values are prone to a null dereference, if the invoked method might return null. We group the dereferenced methods return values by the invoked method and count how often the method is invoked and how often the returned value is checked for null. We call the ratio $null\,checks/invocations$ *method nullability*.

For the accompanying example in Listing 6 we compile the method nullabilities in Table 3.3. `[getLeft(), getRight()]` have a nullability of 1, meaning that all dereferenced return values are checked for null. Contrary, `getMark()` has a nullability of 0, meaning that no dereferenced return value has been checked for null.

In the experiment in Chapter 4 we apply the analysis to a large corpus of API clients. We calculate the nullability for many methods and try to differentiate between *nullable* and *non-nullable* methods based on the method's usage.

3.4 Implementation Notes

```
1  // Method descriptor #17 (LNode;)V
2  // Stack: 2, Locals: 4
3  public void visit(Node n);
4     0  aload_1 [n]
5     1  checkcast MarkNode [18]
6     4  astore_2 [node]
7     5  aload_2 [node]
8     6  invokevirtual MarkNode.getLeft() : Node [20]
9     9  checkcast MarkNode [18]
10   12  astore_3 [left]
11   13  aload_2 [node]
12   14  invokevirtual MarkNode.getMark() : Mark [24]
```

```
  17  invokevirtual Mark.becomeGrey() : void [28]
  20  aload_0 [this]
  21  aload_3 [left]
  22  invokespecial MarkNodeVisitor.isRightToLeft(MarkNode) : boolean [33]
  25  ifeq 39
  28  aload_2 [node]
  29  invokevirtual MarkNode.getRight() : Node [37]
  32  checkcast MarkNode [18]
  35  astore_2 [node]
  36  goto 41
  39  aload_3 [left]
  40  astore_2 [node]
  41  aload_2 [node]
  42  ifnull 50
  45  aload_2 [node]
  46  aload_0 [this]
  47  invokevirtual MarkNode.accept(NodeVisitor) : void [40]
  50  return
    Line numbers:
      [pc: 0, line: 6]
      [pc: 5, line: 7]
      [pc: 13, line: 8]
      [pc: 20, line: 9]
      [pc: 28, line: 10]
      [pc: 36, line: 11]
      [pc: 39, line: 12]
      [pc: 41, line: 14]
      [pc: 45, line: 15]
      [pc: 50, line: 17]
    Local variable table:
      [pc: 0, pc: 51] local: this index: 0 type: MarkNodeVisitor
      [pc: 0, pc: 51] local: n index: 1 type: Node
      [pc: 5, pc: 51] local: node index: 2 type: MarkNode
      [pc: 13, pc: 51] local: left index: 3 type: MarkNode
    Stack map table: number of frames 3
      [pc: 39, append: {MarkNode, MarkNode}]
      [pc: 41, same]
      [pc: 50, same]
```

Listing 7: Java bytecode version of Listing 6.

```
public void visit(Node)
   {
      MarkNodeVisitor r0;
      Node r1, $r3, $r5;
      MarkNode r2, r6, r7;
      Mark $r4;
      boolean $z0;
      r0 := @this: MarkNodeVisitor;
      r1 := @parameter0: Node;
      r6 = (MarkNode) r1;
      $r3 = virtualinvoke r6.<MarkNode: Node getLeft()>();
      r2 = (MarkNode) $r3;
      $r4 = virtualinvoke r6.<MarkNode: Mark getMark()>();
      virtualinvoke $r4.<Mark: void becomeGrey()>();
      $z0 = specialinvoke r0.<MarkNodeVisitor: boolean isRightToLeft(MarkNode)>(r2);
      if $z0 == 0 goto label1;
      $r5 = virtualinvoke r6.<MarkNode: Node getRight()>();
      r7 = (MarkNode) $r5;
      goto label2;
    label1:
      r7 = r2;
```

```
22     label2:
23       if r7 == null goto label3;
24       virtualinvoke r7.<MarkNode: void accept(NodeVisitor)>(r0);
25     label3:
26       return;
27   }
```

Listing 8: Jimple version of Listing 6.

The analysis is implemented as an intra-procedural analysis in the SOOT [36, 46] framework.[1] SOOT transpiles bytecode into the Jimple *IR* (intermediate representation), on which the analysis is run. Jimple programs are typed and based on three-address statements. The language is stackless, in constrast to Java bytecode. Jimple has only 15 statements compared to over 200 instructions in Java bytecode. [10] Those features turn out to be very convenient to implement our analysis.

As an example we look at the procedure to extract the type of the receiver of `becomeGray()` on line four Listing 6. To get the receiver type in Java source code, we find it to be return type of `node.getMark()`. This method is defined in another source file, hence we lack the necessary type information to determine its return type. We could run a type checker on the sources to get this information, but from there the step to compiled bytecode is not that big anymore. The method from Listing 6 looks like Listing 7 when compiled to bytecode. In bytecode we need a stack machine to extract the receiver type. The method `becomeGray()` is invoked on the stack top at label 17. We find that the top of the stack is the return value of `getMark()` invoked at label 14. From the instruction `invokevirtual MarkNode.getMark() : Mark` we can get `Mark` as the receiver type we are looking for. The task becomes even easier in Jimple, as it is closer to Java source code. In Listing 8 we see again the same method as in Listing 6, but in Jimple IR. We locate the invocation of `becomeGrey()` at line 14. The method is invoked on a variable `$r4`. The practiced Java source code reader can instantly find that `$r4` is declared to be of the type `Mark` at line 6. Jimple declares additional temporary variables to account for chained invocations. This makes it easier to reason about data flow, as complex constructs are replaced by simpler ones. Jimple might be more verbose than source code, but it is composed only of a few simple abstractions.

The same elegance of Jimple shines in complex boolean expressions: As they are lazily evaluated in Java, in Jimple they are represented as sequence of `if`s checking temporary variables and `goto`s to the corresponding labels. For many situations like this, which we suspected to require special care, the Jimple IR provides simple solutions.

3.5 Summary

In this chapter we outline the analysis we apply on API clients to detect nullable methods in the APIs. The method nullability is the ratio of the number of null checks that check the value returned from the method and the number of invocations of the methods. We resolve from an outgoing method invocation the invoked method, the class declaring the invoked method and the artifact containing the declaring class to build the universally unique method identifier. The dataflow analysis to associate null checks with invoked methods is flow-sensitive and intra-procedural. It only looks at parameters and local variables, but not at fields or array elements as the could be changed by concurrent threads or side-effects of invoked methods. The analysis is applicable to the output of COLUMBO, so that we can trace the null check of a method return value in different API clients to the same API method.

[1] `https://sable.github.io/soot`

4 Experiments

We run the analysis described in Chapter 3 on thousands of artifacts collected by the approach outlined in Chapter 2 to compute the nullability of each invoked method by API clients.

4.1 Setup

For our experiment we deploy COLUMBO to a multi-core Ubuntu server, as shown in Figure 4.1. The collection and analysis runs on a 64 bit Ubuntu machine with 32 cores at 1.4 GHz and 128 GB of RAM. Additionally to COLUMBO we install ActiveMQ[1] as a JMS server and reserve some disk space for the local Maven repository. ActiveMQ stores the messages written to the JMS queues by COLUMBO. The dependency task from COLUMBO writes downloaded JAR and POM artifacts to the local Maven repository, so that they can be read by the downstream analysis. APIs are collected with the query `g:org.apache* AND p:jar`, which collects all JAR artifacts with a groupId starting with `org.apache`. As not all Apache APIs start have a matching groupId, *e.g.,*`commons-io`, we do not collect their dependents directly. But some dependents of matching Apache APIs might as well depend on `commons-io` As we collect all dependencies and track all invocations in the dependents, we still get some data about `commons-io` usage. We run 16 worker threads to collect the dependencies of multiple clients in parallel. The collected clients are analyzed in 24 concurrent processes. It took two days to analyze 45'638 clients. The results of the analysis are stored in a Neo4j[2] graph database, from which they are aggregated to answer our research questions.

4.2 Results

In this section we answer the research questions and discuss the implications of the results.

[1] `http://activemq.apache.org/`
[2] `https://neo4j.com/`

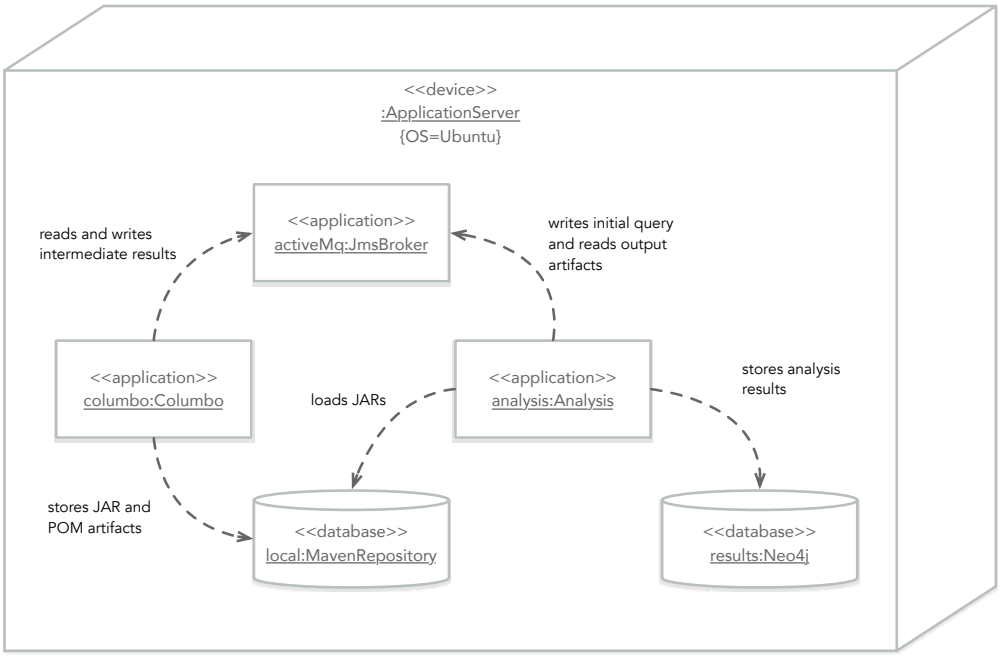

Figure 4.1: Deployment diagram of COLUMBO for our experiment.

4.2.1 RQ1: How much API usage can be collected by dependency sub-graph extraction?

We analyze 37'335'531 method bodies belonging to 4'776'388 classes in 45'638 clients collected by COLUMBO. Those method bodies contain 152'144'309 outgoing method invocations, of which 19.6% invocations have their returned value dereferenced. Only for 2.7% out of all invocations can we not infer the universally unique identifier for the invoked method. This is caused by missing dependencies and errors in the inference of the method identifier. We collect only dependencies in the `compile` scope. Dependencies in the scopes `system`, `provided` *etc.* are not collected and therefore the JARs of the classes cannot be found. The dereferenced invocations are are tracked to 2'403'667 methods. Half of those methods have at most two invocations, 95% have no more than 16 invocations. `StringBuilder.append(String)` accounts for a whopping 7'950'360 invocations.

Figure 4.2 shows the composition of the dataset consisting of dereferenced invocations extracted by COLUMBO. We query for Apache client usage, which accounts for 7.7% of all invocations tracked to 10.3% of all methods. As Apache artifacts often rely on other Apache artifacts, we also analyze those as clients of their Apache dependencies. We count 7.3% of invocations of Apache artifacts to methods defined in themselves. More than half of all invocations are calling JRE methods. But the invoked JRE methods account for only 0.1% of all methods. The JRE standard libraries are used extensively. We also note that about two thirds of invoked methods cannot be tracked to the JRE or Apache, but belong to other dependencies or the calling artifacts itself.

We conclude that COLUMBO, our implementation of dependency sub-graph extraction, finds significant client usage of APIs. We find many Apache artifacts that depend on other Apache artifacts, dependencies in their own ecosystem. It might be interesting to examine intra-artifact and intra-ecosystem usage as an oracle for usage patterns.

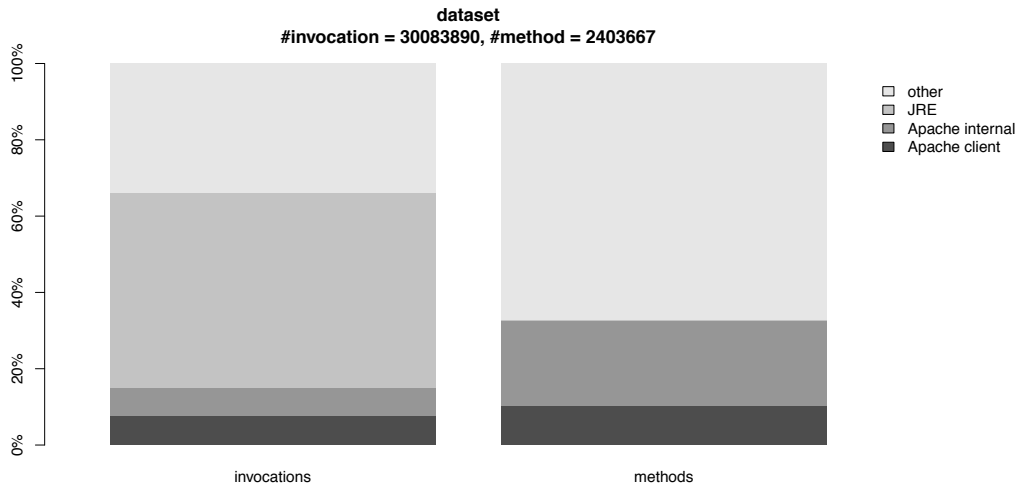

Figure 4.2: Composition of dataset of dereferences invocations extracted by COLUMBO.

4.2.2 RQ2: How frequent are null checks?

In accordance with Osman *et al.* [32], we find that about one out of three conditionals is a null check. We count 30'739'729 conditional expressions, 31.4% of which are null checks. This result implies that null checks are responsible for a significant share of cyclomatic complexity [26]. We cannot conclude that using null should be avoided by all means to reduce complexity and increase maintainability of API clients, as null values and the corresponding null checks are used to represent and handle a domain state. Using an alternative to `null`, *e.g.,* Java 8 `Optional`, to represent this state moves the complexity elsewhere, but does not necessarily reduce it. But as Osman *et al.* also find varying semantics of `null` in different contexts, often representing a failure [33], and as neither the Java compiler nor development tools detect null references, null usage should be documented.

4.2.3 RQ3: How many null checks are associated with method return values?

Our analysis finds 45.5% of null checks to be associated with the return value of an invoked method. This result is significant, but not as high as the 70% reported by Osman *et al.* [32, 33]. We attribute this discrepancy to the differing scopes of analysis: In this study we only track assignments to method scope variables. Osman *et al.* included fields and array elements as well. Both analyses are intra-procedural, but our analysis is flow-sensitive, while Osman *et al.*'s is based on the heuristic that a null-checked value is assigned on any source code line before the null check. The analysis in this study is built with the goal of a low false positive rate, therefore we protect it against possible external state changes from concurrency or side effects of outgoing method invocations. As the remaining 55.5% of null checks cannot be associated with method invocations by our analysis, they are associated with parameters, fields or array elements. Presumably, some of those values are also assigned method return values, therefore we consider the 45.5% of null checks associated with method return values to be an under-approximation.

```
1 public SolrInputDocument readDoc(XMLStreamReader parser) throws XMLStreamException {
2    // [...]
3    Collection<SolrInputDocument> subDocs = null;
4    // [...]
```

```
5     boolean complete = false;
6     while (!complete) {
7        // [...]
8        switch (event) {
9           // [...]
10          case XMLStreamConstants.END_ELEMENT:
11             // [...]
12             if (subDocs != null && !subDocs.isEmpty()) {
13                doc.addChildDocuments(subDocs);
14                subDocs = null;
15             }
16             complete = true;
17             // [...]
18             break;
19          case XMLStreamConstants.START_ELEMENT:
20             // [...]
21             if (subDocs == null) {
22                subDocs = Lists.newArrayList();
23             }
24             subDocs.add(readDoc(parser));
25             // [...]
26             break;
27       }
28    }
29    // [...]
30 }
```

Listing 9: Shortened method from `XMLLoader` of `org.apache.solr:solr-core:4.8.1`. Note the temporary live state `subDocs`.

We manually inspect 100 random samples that the analysis identifies as null checks of method return values to verify the correctness of the analysis. We verify that there exists an intra-procedural path on which the return value from the method invocation ends up being checked for null. Some of those paths might still be infeasible due to inter-procedural dependencies, but resolving this problem is outside the scope of a intra-procedural analysis. For 95 samples the analysis correctly identifies the null check to check the respective method return value. Mostly, the method return value is checked immediately after the method invocation. There is also a handful of cases in which null is used as a temporary live state over some loop iteration, as in Listing 9. The variable `subDocs` is initialized with null. At a start tag in an XML document `subDocs` is lazily initialized with `Lists.newArrayList()`. At an end tag it is used again, but it is checked for null as there might be invalid XML documents containing an end tag, but not a corresponding start tag. If it exists though, it is assigned null again. The intention of these null checks is not to verify that the method actually returned a non-null value, but to check in which state the procedure is. The `subDocs` variables is only alive for some iterations in the loop, but needs to be declared outside the loop scope to reuse it over multiple iterations.

```
1  public Model getArchetypePom(File jar) throws XmlPullParserException, UnknownArchetype,
2        IOException {
3     String pomFileName = null;
4     // [...]
5     while (enumeration.hasMoreElements()) {
6        // [...]
7        String entry = el.getName();
8        if (entry.startsWith("META-INF") && entry.endsWith("pom.xml")) {
9           pomFileName=entry;
10       }
11    }
12    if (pomFileName == null) {
13       return null;
14    }
```

```
15    // [...]
16 }
```

Listing 10: Shortened method from `DefaultArchetypeArtifactManager` of `org.sonatype.maven.archetype:archetype-common:0.8.5`. Note the dereference of `entry`, the assigment `pomFileName` to `entry`, and the null check.

```
1  protected boolean normalizeDocument(Element e,
2                                      boolean cdataSections,
3                                      boolean comments,
4                                      boolean elementContentWhitepace,
5                                      boolean namespaceDeclarations,
6                                      boolean namespaces,
7                                      boolean splitCdataSections,
8                                      DOMErrorHandler errorHandler) {
9     // [...]
10    for (int i = 0; i < nnm.getLength(); i++) {
11       Attr a = (Attr) nnm.item(i);
12       String prefix = a.getPrefix(); // todo : this breaks when a is null
13       if (a != null && XMLConstants.XMLNS_PREFIX.equals(prefix)
14             || a.getNodeName().equals(XMLConstants.XMLNS_PREFIX)) {
15          // [...]
16       }
17    }
18    // [...]
19 }
```

Listing 11: Shortened method from `AbstractDocument` of `fr.avianey.apache-xmlgraphics:batik:1.8`. Note the dereference of `a` before the null check. Interestingly the comment notes this mistake.

Five samples are classified as false positives: In all falsely classified samples the value is dereferenced before it is checked for null, hence an exception is thrown at this point and the method return value never ends up being checked. See Listing 10 as an example. One curious sample in Listing 11 comments the dereference as unsafe.

The manual inspection reveals that the analysis is precise, although it does not rule out some infeasible paths. The issue is that the analysis does not verify whether or not the dereference of the method return value is guarded by a null check. It only tracks if a method return value is dereferenced and if it is checked for null, but it does not extract the order in which those operations happen. To circumvent this limitation the analysis would need to be path-sensitive. To gain path-sensitivity, the analysis could be extended with a *belief set* about the nullity of a value that is propagated along the paths in the control flow graph, similar to the approach introduced by Engler *et al.* [11] and formalized by Dillig *et al.* [7].

4.2.4 RQ4: How is method nullability distributed?

We calculate the nullability for each invoked method as defined in Section 3.3. Figure 4.3 shows histograms of the nullability of all method that are invoked and their returned value is dereferenced. The dereference condition excludes all methods that do not return a value or return a value of a primitive type, as those values cannot be dereferenced. We notice that most of the methods have a nullability of 0, *i.e.,* they are never checked for null. Some methods have a nullability of 1, *i.e.,* they are always checked for null. Then there are a other methods with a nullability between the two extremes, *i.e.,* they are sometimes checked and sometimes not.

Half of the methods in the dataset are invoked at most twice, which could lead to an over-representation of methods with a nullability of 0 or 1: For a method that is only invoked once we can only measure a nullability of 0 or 1. To rule out those extreme cases, we look at the methods with at least ten invocations

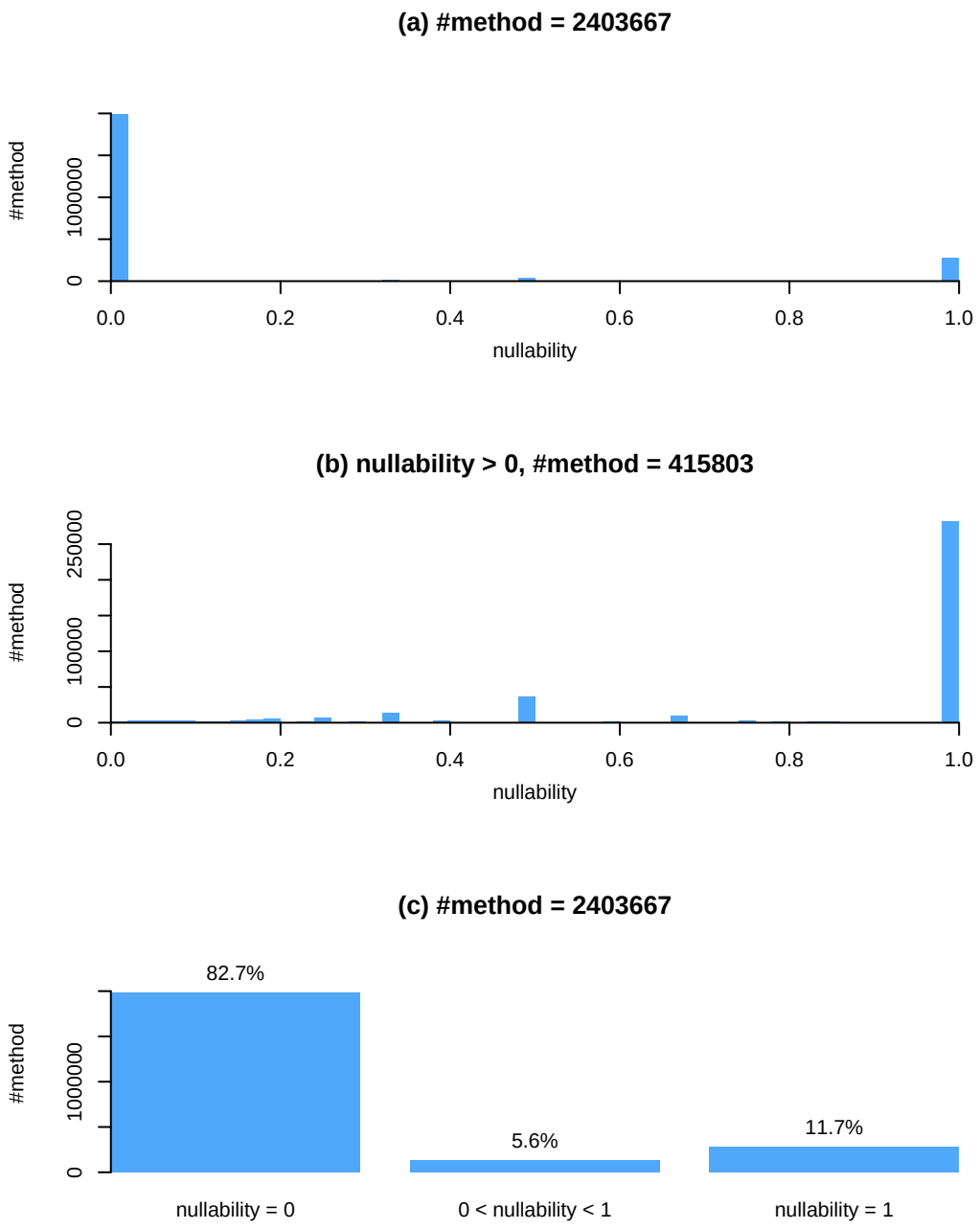

Figure 4.3: Method nullability of all used methods with at least 1 invocation.

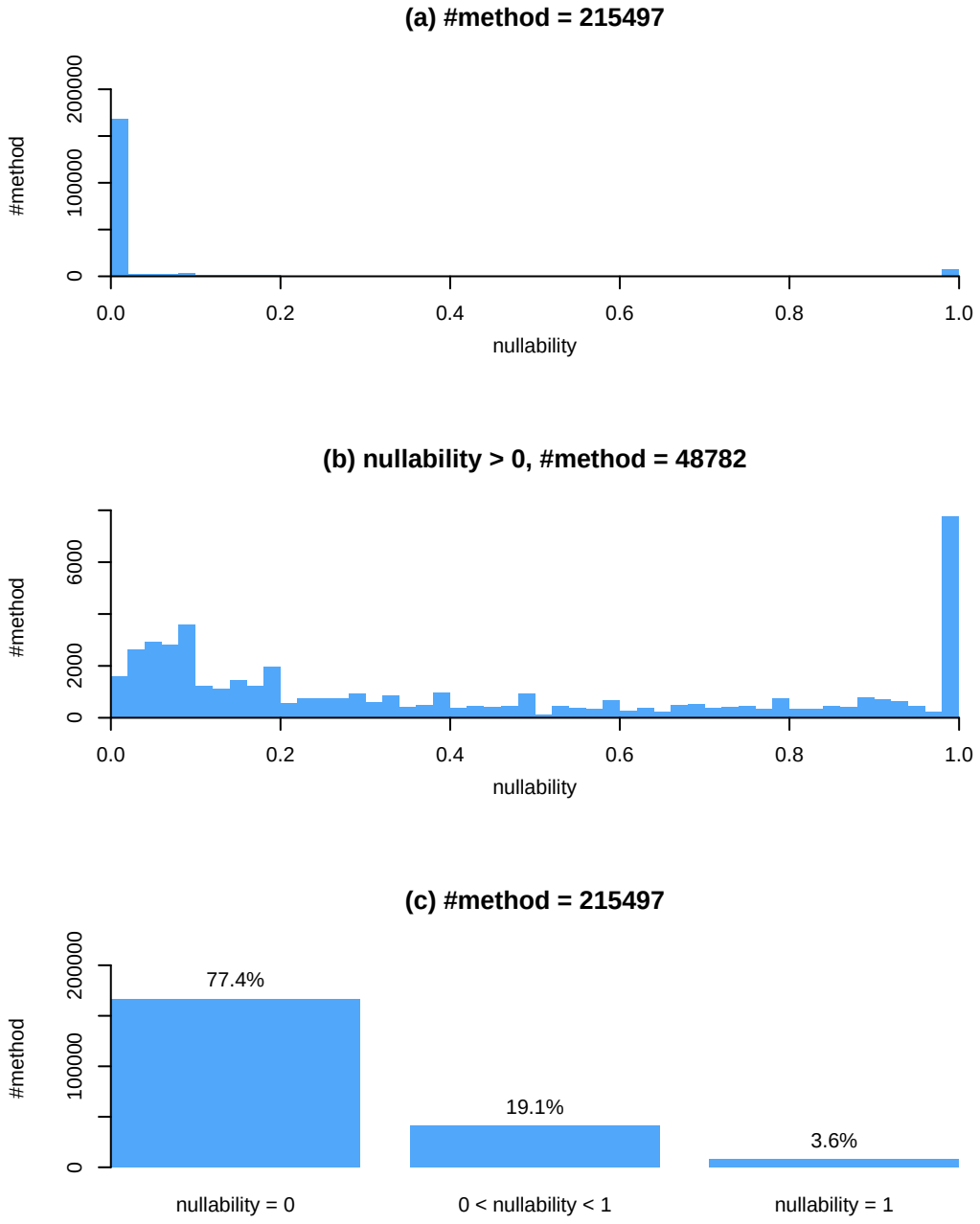

Figure 4.4: Method nullability of all used methods with at least 10 invocations.

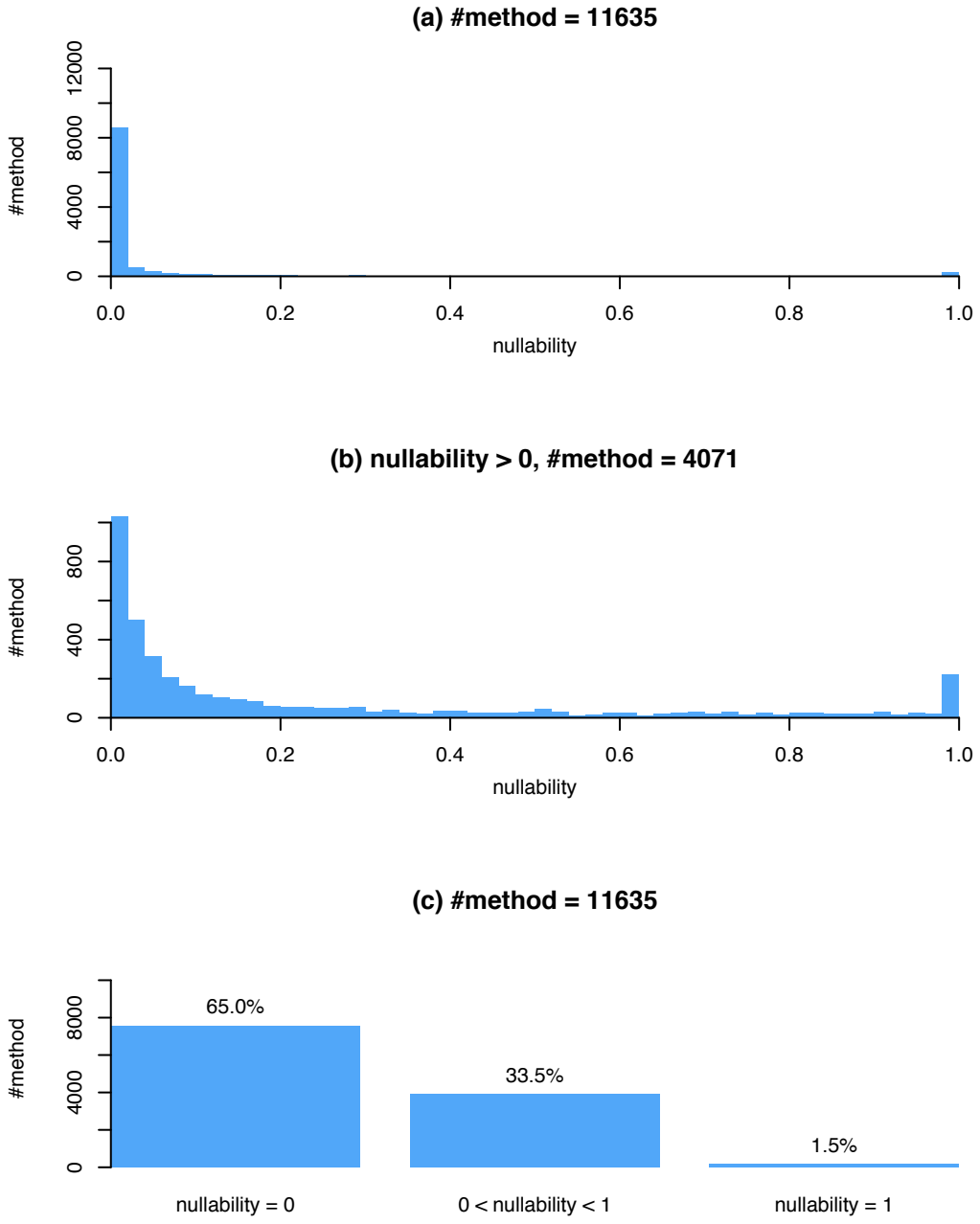

Figure 4.5: Method nullability of all used methods with at least 100 invocations.

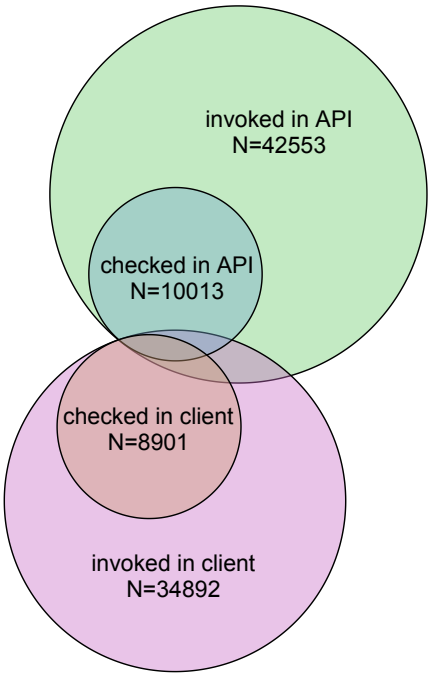

Figure 4.6: An illustration of method usage within the API itself and in client usage of Apache methods with at least 10 invocations.

and notice the same three-class pattern, see Figure 4.4. We see that the extremes are indeed over-represented when including methods with a handful of invocations, but the sample now includes only about a tenth of the original set or 215'497 methods.

If we increase the lower invocation bound to 100, we find in Figure 4.5 that 65.0% of the invoked methods are never checked for null, 33.5% are sometimes checked and 1.5% are always checked. Those three classes are clearly separable. The class of sometimes-checked methods exhibits an interesting distribution, as there are many methods that are rarely-checked and only a few that are often checked.

The focus of the analysis lies on client usage of Apache APIs. To some extent the Apache artifacts are their own clients, as they call the APIs they define themselves. As developers of an API generally have a better understanding of the details of the APIs than the developers of API clients do, we check if we notice a difference in the measured nullability distribution. We partition the Apache API usage into the internal usage, *i.e.,* how an artifact uses its own APIs, and client usage, *i.e.,* how other artifacts use the provided APIs. Note that the two datasets are not discriminated to the same methods that are used both internally and by clients, and we do not analyze all Apache artifacts for which we collect clients. Refer to Figure 4.6 to find that only some methods are invoked both internally and by clients. We notice a difference between the client usage in Figure 4.7 and the internal usage in Figure 4.8. Both datasets show the three separable classes and a similar distribution of sometimes-checked methods, yet they disagree on the sizes of the classes. In client usage we find 25.2% of the methods to be sometimes checked, but only 17.4% of the methods in internal usage. As we measure more rarely-checked than often-checked methods, we suspect that some null checks are unnecessary. We look at the implementations of the rarely-checked methods in client usage to see if they are nullable or not. We determine that 29 of 100 methods are not nullable, as they always instantiate the returned object. Most of those methods return collections. For the remaining methods we cannot determine the nullability from the method implementation alone. Some contain an

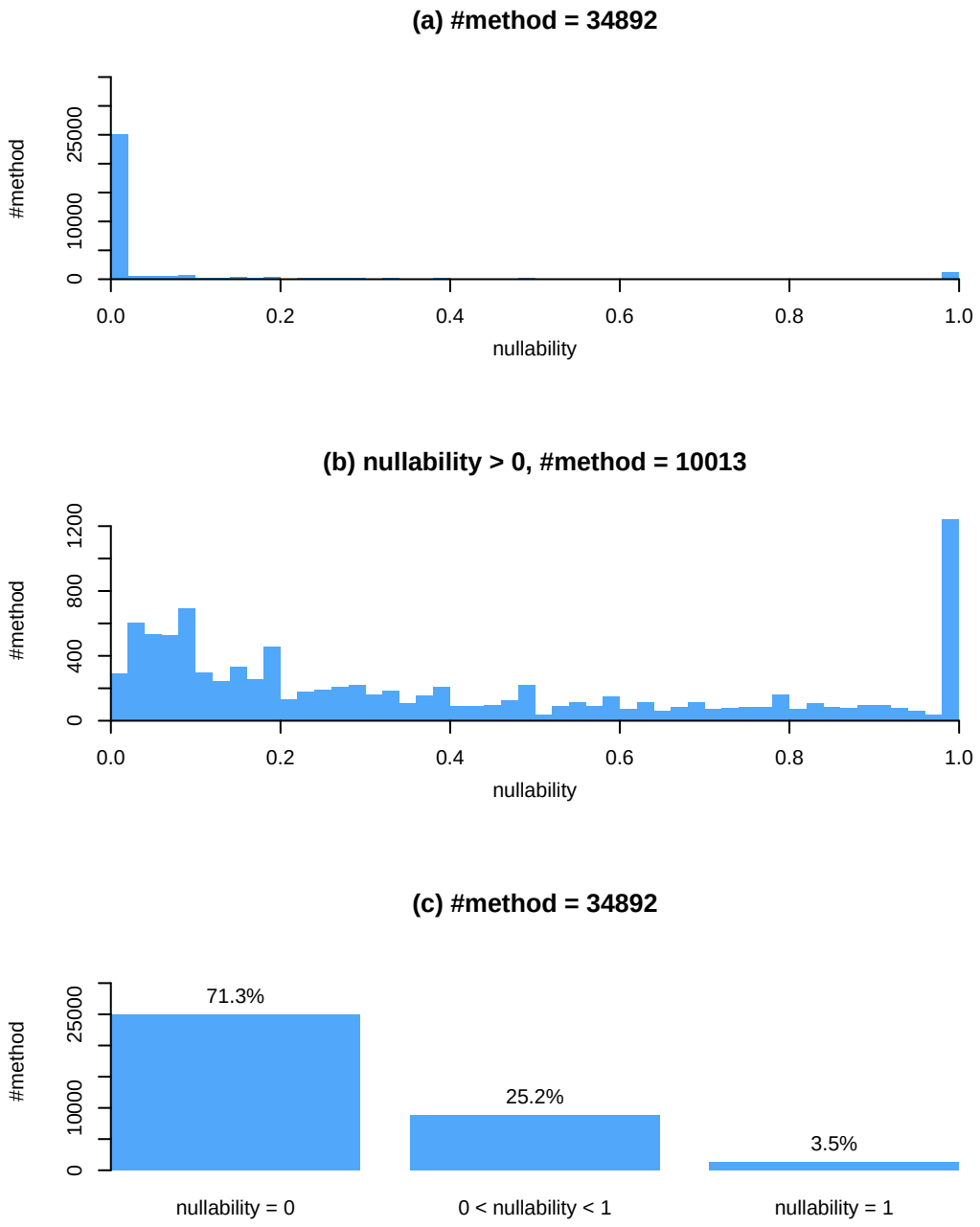

Figure 4.7: Method nullability of Apache methods used in clients with at least 10 invocations.

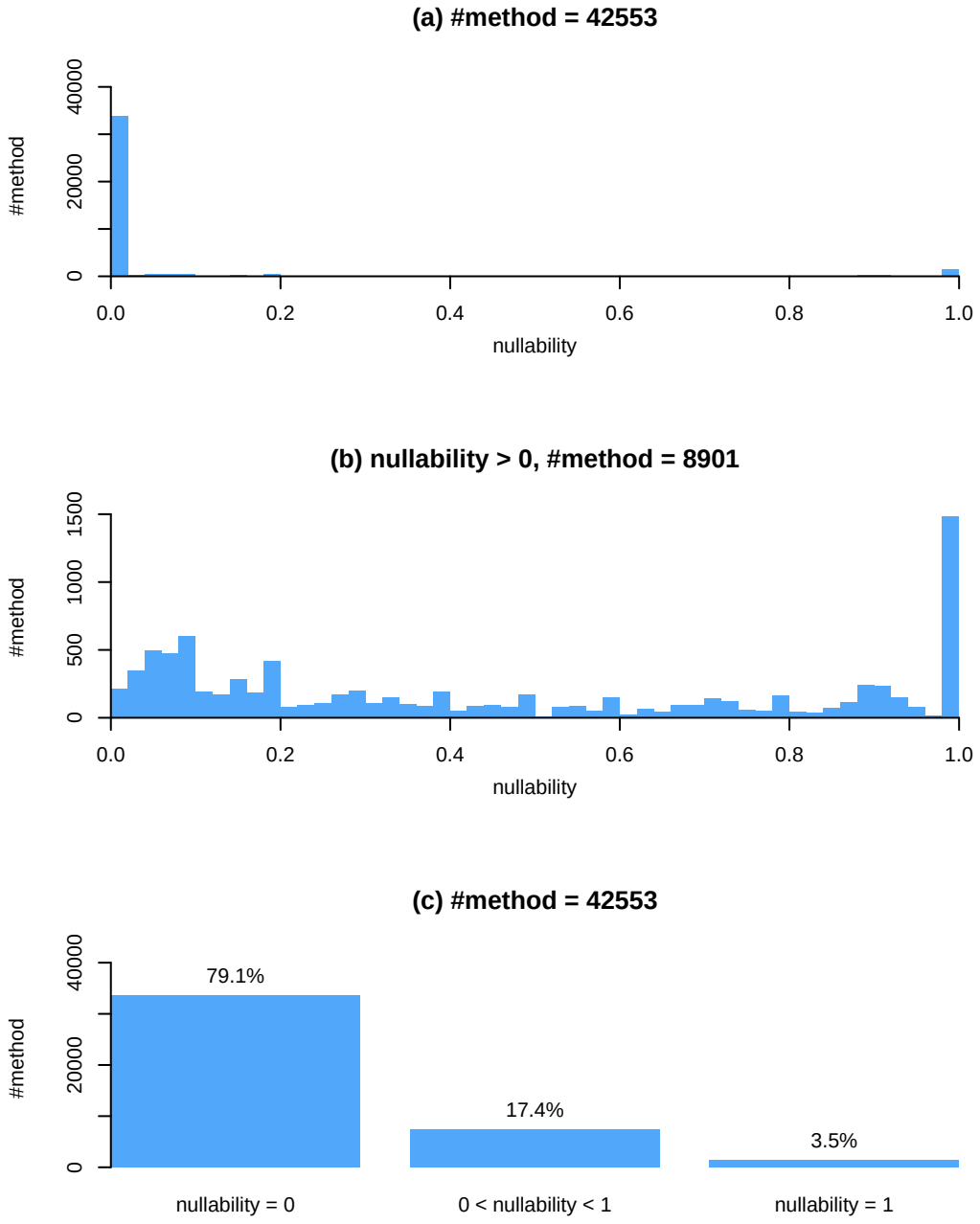

Figure 4.8: Method nullability of internally used Apache methods with at least 10 invocations.

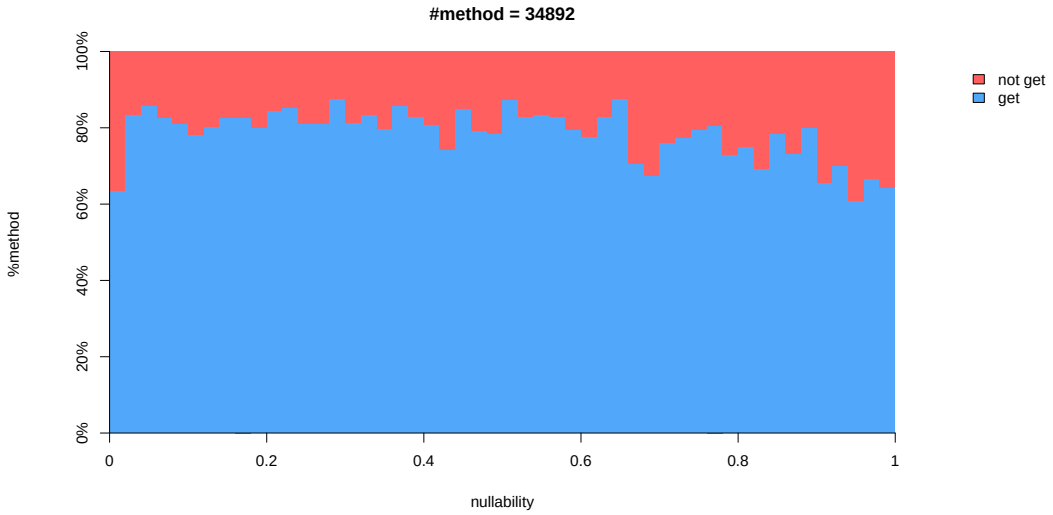

Figure 4.9: Starts of method names of Apache methods with at least 10 invocations.

explicit `return null;` statement, but many return a field value or the result of another method invocation. Checking the return values of the 29 not nullable method is superfluous, as they will never be null. Those checks could be omitted to improve code readability, performance [36] and maintainability [19].

During the manual inspections of the rarely-checked methods, we notice that most of those methods start with the verb `get`. To learn if a `get` method indicates a low nullability, we partition the methods into `get` and non-`get` methods in Figure 4.9. The distribution of `get` methods shows no correlation with a low nullability. Methods whose names start with `get` are spread quite uniformly over the whole nullability spectrum. To our surprise, we find that most methods are `get` methods. We attribute this finding to parts to the nature of the dataset: Only methods whose return value is dereferenced are included, therefore the verb `get` is descriptive, though general. We find that `getters`, *i.e.,* simple methods that return fields, cannot be identified by their verb, as we find many counter examples in the inspected samples. Yet would be interesting to check if rarely-checked methods are often getters.

4.2.5 RQ5: How is method nullability documented?

Osman *et al.* find the addition of a null check to be the most prominent bug fix pattern. In the inspected samples they find many of the added null checks to check a method return value [33]. Those bugs obviously happen because the developers are not aware of the nullability of a returned value. They are either not aware that the method can be invoked in a context for which it returns null, or they are not aware that the method can return null at all. We consider the nullability of a method to be a part of its contract. As Java has no formal contract concept, it should be at least documented, as many methods are. We compare the nullability of methods measured by the usage with the documented nullability. As a dataset for this experiment we use the usage of JRE methods, as they have an extensive and central documentation.[3] We observe a similar method nullability distribution for the JRE in Figure 4.10 as we do for the Apache APIs. For each method we search for its documentation and extract the statement about the return value for the method. Zhai *et al.* [48] generate implementations of JRE methods from their documentation and report accurate models, which supports the claim that the JRE documentation is quite precise. We detect if the

[3] `https://docs.oracle.com/javase/8/docs/api/`

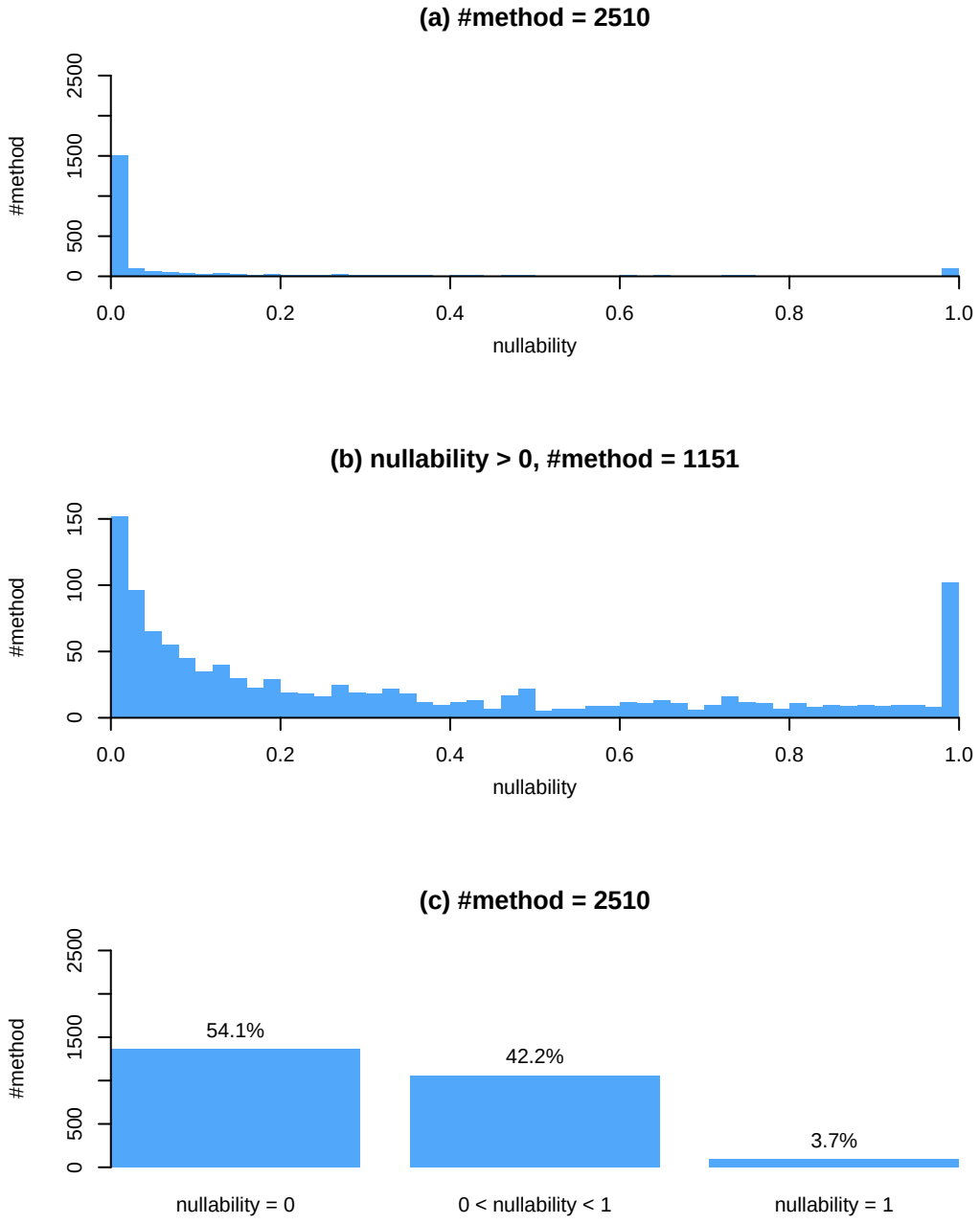

Figure 4.10: Method nullability of JRE methods with at least 10 invocations.

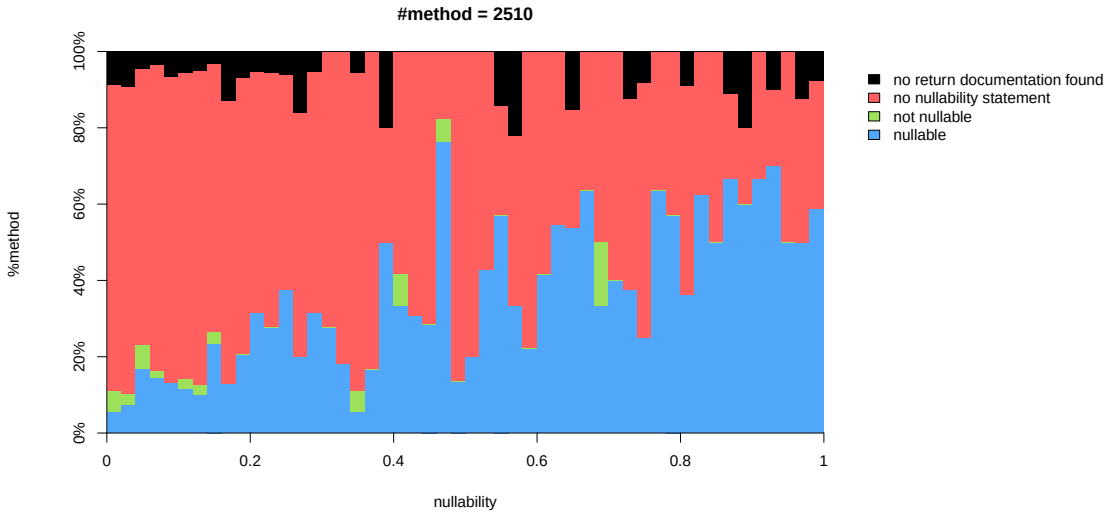

Figure 4.11: Method nullability as documented in the JavaDoc of collected JRE methods for methods with at least 10 invocations.

word `null` occurs in this statement. If `null` appears in a negation as `non-null`, `not null` or `never null`, the method is classified as not nullable. If `null` appears without a negation, the method is classified as nullable. If `null` does not appear at all, the method's nullability is unknown. Figure 4.11 shows the nullability correlation of the analysis of the documentation and the method usage. Only a few methods have no documentation about the return value. Most return documentations do not make a statement about the nullability, but if they do, it is mostly stating that the method is nullable. We notice that there exists a correlation between the measured and the documented nullability. Methods that are rarely checked for null, rarely document the nullability. Methods that are often checked for null, often document the nullability.

Although we did not apply natural language processing to determine the statement about nullability, we notice a lack of documentation about nullability. Many often-checked methods mention the nullability, but most rarely-checked methods do not. We think it would help developers if the methods that are not nullable would mention it. If all methods were documented with its nullability, we suppose the number of superfluous and missing null checks could be reduced. For this purpose, we created an IDE plugin that looks up the nullability of an invoked method in our experimental dataset, enabling the developer to make an informed decision on whether the return value should be checked for null or not. As shown in Figure 4.12, the plugin shows a tooltip with the nullability information of the hovered method invocation. In this case, we find a high nullability of 84%, indicating that it should be further inspected. Figure 4.13 displays the same tooltip for a method with a low nullability of 6%. As we have noticed in the experiment that there are many unnecessary null checks, we can be quite sure that the returned value does not need to be checked for null. In fact the documentation for this method explicitly mentions that it never returns null. If the dataset does not contain information about the invoked method, it displays the lack of information as in Figure 4.14.

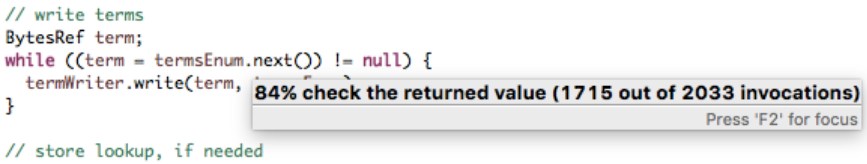

Figure 4.12: Tooltip showing that `termsEnum.next()` is checked for null in 84% of the dereferenced cases.

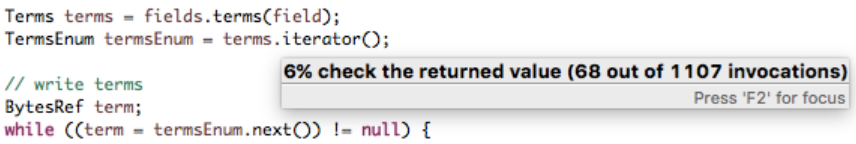

Figure 4.13: Tooltip showing that `terms.iterator()` is checked for null in 6% of the dereferenced cases.

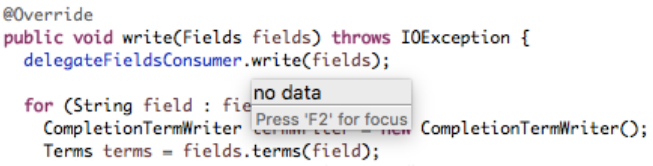

Figure 4.14: Tooltip showing that there is no data for `delegateFieldsConsumer.write(fields)`.

4.3 Summary

We automatically analyze over 45'638 artifacts clients of Apache APIs within two days. Client usage accounts for 7.7% of all detected invocations tracked to 10.3% of all invoked methods. Of 30'739'729 detected conditional expressions, 31.4% are null checks. We find 45.5% of null checks to be associated with the return value of an invoked method, identifying null returning methods as a major driver of cyclomatic complexity. In a manual inspection to verify the correctness of the analysis we find some samples of temporary live variables. The detected null checks in those cases are not installed to check if a method returned null, but to determine the state of an iterative procedure. We compute the nullability of the invoked methods and find that 65.0% of them are never checked for null, 33.5% are sometimes checked and 1.5% are always checked. There are many methods that are rarely checked, and we find many superfluous null checks among them. In the JRE API documentation, method nullability is incompletely documented, although it is an import part of a method's contract. To fill this gap, we create an IDE plugin that shows the nullability of a hovered method.

5 Threats To Validity

We identify the following threats to validity for our results:

5.1 Threats To External Validity

The generalizability of the results is dependent on the coverage of the dataset collection.

We only analyzed open source clients of Apache APIs for the JVM. The results might differ for other languages, but as many languages have nullable types as defaults, we suspect error patterns similar to null dereferences to be common. *e.g.,* Dillig *et al.* [8] analyze the Linux kernel written in C to find undetected null dereferences. In my personal experience with the Python, JavaScript, Ruby and Smalltalk languages, null dereferences were a constant companion, although Ruby and Smalltalk have language constructs to catch a null dereference errors by extending `method_missing` and `doesNotUnderstand` of the null object. Apache APIs are mature and widely used in business applications: The *commons* libraries as an extension of the JRE libraries, TOMCAT as an application server, CASSANDRA as a database and LUCENE as a full text search are all mature projects that have been refined for years and are still actively developed. Committers to Apache APIs have to be invited, this methodology governs the evolution of the API. We see a bigger threat in selecting only open source clients of Apache APIs. There are no quality standards for clients, which may influence the results. We also lack any closed source projects in our dataset, which might exhibit different usage patterns as they are developed under different circumstances with possibly different quality standards.

The dataset collection is based on the Maven infrastructure. We rely on the correctness of the dependencies declared in a POM and their availability in public Maven repositories. Projects that use another solution for dependency management cannot be discovered by our implementation, *e.g.,* OSGi,[1] Ivy[2] modules or dependencies that are distributed as files together with the depending module. If those projects used the Apache API differently, it would influence the results we collect. As Maven is widespread in the Java ecosystem, we do not consider the restriction to Maven artifacts to be critical.

The search for APIs relies on Maven Central search exclusively. Thus, all artifacts that are not indexed

[1] `https://www.osgi.org/`
[2] `http://ant.apache.org/ivy/`

in this index cannot be discovered. While Maven Central is still a popular repository, jcenter and clojars are gaining popularility and size. E.g. jcenter is the default repository in Android Studio.[3]

A similar restriction is imposed by the usage of mvnrepository to find dependents of artifacts. Although mvnrepository aggregates over many repositories, we do not know how the computation of the dependents works. It might only include transitive dependencies to a certain level, or only dependencies in a certain scope are listed. We inspect those issues and found transitive and test dependencies to be included, therefore we suppose that the coverage of dependents is quite complete. Of course only dependents that are indexed in covered repositories are discovered. Many private businesses operate private repositories that index the artifacts of the products they produce. As they are not publicly available, a lot of API clients are not covered in our dataset. Only publicly available APIs and API clients are collected.

Our evaluation is based on the premise that the majority of the analyzed API clients uses the APIs correctly, *i.e.,* most usages do not contain bugs. If this premise does not hold, we cannot make suggestions how to use the API, as there is no evidence of correct usage.

5.2 Threats To Construct Validity

The appropriateness of our measures depends on the correctness of their implementation.

In the evaluation of the experiment we detected some false positive matches in the results. The analysis extracts which method return values are checked for null and which return values are dereferenced. But it does not verify that the dereference is actually guarded by a null check to count it as checked. We manually inspect a random sample from results and found only a few instances where the null check is not guarding the dereference of the value. In the manual inspection we also found that not all null checks are guards, some are used to break the control flow in a loop or recursion. To overcome this limitation, a path-sensitive analysis is required.

Another over-approximation is caused by the lack of constraints on some paths in the control flow graph. Language constructs like an `instanceof` check imply that the successfully checked value is non-null, because null does not pass this check. Similarly, `Objects.requireNonNull(value)` or custom assertions like `Assert.notNull(value)` impose non-nullness on the remainder of the code block. Again, a path-sensitive analysis could circumvent these issues.

[3] `https://developer.android.com/studio/build/index.html`

6 Conclusions and Future Work

In this study we look at the usage of method return values as they make up the main class of null-checked values. We compute the nullability of each invoked method, *i.e.,* the ratio between null checks and dereferenced invocations of the returned values. We extract this information from Maven JAR artifacts. The JARs are collected with a API client collection framework called COLUMBO, a fast, scalable and configurable approach to find APIs and their respective clients by exploiting Maven's dependency management system. The API clients are analyzed in an intra-procedural dataflow analysis that associate dereferenced invocations and null checks. Every invoked method is tracked to its declaring artifact, thus generating a universally unique identifier for each method, allowing the aggregation of invocations of the same method over whole ecosystems instead of just projects.

We run the analysis on 45'638 Apache API clients to compute the nullability of Apache API methods. We find that 65.0% of dereferenced Apache API method are never checked for null, 33.5% are sometimes checked and 1.5% are always checked. Some analyzed artifacts are Apache artifacts themselves, as they depend on other Apache APIs. The comparison between the usage of internal Apache API usage and client usage shows significantly less rarely-checked methods in internal usage than in client usage. A manual inspection of the methods rarely checked in client usage shows that about a third of them can never return null, hence checking the return value for null is superfluous and hinders code readability. In the Apache API clients we also analyze their usage of the JRE and we find a similar nullability distribution as in Apache usage. We consider method nullability an important part of a method contract, but we find it to be incompletely documented in the JRE API documentation. Most method documentations do not make a statement about its nullability. To bridge this gap, we integrate the nullability data an IDE plugin, that shows developers the measured nullability for each method, giving them an estimation of the potential null return.

6.1 Future Work

In this section we outline possible extensions of this study.

6.1.1 Generalizability

To generalize the results we draw from the analysis of Apache API clients, the scope of the dataset should be broadened to different datasets. Wide datasets could be around Mozilla artifacts or the Eclipse and Spring frameworks. Narrow datasets could be the clients of a single popular products, *e.g.,* Elasticsearch,[1] Hibernate,[2] Guava[3] or JUnit.[4] This might reveal patterns that are ecosystem specific — or general. Some APIs may declare `Optional` return types, use `@Nullable` and `@NonNullable` annotations [6, 35] or employ documentation policies to make the issue of returned nulls easier to detect for clients. Different policies may have different effects, some might be more effective than others.

Artifacts compiled for the JVM from other languages than Java, *e.g.,* Scala, or derivations, *e.g.,* Android, could also be in the focus of the analysis to compute the nullabilities in popular frameworks or the standard libraries. Android applications are compiled to Dalvik bytecode, which is not compatible with Java bytecode. As our analysis is based on SOOT's Jimple IR, we could run the same analysis on Android applications as on Java applications, as Bertels *et al.* provide a translation from Dalvik to Jimple [3].

A comparison with some closed source ecosystems might reveal how APIs shaped by corporate guidelines compare to open source APIs concerning nullability.

Our analysis treats every usage of an API as equal. This bears the risk of basing our suggestions on wrong usages. If we included a measure for quality or trust into the computation of method nullability, the risk to make wrong suggestions could be reduced and the generalizability increased. *e.g.,* the usages in projects with Travis CI[5] builds and high test coverage could be weighted more than those without.

6.1.2 Experiments

We find insufficient documentation of method nullability, yet the applied analysis to extract nullability statements from the documentation is quite primitive. We plan on extending the nullability statement extraction, apply it on multiple projects and manually inspect the documentation and usages of some methods.

We notice that most nullable methods return null only in some contexts, depending on either the state of parameters or the method receiver. Our analysis does not account for this context, hence we were unable to detect in which situations these methods should be checked and in which they do not. An extended scope of the analysis with symbolic execution or inter-procedural call graphs could identify patterns for the null and non-null classes. Those patterns could then be matched against other projects to find missing and superfluous null checks.

The analysis could also be extended with path-sensitivity to increase its accuracy. Constraints on nullness of variables, fields and array elements can be propagated along the paths in the control flow graph, *e.g.,* an `instanceof` check ensures that the checked value is non-null, as do premature returns after a null checks like in `if (a == null) return; a.b(); // a is not null`. Path-sensitivity is essential to decide whether or not a dereference is guarded by a null check. With an analysis with a high precision and recall it could be possible to detect null dereference bugs by detecting outliers in the method usage.

As we find many unnecessary null checks, it might be feasible to do more experiments to find rules to detect them reliably. In the manual inspection of rarely-checked methods we verify if the method guarantees to return a non-null value at all return statements. This process could be automated with an intra-procedural, path-sensitive backward dataflow analysis, where nullness constraints are checked on the returned value. Maybe a cut-off point at a low nullability can be found, where no method with a lower nullability needs to be checked for null.

[1] `https://www.elastic.co/products/elasticsearch`
[2] `http://hibernate.org/`
[3] `https://github.com/google/guava`
[4] `http://junit.org/junit4/`
[5] `https://travis-ci.org/`

We could correlate method nullability with bugs to identify the method usage patterns that lead to bugs. From bug reports that are annotated with commits that insert a null check we could identify the method whose return value is checked for null. A comparison of the method nullability as extracted from its global usage and the method nullability measured in bug fixes might reveal methods that are notoriously causing bugs. Instead of basing bug fixes on reports, we might as well base it on the evolution of artifacts over their versions. As our dataset includes multiple versions of the same client, we should be able to track the evolution of method nullability without connecting our dataset to source control management systems or issue trackers. If a client adds a null check for a formerly unchecked method in a newer version, this could be a bug fix.

6.1.3 API Client Collection

COLUMBO, our implementation of API client collection, could be extended to search for clients beyond those accessible by Maven Central Search and mvnrepository. If COLUMBO kept its own index of POMs that is synced with the remote repositories, the scope of the artifact search could be broadened and the discovery of dependents would be more transparent. The analysis of dependencies between artifacts could be performed on the extended index itself, without the need to collect any artifacts from remote servers.

In the current implementation the same client artifact is collected multiple times if it is the dependent of different APIs matching the initial query. This duplication occurs often in our analysis, so we skip already analyzed clients to avoid duplication of the analysis work. The duplication detection should be implemented in COLUMBO itself, so that all collected clients are unique. This would improve performance as well.

COLUMBO could be deployed as search service, so that other API usage analyses can externalize the search for dataset candidates. The service would take the query for APIs as a request and respond with a stream of API clients. To restrict network traffic, the service would not respond with resolved JARs, but artifact descriptors instead. A client would resolve the JARs. Instead of running the analysis the client side of a COLUMBO service, it would also be possible to run it within the service infrastructure, similar to Google's BigQuery [12, 15] or the Boa [9] infrastructure. The advantage of this approach would be that service consumers do not need any infrastructure to run the analysis, yet the infrastructure needs to be provided on the provider side.

6.1.4 Application Of Results

We find about two thirds of Apache API methods to be not nullable. This knowledge could be exploited to prune inter-procedural call graphs in a null dereference verification analysis. At every invocation of a non-nullable method, we know that the returned value is non-null, hence no further analysis is required on the outgoing invocation. We can prune this branch form the call graph and replace it immediately with the non-null constraint. Pruning could reduce runtime performance of inter-procedural analysis as the processed call graph.

The IDE plugin that attributes method invocations with the nullability measured in our experiments could be improved. In its current state, detecting potential missing or superfluous null checks is not trivial, as every invocation needs to be inspected manually. Marking usages that disaccord with the overall usage with color in the editor might be a better approach. This way the potential issues would be easy to detect and the usage in agreement with the majority would not distract from the real issues.

7 Related Work

In Section 7.1 we compare the collection of our dataset with the collection used in other API usage studies. Related API usage studies are outlined in Section 7.2. Section 7.3 exhibits the state of the art to identify possible null dereferences and how our heuristics about null-checked methods could improve it.

In an earlier study we found that 35% of conditionals contain a null check and 71% of the values checked against null are method return values [32]. The analysis is intra-procedural without flow-sensitivity, it only looks at the values assigned to the checked variable at the nodes higher in the abstract syntax tree than the null check. These findings imply that null returning methods are a common class. We pick up from there and explore the class of null-returning methods by analyzing their usage. We replicate the findings of this study with an intra-procedural, flow-sensitive analysis of bytecode. In this study we find 31.4% of conditionals to be null checks and 45.5% of them to be associated with method return values. The result on null check frequency is very similar, the lower result in associated method return values can be explained by the more conservative analysis. We only include local variables in our analysis, in our predating analysis we also included fields and array elements.

7.1 Dataset Collection

We find the collection of the dataset to be critical for API usage analysis. In this section we compare our dataset and its collection with other approaches.

With COLUMBO we provide a scalable tool that collects API clients including their dependencies. The collection of our dataset is neither random [27], nor does it target usage of the language's standard libraries [21].

Sawant *et al.* build a typed dataset for five APIs and their usages in 20'263 GitHub projects [42]. They use Eclipse JDT to allow partial compilation from source which solves the same issue as phantom references in SOOT, the analysis framework we used. 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.

Saini *et al.* provide a dataset of 186'392 Maven artifact, most of them with sources, and run FINDBUGS on them [41]. They provide their dataset as repository of artifacts and a database of the FINDBUGS reports.

We do not provide a dataset, but with a COLUMBO we provide a procedure to build such a dataset. A dataset of artifacts becomes outdated and needs to be curated when new versions of artifacts are released. With COLUMBO a dataset can be recollected by running the same query again.

There are several datasets over large parts of GitHub. GHTorrent focusses on project meta-data like issues and committing users, not the source files containing the API calls we are interested in [13]. Google's BigQuery GitHub dataset can be queried for contents of source files. It even runs static analysis remotely [15], but type information is not provided [12]. Dyer *et al.* provide ASTs that include type information of sources in GitHub projects in the Boa dataset [9]. We provide more precise type information and we can track API invocations back to the artifact providing the API.

Lämmel *et al.* check out 6'286 SourceForge projects and manage to build 1'476 of them from source to analyze them for API usage [20]. Instead of building projects from source, we collect binary Maven artifacts with the dependencies declared in the POMs. SOOT, the analysis framework we use, creates phantom references for classes that cannot be found. The analysis runs with only the artifacts of the API and the API client available.

7.2 API Usage

Our data about method nullability complements documentation and determines part of the contracts of the mined methods. We classify our approach into the *behavioral specification* category in the API mining survey of Robillard *et al.* [40].

Ramanathan *et al.* mine method pre-conditions in client code [38, 39], while we mine post-conditions. Acharya *et al.* extract specifications of X11 API methods by analyzing client code and compare the extracted specifications against the documented specification [1].

Developers demand a good API documentation and usage examples [14]. We contribute method nullability to complement existing documentation, which we find to be incomplete in this aspect. Buse *et al.* motivate their approach by the lack of usage examples in the documentation. The synthesized examples are evaluated in a human study, in which the participants preferred the examples over code search [4]. Uddin *et al.* also note that the documentation of the API usage patterns they detected is incomplete [45]. Contrary to our results of incomplete documentation, Pandita *et al.* find their specifications derived from the documentation of the Facebook API for C# to comply with human written contracts for the same methods. They note that the human written contracts included more null checks than the derived contracts, yet claim that those assertions are implied by the derived contracts [34].

As an alternative to documentation, there are approaches that explore APIs to find coupled types, methods or protocols. Michail searches for reuse in inheritance trees, so that commonly overwritten methods can be discovered [28, 29]. De Roover *et al.* explore APIs on multiple dimension, *i.e.,* scopes of projects and APIs, metrics and metadata of APIs presented in project- or API-centric views [5]. Mandelin *et al.* provide an IDE tool that finds snippets illustrating how to get an object of a desired type, given some input types [25]. Nguyen *et al.* mine usage patterns of one or multiple objects to identify protocols that can then be checked against user code to find anomalies [31]. They report some patterns of missing null checks. Pradel *et al.* check mined usage patterns to find bugs and report a true positive rate of 51% [37]. Thummalapenta *et al.* detect hot and cold spots in APIs based on their usage to identify possible starting points for developers learning the API [43]. Zie, Zhong *et al.* search for representative API usage patterns to assist developers using the API [47, 50]. Zhang *et al.* provide a tool that suggests parameters to use in method invocations [49].

7.3 Null Dereference Analysis

Finding possible null dereferences is usually done by checking whether a dereference is null safe or not based on a dataflow analysis. Null dereferences are mostly fixed by wrapping the dereference in a null check [33]. Instead of trying to find a path that results in a null dereference, we associate those null checks with the API methods, whose returned value they check. We infer the nullability of a method by its usage.

Papi, Dietl *et al.* propose to avoid null dereferences by extending the type system with `@Nullable` and `@NonNullable` annotations [6, 35]. Method, field, variable and collection item declarations can be annotated, so that a static checker can verify that no nullable value is ever dereferenced. It requires manual work though to add those annotations to existing projects, as the checker cannot infer the nullability for all values. Not all dependencies may use these annotations, which makes their usage cumbersome, as any API invocations need to be annotated in the client project.

Hovemeyer *et al.* use intra-procedural dataflow analysis to find null dereferences in Java code and continuously improve their FINDBUGS tool [16–18]. Their approach is path-sensitive, considers null checks, and aims at a fast and simple analysis with a low false positive rate. With this analysis 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 known as notoriously returning null. By looking at the null checks of method return values we can identify the null returning methods.

Tomb *et al.* build an inter-procedural analysis to detect runtime exceptions [44]. It uses symbolic execution and a constraint solver to provoke an exception and only reports the locations where an exception could be provoked. They note that the inter-procedural analysis only increases the precision slightly compared to intra-procedural analysis. Our findings contradict this statement, as we find a large share of null checks checking values returned from method invocations, which is impossible to detect for an intra-procedural analysis. Additionally, my personal experience developing business applications with Java disagrees with the findings of Tomb *et al.*, as I remember many debug sessions of NullPointerExceptions, where the dereferenced value was returned from a method.

Loginov *et al.* target a *sound* inter-procedural analysis, verifying dereferences to be null-safe [23]. Their implementation, called SALSA, can verify 90% of all dereferences. This leaves 10% of dereferences that cannot be proven to be safe. Madhavan and Komondoor also provide a *sound* analysis that is applied on a single dereference instead of a whole program, reporting 16% of dereferences that cannot be proven to be safe [24]. We consider this rate of warnings to be high for manual inspection, but it is certainly useful as a verification technique. We do not strive for a complex sound inter-procedural analysis, instead we provide a heuristic to increase the precision of a simple intra-procedural analysis.

Nanda and Sinha present XYLEM, an inter-procedural, path- and context-sensitive dataflow analysis [30]. They want the analysis to output "useful" defect reports that do not need to be sound. Contrary to Tomb et. al. [44] they report a 16 times higher number of possible null dereferences in inter-procedural than in intra-procedural analysis [16], yet they claim a low false positive rate of 4 out of 86 warnings. These results are challenged by Ayewah and Pugh [2]. They examine the warnings generated by XYLEM and accounted only 60% of them to be plausible. They argue that if a checker warns for the same locations in many versions of stable software, then the potential null dereference might never have manifested in execution and is not causing problems. Hence it was not only about finding all mistakes, but finding the important ones. We propose heuristics about method invocations to reduce the number of warnings and giving an estimate on the severity of the warning.

Acknowledgements

I would like to thank Haidar Osman for guiding me through this project, infecting me with his good mood and humour, for the discussions we had about the interpretation of the results and all other unrelated discussions. I thank Oscar Nierstrasz for his patience, trust and encouragement when I needed it. Thanks to all my friends and family that showed interest in what I am doing, asked me questions about it and helped me better understand how to formulate the story I wanted to tell. Also, thank you for all the distractions from the project. A special thanks also goes to Fiva, whose music accompanied and motivated me for most of the writing process.

Bibliography

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 65–72. ACM, 2010.

[3] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[4] Raymond PL Buse and Westley Weimer. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, pages 782–792. IEEE Press, 2012.

[5] Coen De Roover, Ralf Lammel, and Ekaterina Pek. Multi-dimensional exploration of API usage. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 152–161. IEEE, 2013.

[6] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.

[7] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 435–445, New York, NY, USA, 2007. ACM.

[8] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[9] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE'13, pages 422–431, 2013.

[10] Arni Einarsson and Janus Dam Nielsen. BigQuery GitHub data. `http://www.brics.dk/SootGuide/`. Accessed: 2017-02-06.

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[12] Google. BigQuery GitHub data. `https://cloud.google.com/bigquery/public-data/github`. Accessed: 2016-11-21.

[13] Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE, 2012.

[14] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, page 12. ACM, 2014.

[15] Felipe Hoffa. Static JavaScript code analysis inside a SQL query: JSHint+GitHub+BigQuery. `https://medium.com/google-cloud/static-javascript-code-analysis-within-bigquery-ed0e3011732c#.w2e81q2pc`. Accessed: 2016-11-21.

[16] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[17] David Hovemeyer and William 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*, pages 9–14. ACM, 2007.

[18] David Hovemeyer, Jaime Spacco, and William 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*, PASTE '05, pages 13–19, New York, NY, USA, 2005. ACM.

[19] Shunji Kimura, Kazuhiro Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Does return null matter? In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 244–253. IEEE, 2014.

[20] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[21] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[22] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[23] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 213–224. ACM, 2008.

[24] Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. *ACM Sigplan Notices*, 46(10):1033–1052, 2011.

[25] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *ACM SIGPLAN Notices*, volume 40, pages 48–61. ACM, 2005.

[26] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 4:308–320, 1976.

[27] D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in API usage of object-oriented software. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 43–52, Sept 2013.

[28] Amir Michail. Data mining library reuse patterns in user-selected applications. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 24–33. IEEE, 1999.

[29] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[30] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering*, pages 133–143. IEEE Computer Society, 2009.

[31] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392. ACM, 2009.

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

[33] Haidar Osman, Mircea Lungu, and Oscar Nierstrasz. Mining frequent bug-fix code changes. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 343–347. IEEE, 2014.

[34] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 815–825. IEEE, 2012.

[35] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[36] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.

[37] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 925–935. IEEE Press, 2012.

[38] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th international conference on Software Engineering*, pages 240–250. IEEE Computer Society, 2007.

[39] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, volume 42, pages 123–134. ACM, 2007.

[40] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[41] Vaibhav Saini, Hitesh Sajnani, Joel Ossher, and Cristina V Lopes. A dataset for maven artifacts and bug patterns found in them. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419. ACM, 2014.

[42] Anand Ashok Sawant and Alberto Bacchelli. A dataset for API usage. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 506–509. IEEE Press, 2015.

[43] Suresh Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336. IEEE Computer Society, 2008.

[44] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 97–107. ACM, 2007.

[45] Gias Uddin, Barthélémy Dagenais, and Martin P Robillard. Temporal analysis of API usage concepts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 804–814. IEEE Press, 2012.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[47] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[48] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 380–391, New York, NY, USA, 2016. ACM.

[49] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836. IEEE Press, 2012.

[50] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.