REPFINDER: Finding Replacements for Missing APIs in Library Update

Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China

***Abstract*—Libraries are widely adopted in developing software projects. Library APIs are often missing during library evolution as library developers may deprecate, remove or refactor APIs. As a result, client developers have to manually find replacement APIs for missing APIs when updating library versions in their projects, which is a difficult and expensive software maintenance task. One of the key limitations of the existing automated approaches is that they usually consider the library itself as the single source to find replacement APIs, which heavily limits their accuracy.**

In this paper, we first present an empirical study to understand characteristics about missing APIs and their replacements. Specifically, we quantify the prevalence of missing APIs, and summarize the knowledge sources where the replacements are found, and the code change and mapping cardinality between missing APIs and their replacements. Then, inspired by the insights from our study, we propose a heuristic-based approach, REPFINDER, to automatically find replacements for missing APIs in library update. We design and combine a set of heuristics to hierarchically search three sources (deprecation message, own library, and external library) for finding replacements. Our evaluation has demonstrated that REPFINDER can find replacement APIs effectively and efficiently, and significantly outperform the state-of-the-art approaches.

I. INTRODUCTION

Libraries are widely adopted in developing software projects [86]. Developers need to update library versions in their projects under various circumstances, e.g., using new functionalities and bug fixes [21], solving dependency conflicts [87, 88], mitigating vulnerability risks [11, 20, 62, 63, 86, 101], and harmonizing inconsistent library versions [34]. As libraries evolve (i.e., new library versions are released), library developers might introduce missing APIs (i.e., APIs that are missing in the new library versions) by deprecating, removing or refactoring APIs. As a consequence, client projects fail to compile after library update, while client developers have to manually investigate how to replace usages of missing APIs. Due to this difficult and expensive manual analysis, client developers may choose not to update library versions and thus cause technical lags [16, 19, 21, 39, 44, 69, 97].

To achieve automated library update, various API adaptation approaches have been introduced [15, 91]. In general, API adaptation can be decomposed into two tasks; i.e., the first is to find what is the replacement of a missing API (e.g., [13, 18, 89, 95]), and the second is to identify how usages of a missing API are actually replaced by usages of its replacement API (e.g., [24, 81]). The first task, which is the focus of this paper, has been achieved by four groups of approaches. Manual approaches [13, 29, 60] require library developers to specify the mapping between missing APIs and their replacements or to record API refactoring actions in IDE. However, the involvement of library developers is often not available. Similarity-based approaches [26, 37, 95] utilize textual, metric and structural similarities to build the mapping. Usage-based approaches [17, 18, 74] identify the mapping from API usage changes in library's instantiation code or own code. However, they become infeasible when missing APIs are not used in library's instantiation code or own code. Hybrid approaches [50, 89] combine and extend similarity-based and usage-based approaches, and hence also share their limitation. All approaches share the same limitation that they consider the library itself as the single source to find replacement APIs, and only Wu et al. [89] further consider libraries of the same vendor. Therefore, the accuracy of these approaches are limited.

To address the limitations, we first conduct an empirical study to characterize missing APIs and their replacement APIs. Here, we regard public methods in public classes as APIs. Specifically, we quantify the prevalence of missing APIs in 85 major version updates, 771 minor version updates and 1,048 patch version updates from 200 widely-used libraries. On average, 720, 330 and 244 APIs are missing in major, minor and patch version updates. Moreover, we manually find the replacement APIs for 738 missing APIs in 99 version updates, and summarize the knowledge sources where the replacements are found, and the code change and mapping cardinality between missing APIs and their replacements. Deprecation message, own library, and external library are the three sources to find replacements, respectively accounting for 13.6%, 50.8% and 14.6% of the missing APIs. Code change from missing APIs to their replacements can be categorized into refactoring (64.4%), substitution (4.2%) and deletion (13.6%). One-to-one mapping (91.1%) is the commonest cardinality between missing APIs and their replacements.

Then, inspired by our study, we propose a heuristic-based approach, named REPFINDER, to automatically find replacement APIs for missing APIs in library update. REPFINDER takes as inputs the old library version before update, the new library version after update, and an API that resides in the old library version but is missing in the new library version, and returns the replacement APIs. REPFINDER searches the three sources, i.e., deprecation message, own library and external library, to find replacement APIs. Specifically, we design a set of heuristics for each source based on our empirical study, and hierarchically combine them to find replacement APIs.

To evaluate the effectiveness of REPFINDER, we compared REPFINDER with two state-of-the-art approaches, i.e., REFDIFF [75] and AURA [89], on 683 missing APIs. REPFINDER significantly outperformed REFDIFF and AURA in recall by up to 213.6% while having a slight decrease in precision by

5.4%. Further, we evaluated the efficiency of REPFINDER. REPFINDER took one seconds on average to find replacements for a missing API. Moreover, we applied REPFINDER to library update on 32 projects. REPFINDER successfully found replacements for all missing APIs for 65.6% projects.

In summary, this work makes the following contributions.

- We conduct an empirical study to characterize missing APIs and their replacement APIs in library update.
- We propose a heuristic-based approach, named REPFINDER, to find replacements for missing APIs in library update.
- We conduct extensive experiments to demonstrate the effectiveness and efficiency of REPFINDER.

II. AN EMPIRICAL STUDY

In this section, we present an empirical study to characterize missing APIs and their replacements to inspire approach design.

A. Study Design

We designed our study to answer the following four research questions about missing APIs and their replacements.

- **RQ1 Prevalence Analysis:** How prevalent are missing APIs in library evolution? (Sec. II-C)
- **RQ2 Source Analysis:** What are the knowledge sources for finding replacements for missing APIs? (Sec. II-D)
- **RQ3 Change Analysis:** What is the code change between missing APIs and their replacements? (Sec. II-E)
- **RQ4 Cardinality Analysis:** What is the mapping cardinality between missing APIs and their replacements? (Sec. II-F)

Before elaborating our RQ design, we define some library terms to avoid confusion. Library update refers to updating an old library version to a new library version. Hereafter, old library version refers to the library version before library update, and new library version refers to the library version after library update.

We designed **RQ1** to quantify the prevalence of missing APIs in library evolution. To this end, for each of the collected library version updates (Sec.II-B), we analyzed the number of public classes and the number of public methods in public classes that are defined in the old library version but are missing in the new library version through code differencing [33]. Defined by semantic versioning [64], version numbers must take the form of *X.Y.Z*, where *X*, *Y* and *Z* denotes the major, minor and patch version. Bug fixes not affecting APIs increment *Z*, backwards compatible API changes or additions increment *Y*, and backwards incompatible API changes increment *X*. Generally, client developers need no integration effort if updating to a patch or minor version, but need some integration effort if updating to a major version. Therefore, we conducted our prevalence analysis via distinguishing major version updates (the old and new library version have different major version), minor version updates (the old and new library version have the same major version but different minor version), and patch version updates (the old and new library version have the same major and minor version but different patch version). Our results from **RQ1** aim to motivate the need of an automated approach to find replacements for missing APIs.

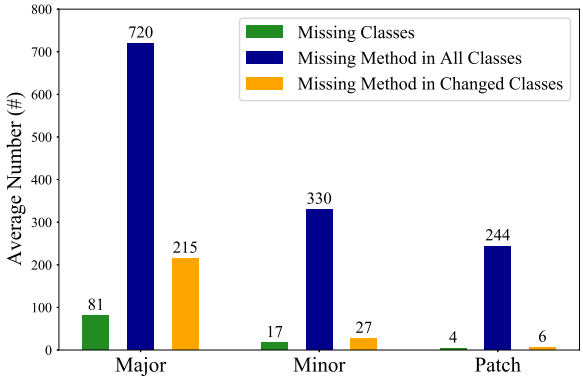


Fig. 1. Prevalence of Missing APIs in Library Evolution

We designed **RQ2**, **RQ3** and **RQ4** to characterize the knowledge sources for finding replacements for missing APIs, and the code change and mapping cardinality between missing APIs and their replacements. To this end, for each of the collected missing APIs in the collected library version updates (Sec. II-B), two of the authors separately found its replacements by analyzing API usages in client projects, looking at library documentations, and searching internet resources, and recorded its knowledge source, code change and mapping cardinality. Then, they discussed and investigated inconsistent cases together to reach consensus, and categorized knowledge sources, code change and mapping cardinality. Our results from **RQ2**, **RQ3** and **RQ4** aim to capture the characteristics of missing APIs and their replacements to inspire the design of an automated approach.

B. Data Collection

To prepare library version updates for our RQs, we decided to choose libraries that were truly used by client projects so that i) the characteristics about missing APIs and their replacements would be more realistic and more representative and ii) our manual analysis would become more accurate as we could use client projects to help to confirm whether the replacements we found were correct. Hence, we selected the GitHub Java projects that were created after 2013, used Maven as the build tool, and had more than 20 stars. These criteria were adopted to ensure project quality and ease the extraction of libraries, which restricted our selection to 2,567 projects. From these projects, we extracted a total of 11,419 used libraries and 31,393 used library versions, and collected library API calls using JavaParser [76].

For **RQ1**, we chose 200 libraries that were most widely used according to our collected library API calls. 3,030 library versions from these libraries were used, from which we generated a version update for any two adjacent major, minor and patch versions. If multiple library versions shared the same major, minor or patch version, we randomly selected one of them. Finally, we prepared 85 major version updates, 771 minor version updates, and 1,048 patch version updates.

For **RQ2**, **RQ3** and **RQ4**, we selected libraries whose library APIs were called by more than ten times across projects to increase the possibility of finding missing APIs called by projects, which resulted in 999 libraries. Of these libraries, 230 libraries had at least two library versions used in projects. 841 library

TABLE I. Examples of Missing APIs and Their Replacements in Library Update

No.	Library	Version Update	Missing API	Replacement API	Source
1	org.mapdb mapdb	0.9.3 0.9.13	org.mapdb. DBMaker.writeAheadLogDisable()	org.mapdb. DBMaker.transactionDisable()	Deprecation Message
2	org.apache.lucene lucene-core	5.1.0 6.0.0	org.apache.lucene.search. PhraseQuery.add(Term)	org.apache.lucene.search. PhraseQuery.Builder.add(Term)	Own Library
3	org.apache.lucene lucene-core	3.0.3 4.0.0	org.apache.lucene.analysis.standard. StandardAnalyzer.StandardAnalyzer(Version)	org.apache.lucene.analysis.standard. StandardAnalyzer.StandardAnalyzer(Version)	Vendor Library
4	org.elasticsearch elasticsearch	1.7.1 2.0.0	org.elasticsearch.common.joda.time.format. DateTimeFormatter.print(long)	org.joda.time.format. DateTimeFormatter.print(long)	Dependency Library
5	org.elasticsearch elasticsearch	0.20.6 1.2.1	org.elasticsearch.common.trove.list.array. TIntArrayList.toArray()	gnu.trove.list.array. TIntArrayList.toArray()	Similar Library
6	org.jsoup jsoup	1.7.2 1.13.1	org.jsoup.select. Elements.contains(Object)	java.util. ArrayList.contains(Object)	JDK Library
7	org.apache.solr solr-solrj	4.10.4 6.6.2	org.apache.solr.client.solrj.util. ClientUtils.toSolrInputDocument(SolrDocument)	NA	Deprecation Message

versions from these 230 libraries were used in projects, from which we generated version updates in three ways. First, we selected the smallest and the largest version as a version update to simulate the largest version gap in library update. Second, we selected any two adjacent versions as a version update to simulate timely library update. Third, we selected any two adjacent major versions as a version update to simulate major library version update where missing APIs are very common. If multiple library versions had the same major version, we selected the version with the highest API usage. We generated 183 major version updates, 320 minor version updates, and 149 patch version updates. Of these version updates, 77 major version updates, 19 minor version updates and 3 patch version updates had missing APIs called across projects, resulting in 738 missing APIs that were from 37 libraries and truly called across projects.

C. Prevalence Analysis (RQ1)

Fig. 1 presents the average number of missing public classes and missing public methods in major, minor and patch version updates. We can observe that major version updates introduce the most missing public classes and missing public methods, which is consistent to semantic versioning; and minor version updates and patch version updates also introduce some missing public classes and missing public methods, which actually violates semantic versioning. Averagely, 81 public classes and 215 public methods are missing in a major version update. If we include the public methods in missing public classes, a total of 720 public methods are missing. In a minor and patch version update, 17 and 4 public classes and 27 and 6 public methods are missing. These results indicate that missing APIs are prevalent and severe in major version updates, and also occur in minor and patch version updates.

D. Source Analysis (RQ2)

We summarize three sources, i.e., deprecation message, own library, and external library, where the replacement APIs can be found. First, for 100 (13.6%) of the 738 missing APIs, the deprecation message in JavaDoc gives the hint about the replacement APIs. Specifically, 94.0% of the deprecation messages include a link tag for developer to navigate to the replacement API, while others (6.0%) do not include a link but list the replacement in text. However, the deprecation message is not always in the JavaDoc of the old library version, but can be in the JavaDoc of a library version that is released before the new library version. This result indicates that API changes might not be always documented in JavaDoc.

Example 2.1: The second row of Table I reports an example that the API *org.mapdb.DBMaker.writeAheadLogDisable()* in the old version 0.9.3 of the library *mapdb* is missing in the new version 0.9.13. There is no deprecation message in the JavaDoc of the old version 0.9.3. It turns out that the deprecation message is in the JavaDoc of the version 0.9.4, and says that "use *transactionDisable()* instead" with a link to the replacement.

Second, for 375 (50.8%) of the 738 missing APIs, their replacement APIs can be found in their own library, i.e., the new library version. As refactoring is a common practice in library evolution, some APIs are refactored to become missing in the new library version (see Sec. II-E for a detailed discussion). Notice that for the missing APIs whose replacements are found by deprecation message, their replacements are actually also in the new library version, but here we do not include them.

Example 2.2: For the second example in Table I, no deprecation message can be found in the versions of the library *lucene-core* for the missing API *org.apache.lucene.search.PhraseQuery.add(Term)* in the version 6.0.0. After investigating the source code of the version 6.0.0, we find that the missing API is moved from the class *PhraseQuery* to its inner class *Builder*.

Third, for 108 (14.6%) of the 738 missing APIs, their replacements can be found in related external libraries. Specifically, the replacements are found in the library with the same vendor (i.e., vendor library) for 45 missing APIs, the library that is declared as a direct dependency (i.e., dependency library) for 29 missing APIs, the library that provides a similar API to the missing API (i.e., similar library) for 18 missing APIs, and the JDK library for 16 missing APIs.

Example 2.3: In the third example in Table I, the API locates in the version 3.0.3 of the library *lucene-core*, but is missing in the version 4.0.0. It turns out that the lucene project evolves into a multiple module project, and it reorganizes its modules after the version 3.0.3. As a result, the API, originally a part of the module corresponding to the library *lucene-core*, is moved into another module corresponding to the library *lucene-analyzer-common* that shares the same vendor.

Example 2.4: In the fourth example in Table I, the library

TABLE II. Code Change between Missing APIs and their Replacements

Code Change	Change Level	Change Action	Deprecation Message	Own Library	External Library	None
Refactoring	Class	Move Class	-	71	56	-
		Rename Class	15	62	-	-
	Method	Pull Up Method	46	62	39	-
		Push Down Method	-	5	-	-
		Change Method Signature	8	89	3	-
		Move Method	3	16	-	-
Substitution	Class	Substitute by Method with Different Name from Another Class	18	-	-	-
		Substitute by Method with Same Name from Another Class	8	-	-	-
	Method	Substitute by Method with Different Name from Own Class	-	5	-	-
Deletion	Class	Delete Class	-	-	-	33
	Method	Delete Method	-	-	-	67
Composition	NA	NA	2	65	10	-

joda-time is used by the library *elasticsearch* by copy-and-paste in the version 1.7.1; i.e., the source code of *joda-time* is directly included in the packages of *elasticsearch*. However, in the version 2.0.0, the source code of *joda-time* has already been removed from *elasticsearch*, which introduces the missing API. *elasticsearch* declares *joda-time* as a direct dependency as it itself also uses the APIs in *joda-time*.

Example 2.5: The fifth example in Table I is similar to Example 2.4. The library *trove* is used by the library *elasticsearch* by copy-and-paste in the version 0.20.6. However, in the version 1.2.1, the source code of *trove* has already been removed from *elasticsearch*, which introduces the missing API. The difference from Example 2.4 is that *elasticsearch* does not declare *trove* as a direct dependency after removing the source code of *trove*.

Example 2.6: The sixth example in Table I shows that the API *jsoup.select.Elements.contains(Object)* in the old version 1.7.2 of library *jsoup* is missing in the new version 1.13.1. It turns out that the class *Elements* in the old version 1.7.2 inherits the class *ArrayList* in the JDK library, and overrides the method *contains(Object)*. However, in the new version 1.7.2, the method *contains(Object)* is removed but the inheritance still exists, and thus the replacement API is the method *contains(Object)* in the class *ArrayList* in the JDK library.

Finally, of the remaining 155 missing APIs, 100 (13.6%) missing APIs have no replacement; i.e., these missing APIs are simply removed. Here we take a conservative approach to conclude that a missing API has no replacement. We first use the previous three sources to check whether we can find a replacement. If not, we then look for clear evidences of no replacement; e.g., deprecation message and release note that clearly say the removal of a functionality, and the functionality of the missing API is inlined in other methods. Still, we fail to find replacement for 55 missing APIs but cannot conclude no replacement.

Example 2.7: The last example in Table I shows that the API *org.apache.solr.client.solrj.util.ClientUtils.toSolrInputDocument(SolrDocument)* in the old version 4.10.4 of the library *solr-solrj* is missing in the new version 6.6.2. The deprecation message clearly says that "this method will be removed in Solr 6.0", and we make the conclusion of no replacement.

E. Change Analysis (RQ3)

We categorize three basic code changes from missing APIs to their replacements, i.e., refactoring, substitution and deletion, and a composition of these basic code changes. Table II reports a detailed categorization and its correlation to the sources discussed in Sec. II-D. Here, we denote the old library version as l_o, the missing API as m_o, the residing class of m_o in l_o as c_o, the replacement API as m_n, the residing class of m_n as c_n, and the residing library of m_n as l_n.

The first category is refactoring, covering 475 (64.4%) of the missing APIs. If m_n does not reside in l_o, we consider the change from m_o to m_n as refactoring. Specifically, as shown in the second column, refactoring can be conducted at two different levels. At the class level, c_o can be moved into a different package in l_n, while m_n is the same to m_o except for the residing package name. c_o can be renamed in l_n, while m_n is the same to m_o except for the residing class name. At the method level, m_o can be pushed up or pushed down along the inheritance tree. The signature of m_o can be changed in l_n, including its return type, method name and parameter type. m_o can also be move into a different class in l_n.

The second category is substitution, covering 31 (4.2%) of the missing APIs. Substitution means that there exists another API that can do the similar job to the missing API. Both APIs can co-exist and later the missing API is deprecated and finally removed. Thus, if m_n also resides in l_o, we consider the change from m_o to m_n as substitution. In particular, as shown in the second column, substitution can be conducted at two different levels. At the class level, m_o can be substituted by a method from another class, either with the same method name or a different method name. At the method level, m_o can be substituted by a different method from its own class.

The third category is deletion, covering 100 (13.6%) of the missing APIs, which means there is no need in the functionalities the API provides and the library developer simply removes it in the new library version. Thus, only the 100 missing APIs we conclude no replacement (i.e., the last column in Table II) belong to this category. As shown in the second column, deletion can be conducted at two different levels. At the class level, c_o can be deleted and thus m_o in c_o is also deleted. At the method level, m_o can be deleted while c_o still exists.

Apart from these basic code changes, there are more complex code changes, which are a composition of the three basic code changes. For example, the signature of m_o can be changed in l_n, and then the method is pushed up into its super class; or

TABLE III. Examples of Missing APIs and Their Replacements with Various Cardinality

No.	Library	Version Update	Missing API	Replacement APIs
1	redis.clients jedis	2.2.1 2.5.1	redis.clients.jedis. JedisPoolConfig.setMaxActive(int)	org.apache.commons.pool2.impl.GenericObjectPoolConfig.setMaxTotal(int) org.apache.commons.pool2.impl.GenericObjectPoolConfig.setMaxIdle(int)
2	org.apache.lucene lucene-core	3.0.3 4.0.0	org.apache.lucene.document. NumericField.setIntValue(int)	org.apache.lucene.document.IntField.setIntValue(int) org.apache.lucene.document.LongField.setIntValue(int) org.apache.lucene.document.FloatField.setIntValue(int) org.apache.lucene.document.DoubleField.setIntValue(int)
3	org.apache.lucene lucene-core	2.9.3 3.5.0	org.apache.lucene.search. Hits.iterator()	org.apache.lucene.search.IndexSearcher.search(Query, int) org.apache.lucene.search.ScoreDoc.doc org.apache.lucene.search.IndexSearcher.doc(int)

c_o is moved into a different package in l_n, while the signature of m_o is changed. This category accounts for a total of 77 (10.4%) missing APIs.

F. Cardinality Analysis (RQ4)

From the 583 missing APIs that we find replacements, we categorize five types of mapping cardinality, i.e., one-to-one, one-to-many, one-to-some, many-to-one and many-to-many. We observe that most of the missing APIs (531, 91.1%) have a one-to-one mapping to their replacements. The first six examples in Table I belong to this category. Besides, 26, 6, 4 and 16 missing APIs have a one-to-many, one-to-some, many-to-one and many-to-many mapping to their replacements. Here, one-to-some mapping means the missing API has more than one replacement, and which one to use depends on the API usage context.

Example 2.8: The first example in Table I shows an example of one-to-many mapping. The missing API sets the field *maxActive*. Its two replacements respectively set the field *maxTotal* and *maxIdle*. As *maxTotal - maxIdle* equals *maxActive*, the missing API can be replaced by the two replacements together.

Example 2.9: The second example in Table I shows an example of one-to-some mapping. The missing API belongs to the class *NumericField*, which is later replaced by specific classes *IntField*, *LongField*, *FloatField* and *DoubleField*. Depending on the type of numeric field used, the missing API can be replaced by the API from one of the four classes.

Example 2.10: The third example in Table I shows an example of many-to-many mapping. The missing API is the method *iterator()* from the class *Hits*, which is often used in the scenario to iterate over all hits for a search query and get the document for each hit. However, in the version 3.5.0, *lucene-core* provides another set of APIs to realize this scenario. Therefore, these APIs, often used together, should be replaced together.

G. Insights

From our study results, we have several insights. **I1:** tools are needed to help developers find replacement APIs for missing APIs in library update, as missing APIs are prevalent, especially in major version updates. **I2:** multiple sources should be leveraged together to find replacements for missing APIs, as a single source alone usually fails to find replacements for all missing APIs. **I3:** replacement APIs can be found by searching deprecation message in JavaDoc or searching similar methods in the own library or some external libraries, as missing APIs might have deprecation message to indicate their replacements, and are often refactored or substituted into a similar method in the own library or some external libraries. **I5**: it is still useful to only find one-to-one mapping for missing APIs, as most missing APIs have a one-to-one mapping to their replacements.

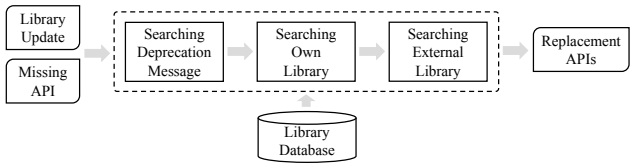


Fig. 2. Approach Overview of REPFINDER

III. OUR APPROACH

Based on the insights from our study, we propose a heuristic-based approach, named REPFINDER, to automatically find replacements for missing APIs in library update. As shown in Fig. 2, it takes as inputs a library update (i.e., an old library version l_o before the update and a new library version l_n after the update) and a missing API m_o that is resides in l_o but is missing in l_n, and returns a set of replacement APIs $\mathcal{M}$. REPFINDER has three steps, i.e., parsing deprecation message, searching own library and searching external library, to find $\mathcal{M}$. Each step leverages one of the three sources to heuristically find $\mathcal{M}$. REPFINDER has a library database, where the jar file, JavaDoc file and POM file for all library releases and JDK libraries are stored. We have a pipeline to regularly crawl them from Maven.

To ease the approach presentation, we formally define two terms. A library version l is denoted as a three-tuple $\langle group, artifact, version\rangle$, where $group$ and $artifact$ denote the vendor and name of the library, and $version$ denotes the version number of the library. An API, which is considered as a public method in a public class in this work, is denoted as a six-tuple $\langle lib, pkg, cls, ret, name, param\rangle$, where lib, pkg and cls denote its residing library version, package and class, ret denotes its return type, $name$ denotes its method name, and $param$ denotes a list of its parameter types.

A. Searching Deprecation Message

Given l_o, l_n and m_o, the first step of REPFINDER is to find replacement APIs by searching deprecation message. As deprecation often follows a *deprecate-replace-remove* cycle, m_o might be deprecated in a library version before l_n. Thus, REPFINDER first retrieves a sorted list of the JavaDoc files for all the library versions from l_o to l_n, denoted as $\mathcal{D}$, from our library database. As revealed by our empirical study, a deprecation message may use a hyper link or a text to indicate the replacements. Besides, there is no deprecation message for m_o if $m_o.cls$ is deprecated

P1. Use [the] ***text***
P2. Use [the] <code>***text***</code>
P3. Use [the] ***text*** [instead]
P4. Replaced [by][with] ***text***
P5. Replaced [by][with] ***text***
P6. In favor of ***text***
P7. In favor of ***text***
P8. Rename to ***text***
P9. Rename to ***text***
P10. Call ***text***
P11. Move to [the] ***text***
P12. [may][will] be removed

Fig. 3. Matching Patterns for Deprecation Message

as a whole. Thus, REPFINDER then iterates over $\mathcal{D}$ in a reverse order to search deprecation messages with following heuristics.

- *Method Deprecation with Link.* REPFINDER analyzes the deprecation message by matching patterns in Fig. 3 summarized through our empirical study. If a pattern is matched, it extracts the hyper link of the replacement APIs and resolves their pkg, cls, ret, $name$ and $param$ from the documentation.
- *Method Deprecation with Text.* REPFINDER extracts the textual information (e.g., cls, $name$ and $param$) about the replacement APIs by matching patterns in Fig. 3. If matched, it uses the textual information to validate the existence of the replacement APIs in JavaDoc. If validated, it resolves their pkg, cls, ret, $name$ and $param$ from their documentation.
- *Class Deprecation with Link.* REPFINDER extracts the hyper link of the replacement class of $m_o.cls$ by matching patterns in Fig. 3. If matched, it identifies the replacement API whose method name equals to $m_o.name$ and whose parameter types equal to $m_o.param$. If identified, it resolves its pkg, cls, ret, $name$ and $param$ from the documentation.
- *Class Deprecation with Text.* REPFINDER extracts the name of the replacement class of $m_o.cls$ by matching patterns in Fig. 3. If matched, it validates the existence of the replacement class in JavaDoc. If validated, it finds the replacement API whose method name equals to $m_o.name$ and whose parameter types equal to $m_o.param$. If found, it resolves its pkg, cls, ret, $name$ and $param$ from the documentation.

It is worth mentioning that i) we use exact matching to find replacement classes and APIs to improve the accuracy; ii) we may find multiple replacement APIs as the method deprecation message might list multiple replacement APIs; and iii) once the last pattern in Fig. 3 is matched, we conclude no replacement.

Finally, REPFINDER validates whether the replacement APIs exist in l_n by searching the JavaDoc file of l_n because the replacement APIs might be resolved from the JavaDoc file of a library version before l_n. If validated, REPFINDER resolves the lib, puts the replacement APIs to $\mathcal{M}$, and returns $\mathcal{M}$. If not, REPFINDER continues to the next two steps.

B. Searching Own Library

Given l_o, l_n and m_o, the second step of REPFINDER is to find the replacement API by searching l_n. REPFINDER retrieves the jar file of l_n from our library database and finds the replacement API by sequentially applying following heuristics. Our idea is to first find the API with identical name in super or similar classes as m_o may be moved to such classes (see Table II), and then find the API with similar name in super or similar classes as m_o may be moved to such classes and changed (see Table II).

- *Search Identical API in Super Classes.* REPFINDER checks whether $m_o.cls$ exists in l_n. If yes, it analyzes $m_o.cls$ to obtain its parent class and grant parent class. Here, we empirically only track two super classes based on observations from our empirical study. Then, it checks whether each super class exists in l_n. If yes, it analyzes the super class in l_n to find the replacement API whose name equals to $m_o.name$ and whose parameter types equal to $m_o.param$. If found, it puts the replacement API to $\mathcal{M}$ and returns $\mathcal{M}$. Notice that if we find the replacement API from both its parent class and grant parent class, we use the replacement API from its parent class.
- *Search Identical API in Similar Classes.* REPFINDER checks whether $m_o.pkg$ exists in l_n. If yes, it iterates over each class cls under $m_o.pkg$ in l_n to compute the distance from cls to $m_o.cls$ by tokenizing class name according to camel case and changing plurals into singles before applying Levenshtein distance [52]. For the classes that are similar to $m_o.cls$ (i.e., the class distance is smaller than th_c), it analyzes the class from the smallest class distance to the largest class distance to find the replacement API whose name equals to $m_o.name$ and whose parameter types equal to $m_o.param$. Once it finds the replacement API, it puts it to $\mathcal{M}$ and returns $\mathcal{M}$. Besides, if $m_o.pkg$ does not exist in l_n, it searches across all packages in l_n in the same way to the above procedure.
- *Search Similar API in Super Classes.* This heuristic shares the same procedure to the first heuristic except that it finds the replacement API that is similar but not identical to m_o. To this end, we borrow the distance metric from Nguyen et al.'s work [59], which is computed as a weighted sum of the distances of return types, method names and parameter types, as shown in Eq. 1, where $dis()$ computes the Levenshtein distance [52].

$$\begin{aligned} dis(m_o, m_n) = {} & 0.25 * dis(m_o.ret, m_n.ret) \\ & + 0.50 * dis(m_o.name, m_n.name) \\ & + 0.25 * dis(m_o.param, m_n.param) \end{aligned} \quad (1)$$

  For the APIs whose distance to m_o is smaller than th_m^1, it selects the API with the smallest distance as the candidate API, which will be finally determined in the last step.
- *Search Similar API in Similar Classes.* This heuristic has the same procedure to the second heuristic except that it finds the replacement API that is similar but not identical to m_o. Thus, it applies the same distance metric to the third heuristic to find candidate API except that a different threshold th_m^2 is used.

C. Searching External Library

Given l_o, l_n, m_o and the candidate API, the last step of REPFINDER is triggered only when REPFINDER does not find the replacement APIs or only finds the candidate API in the previous steps. Our idea is to find the API with identical name in super or identical classes in some external libraries as m_o may be moved to such classes (see Table II).

To this end, it first collects four lists of external libraries, respectively for JDK library, dependency library, vendor library and similar library (see Sec. II-D). Specifically, it retrieves the jar file of a specific JDK library version, which can be configured by users as they often know the JDK library version their projects use, from our library database, and puts it to $\mathcal{L}_j$. Besides, it retrieves the POM file of l_n, uses the method in Wang et al. [86] to parse the POM file for a list of direct library dependencies of l_n, retrieves their jar file from our library database, and puts them to $\mathcal{L}_d$. Further, it queries our library database for all the libraries with the same group to $l_o.group$ but a different artifact to $l_o.artifact$. For each of such vendor libraries, it selects the same version number to $l_n.version$ if exists; otherwise, it selects the version number whose release date is no later than l_n but is closest to l_n, and then retrieves its jar file from our library database and puts it to $\mathcal{J}_v$. Finally, it uses tokens in $m_o.pkg$, $m_o.cls$ and $m_o.name$ to query our library database, which is indexed with libraries' group and artifact, selects the top ten hits as similar libraries, selects their version number whose release date is no later than l_n but is closest to l_n, and puts their jar files to $\mathcal{J}_s$.

Then, REPFINDER sequentially iterates over the four types of external libraries $\mathcal{L}_j$, $\mathcal{L}_d$, $\mathcal{L}_v$ and $\mathcal{L}_s$ to find the replacement API. This order is inspired by our empirical study as $\mathcal{L}_j$ has the highest possibility to provide the replacement while $\mathcal{L}_s$ has the lowest possibility. Once a replacement API is found during the iteration, REPFINDER returns it. Only if no replacement API is found in these external libraries, REPFINDER returns the candidate API as the replacement. REPFINDER uses following two heuristics to find the replacement in an external library l_e.

- *Search Identical API in Super Classes.* This heuristic has the same procedure as the first heuristic in Sec. III-B except that it checks whether each super class exists in l_e but not l_n, and if yes, it finds the identical API in the super class in l_e.
- *Search Identical API in Identical Classes.* This heuristic is similar to the second heuristic in Sec. III-B. It checks whether $m_o.pkg$ exists in l_e. If yes, it iterates over each class cls under $m_o.pkg$ in l_e to find the class whose name is identical to $m_o.cls$. If found, it analyzes the class to find the replacement API whose name is identical to $m_o.name$ and whose parameter types are identical to $m_o.param$. Once it finds the replacement API, it puts it to $\mathcal{M}$ and returns $\mathcal{M}$. Besides, if $m_o.pkg$ does not exist in l_e, it searches across all packages in l_e in the same way to the above procedure.

IV. EVALUATION

We have implemented a prototype of REPFINDER in 18.2K lines of Java code, and released the source code at our website https://repfinder.github.io/ with our experimental data set.

A. Evaluation Setup

To evaluate the effectiveness and efficiency of REPFINDER, we designed our evaluation to answer four research questions.

- **RQ5 Effectiveness Evaluation:** How is the effectiveness of REPFINDER, compared with state-of-the-art approaches?
- **RQ6 Efficiency Evaluation:** How is the time overhead of REPFINDER in finding replacement APIs?
- **RQ7 Sensitivity Analysis:** How is the sensitivity of each parameter in REPFINDER to the effectiveness of REPFINDER?
- **RQ8 Application Analysis:** How is effectiveness of applying REPFINDER to library update in real-life projects?

Data Set. We used the 683 missing APIs whose replacements were found or considered as none in our study (see Sec. II) as the data set for answering **RQ5**, **RQ6** and **RQ7**; i.e., we did not include the 55 uncertain missing APIs. Besides, to answer **RQ8**, we used 32 GitHub Java projects to update their used libraries.

Comparison Approaches. For **RQ5**, we selected two state-of-the-art approaches: i) AURA [89], which is a hybrid approach that combines similarity-based and usage-based approaches. We selected it because it achieved the best performance over previous approaches. ii) REFDIFF [75], which is the state-of-the-art refactoring detection tool. Although REFDIFF is not designed to find replacement APIs, we selected it because our study showed that a large part of missing APIs were refactored into their replacement APIs. Notice that we failed to compare REPFINDER with HIMA [50] which is another hybrid approach, as it requires libraries to use SVN but now libraries seldom use SVN. AURA and REFDIFF take as inputs two library versions, and detect all API mappings or API refactorings between the two versions. To align with our data set, we filtered their results to check whether they found the replacements for the missing API in our data set.

Evaluation Metrics. We used precision and recall as the indicator of effectiveness as they were widely used in prior work. For the ease of computation for precision and recall, similar to AURA [89], we converted a one-to-many mapping as many one-to-one mappings, a one-to-some mapping as a one-to-one mapping (i.e., we treated finding one of the some replacements as correct), and a many-to-one mapping as a one-to-one mapping and a many-to-many mapping as many one-to-one mappings (i.e., we only targeted the missing API). Due to such conversion, we had 735 mappings to find for 683 missing APIs.

TABLE IV. Effectiveness Comparison to State-of-the-Art

Approach	TP	FP	FN	Pre.	Rec.
AURA	159	135	576	0.54	0.22
REFDIFF	201	18	534	0.92	0.27
REPFINDER	509	75	226	0.87	0.69

B. Effectiveness Evaluation (RQ5)

Table IV reports the overall results of AURA, REFDIFF and REPFINDER with respect to true positive (*TP*), false positive (*FP*), false negative (*FN*), precision (*Pre.*) and recall (*Rec.*). We can see that REPFINDER significantly increased the number of true positives and reduced the number of false negatives, and thus significantly outperformed AURA and REFDIFF in recall by 213.6% and 155.6%. However, REPFINDER had a higher number of false positives than REFDIFF, resulting in a 5.4% decrease in precision than REFDIFF; but REPFINDER had a lower number of false positives than AURA, resulting in a 61.1% increase in precision than AURA. Besides, although REFDIFF was not originally designed for finding replacements, it achieved the highest precision as it could accurately detect refactorings.

TABLE V. Effectiveness Comparison w.r.t. Sources

Source	Num.	AURA					REFDIFF					REPFINDER				
		TP	FP	FN	Pre.	Rec.	TP	FP	FN	Pre.	Rec.	TP	FP	FN	Pre.	Rec.
Deprecation Message	111	11	9	100	0.55	0.10	51	1	60	0.98	0.46	94	15	17	0.86	0.85
Own Library	400	74	95	326	0.44	0.19	54	14	346	0.79	0.14	249	72	151	0.81	0.62
External Library	124	0	31	124	0.00	0.00	0	3	124	0.00	0.00	83	11	41	0.97	0.73
None	100	74	0	26	1.00	0.74	96	0	4	1.00	0.96	84	0	16	1.00	0.75

TABLE VI. Effectiveness Comparison w.r.t. Code Changes

Code Change	Change Level	Change Action	Num.	AURA		REFDIFF		REPFINDER	
				Pre.	Rec.	Pre.	Rec.	Pre.	Rec.
Refactoring	Class	Move Class	141	0.26	0.11	0.00	0.00	0.89	0.78
		Rename Class	80	0.88	0.54	1.00	0.05	0.95	0.86
	Method	Pull Up Method	147	0.33	0.10	0.87	0.53	0.99	0.95
		Push Down Method	5	0.25	0.20	1.00	0.20	0.80	0.80
		Change Method Signature	103	0.30	0.08	0.81	0.20	0.82	0.62
		Move Method	28	0.00	0.00	0.00	0.00	0.50	0.18
Substitution	Class	Substitute by Method with Different Name from Another Class	26	0.00	0.00	0.00	0.00	0.48	0.50
		Substitute by Method with Same Name from Another Class	8	0.00	0.00	1.00	0.13	0.89	1.00
	Method	Substitute by Method with Different Name from Own Class	5	0.00	0.00	0.00	0.00	0.25	0.20
Deletion	Class	Delete Class	33	1.00	0.79	1.00	1.00	1.00	0.73
	Method	Delete Method	67	1.00	0.72	1.00	0.87	1.00	0.76
Composition	NA	NA	92	0.16	0.03	0.00	0.00	0.54	0.23

Moreover, Table V breaks down the effectiveness results according to the sources where the replacements are found, where the second column (i.e., *Num.*) reports the number of mappings each approach should find (i.e., their sum should be 735). We can observe that AURA and REFDIFF achieved 0 precision and recall for missing APIs whose replacements were found in external libraries, while REPFINDER had a precision and recall of 0.97 and 0.73. It owes to the fact that AURA and REFDIFF are designed to not use the source of external libraries. Besides, for the source of deprecation message, AURA and REFDIFF could still find some replacement APIs by using the libraries, but REPFINDER achieved the highest recall by directly leveraging the knowledge in deprecation message. For the source of own library, REPFINDER achieved the highest precision and recall as our heuristics are designed based on a deep understanding of the characteristics of missing APIs and their replacements. For the missing APIs with no replacement, REPFINDER had a lower recall. These results demonstrate the importance of combining different knowledge sources for finding replacements.

In addition, Table VI breaks down the effectiveness results according to the code changes between missing APIs and their replacements. We can observe that for refactoring, REFDIFF had a 0 precision for *move class* and *move method* and a low recall, because i) REFDIFF often works at the commit level but it may work not well at the library version level due to the large amount of changes between versions, and ii) replacements can be in external libraries. AURA achieved low precision and recall across most change actions, because i) some missing APIs have limited usages and ii) replacements can be in external libraries. Instead, REPFINDER achieved a balanced precision and recall except for *move method*, because the missing API can be moved into a class that has a dissimilar name from its original residing class. For substitution, AURA and REFDIFF almost failed to find replacements. REPFINDER also achieved low precision and recall because the missing API can be substituted by a dissimilar method (in a dissimilar class). For deletion, REPFINDER had a lower recall than AURA and REFDIFF because AURA and REFDIFF found no replacement for a large number of missing APIs. For composition, all approaches had low precision and recall, while REPFINDER was the best. These results indicate that syntactic similarity measures used in REPFINDER may not be good enough to find replacement APIs, and some semantic similarity measures may be designed to further improve REPFINDER.

> REPFINDER significantly improved the state-of-the-art approaches in recall by up to 213.6%, while having a slight decrease in precision by 5.4%. We believe a slight decrease in precision is acceptable as recall increases satisfactorily.

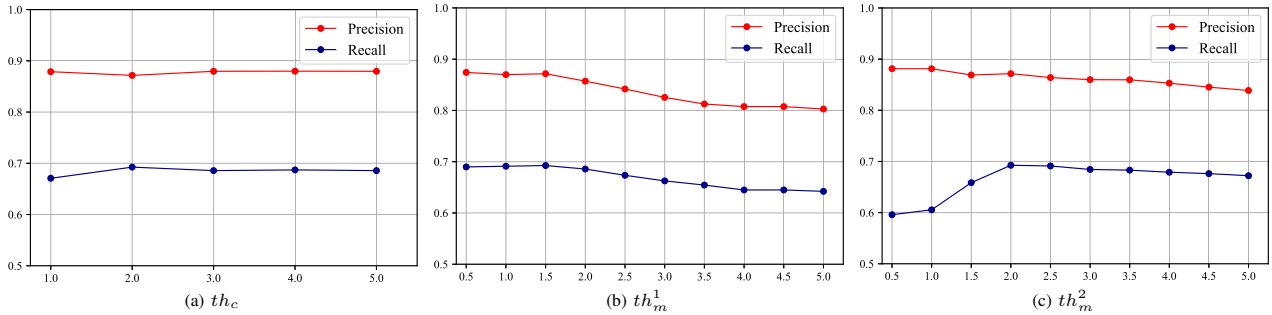
C. Efficiency Evaluation (RQ6)

REPFINDER took 689 seconds for finding replacements for 683 missing APIs. On average, REPFINDER took about one second to find the replacements for a missing API. We believe the efficiency of REPFINDER is good, and it can be practically used by developers. The good efficiency of REPFINDER owes to the availability of a library database as well as our lightweight design of heuristics. Notice that we did not compared the time overhead of AURA and REFDIFF because they found all API mappings or refactorings between two library versions.

> REPFINDER took about one second to find the replacements for a missing API, which was acceptable for practical usage by developers.

D. Sensitivity Analysis (RQ7)

Three thresholds, i.e., the class distance threshold th_c and two method distance thresholds th_m^1 and th_m^2, are configurable in the second step of REPFINDER (see Sec. III-B). The default configuration is 2, 1.5 and 2, which is used in the experiment for



(a) th_c (b) th_m^1 (c) th_m^2

Fig. 4. Results of Parameter Sensitivity Analysis

TABLE VII. Code Change between Missing APIs and their Replacements

Code Change	Change Level	Change Action	Deprecation Message	Own Library	External Library	None
Refactoring	Class	Move Class	-	5	50	-
		Rename Class	-	3	-	-
	Method	Pull Up Method	15	8	-	-
		Push Down Method	-	-	-	-
		Change Method Signature	1	5	1	-
		Move Method	-	-	-	-
Substitution	Class	Substitute by Method with Different Name from Another Class	1	-	2	-
		Substitute by Method with Same Name from Another Class	4	1	-	-
	Method	Substitute by Method with Different Name from Own Class	1	-	-	-
Deletion	Class	Delete Class	-	-	-	-
	Method	Delete Method	-	-	-	4
Composition	NA	NA	-	1	-	-

RQ5, **RQ6** and **RQ8**. To evaluate their sensitivity to the effectiveness of REPFINDER, we re-configured one threshold and fixed the other two, and re-ran REPFINDER against our data set. Specifically, th_c was configured from 1 to 5 by a step of 1, and th_m^1 and th_m^2 were configured from 0.5 to 5 by a step of 0.5.

Fig. 4 presents the impact of three thresholds on the precision and recall of REPFINDER, where x-axis denotes the value of threshold, and y-axis denotes the precision or recall. Overall, as th_c increased, the recall of REPFINDER first increased and then stabilized, while its precision was almost stable. Thus, we believe 2.0 is a good value for th_c. As th_m^1 increased, the recall and precision of REPFINDER were first stable and then decreased. Thus, we believe 1.5 is a good value for th_m^1. As th_m^2 increased, the recall of REPFINDER first greatly increased and then slightly decreased, while its precision slightly decreased. Hence, we believe 2 is a good value for th_m^2.

Overall, the sensitivity of the configurable parameters to the effectiveness of REPFINDER is acceptable.

E. Application Analysis (RQ8)

To apply REPFINDER to library update in real-life projects, we initially selected 168 GitHub Java projects which had more than 1000 stars and used Maven as the build tool. We analyzed the library dependencies in these projects, and found that 121 projects used 690 outdated library versions. Here we set our goal to update these outdated library versions to their latest version. Thus, we analyzed library API usage in these projects, and found that 59 outdated library versions in 33 projects had 105 used APIs missing in their latest version. Two of the authors followed the same procedure to Sec. II-A to manually find the replacements, but could not determine the replacements for 3 missing APIs. Hence, we ran REPFINDER against these 102 missing APIs, involving 57 outdated library versions in 32 projects. Notice that these 102 missing APIs had a overlap of 6 missing APIs to our data set in **RQ5**. Moreover, we also categorized code changes of these missing APIs to their replacements in Table VII. Compared to the results in Table II, three change actions, i.e., push down method, move method, and delete class, were not covered, indicating a relatively good representativity of this data set.

TABLE VIII. Effectiveness Comparison to State-of-the-Art

Approach	TP	FP	FN	Pre.	Rec.
AURA	10	25	104	0.29	0.09
REFDIFF	29	5	85	0.85	0.25
REPFINDER	84	4	30	0.95	0.74

REPFINDER achieved a precision of 0.95 and a recall of 0.74 on the 114 mappings for these 102 missing APIs, which were comparable to the results in **RQ5**, demonstrating the generality of REPFINDER. Specifically, REPFINDER successfully found replacements for all missing APIs for 21 (65.6%) projects, for partial missing APIs for 7 (21.9%) projects, but for no missing APIs for 4 (12.5%) projects. This result indicates that REPFINDER can be effectively used in real-life projects.

Moreover, we also compared REPFINDER with AURA and REFDIFF on this data set, as reported in Table VIII. While REFDIFF and REPFINDER achieved similar performance when compared to the results in Table IV, AURA had a performance degradation. Besides, we also broke down the effectiveness results on this data set according to the sources and code

TABLE IX. Effectiveness Comparison w.r.t. Sources

Source	Num.	AURA					REFDIFF					REPFINDER				
		TP	FP	FN	Pre.	Rec.	TP	FP	FN	Pre.	Rec.	TP	FP	FN	Pre.	Rec.
Deprecation Message	22	0	12	22	0.00	0.00	18	1	4	0.95	0.82	21	1	1	0.96	1.00
Own Library	35	9	4	26	0.70	0.26	8	2	27	0.80	0.23	17	1	18	0.94	0.49
External Library	53	0	9	53	0.00	0.00	0	2	53	0.00	0.00	34	11	19	0.96	0.81
None	4	1	0	3	1.00	0.25	3	0	1	1.00	0.75	2	0	2	1.00	0.50

TABLE X. Effectiveness Comparison w.r.t. Code Changes

Code Change	Change Level	Change Action	Num.	AURA		REFDIFF		REPFINDER	
				Pre.	Rec.	Pre.	Rec.	Pre.	Rec.
Refactoring	Class	Move Class	55	0.40	0.07	0.00	0.00	0.96	0.87
		Rename Class	3	0.00	0.00	1.00	0.67	1.00	1.00
	Method	Pull Up Method	23	0.13	0.09	0.95	0.87	0.96	0.96
		Push Down Method	-	-	-	-	-	-	-
		Change Method Signature	7	0.75	0.43	0.33	0.14	1.00	0.43
		Move Method	-	-	-	-	-	-	-
Substitution	Class	Substitute by Method with Different Name from Another Class	3	0.00	0.00	0.00	0.00	1.00	0.33
		Substitute by Method with Same Name from Another Class	5	0.00	0.00	0.00	0.00	0.80	0.80
	Method	Substitute by Method with Different Name from Own Class	1	0.00	0.00	0.00	0.00	1.00	1.00
Deletion	Class	Delete Class	-	-	-	-	-	-	-
	Method	Delete Method	4	1.00	0.25	1.00	0.75	1.00	0.50
Composition	NA	NA	13	0.00	0.00	0.00	0.00	0.00	0.00

changes, as shown in Table IX and X. Compared to the results in Table V and VI, most of the findings still hold. These results further demonstrate the generality of REPFINDER.

REPFINDER effectively found replacements for all missing APIs for 65.6% projects when all their outdated library versions were updated to the latest version.

F. Discussion

Limitations. First, REPFINDER mainly supports one-to-one mappings, and only provides partial support for one-to-many and one-to-some mappings by searching deprecation messages. We believe it is still useful as the majority of missing APIs have a one-to-one mapping to their replacements. One potential way to enhance REPFINDER is to consider the APIs called around the missing API and its found replacement to determine the possibility of a many-to-many mapping. Second, REPFINDER does not achieve good performance when the replacement API is dissimilar from the missing API or the residing class of replacement API is dissimilar from the residing class of missing API because REPFINDER only leverages syntactic similarity measures. We plan to leverage semantic similarity measures (e.g., code representation learning [1, 98]) to improve REPFINDER. Third, REPFINDER is currently implemented for the Java programming language. It is interesting to explore the generality of REPFINDER to other programming languages.

Threats. The main threat to our empirical study and evaluation is the manual construction of the replacements for missing APIs. To mitigate the threat, two of the authors conducted an independent analysis followed by a group discussion to make conclusions about replacement APIs. Further, we also intentionally focused on missing APIs that were actually used in client projects such that we could use the API usage information to help conclude whether the replacement APIs identified by two of the authors were correct or not.

V. RELATED WORK

We discuss the closely relevant work in five aspects, i.e., API adaptation, API deprecation, API breaking, API evolution and refactoring detection.

A. API Adaptation

A number of API adaptation approaches have been developed to update usages of missing APIs in client projects to usages of their replacement APIs for the ease of library update. To find the mapping between a missing API and its replacement, Chow and Notkin [13] and Nita and Notkin [60] designed method to allow library developers to manually specify the mapping. Henkel and Diwan [29] developed an IDE plugin to allow library developers to record API refactoring actions and allow client developers to replay them. These approaches often provide accurate mappings due to the manual involvement of library developers. In practice, however, such involvement is often not available, which hinders the generalizability of these approaches. Godfrey and Zou [26] proposed a semi-automated origin analysis using similarities of name, declaration, complexity metrics and call dependencies. Inspired by this approach, Kim et al. [37] extended the similarity measures and automated Godfrey and Zou's approach. Xing and Stroulia [95] used a set of heuristics to infer replacement APIs based on API changes identified from logical design models of two library versions. These similarity-based approaches identify replacement APIs from the source of the library itself but do not consider other sources. Schäfer et al. [74] proposed to mine the mapping from API usage changes in library's instantiation code (e.g., client projects), and Dagenais and Robillard [17, 18] tried to mine the mapping from API usage changes in library's own code. These usage-based approaches heavily rely on API usages in instantiation code or own code, and are practically infeasible as API usages are often not rich (since library's own code does not call every API, and only a small portion of APIs are called

across client projects [86]). Wu et al. [89] combined similarity-based and usage-based approaches, and considered the source of libraries of the same vendor. Meng et al. [50] analyzed commit message to infer the mapping in two consecutive commits and confirmed the mapping by analyzing source code, expanded the mapping in the similar way to similarity-based and usage-based approaches, and aggregated the mapping across commits. These hybrid approaches share similar limitations to similarity-based and usage-based approaches. Cossette and Walker [15] reported a retroactive study with five Java libraries to manually evaluate a set of techniques. Lamothe and Shang [42] explored how documentation and commits could be leveraged to find the mapping for Android APIs. Wu et al. [91] investigated how imperfect mappings affected client developers in updating libraries. It is worth mentioning that several approaches have been proposed for finding API mappings across similar libraries (e.g., [78, 79]) and different programming languages (e.g., [10, 99]).

To update usages of a missing API in client projects to usages of its replacement, several approaches have been proposed, e.g., by replacing calls to a deprecated API with its bodies [61], by type constraint analysis [2], by heuristic rules [81, 94], and by historical update examples [24, 43, 59, 80, 96]. Our approach is orthogonal to these approaches.

B. API Deprecation

Various empirical studies have been conducted to understand API deprecation in different programming languages. Zhou et al. [100] found that the classic *deprecate–replace–remove* cycle is often not followed as many APIs were removed without prior deprecation, many deprecated APIs were un-deprecated later, and many removed APIs are even resurrected. Some studies explored why developers deprecated APIs [53, 70, 71], e.g., avoiding bad code practices, functional and security bugs, redundant methods, merged into existing methods, and renaming methods. These studies reveal the importance of reacting to API deprecation. Otherwise, the quality or maintainability of client projects might be hurt. However, some studies studied how developers reacted to API deprecation [45, 68, 70, 72, 73, 84], and found that deprecated APIs were still widely used due to reasons like no suitable replacements and high update effort. Besides, some studies investigated the quality of documentation for deprecated APIs [8, 9, 38, 45, 58, 84]. They found that 67%, 78%, 67%, 53% and 78% of the APIs were deprecated with replacement messages in Java, C#, JavaScript, Python and Android. Apart from API documentation, we leverage other sources for finding replacement APIs. Except for API deprecation, Cogo et al. [14] analyzed how often and why package releases were deprecated in npm, and how client packages adopted deprecated releases.

C. API Breaking

Many empirical studies have investigated API breaking to analyze how and why developers break APIs [4, 5, 7, 35, 93], how API breaking impacts client projects [67, 92], etc. Besides, several advances have been made to detect API breaking. Theorem proving [25, 27, 41, 47, 48] and symbolic execution [56, 82] are used to detect behavioral API breaking, but suffer scalability issues. To be scalable, testing techniques are used to dynamically detect behavioral API breaking in Java [12, 28, 77], and heuristics are applied to statically detect signature API breaking (e.g., changes to API signatures) in Java [6]. In JavaScript, testing techniques are used to identify signature breaking [51, 55] and behavioral breaking [57], and heuristics are used to detect both signature and behavioral breaking [54]. Our approach is focused on signature API breaking as there is no need to find replacements for behavioral breaking APIs.

D. API Evolution

Several empirical studies have explored API evolution to understand how refactoring affects API evolution [22, 23, 36, 40], how client developers react to API evolution [30, 31, 85], how API stability is measured [49, 66], how Android API evolution affects apps' user ratings [3], how APIs are changed and used in Apache and Eclipse [90], how API evolution triggers stack overflow discussions [46], how and why APIs are evolved [32], etc. Our empirical study on missing APIs has a different goal than these studies, and provides insights for our approach.

E. Refactoring Detection

Several refactoring detection approaches have been proposed, e.g., RefDiff [75], RepFinder [65] and RMiner [83]. However, such tool cannot be directly applied to find replacement APIs because they are mostly work at the commit level and library version-level changes would be much more complex than commit-level changes.

VI. Conclusions

In this paper, we have presented an empirical study to understand the characteristics of missing APIs and their replacements. Inspired by our study results, we have propose a heuristic-based approach, named RepFinder, to automatically find replacements for missing APIs in library update. The key idea of RepFinder is to leverage multiple sources to find replacements. Our evaluation has demonstrated that RepFinder can find replacement APIs effectively and efficiently, and significantly outperformed the state-of-the-art approaches.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067). Bihuan Chen is the corresponding author of this paper.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, no. 4, p. 81, 2018.

[2] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *OOPSLA*, 2005, pp. 265–279.

[3] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, 2014.

[4] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *FSE*, 2016, pp. 109–120.

[5] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, 2020.

[6] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *SANER*, 2018, pp. 507–511.

[7] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how java developers break apis," in *SANER*, 2018, pp. 255–265.

[8] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate apis with replacement messages? a large-scale analysis on java systems," in *SANER*, 2016, pp. 360–369.

[9] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "On the use of replacement messages in api deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 306–321, 2018.

[10] N. D. Bui, Y. Yu, and L. Jiang, "Sar: learning cross-language api mappings with little knowledge," in *ESEC/FSE*, 2019, pp. 796–806.

[11] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *SANER*, 2015, pp. 516–519.

[12] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *ICSE*, 2020, pp. 112–124.

[13] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *ICSM*, 1996, pp. 359–368.

[14] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, 2021.

[15] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *FSE*, 2012, p. 55.

[16] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *ICSE*, vol. 2, 2015, pp. 109–118.

[17] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *ICSE*, 2009, pp. 599–602.

[18] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, p. 19, 2011.

[19] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *SANER*, 2017, pp. 2–12.

[20] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *MSR*, 2018, pp. 181–191.

[21] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *CCS*, 2017, pp. 2187–2200.

[22] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *ICSM*, 2005, pp. 389–398.

[23] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, 2006.

[24] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *ISSTA*, 2019, pp. 204–215.

[25] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *ASE*, 2014, pp. 349–360.

[26] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.

[27] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 241–258, 2013.

[28] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *ISSRE*, 2018, pp. 112–122.

[29] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *ICSE*, 2005, pp. 274–283.

[30] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *ICSME*, 2015, pp. 251–260.

[31] A. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse, "How do developers react to api evolution? a large-scale empirical study," *Software Quality Journal*, vol. 26, no. 1, pp. 161–191, 2018.

[32] D. Hou and X. Yao, "Exploring the intent behind api evolution: A case study," in *WCRE*, 2011, pp. 131–140.

[33] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: Generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, p. 679–690.

[34] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," in *ESEC/FSE*, 2020, pp. 518–529.

[35] K. Jezek, J. Dietrich, and P. Brada, "How java apis break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[36] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *ICSE*, 2011, pp. 151–160.

[37] S. Kim, K. Pan, and E. J. Whitehead, "When functions change their names: Automatic detection of origin relationships," in *WCRE*, 2005, pp. 10–pp.

[38] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. Le Traon, "Api document quality for resolving deprecated apis," in *APSEC*, vol. 2, 2014, pp. 27–30.

[39] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[40] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," *Information and Software Technology*, vol. 93, pp. 186–199, 2018.

[41] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *CAV*, 2012, pp. 712–717.

[42] M. Lamothe and W. Shang, "Exploring the use of automated api migrating techniques in practice: An experience report on android," in *MSR*, 2018, pp. 503–514.

[43] M. Lamothe, W. Shang, and T.-H. P. Chen, "A3: Assisting android api migrations using code examples," *IEEE Transactions on Software Engineering*, 2020.

[44] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," in *NDSS*, 2017.

[45] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2058–2098, 2020.

[46] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *ICPC*, 2014, pp. 83–94.

[47] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," in *ESEC/FSE*, 2003, pp. 287–296.

[48] S. McCamant and M. D. Ernst, "Early identification of incompatibilities in multi-component upgrades," in *ECOOP*, 2004, pp. 440–464.

[49] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *ICSM*, 2013, pp. 70–79.

[50] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *ICSE*, 2012, pp. 353–363.

[51] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in node. js libraries," in *ECOOP*, 2018.

[52] F. P. Miller, A. F. Vandome, and J. McBrewster, *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009.

[53] A. Mirian, N. Bhagat, C. Sadowski, A. P. Felt, S. Savage, and G. M. Voelker, "Web feature deprecation: a case study for chrome," in *ICSE-SEIP*, 2019, pp. 302–311.

[54] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," in *OOPSLA*, 2020, pp. 1–25.

[55] A. Møller and M. T. Torp, "Model-based testing of breaking changes in node. js libraries," in *ESEC/FSE*, 2019, pp. 409–419.

[56] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *ASE*, 2018, pp. 441–451.

[57] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *MSR*, 2020, pp. 466–476.

[58] R. Nascimento, A. Brito, A. Hora, and E. Figueiredo, "Javascript api deprecation in the wild: A first assessment," in *SANER*, 2020, pp. 567–571.

[59] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim,

and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *OOPSLA*, 2010, pp. 302–321.
[60] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *ICSE*, 2010, pp. 205–214.
[61] J. H. Perkins, "Automatically generating refactorings to support api evolution," in *PASTE*, 2005, pp. 111–114.
[62] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *ICSME*, 2015, pp. 411–420.
[63] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *ICSME*, 2018, pp. 449–460.
[64] T. Preston-Werner, "Semantic versioning 2.0. 0," *http://semver. org*, 2013.
[65] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *ICSM*, 2010, pp. 1–10.
[66] S. Raemaekers, A. Van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *ICSM*, 2012, pp. 378–387.
[67] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.
[68] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *FSE*, 2012, pp. 1–11.
[69] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia, and F. Ferrucci, "Do developers update third-party libraries in mobile apps?" in *ICPC*, 2018, pp. 255–265.
[70] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *ICSE*, 2018, pp. 561–571.
[71] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli, "Why are features deprecated? an investigation into the motivation behind deprecation," in *ICSME*, 2018, pp. 13–24.
[72] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis," in *ICSME*, 2016, pp. 400–410.
[73] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to api deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.
[74] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE*, 2008, pp. 471–480.
[75] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.
[76] N. Smith, D. van Bruggen, and F. Tomassetti, "Javaparser: Visited," *Leanpub, oct. de*, 2017.
[77] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.
[78] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *WCRE*, 2012, pp. 289–298.
[79] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *WCRE*, 2013, pp. 192–201.
[80] F. Thung, S. A. Haryono, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automated deprecated-api usage update for android apps: How far are we?" in *SANER*, 2020, pp. 602–611.
[81] F. Thung, H. J. Kang, L. Jiang, and D. Lo, "Towards generating transformation rules without examples for android api replacement," in *ICSME*, 2019, pp. 213–217.
[82] A. Trostanetski, O. Grumberg, and D. Kroening, "Modular demand-driven analysis of semantic difference for program versions," in *SAS*, 2017, pp. 405–427.
[83] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *ICSE*, 2018, pp. 483–494.
[84] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *ESEC/FSE*, 2020, pp. 233–244.
[85] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" in *ICSOC*, 2014, pp. 245–259.
[86] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *ICSME*, 2020, pp. 35–45.
[87] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: monitoring dependency conflicts for python library ecosystem," in *ICSE*, 2020, pp. 125–135.
[88] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *ESEC/FSE*, 2018, pp. 319–330.
[89] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *ICSE*, 2010, pp. 325–334.
[90] W. Wu, F. Khomh, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of api changes and usages based on apache and eclipse ecosystems," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2366–2412, 2016.
[91] W. Wu, A. Serveaux, Y.-G. Guéhéneuc, and G. Antoniol, "The impact of imperfect change rules on framework api evolution identification: an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1126–1158, 2015.
[92] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *SANER*, 2017, pp. 138–147.
[93] L. Xavier, A. Hora, and M. T. Valente, "Why do we break apis? first answers from developers," in *SANER*, 2017, pp. 392–396.
[94] Y. Xi, L. Shen, Y. Gui, and W. Zhao, "Migrating deprecated api to documented replacement: Patterns and tool," in *Internetware*, 2019, pp. 1–10.
[95] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.
[96] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *ICPC*, 2019, pp. 335–346.
[97] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *ICSR*, 2018, pp. 95–110.
[98] F. Zhang, B. Chen, R. Li, and X. Peng, "A hybrid code representation learning approach for predicting method names," *Journal of Systems and Software*, vol. 180, 2021.
[99] W. Zheng, Q. Zhang, and M. Lyu, "Cross-library api recommendation using web search engines," in *ESEC/FSE*, 2011, pp. 480–483.
[100] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *FSE*, 2016, pp. 266–277.
[101] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security*, 2019.