Assessing the Threat of Untracked Changes in Software Evolution

Andre Hora
FACOM, UFMS, Brazil
hora@facom.ufms.br

Danilo Silva, Marco Tulio Valente
ASERG Group, DCC, UFMG, Brazil
{danilofs,mtov}@dcc.ufmg.br

Romain Robbes
SwSE Group, Free University of Bozen-Bolzano, Italy
rrobbes@unibz.it

ABSTRACT

While refactoring is extensively performed by practitioners, many Mining Software Repositories (MSR) approaches do not detect nor keep track of refactorings when performing source code evolution analysis. In the best case, keeping track of refactorings could be unnecessary work; in the worst case, these *untracked changes* could significantly affect the performance of MSR approaches. Since the extent of the threat is unknown, the goal of this paper is to assess whether it is significant. Based on an extensive empirical study, we answer positively: we found that between 10 and 21% of changes at the method level in 15 large Java systems are untracked. This results in a large proportion (25%) of entities that may have their histories split by these changes, and a measurable effect on at least two MSR approaches. We conclude that handling untracked changes should be systematically considered by MSR studies.

CCS CONCEPTS

• **Software and its engineering → Software evolution**;

KEYWORDS

Mining Software Repositories, Software Evolution, Refactoring

ACM Reference Format:
Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. 2018. Assessing the Threat of Untracked Changes in Software Evolution. In *ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180212

1 INTRODUCTION

Mining Software Repositories (MSR) techniques are helping to improve our understanding of software development and contributing to the implementation of a new generation of software engineering tools. Many of the existing MSR techniques are based on the analysis of changes performed on source code repositories [30]. For example, change-based techniques have been proposed to support library migration [27, 45, 66], change prediction [69], bug fixing [36], warnings prioritization [5, 9, 11, 34, 35], and expertise calculation [3, 44, 51, 62], to name a few.

Generally, these techniques operate on the level of the individual methods and classes changed on each commit. For example, by mining the history of the methods in a system, change-based techniques can learn that methods that depend on type `A` are often updated to depend on an improved type `A'`. A recommendation can then be provided to alert developers about the methods not yet updated in the system. However, these techniques require tracking the changes along all versions of each individual method (or class) in the system under analysis. The challenge is that some changes invalidate this tracking when it is solely based on the names of the tracked entities. Specifically, refactorings may introduce discontinuities in name-based tracking strategies. For instance, a method rename or move can be misinterpreted as the disappearance of a method and the appearance of a brand new one, splitting its history, and thus invalidate the tracking. We call the changes that affect the names of code entities *untracked changes* (this definition is detailed in Section 2). If not properly handled, untracked changes may have a negative impact on the accuracy of MSR-based techniques.

While the threat of untracked changes has been acknowledged in the literature (several examples are presented in Section 3), to our knowledge, the actual extent of these changes on MSR studies has not been investigated. Assessing the extent of threats to MSR studies is essential; prior work has investigated bias in bug-fix datasets [8], or non-essential changes [30]. In this paper, we perform such a study: we assess the frequency, extension, and impact of the threat of untracked changes. We detail our methodology in Section 4 and our selection of case studies (15 popular Java systems) in Section 5. We then answer the following research questions:

- *RQ1. What is the frequency of untracked changes?* In Section 6, we find that between 10 and 21% of method-level changes and 2 and 15% of class-level changes are untracked. The most common ones are due to rename, extract, and move method.
- *RQ2. What is the extension of untracked changes?* Section 7 shows that 25% of entities have at least one untracked change in their histories, and thus may have their history split. In the most changed entities, the proportion raises to 37%.
- *RQ3. What is the impact of untracked changes in existing MSR-based approaches*? In Section 8, we investigate the concrete impact of untracked changes in two MSR approaches, namely API evolution and API co-usage rule mining. We find that, on the median, up to 6.9% and 5.3%, respectively, more rules are discovered when untracked changes are considered.

Thus, the contributions of this work are twofold: (i) we provide an empirical study to assess the frequency and extension of untracked changes at a large scale and (ii) we measure the concrete impact of untracked changes in two MSR-based approaches.

We discuss the implications and threats to validity of our study in Sections 9 and 10, respectively. Finally, we cover related work (studies of threats to MSR studies, and approaches to detect untracked changes) in Section 11, and conclude in Section 12.

2 TRACKED AND UNTRACKED CHANGES

This section defines the two categories of changes that we analyze in this study. During software evolution, two kinds of changes can occur to code entities such as classes and methods: tracked and untracked changes. A *tracked change* preserves the entity name and modifies its source code. Thus, an entity in version *n* can be directly matched to its following version *n+1*. In contrast, an *untracked change* modifies the entity name, and may also modify its source code. They occur due to, for example, rename or move refactorings. In addition, an *untracked change* may also spawn a new entity, for example, as the result of extract method refactorings. Therefore, an entity in version *n* is not matched to its version *n+1*, unless the untracked change is resolved with the support of a refactoring detection approach (*e.g.,* [32, 57, 58]).[1]

Figure 1 illustrates examples of tracked and untracked changes in four source code versions. Each box represents a class and its methods, and each arrow represents an evolutionary change between two consecutive versions. Class `Foo` and its method `mA()` undergo some tracked changes (solid arrow), since they do not change entity names. Thus, to understand the evolution of `Foo` and `mA()`, one can trace them backward or forward. For example, the last version of `mA()` can be straightforwardly traced back to its first version, and vice-versa. On the other hand, class `Bar` and its method `mB()` undergo some untracked changes (dashed arrow). Method `mB()` is renamed to `mX()` in version 2, and then to `mY()` in version 4; class `Bar` is also renamed to `Baz` in version 4. In this case, `mY()` in version 4 may not be traced back to its originating method `mB()` in version 1, due to the method renaming. Consider now the more challenging scenario: method `mC()` is moved from class `Bar` to `Qux` in version 3, and then, method `mE()` is extracted from `mC()` in version 4. Notice that two non-trivial untracked changes happen: move and extract method. In this case, methods `mC()` and `mE()` in version 4 have both their origin in version 1 of `mC()`. Therefore, if the untracked change is not resolved, method `mE()` may be simply misdetected as a new method, and its actual origin may not be found.

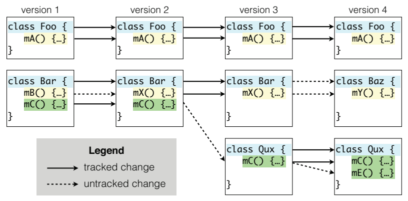


Figure 1: Example of tracked and untracked changes.

Untracked changes at the class level (*e.g.,* class renaming or moving) behave similarly, and, if unresolved, can be misdetected in the same fashion, albeit on a larger scale, since losing track of a class means also losing track of its methods.

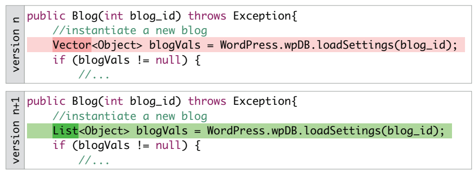
[1]Another name to *tracked* and *untracked changes* could be *history-preserving* and *history-destroying changes*, respectively.

3 THE THREAT OF UNTRACKED CHANGES

In this section we present three MSR research lines affected by untracked changes. We illustrate each scenario with real-world examples extracted from open-source projects. We conclude the section by generalizing our discussion to other affected research lines, and state the problem we investigate.

3.1 Scenario 1: Mining Code Evolution

The first scenario comes from a research line intended to mine code evolution. The main idea is to compare two versions of a code entity, and to learn from the differences. Applications of this line are broad, including learning how bugs are fixed [36], detection of behavioral breaking changes [59], and extraction of rules to support library migration [27–29, 45, 66]. Figure 2 illustrates an example in the context of library migration, from project WordPress-Android.[2] Version *n* has a reference to class `Vector` that is replaced by a reference to `List` in version *n+1*. The example shows a tracked change: method `Blog()` in version *n* is directly matched to method `Blog()` in version *n+1*. By comparing the differences between both method versions, one may infer the rule `Vector` → `List`.

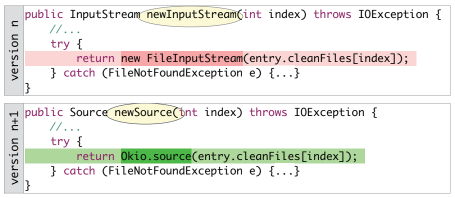


```
public Blog(int blog_id) throws Exception{
    //instantiate a new blog
    Vector<Object> blogVals = WordPress.wpDB.loadSettings(blog_id);
    if (blogVals != null) {
        //...
```

```
public Blog(int blog_id) throws Exception{
    //instantiate a new blog
    List<Object> blogVals = WordPress.wpDB.loadSettings(blog_id);
    if (blogVals != null) {
        //...
```

Figure 2: Two versions of method `Blog()`. Rule `Vector` → `List` can be inferred by comparing both method versions (project WordPress-Android).

Consider the example in Figure 3, from project OkHttp.[3] Version *n* has an invocation to `FileInputStream()` that is replaced by an invocation to `Okio.source()` in version *n+1*. The example presents an untracked change: method `newInputStream()` in version *n* is renamed to `newSource()` in version *n+1*. Thus, unless the method renaming is resolved, this change may be seen as a removal of `newInputStream()` and an addition of `newSource()`. In this case, these methods would not be matched, and, consequently, the rule `FileInputStream()` → `Okio.source()` would not be inferred.



```
public InputStream newInputStream(int index) throws IOException {
    //...
    try {
        return new FileInputStream(entry.cleanFiles[index]);
    } catch (FileNotFoundException e) {...}
}
```

```
public Source newSource(int index) throws IOException {
    //...
    try {
        return Okio.source(entry.cleanFiles[index]);
    } catch (FileNotFoundException e) {...}
}
```

Figure 3: Method `newInputStream()` is renamed to `newSource()`. Rule `FileInputStream()` → `Okio.source()` may not be detected, after a renaming (project OkHttp).

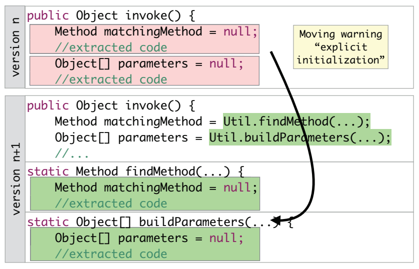
[2]Change available at: https://goo.gl/NHvjgH
[3]Change available at: https://goo.gl/2NvgC1

The literature recognizes the issues caused by untracked changes such as class or method renaming. Commonly, they are treated as threats to validity or limitations. For example, in the context of API evolution, Hora et al. [27] state: "*API changes are automatically produced by applying the [...] technique on the set of method call changes between two versions of one method*". Similarly, in the context of bug fixing, Kim et al. [36] notice: "*Our approach is based on comparing a source code file of two versions, so the bugs captured and fixes suggested are only file-by-file based*". In both cases, rename refactorings are not resolved. Raemaekers et al. [50] plainly state: "*We detect renamed or moved units as units that are removed first and added later*". In the domain of behavioral breaking changes, Soares et al. [59] note: "*A similar thing occurs when renaming a class. We cannot compare the renamed method's behavior directly*".

3.2 Scenario 2: Prioritizing Code Warnings

The second scenario comes from a research line meant to prioritize warnings (or code violations) reported by static analysis tools, such as FindBugs [4], PMD [13], and Checkstyle [10]. These tools are used to find common programming issues, related to performance, security, legibility, etc. In practice, static analysis tools are known to report too many warnings. Consequently, most reported warnings are unlikely to be fixed by developers, which are known as false positives [35, 48]. In this context, researchers propose approaches to filter out false positives (*e.g.,* [5, 9, 11, 14, 34, 35, 65]). A common approach is to detect the most fixed warnings over time, which are more likely to be relevant than warnings never fixed by any developer. For example, the warning "*method parameters should be final*"[4] is systematically fixed in project Easy Properties.[5] Thus, such warning could receive a high prioritization in this project.

Consider now the example in Figure 4, which shows two versions of method `invoke()` in project Apache Tomcat.[6] Version *n* contains a warning (*i.e.,* "*explicit initialization with null*"[7]) that seems to be fixed in version *n+1*. However, the warning is not fixed but only moved to methods `findMethod()` and `buildParameters()` in version *n+1*, as the result of a extract method refactoring. In this case, approaches may misclassify the warning as fixed if the extract method refactoring is not taken into account.



```
public Object invoke() {
    Method matchingMethod = null;
    //extracted code
    Object[] parameters = null;
    //extracted code
```

```
public Object invoke() {
    Method matchingMethod = Util.findMethod(...);
    Object[] parameters = Util.buildParameters(...);
    //...
static Method findMethod(...) {
    Method matchingMethod = null;
    //extracted code
static Object[] buildParameters(...) {
    Object[] parameters = null;
    //extracted code
```

Figure 4: Warning "*explicit initialization with null*" is moved from method `invoke()` to `findMethod()` and `buildParameters()`, after a extract method (project Tomcat).

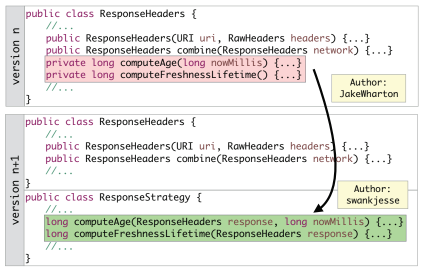
[4]http://checkstyle.sourceforge.net/config_misc.html#FinalParameters
[5]Change available at: https://goo.gl/mBL0QK
[6]Change available at: https://goo.gl/5LbyQT
[7]http://checkstyle.sourceforge.net/config_coding.html#ExplicitInitialization

As in Scenario 1, the literature also notices theses threats, where missing or misclassifing results may be produced when code changes are not tracked. For example, Ayewah and Pugh [5] state: "*If a method is renamed or moved to another class, any issues in that method will be reported as being removed [...]*". Peters and Zaidman [48] note: "*[Our tool] is unable to determine if an entity in revision n has been renamed in revision n+1*". Kim and Ernst [34] mention: "*We exclude removed warnings due to any file deletion. If there is a file deletion during a fix, all warnings in the files are removed*" (note that a class rename can be misdetected as a file deletion).

3.3 Scenario 3: Detecting Code Authorship

The last research line is intended to detect code authorship. In short, the idea is to infer expert developers who are best qualified to maintain certain code files (*e.g.,* [1, 3, 18, 19, 23, 44, 47, 51, 56, 62]). Approaches in this area often take advantage of facilities provided by SCM tools such as Git, SVN, and CVS. For example, Git provides the facility `git blame`[8], which shows what revision and author last modified each line of a file. By using this information, approaches in this domain can discover the developer who created a file, the developer who most modified a file, among others. However, this process is sensitive to refactoring, such as moving [3].

Consider the example in Figure 5, which presents a move method refactoring in project OkHttp.[9] Methods `computeAge()` and `computeFreshnessLifetime()` are moved from class `ResponseHeaders` in version *n* to class `ResponseStrategy` in version *n+1*. As SCM tools cannot track fine-grained refactorings, the actual author of these methods (*i.e., JakeWharton*) is lost, after a move method. In this case, the author in version *n+1* would be misdetected as *swankjesse*, *i.e.,* the one who performed the move refactoring.



```
public class ResponseHeaders {
    //...
    public ResponseHeaders(URI uri, RawHeaders headers) {...}
    public ResponseHeaders combine(ResponseHeaders network) {...}
    private long computeAge(long nowMillis) {...}
    private long computeFreshnessLifetime() {...}
    //...
}
```

```
public class ResponseHeaders {
    //...
    public ResponseHeaders(URI uri, RawHeaders headers) {...}
    public ResponseHeaders combine(ResponseHeaders network) {...}
    //...
}
public class ResponseStrategy {
    //...
    long computeAge(ResponseHeaders response, long nowMillis) {...}
    long computeFreshnessLifetime(ResponseHeaders response) {...}
    //...
}
```

Figure 5: Actual author of methods `computeAge()` and `computeFreshnessLifetime()` (*i.e.,* JakeWharton) is lost, after a move method (project OkHttp).

These limitations are recognized by related literature. For instance, Avelino et al. [3] state: "*The full development history of a file can be lost in case of renaming operations, copy or file split. We address the former problem using Git facilities. However, we acknowledge the need for further empirical investigation to assess the true impact of the other cases*". Similarly, Spinellis [62] note: "*The git blame command works by traversing backwards a repository's history [...]. Consequently, deleted snapshots would create a discontinuity between them, and prevent the tracing of code between them*".

[8]https://git-scm.com/docs/git-blame
[9]Change available at: https://goo.gl/5HYf5p

3.4 Other Scenarios

The aforementioned scenarios show some MSR research lines and studies affected by the threat of untracked code changes. Notice, however, that the list of presented research lines is not intended to be exhaustive. In addition to the previous ones, other lines include studies and threats in the context of (i) bug introducing change detection[10] (*e.g.,* [2, 12, 26, 37–39, 53, 54, 67, 68]): "*It is also possible to miss bug-introducing changes when a file changes its name since the algorithm does not track such name changes*" [38]; (ii) code evolution understanding (*e.g.,* [6, 7, 42, 61, 62]): "*To distinguish cases where a method was removed and a new one added from cases when a method was renamed, we use a heuristic that maps methods with different names [...]*" [7, 42]; (iii) code evolution visual supporting (*e.g.,* [21, 22]): "*[Our tool] does not perform any recovery of refactorings such as renaming, pull or push down methods*" [22], among other.

3.5 Problem: What is the Extent of the Threat of Untracked Changes?

The threat of untracked changes may be faced (at a higher or a lower level) by several MSR-based studies. In practice, however, this threat is commonly not assessed by researchers, therefore, we are still unaware about its real size. Based on that, one important question appears: *what is the frequency, extension, and impact of the threat of untracked changes?* Answering this question has practical consequences: researchers can better quantify the threat's impact in MSR-based studies. Consequently, they will have better support to decide whether the threat is large enough that they should address it, or if they can ignore it. In the remainder of this paper, we aim to answer this question.

4 ASSESSING UNTRACKED CHANGES

4.1 Detecting Untracked Changes

RefDiff. We rely on RefDiff [58] to detect both tracked and untracked changes. RefDiff is a tool that identifies refactorings performed in the version history of a system. The tool relies on a combination of heuristics based on static analysis and code similarity to detect 11 well-known refactoring operations that can lead to untracked changes at the class or method levels. RefDiff also detects tracked changes at both class and method levels, as summarized in Table 1. Essentially, RefDiff receives as input two versions v_1 and v_2 of a system, and outputs a list of changes performed in v_2, when compared to v_1.

Table 1: Types of tracked and untracked changes.

Change	Type
Tracked	Same Class, Same Method
Untracked	Rename Class, Move Class, Move and Rename Class, Extract Interface, Extract Superclass, Rename Method, Move Method, Extract Method, Inline Method, Pull Up Method, Push Down Method

[10]These studies are often affected by the same issues presented in Scenario 3, since they often make use SCM facility tools.

RefDiff's authors provide two evaluations of their tool. First, they evaluated the tool using an oracle with well-known refactoring instances performed by students in seven Java projects. As presented in Table 2 (column Eval #1), RefDiff achieved a precision of 100% in all refactoring types; overall recall was 93.9%, ranging from 60% (Pull Up Method) to 100% (Rename Class and Move Method). Although the tool detects these refactorings, in their evaluation, RefDiff's authors did not consider two refactoring operations: Move and Rename Class and Extract Interface. In this evaluation, RefDiff also outperformed the results of similar tools, including Refactoring Miner [57, 64], Refactoring Crawler [16], and RefFinder [32]. These mentioned tools achieved a precision of 96%, 42%, and 26%, respectively; regarding recall, the results were 73%, 36%, and 64%, respectively. *RefDiff thus constitutes, at the time of writing, the state of the art in refactoring detection.* RefDiff's authors provide a second evaluation, using 102 real refactoring instances from ten GitHub projects. As presented in Table 2 (column Eval #2), with these refactorings, RefDiff achieved an overall precision of 85.4% and an overall recall of 93.6%.

Table 2: Precision and recall of RefDiff.

Refactoring	Eval #1 [58]		Eval #2 [58]		Eval #3	
	Prec	Recall	Prec	Recall	Prec	Recall
Rename Class	100	100	100	100	95.0	87.5
Move Class	100	96.8	100	100	98.3	89.5
Extract Superclass	100	87.5	100	100	-	66.7
Move and Rename Class	-	-	-	-	71.4	100
Extract Interface	-	-	-	-	100	-
Rename Method	100	94.3	88.0	91.7	89.7	92.3
Move Method	100	100	95.5	87.5	92.4	100
Extract Method	100	89.7	73.5	100	79.2	66.7
Inline Method	100	98.1	71.4	83.3	93.8	83.3
Pull Up Method	100	60.0	100	100	100	66.7
Push Down Method	100	97.1	100	100	100	-
All Refactorings	100	93.9	85.4	93.6	89.1	89.8

To increase our confidence on RefDiff's accuracy, we manually validated a set of refactorings detected by the tool in the version history of the projects investigated in this paper (which are presented in Section 5). First, we executed RefDiff in the commit history of these systems. We then randomly selected 383 of such refactorings for manual validation by the second author of this paper (with this sample size, we ensure a confidence level of 95% and confidence interval of 5% [63]).[11] For each refactoring, he inspected the textual difference produced by GitHub for the respective commit; he then classified the detected refactorings as true or false positives. As presented in Table 2 (column Eval #3), the overall precision in this third evaluation was 89.1%, ranging from 71.4% (Move and Rename Class) to 100% (Extract Interface and Pull Up/Push Down Method).

Finally, we evaluated RefDiff's recall using well-know refactorings performed in the version history of the projects investigated in this paper. To create this gold set, we first searched in the textual description of the commits performed in these systems for regular expressions denoting the refactorings considered in the study (*e.g.,* "extract method"). Then, the second author of this paper checked

[11]This selection includes instances of all refactorings detected by RefDiff, with the exception of Extract Superclass; since it is a rare refactoring, no instance of this refactoring was included in the random sample of refactorings.

whether these commits indeed include the refactorings declared in their descriptions. In this way, we created a gold set with 127 validated refactoring instances, which was used to compute recall. As shown in Table 2 (column Eval #3), the overall recall in this third evaluation was 89.8%, ranging from 66.7% (Extract Superclass, Extract Method, and Pull Up Method) to 100% (Move and Rename Class and Move Method).

Change Graph. To facilitate the evolutionary analysis of classes, methods, and their related changes, we model a graph—the *change graph*—which is built by running RefDiff on a set of system versions (*i.e.,* commits). In this graph, each class or method is represented as a node while each tracked or untracked change is represented as an edge between two nodes. Figure 6 presents the change graph for the example described in Figure 1. We notice the entities represented as nodes and the changes represented as edges.

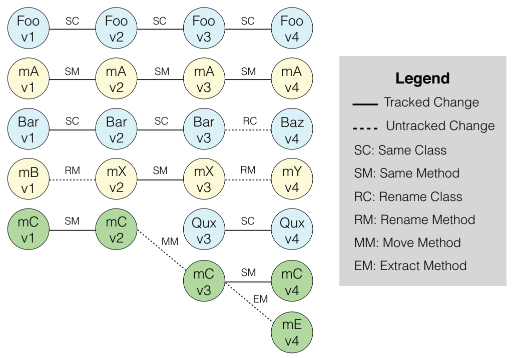


Figure 6: Change graph for the example in Figure 1.

4.2 Measuring Untracked Changes

After generating the *change graph*, we can assess the frequency and extension of tracked and untracked changes. These assessments are later used to answer research questions 1 and 2.

Frequency. The frequency of changes is assessed by counting the number of edges in the change graph. Each change can be then classified as tracked or untracked. In Figure 6, the change graph has 17 changes, from which 12 are tracked and 5 are untracked. Specifically, it contains the following change types: 6 Same Class (SC), 6 Same Method (SM), 1 Rename Class (RC), 2 Rename Method (RM), 1 Move Method (MM), and 1 Extract Method (EM).

Extension. The extension of changes is computed by assessing the paths in the change graph. We consider a path to represent the history of an entity. Each path may be formed by tracked and/or untracked changes. Paths with untracked changes are not desirable, because their histories may be split (if the untracked changes are not resolved), decreasing traceability of the entity history. In Figure 6, for example, the change graph has 7 paths. Three paths include only tracked changes: $Foo_{v1}...Foo_{v4}$ (length 3), $mA_{v1}...mA_{v4}$ (length 3), and $Qux_{v3}...Qux_{v4}$ (length 1). Four paths include at least one untracked changes: $Bar_{v1}...Baz_{v4}$ (length 3), $mB_{v1}...mY_{v4}$ (length 3), $mC_{v1}...mC_{v4}$ (length 3), and $mC_{v1}...mE_{v4}$ (length 3). Thus, traceability is more precise when untracked changes are resolved.

5 EXPERIMENTAL DESIGN

5.1 Selecting Case Studies

In this study we analyze tracked and untracked changes that happen in real-world and popular software systems. We collect the 10 most popular Java systems hosted on GitHub, as sorted by the star metric. We restrict our analysis to systems with more than 1K commits to filter out newer and less active ones. We also filter out projects not related to software systems.[12] In addition to these top 10 popular systems, we added 5 relevant systems from large organizations such as Google, Facebook, and Apache, totalling 15 case studies.

Table 3 presents an overview of the case studies in terms of commits, contributors, stars, forks as well as a short description.[13] The most popular project is RxJava (24,751 stars) while the most forked is Spring Framework (10,518 forks). The commits range from 1,025 (Android Image Loader) to 39,389 (Kotlin) while contributors range from 35 (Android Image Loader) to 837 (Elasticsearch). The selected systems cover distinct domains, such as search engines, programming languages, and software tools.

Table 3: Overview of case studies.

Project	Com.	Cont.	Stars	Forks	Short Description
RxJava	5,109	162	24,751	4,348	Event-based lib.
Elasticsearch	27,738	837	23,057	8,110	Search engine
Retrofit	1,482	110	21,606	4,426	HTTP client
OkHttp	2,963	137	20,189	4,999	HTTP client
Google Guava	4,173	105	16,777	3,933	Core lib. for Java
MPAndroidChart	1,900	57	15,944	4,767	Android view lib.
Glide	1,639	49	15,703	3,238	Android image lib.
Android Image	1,025	35	15,284	6,430	Android image lib.
Kotlin	39,389	159	14,525	1,358	Programming lang.
Spring	14,792	219	14,303	10,518	Support framework
Facebook Fresco	1,342	85	12,775	3,343	Android image lib.
Clojure	3,065	125	6,339	1,064	Programming lang.
Google Guice	1,611	36	5,148	821	Dependency injection
Apache Storm	8,394	259	4,195	3,108	Distributed system
Eclipse Che	3,462	74	4,166	691	Eclipse IDE

5.2 Selecting Commits

After collecting the case studies, we need to select the commits (*i.e.,* versions) to be analyzed. Projects using distributed version control systems such as Git may have several branches under development. To facilitate code evolution analysis, in this study, we focus on the evolution of the main branch. For this purpose, we use the command `git log --first-parent`[14] to select the analyzed commits, since the Git documentation clearly states: "*This option can give a better overview when viewing the evolution of a particular branch*".

5.3 Computing Untracked Changes

In this final step, we perform the approaches described in Section 4. We run RefDiff for each case study and its respective set of commits. We then compute the *change graph* to investigate the frequency and extension of untracked changes. Our source code implementation and results are publicly available.[15]

[12]For example: https://github.com/iluwatar/java-design-patterns
[13]The data was collected on 05, June, 2017.
[14]https://git-scm.com/docs/git-log#git-log---first-parent
[15]Available at: https://github.com/andrehora/rastreability

6 RQ1: WHAT IS THE FREQUENCY OF UNTRACKED CHANGES?

Table 4 presents the frequency of untracked changes at the class and method levels. At the *class level*, the systems with proportionally more untracked changes are Apache Storm (15%), Eclipse Che (14%), and RxJava (13%). In contrast, the ones with less untracked changes are Google Guava (2%), MPAndroidChart (2%), and Facebook Fresco (3%). When considering all projects, 6% of the changes are untracked at class level. At the *method level*, the systems with more untracked changes are Glide (21%), Android Image (21%), and Spring (20%), while RxJava (10%), Kotlin (13%), and MPAndroidChart (13%) have less. When considering all projects, 16% of the method changes are untracked. Thus, the threat of untracked changes happens at both levels, but is more frequent for methods.

Table 4: Frequency of untracked changes per project.

Project	Frequency of Changes				
	Total	Tracked		Untracked	
		Class	Method	Class (%)	Method (%)
RxJava	50,989	16,195	29,095	2,339 (13%)	3,360 (10%)
Elasticsearch	245,991	97,791	11,9691	4,961 (5%)	23,548 (16%)
Retrofit	8,786	4,468	3,285	302 (6%)	731 (18%)
OkHttp	20,200	5,778	11,247	669 (10%)	2,506 (18%)
Google Guava	42,893	17,987	19,995	391 (2%)	4,520 (18%)
MPAndroidChart	11,775	4,891	5,871	111 (2%)	902 (13%)
Glide	10,640	4,069	4,985	228 (5%)	1,358 (21%)
Android Image	2,338	984	1,006	82 (8%)	266 (21%)
Kotlin	236,144	97,396	116,981	4,523 (4%)	17,244 (13%)
Spring	108,811	42,214	51,519	2,443 (5%)	12,635 (20%)
Facebook Fresco	7,222	2,786	3,674	95 (3%)	667 (15%)
Clojure	10,333	4,616	4,650	165 (3%)	902 (16%)
Google Guice	15,300	5,869	7,822	360 (6%)	1,249 (14%)
Apache Storm	14,260	5,502	6,716	961 (15%)	1,081 (14%)
Eclipse Che	24,602	8,284	12,281	1,301 (14%)	2,736 (18%)
All Projects	810,284	318,830	398,818	18,931 (6%)	73,705 (16%)

Table 5 shows the frequency of untracked changes per type. The most frequent untracked changes happen at the method level and are due to Rename Method (23K), Extract Method (21K), and Move Method (20K). In contrast, the least frequent ones are due to Extract Superclass (0.3K), Extract Interface (0.8K), and Push Down Method (0.8K). Therefore, keeping track of classical untracked changes such as class or method renaming is important—but not enough. Other changes such as method extraction and moving should also be addressed to ensure a more complete tracking.

Table 5: Frequency of untracked changes per type.

Level	Change Type	#	%
Class	Move Class	11,319	12%
	Rename Class	4,893	5%
	Move and Rename Class	1,585	2%
	Extract Interface	822	1%
	Extract Superclass	312	<1%
Method	Rename Method	23,921	26%
	Extract Method	21,483	23%
	Move Method	20,198	22%
	Inline Method	5,511	6%
	Pull Up Method	1,718	2%
	Push Down Method	874	1%

Untracked changes constitute up to 21% of the changes at the method level and up to 15% at the class level, therefore they should not be neglected. This may directly affect MSR studies that compare two versions of one class or method (such as the ones presented in Scenarios 1 and 2). Moreover, if *one in five* changes can result in losing track of the entity, this is a sign that this threat is also relevant for MSR studies that rely on entity traceability over several versions (as presented in Scenario 3). In this context, further investigation is performed in the next research question.

Summary. The ratio of untracked changes ranges from 10% to 21% for methods, and from 2% to 15% for classes. In practice, thus, the threat is more frequent at the method level.

7 RQ2: WHAT IS THE EXTENSION OF UNTRACKED CHANGES?

In this research question we assess the extension of untracked changes. We first measure the amount of entities (*i.e.,* classes and methods) that includes untracked changes in their histories. We further detail the analysis by assessing the paths of these entities.

Amount of entity histories with untracked changes. Entities with untracked changes in their histories are not desirable, because their histories may be split. To better understand this threat, Table 6 presents the number of entity histories (i) with only tracked changes and (ii) with tracked and untracked changes. The systems with proportionally more entity histories with untracked changes are Android Image (41%), MPAndroidChart (34%), and OkHttp (32%). The systems with less are Kotlin (18%), RxJava (19%), and Facebook Fresco (21%). When considering all systems, 25% of the entities have at least one untracked change in their histories. Thus, a relevant amount (a quarter) of entities potentially have their history split.

Table 6: Amount of entity histories with untracked changes.

Project	Entity Histories			
	Total	Tracked only	Tracked & Untracked	%
RxJava	21,419	17,265	4,154	19%
Elasticsearch	61,045	42,721	18,324	30%
Retrofit	2,306	1,697	609	26%
OkHttp	6,407	4,388	2,019	32%
Google Guava	12,739	9,493	3,246	25%
MPAndroidChart	1,796	1,194	602	34%
Glide	3,663	2,579	1,084	30%
Android Image	517	306	211	41%
Kotlin	77,441	63,342	14,099	18%
Spring	38,565	27,513	11,052	29%
Facebook Fresco	2,520	1,992	528	21%
Clojure	2,641	1,939	702	27%
Google Guice	5,265	4,140	1,125	21%
Apache Storm	6,920	5,087	1,833	26%
Eclipse Che	11,521	8,388	3,133	27%
All Projects	254,765	192,044	62,721	25%

In practice, this threat is more critical when an entity has more changes in its history, meaning that a long track can be lost. To better understand the most changed entity histories, Table 7 focuses on the top 25% most changed entities. In this case, the systems with more entities with untracked changes are Android Image (58%), Google Guice (56%), and OkHttp (48%). In contrast, the systems with less are Facebook Fresco (22%), Google Guava (24%), and Eclipse Che

(31%). Considering all systems, 37% of the top-25% most changed entities have at least one untracked change in their histories. We notice that the proportion of entity histories with untracked changes increase when compared to the previous analysis (overall, from 25% to 37%). This shows that the most changed entities (*i.e.,* entities that are constantly evolving, and, consequently, that are the most important and critical) are also likely to have their histories split.

Table 7: Amount of entity histories with untracked changes in the top 25% most changed entities.

Project	Entity Histories			
	Total	Tracked only	Tracked & Untracked	%
RxJava	5,355	3,457	1,898	35%
Elasticsearch	15,261	9,456	5,805	38%
Retrofit	577	383	194	34%
OkHttp	1,602	831	771	48%
Google Guava	3,185	2,427	758	24%
MPAndroidChart	449	298	151	34%
Glide	916	587	329	36%
Android Image	129	54	75	58%
Kotlin	19,360	12,158	7,202	37%
Spring	9,641	6,267	3,374	35%
Facebook Fresco	630	490	140	22%
Clojure	660	442	218	33%
Google Guice	2,023	895	1,128	56%
Apache Storm	1,730	1,128	602	35%
Eclipse Che	2,880	1,987	893	31%
All Projects	64,398	40,860	23,538	37%

Length of entity histories with untracked changes. To further assess the data previously presented, Figure 7 shows the path length distributions of the top-25% most changed entities. Each box plot presents the path lengths of the entity histories with only tracked changes (left) and with tracked and untracked changes (right).[16] Overall, entity histories with untracked changes have higher length. For example, the median path length for RxJava is 3 for entity histories with only tracked changes and 5 for entity histories with tracked and untracked changes, the third quartile is 5 against 9, and the upper whisker is 8 against 18. Thus, entities with untracked changes in their histories also tend to have a longer lifespan. If untracked changes are properly resolved, this long lifespan can help MSR studies focused on traceability analysis to be more precise.

Summary. The ratio of entities with untracked changes in their histories varies from 18% to 41%. For the most changed entities, this proportion is higher, between 22% and 58%. Overall, these entities also tend to have a long lifespan.

8 RQ3: WHAT IS THE IMPACT OF UNTRACKED CHANGES?

To further assess the practical impact of untracked changes in MSR studies, we evaluate how untracked changes affect two MSR approaches based on association rule mining.

The first approach focuses on API evolution rule mining [27, 45, 66], and infers rules in the format `Removed` ⟶ `Added`, indicating that references to class `Removed` are likely to be replaced by references to class `Added`. To compute the rules, this approach takes as input

[16]Outliers are not presented to improve visualization of quartiles.

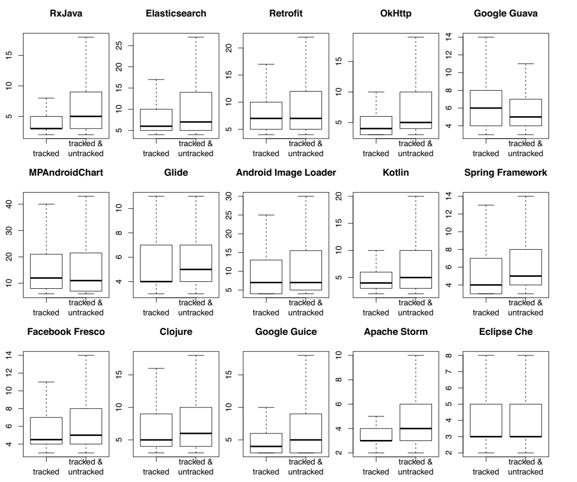


Figure 7: Length of entity histories in the top 25% most changed entities.

a set of transactions; each transaction represents a change between two versions of one method. The transaction elements are class references that were added or removed in the change. For example, by mining the two versions of method `Blog()` (Figure 2, Scenario 1), we detect that a reference to class `Vector` is removed and a reference to class `List` is added. Thus, a transaction with the elements `Removed-Vector` and `Added-List` is generated. Consequently, we may infer the rule `Removed-Vector` ⟶ `Added-List`.

The second approach targets API co-usage rule mining [43] to infer rules in the format `UseA` ⟶ `UseB`, indicating that class `UseA` is likely to be used with class `UseB`. This approach also takes as input a set of transactions representing method level changes. However, in this case, the transaction elements are only the added class references in the change. For example, by mining the two versions of method `getState()` in Apache Storm,[17] the co-usage rule `Map` ⟶ `HashMap` may be inferred, suggesting that `Map` is likely to be used with `HashMap`. To reduce the threat of noise generated by large changes [43, 46], in both MSR approaches, we only consider transactions with less than 5 elements.

For each approach, we assess (i) the amount of mined rules and (ii) the quality of mined rules in terms of recall and precision. These assessments are performed in four setups to cover distinct rule support count (1 and 3) and confidence (10% and 90%) [69]. We then compare these setups in two scenarios: when mining only tracked changes and when mining both (tracked and untracked) changes.

Assessing the amount of mined rules. In this analysis, we verify whether there is an improvement on the amount of mined rules when including untracked changes. To assess the amount of rules, we run the association rule mining approaches for all setups and case studies, and collect the inferred rules. For example, Table 8 shows the results for API co-usage approach in setup 1 (support=1, confidence=10%). In this case, the improvement on the amount of rules when including untracked changes ranges from 0.6% to 10.8%.

[17]Apache Storm change available at: https://goo.gl/tzhYiK

Table 8: Amount of co-usage rules for tracked and both (tracked and untracked) changes in setup 1 (support=1, confidence=10%). "Imp": improvement percentage.

Project	Tracked only	Tracked & Untracked
	#Rules	#Rules (Imp)
RxJava	993	999 (+0.6%)
Elasticsearch	1,375	1,490 (+8.4%)
Retrofit	404	435 (+7.7%)
OkHttp	524	548 (+4.6%)
Google Guava	1,179	1,233 (+4.6%)
MPAndroidChart	321	324 (+0.9%)
Glide	576	633 (+9.9%)
Android Image	172	185 (+7.6%)
Kotlin	333	337 (+1.2%)
Spring	1,054	1,104 (+4.7%)
Facebook Fresco	362	394 (+8.8%)
Clojure	396	399 (+0.8%)
Google Guice	1,204	1,274 (+5.8%)
Apache Storm	678	714 (+5.3%)
Eclipse Che	1,976	2,189 (+10.8%)

Figure 8 presents the improvement on the amount of rules for each setup and approach.[18] For the API co-usage approach, the median improvements are 5.3%, 1.7%, 4.3%, and 0%. The gain are clearly focused on setups 1, 2, and 3; in these cases, the third quartiles are 8%, 4.7%, and 6.6%. That is, 25% of the case studies in setup 1 can produce at least 8% more rules when resolving untracked changes. For the API evolution approach, the results are slightly better: the medians are 6.9%, 6.2%, 3.3%, and 0% while the third quartiles are 10.2%, 8.5%, 9.6%, and 1.9%. Interestingly, in some systems, setups 2 and 4 produce a reduction in the amount of mined rules (*e.g.*, first quartile is -4.7% for API evolution in setup 4). In these systems, the new data produced by mining untracked changes contribute to reduce the confidence of some rules that are mined considering only tracked changes. As a result, these rules no longer attend the higher confidence threshold (90%) of setups 2 and 4.[19]

Summary. The amount of mined rules usually improves when taking into account untracked changes. The median improvements range from 0% to 6.9% for API evolution, and from 0% to 5.3% for API co-usage.

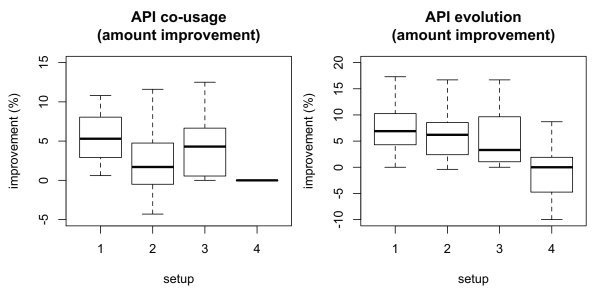


Figure 8: Improvement in amount of mined rules

[18]Outliers are not presented to improve visualization of quartiles.

[19]For example, suppose that in 9 out 10 cases the rule A $\longrightarrow$ B applies (reaching a confidence threshold of 90%). After considering untracked changes, we detect 8 new cases were A $\longrightarrow$ B applies and 2 cases where the rule does not apply. Overall, we will have a confidence of (9+8)/20 = 0.85, less than the 90% threshold.

Assessing the quality of mined rules. In this assessment, we verify whether there is an improvement on the quality of mined rules when taking into account untracked changes. To assess quality we perform a rule recommendation analysis to simulate a real usage scenario of the studied rule mining approaches. Following the same experimental design used to evaluate an approach to detect non-essential code changes [30], we iterate over all commits in ascending date order, and verify at commit *n* whether helpful rule recommendations are produced based on the data mined from commits *1* to *n-1*. Specifically, our analysis helps developers who have removed (or used) some class reference c_i at commit *n*, and who would like to find additional class references c_j that need to replace (or be co-used with) c_i. This recommendation analysis infers rules in the format $c_i \longrightarrow c_j$, from which we return a ranked list of class references c_j that were found to have been frequently replaced (or co-used with) c_i in previous commits. We rank recommendations c_j based on the confidence of the inferred rule $c_i \longrightarrow c_j$. Following the guidelines of Kawrykow and Robillard [30], we cap the number of recommendations at ten. Finally, to measure quality we assess recall and precision of the recommendations.

Table 9 shows the quality analysis for API co-usage approach in setup 1. For this setup, for example, the precision improvement ranges from -8.1% (Facebook Fresco) to 26.3% (Retrofit).

Table 9: Quality of co-usage rules for tracked only and both (tracked and untracked) changes in setup 1 (support=1, confidence=10%). "T Rec": total recommendation, "Cor": correct, "Prec": precision, "Imp": improvement percentage.

Project	Tracked only			Tracked & Untracked		
	T Rec	Cor	Prec	T Rec	Cor (Imp)	Prec (Imp)
RxJava	23,584	2,168	9.2	23,188	2,179 (+0.5)	9.4 (+2.2)
Elasticsearch	5,580	501	9	5,636	502 (+0.2)	8.9 (-0.8)
Retrofit	2,053	73	3.6	2,382	107 (+46.6)	4.5 (+26.3)
OkHttp	3,863	680	17.6	3,921	692 (+1.8)	17.6 (0.3)
Google Guava	11,052	1,541	13.9	10,994	1,624 (+5.4)	14.8 (+5.9)
MPAndroidChart	1,795	377	21	1,744	375 (-0.5)	21.5 (+2.4)
Glide	2,949	510	17.3	3,050	520 (+2.0)	17 (-1.4)
Android Image	430	38	8.8	407	36 (-5.3)	8.9 (+0.1)
Kotlin	342	20	5.8	343	20 (0)	5.7 (-0.3)
Spring	5,348	358	6.7	5,346	360 (+0.6)	6.8 (+0.6)
Facebook Fresco	1,161	111	9.6	1,263	111 (0)	8.8 (-8.1)
Clojure	2,448	267	10.9	2,435	267 (0)	11 (0.5)
Google Guice	7,344	829	11.3	7,239	813 (-1.9)	11.2 (-0.5)
Apache Storm	4,481	923	20.6	4,175	913 (-1.1)	21.9 (+6.2)
Eclipse Che	9,584	1,522	15.9	9,713	1,544 (+1.4)	16 (+0.1)

Figure 9 shows the quality improvement for each setup and approach. For the API co-usage approach, the median recall improvement is 0.2%, 0%, 1.6%, and 0%. The median precision improvement is 0.3%, 1.5%, 0.4%, and 0%. Finally, regarding the API evolution approach, the median recall improvement is 0% for all setups. For precision, the median improvement is -1.8%, 0%, -0.7%, and 0%.

While the median improvements are minor, notice that the spread of the boxplots is large: some systems saw a significant decrease in performance, while other saw a large increase instead. This indicates that the impact of untracked changes is difficult to predict, and needs to be evaluated in a case-by-case basis.

Summary. Overall, the quality of mined rules slightly improves when including untracked changes. However, some systems saw much larger improvements and the impact is difficult to predict.

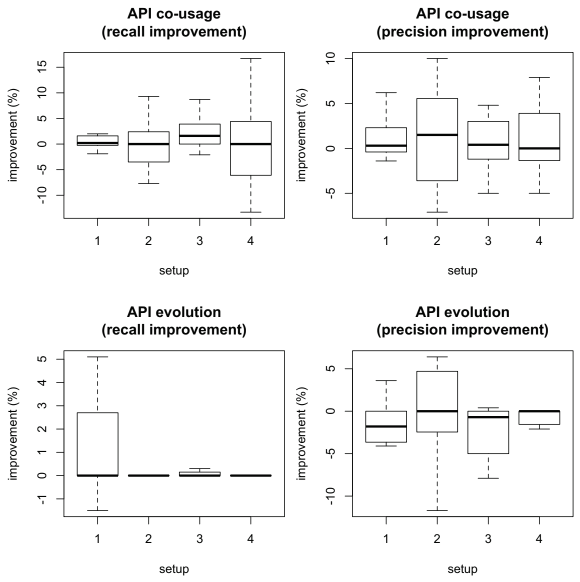


Figure 9: Improvement in quality of mined rules.

9 FINDINGS AND IMPLICATIONS

Untracked changes are more frequent for methods than classes. The ratio of untracked changes is not irrelevant; it ranges from 10% to 21% at the method level and from 2% to 15% at the class level. Despite happening at both levels, untracked changes are more frequent for methods. Thus, we suggest MSR studies to resolve untracked changes, especially when performing method level comparisons, to access neglected but potentially relevant new mining data.

Keeping track of untracked changes such as renaming is important, but not enough. The most frequent untracked changes are due to method renaming (26%), extraction (23%), and moving (22%). At class level, the most frequent are due to class moving (12%) and renaming (5%). Overall, renamings are responsible of around 1/3 of the untracked changes while other changes represent 2/3. Therefore, in addition to renaming, we recommend to address extraction and moving for a more complete resolution of untracked changes.

Untracked changes commonly cause splits in entity histories. We detect that a quarter of the studied entities potentially have their history split because they include untracked changes. For the most changed entities (*i.e.,* entities that are constantly evolving), this proportion is even higher (37%). As these entities tend to have a long lifespan, in practice, the issue is magnified. Therefore, we recommend untracked change resolution when performing traceability analysis, for more precise entity lifespans.

Resolving untracked changes can positively or negatively affect MSR approaches. We assessed two specific MSR approaches (API evolution and co-usage rule mining), and detected that their results can be improved when untracked changes are resolved. In our empirical study, more rules were produced with slightly better overall, but *very variable* quality. Thus, we suggest MSR researchers to adopt refactoring detection techniques to potentially improve their mining results on a case-by-case basis.

10 THREATS TO VALIDITY

Generalization of results. We focused our analysis on 15 popular, real-world, and publicly available (as they are hosted on GitHub) Java systems. This is twice than similar studies [25, 30]. Despite these observations, our findings—as usual in empirical software engineering—may not be directly generalized to other systems, especially to commercial and to systems implemented in other programming languages. Further, the variability observed in Figure 9 indicates that the effect of untracked changes is hard to predict, a further incentive for replication.

Detection of untracked changes. We may underestimate or overestimate the threat of untracked changes. We use RefDiff [58] to detect untracked changes: any false positive reported by RefDiff will make us overestimate the threat, while any false negative will make us underestimate it. We note however that RefDiff is a state of the art approach; its performance was found to exceed other refactoring detection approaches [58], with precision varying from 85.4% to 100% (meaning RefDiff reports few false positives), and recall ranging from 93.6% to 93.9% (meaning RefDiff reports few false negatives). Despite these observations, we took additional precautions, by evaluating the precision and recall of RefDiff in our dataset with two manual assessments (as reported in Section 4); in this case, we found an overall precision and recall of 89.1% and 89.8%, respectively. Therefore, to our knowledge, the risk of this threat is reduced.

Impact of the threat. The threat of untracked changes may not translate in actual issues to MSR-based studies. To address this, we investigated the threat impact in two specific approaches (API evolution and co-usage rule mining), as reported in research question 3. We found that the performance of these MSR approaches can be improved when untracked changes are resolved, but the improvements are very variable.

11 RELATED WORK

11.1 Assessing threats in empirical studies

Several studies have been performed to assess the quality of the data in software repositories, and whether issues found in the data may be problematic. A systematic review of empirical software engineering studies by Liebchen and Shepperd [41] found that (as of 2008) a very small portion of them explicitly mentioned data quality, indicating a low awareness of data quality issues.

The closest research to our study is the work by Kawrykow and Robillard [30]; they investigated the effect of *non-essential* changes in version histories. A *non-essential* change is a change that affects the source code of an entity without changing its behaviour. Example of non-essential changes are renaming a local variable in a method body, adding or removing the `this` keyword, or even white-space changes. Approaches that do change recommendations based on past co-change patterns (*e.g.,* class A and B change together 90% of the time) such as Zimmermann *et al.* [69], may make spurious recommendations if non-essential changes are taken into account. They define several categories of non-essential changes, and find that, in 7 software systems, up to 15.5% of changes to methods of a software system are non-essential, and that removing these changes increased the precision of a change recommendation approach by 10%, while it decreased its recall by 4%. Of note, the non-essential changes do not include the untracked changes. Quoting Kawrykow and Robillard: "*We consider the actual renaming of the code element to be an essential change, but argue that the textual reference updates induced by that renaming are non-essential*". Thus, one kind of change investigated by both papers is related (*i.e.,* renamings), but the set of changes is distinct.

Herzig and Zeller [25] investigated the impact of tangled code changes, that is, changes unrelated to each other that belong to the same commit. They found that up to 15% of bug fixes in 5 Java projects contained changes unrelated to the bug fix, based on a manual analysis. A subsequent automated analysis found that at least 16.6% of files that are associated with a bug report are actually not related to the report. A follow-up study by Herzig et al. [24] found that the impact of tangled changes on defect prediction regression models was significant. Dias et al. [15] proposed a technique to untangle fine-grained code changes.

Beyond source code changes, defect prediction approaches and datasets have been investigated for threats as well. Bird et al. [8] investigated whether the subset of bug reports correctly linked from commits to bug tracking systems is an accurate representation of the overall population of bugs, and found evidence of strong and systematic biases. In particular, for several systems, less severe bugs were more likely to be linked than more severe ones, raising concerns about the accuracy of bug prediction approaches. A follow-up study by Rahman et al. [52] found that the threat was somewhat alleviated when the amount of data available was large enough.

Posnett et al. [49] raised the issue of the ecological fallacy, which states that findings that are found at one level of analysis (*e.g.,* files of a software system), may not be valid at an aggregated level (*e.g.,* packages of that same system). They presented evidence of risks of ecological fallacy for defect prediction models. General threats to the performance of machine learning algorithms include class imbalance and a high number of features; Khoshgoftaar et al. [31] investigated their impact on software defect prediction. Lanza et al. [40] presented a series of reflections on how defect prediction techniques are evaluated. For example, they claim that current evaluations do not consider the effects in further bugs of developers accepting the recommendations produced by a defect predictor.

11.2 Detecting untracked changes

Several approaches attempt, directly or indirectly, to deal with untracked changes. Godfrey and Zou [20] introduced Origin Analysis, a technique that detects scenarios in which an entity (*e.g.,* a class, a function) is split in two, or two entities are merged in a single one. The goal is to allow for a more precise tracking of the lifetime of entities from version to version, that is, to detect a subset of untracked changes. However, the authors validated the approach on a single example (PostgreSQL), and do not attempt to quantify the extent of the threat in the same depth as we do. They found that their technique is able to reduce the number of "apparently deleted" entities by 30% and of "apparently added" entities by 19%.

Many approaches focus on detecting refactorings, which constitute the core of the untracked changes. As examples, we have RefDiff [58], Refactoring Miner [57, 64], Refactoring Crawler [16], and RefFinder [32]. In this work, we relied on RefDiff because it exceeded the precision and recall of the mentioned related tools in an evaluation reported by Silva et. al [58].

A related problem is the detection and the representation of fine-grained changes. Soetens et al. [60] performed a survey on the topic, which we refer to for space reasons. Examples include: detecting fine-grained changes by differencing program ASTs, such as ChangeDistiller [17], which has been used as a building block to detect non-essential changes; detecting *systematic changes* [33]; an alternative to recovering changes is recording them [55].

12 CONCLUSION

To the best of our knowledge, this study is the first to assess the threat of untracked changes, a threat often faced by MSR-based approaches. The empirical study was performed in the context of 15 real-world systems, and relied on RefDiff, the state of the art in refactoring detection, to find untracked changes in history versions. Three research questions were proposed to assess the frequency, extension, and impact of untracked changes. We reiterate the most interesting conclusions from our experimental results:

- *Untracked changes are more frequent for methods than classes.* The amount of untracked changes is not negligible, varying from 10% to 21% for methods, and from 2% to 15% for classes.
- *Keeping track of untracked changes such as renaming is not enough.* The most frequent untracked change types are method renaming (26%), extraction (23%), and moving (22%). Overall, renamings are responsible by 1/3 of the untracked changes while other changes represent 2/3.
- *Untracked changes commonly cause splits in entity histories.* A quarter of the studied entities potentially have their history split. In the top 25% most changed entities, the proportion raises to 37%.
- *Resolving untracked changes can positively or negatively affect MSR approaches.* By assessing two MSR approaches, we detect that their results can be improved when untracked changes are resolved; however, results are very variable.

As future work, we plan to extend this work to assess the impact of untracked changes in other MSR studies such as the ones described in Scenario 2 (warning prioritization) and in Scenario 3 (authorship detection). Moreover, we plan to further assess the branches and merges in the change graph (caused by method extraction and inlining) as a proxy of traceability complexity; in this case, we can measure nodes outdegree and indegree. Finally, we plan to increase the amount of case studies in our empirical analysis to better assess and characterize variability issues.

ACKNOWLEDGMENT

This research is supported by Fundect, CNPq, and FAPEMIG.

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *International Conference on Software Engineering*.

[2] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. 2012. Bug introducing changes: A case study with Android. In *Working Conference on Mining Software Repositories*.

[3] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *International Conference on Program Comprehension*.

[4] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE Software* 25, 5 (2008).

[5] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs fixit. In *International Symposium on Software Testing and Analysis*.

[6] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015).

[7] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2015. The impact of API change-and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering* 41, 4 (2015).

[8] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and balanced?: bias in bug-fix datasets. In *International Symposium on the Foundations of Software Engineering*.

[9] Cathal Boogerd and Leon Moonen. 2009. Evaluating the relation between coding standard violations and faults within and across software versions. In *Working Conference on Mining Software Repositories*.

[10] Oliver Burn. 2007. Checkstyle. (2007).

[11] Leon Moonen Cathal Boogerd. 2008. Assessing the value of coding standards: an empirical study. In *International Conference on Software Maintenance*.

[12] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. An empirical study of dormant bugs. In *Working Conference on Mining Software Repositories*.

[13] Tom Copeland. 2005. PMD applied. (2005).

[14] Cesar Couto, Joao Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. 2013. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal* 21, 2 (2013).

[15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *International Conference on Software Analysis, Evolution and Reengineering*.

[16] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *European Conference on Object-Oriented Programming*.

[17] Beat Fluri, Michael Wuersch, Martin Pixnzger, and Harald Gall. 2007. Change distilling: tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007).

[18] Thomas Fritz, Gail C Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. 2014. Degree-of-knowledge: modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology* 23, 2 (2014).

[19] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. 2010. A degree-of-knowledge model to capture source code familiarity. In *International Conference on Software Engineering*.

[20] Michael W Godfrey and Lijie Zou. 2005. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 31, 2 (2005).

[21] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. 2010. Visually supporting source code changes integration: the Torch dashboard. In *Working Conference on Reverse Engineering*.

[22] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. 2015. Visually characterizing source code changes. *Science of Computer Programming* 98 (2015).

[23] Lile Hattori and Michele Lanza. 2009. Mining the history of synchronous changes to refine code ownership. In *International Working Conference on Mining Software Repositories*.

[24] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering* 21, 2 (2016).

[25] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Working Conference on Mining Software Repositories*.

[26] Andre Hora, Nicolas Anquetil, Stephane Ducasse, and Simon Allier. 2012. Domain specific warnings: are they any better?. In *International Conference on Software Maintenance*.

[27] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *International Conference on Software Maintenance and Evolution*.

[28] Andre Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stephane Ducasse. 2017. How do Developers React to API Evolution? A Large-Scale Empirical Study. *Software Quality Journal* (2017), 1–33.

[29] Andre Hora and Marco Tulio Valente. 2015. apiwave: Keeping Track of API Popularity and Migration. In *International Conference on Software Maintenance and Evolution*.

[30] David Kawrykow and Martin P Robillard. 2011. Non-essential changes in version histories. In *International Conference on Software Engineering*.

[31] Taghi M Khoshgoftaar, Kehan Gao, and Naeem Seliya. 2010. Attribute selection and imbalanced data: problems in software defect prediction. In *International Conference on Tools with Artificial Intelligence*.

[32] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *International Symposium on the Foundations of Software Engineering*.

[33] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *International Conference on Software Engineering*.

[34] Sunghun Kim and Michael D Ernst. 2007. Prioritizing warning categories by analyzing software history. In *International Workshop on Mining Software Repositories*.

[35] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *International Symposium on the Foundations of Software Engineering*.

[36] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. 2006. Memories of bug fixes. In *International Symposium on the Foundations of Software Engineering*.

[37] Sunghun Kim and E James Whitehead Jr. 2006. How long did it take to fix bugs?. In *International Workshop on Mining Software Repositories*.

[38] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008).

[39] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic identification of bug-introducing changes. In *International Conference on Automated Software Engineering*.

[40] Michele Lanza, Andrea Mocci, and Luca Ponzanelli. 2016. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software* 33, 6 (2016).

[41] Gernot A Liebchen and Martin Shepperd. 2008. Data sets and data quality in software engineering. In *International Workshop on Predictor Models in Software Engineering*.

[42] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *International Symposium on the Foundations of Software Engineering*.

[43] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *International Symposium on the Foundations of Software Engineering*.

[44] Andrew Meneely and Oluyinka Williams. 2012. Interactive churn metrics: socio-technical variants of code churn. *Software Engineering Notes* 37, 6 (2012).

[45] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. 2012. A history-based matching approach to identification of framework evolution. In *International Conference on Software Engineering*.

[46] Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller. 2011. Mining evolution of object usage. In *European Conference on Object-Oriented Programming*.

[47] Shawn Minto and Gail C Murphy. 2007. Recommending emergent teams. In *International Workshop on Mining Software Repositories*.

[48] Ralph Peters and Andy Zaidman. 2012. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and Reengineering*.

[49] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2011. Ecological inference in empirical software engineering. In *Automated Software Engineering*.

[50] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *International Conference on Software Maintenance*.

[51] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *International Conference on Software Engineering*.

[52] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample size vs. bias in defect prediction. In *International Symposium on the Foundations of Software Engineering*.

[53] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for inspections: hit or miss?. In *International Symposium on the Foundations of Software Engineering*.

[54] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *International Conference on Software Engineering*.

[55] Romain Robbes. 2008. *Of change and software*. Ph.D. Dissertation. Università della Svizzera italiana.

[56] David Schuler and Thomas Zimmermann. 2008. Mining usage expertise from version archives. In *International Working Conference on Mining Software Repositories*.

[57] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *International Symposium on the Foundations of Software Engineering*.

[58] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: detecting refactorings in version histories. In *International Conference on Mining Software Repositories*.

[59] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making program refactoring safer. *IEEE Software* 27, 4 (2010).

[60] Quinten David Soetens, Romain Robbes, and Serge Demeyer. 2017. Changes as first-class citizens: a research perspective on modern software tooling. *ACM Computing Surveys* 50, 2 (2017).

[61] Diomidis Spinellis. 2015. A repository with 44 years of Unix evolution. In *Working Conference on Mining Software Repositories*.

[62] Diomidis Spinellis. 2017. A repository of Unix history and evolution. *Empirical Software Engineering* (2017).

[63] Mario F Triola. 2006. *Elementary statistics*. Pearson/Addison-Wesley.

[64] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Conference of the Centre for Advanced Studies on Collaborative Research*. 132–146.

[65] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23, 3 (2016).

[66] Wei Wu, Y.-G. Gueheneuc, G. Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering*.

[67] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *International Symposium on the Foundations of Software Engineering*.

[68] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead, Jr. 2006. Mining version archives for co-changed lines. In *International Workshop on Mining Software Repositories*.

[69] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005).