Commits as a Basis for API Misuse Detection

Sebastian Nielebock
OvGU Magdeburg
Germany
sebastian.nielebock@ovgu.de

Robert Heumüller
OvGU Magdeburg
Germany
robert.heumueller@ovgu.de

Frank Ortmeier
OvGU Magdeburg
Germany
frank.ortmeier@ovgu.de

ABSTRACT

Programmers frequently make use of APIs. However, these usages can result in unintended, negative behavior, when developers are not aware of the correct usage or side effects of that API. Detecting those *API misuses* by means of automatic testing is challenging, as many test suites do not cover this unintended behavior.

Instead, API usage patterns are used as specifications to verify the correctness of applications. However, to find meaningful patterns, i.e., those capable of fixing the misuse, the context of the misuse must be considered. Since the developer usually does not know which API is misused, a much larger code section has to be verified against many potential patterns.

In this paper, we present a new idea to enhance API misuse detection by means of commits. We discuss the potential of using commits (1) to decrease the size of the code to be considered, (2) to identify suspicious commits, and (3) to contain API usages which can be used to shepherd API specification mining.

This paper shows first results on the usability of commits for API misuse detection and some insights into what makes a commit suspicious in terms of exhibiting potential API misuses.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; **Software libraries and repositories**;

KEYWORDS

Application programming interfaces, Misuses, Commits

ACM Reference Format:
Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. 2018. Commits as a Basis for API Misuse Detection. In *Proceedings of the 7th International Workshop on Software Mining (SoftwareMining '18), September 3, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3242887.3242890

1 MOTIVATION

By providing Application Programming Interfaces (API) developers of libraries offer other programmers a means of communicating with their implementations. However, as with any kind of communication, misunderstandings may arise when either party is not fluent with the language. Consequently, developers can use an API in a way that was not intended by the API creators and that eventually yields to a negative behavior. We call this an *API misuse*. Detecting API misuses is a crucial and non-trivial task due to the vast possibilities for misuses and negative implications. Usually, automatic test suites do not cover these misuses since programmers are not aware of the misuse. Recent work finds API misuses by verifying API specifications - so-called API usage patterns - against the code. These usage patterns describe frequently recurring applications of API events, e.g., API method sequences [14, 15] or API idioms [1]. The main idea is that frequently recurring patterns are likely to indicate correct usages. However, it is hard to obtain valid API specifications either manually or automatically [7]. In addition, to fix an API misuse it is hard to find a pattern that is exactly tailored to the particular context of the misuse. Particularly, one typically needs to know which API was misused in order to find helpful usage patterns. But this knowledge is usually not available since developers hardly know the particular API misuse beforehand.

We think that these challenges in finding API misuses can be tackled by utilizing commits from version control systems. Commits describe the differences between two code revisions as well as their related metadata. First, we can restrict the misuse detection to those parts that were actually changed or were affected by a change. Second, we can only consider commits that actually used an API event, e.g., adding, changing or removing an API method call. Third, based on the used API events and their context, one can guide the search of API usage patterns, which are related to the actual API usage. Fourth, based on the meta-data of commits one reuse approaches to estimate the likelihood that a misuse or more generally a bug is introduced [3, 6, 13]. Fifth, detecting misuses on commit level helps to detect them as early as possible, which is often referred to as "Just-In-Time Quality Assurance" [3, 13]

These advantages motivate us to consider commits as a basis for API misuse detection. For that purpose, we introduce our notion of how commit analysis and API misuse detection could be combined (Section 2). In Section 3, we present some initial insights into shared properties between API misuse introducing commits, which could be leveraged for enhanced API misuse detection. Finally, we outline some upcoming research questions and challenges (Section 4).

2 API MISUSE DETECTION BY MEANS OF COMMITS

First, we introduce our notion of how API misuses detection and commit analysis can be combined which is presented in Figure 1.

We assume a developer who recently changed an existing code base Ⓐ by means of committing her changes to a code repository. These changes not only include the respective code differences but also meta information, e.g., the committer's name, the commit time, or the commit message. Based on this information, we apply an API-sensitive change analysis Ⓑ to decide whether the current

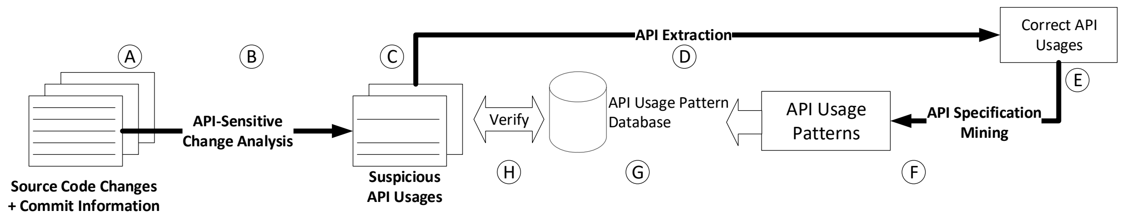

Figure 1: Concept of API Misuse Detection on the Basis of Commits

changeset contains an API usage and to reveal suspicious commits and code sections (C). The analysis encompasses static code analysis as well as the analysis of the commit's meta information, e.g., natural language processing of commit messages. In particular, this analysis can discriminate commits which actually do not use API events, e.g., method calls, for further investigation. This is reasonable since we can assume that a commit whose changes do not relate to API events is very unlikely of introducing an API misuse.

The found suspicious API usages can be utilized to extract the APIs and their respective context (D). A very simple approach is to extract keywords from the code, e.g., the import statements and called method names related to the used APIs, with static analysis and to apply code search to find similar code samples that also contain these or fraction of the same keywords. We assume that these similar samples represent likely correct API usages in a similar context (E). Here, different code search mechanisms can be applied, e.g., textual search [4, 12], search with domain-specific languages [8], or search with automatically generated queries [5, 10]. So far, we restrict it to simple textual search since it is a lightweight approach.

In order to condense the correct usages of the APIs, we apply API specification mining (F). Specification mining [9] infers temporal specifications from code samples, e.g., API method call sequences, describing a correct application of APIs. Since those specifications are rarely documented, these approaches 'guess' them by assuming that frequent application of API events, e.g., method call sequences, represent correct specifications. We plan to use static trace generation of method calls and to mine frequently occurring subsequences as specifications. Finally, we insert these API usage patterns into a database (G) and check them against our suspicious API usage (H). Verification is usually done by conducting static or dynamic analysis as well as symbolic execution [9]. We plan to apply symbolic execution of the analyzed code to check whether it shows the same behavior is indicated by the obtained specifications.

Note that we suggest to use a database since specification mining can be a computationally-intensive step, and thus storing previously inferred patterns can speed up the verification step.

Next, we discuss the step (B) of API-sensitive change analysis, particularly, how one can detect API usages (Section 2.1) and how one can find suspicious commits (Section 2.2).

2.1 Detection of API Usages

Previous work [15] statically analyzed the code to recognize API usages for API specification mining. In particular, they considered the call of a super constructor, casting to a type, method calls on a class and usage of an instance of a class referred to third-party libraries or frameworks. Instead, we want to use this information to detect whether a developer made an API specific change in the current commit. By statically tracing code statements to their originating classes, we can obtain this information. However, we assume that patterns can be inferred from other, correct usages of that APIs. Thus, we have to ensure that the originating class of that API is from a third-party library or an external framework since it is naturally unlikely to find references to project-internal APIs.

One simple approach to do that is to compare the package name of the changed class and the originating package name of the class of the edited API statement, e.g., a method call. Assuming the package naming convention by Java[1], we can expect the first part of a package name is related to the domain of the respective company or organization, followed by the project name and its further package structure, e.g., `de.ovgu.cse.project.algorithm`. Thus, we can assume that if the prefix of the full package name of an API statement is similar or equal to the respective package name of the class, this is not an external or third-party API usage. For instance, the package `de.ovgu.cse.project.algorithm` is more likely to originate from the same project of the package `de.ovgu.cse.project.util` than from `org.eth.cse.project.util`. One can measure similarity by just considering the prefix of package names to decide whether the API is internal or external.

Note that, packages from the Java standard library (prefix `java.*`) could also be excluded since it is very likely that most commits contain such a usage. Without the discrimination of the Java library, we would struggle to filter out commits. However, Java libraries also contain API usage patterns, which are misused.

2.2 Estimate the Suspiciousness of Commits

An additional feature of commit-based misuse detection is to concentrate on changes that are most likely to contain a misuse. Mechanisms to detect these commits utilize metadata of them to build a risk model and thus can be applied not only for API misuses but for bugs in general. Note that we only give an overview of existing approaches, since we do not have a final answer whether the assessment of the suspiciousness of commits with API misuses will significantly differ from commits with 'general' bugs. However, the main benefit we gain from commits is the reduction of code that has to be analyzed, and thus the assessment of the suspiciousness can be seen as an additional lever to prioritize misuse analysis.

[1] https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

In a study of six open source and five commercial projects, Kamei et. al [6] developed a risk model based on a logistic regression model and several different change factors. Their findings show that a higher number of changed files, and therefore a higher number of edited source code lines, as well as the fact that a commit fixed an error, are risk increasing. Instead, a lower average time interval between changes of a file was found to be a risk-decreasing factor. All in all, their model achieved an accuracy of 68% and a 64% recall.

In a study of the Mozilla project [3], the authors found out that first, 25.5% of all commits appear to be bug-introducing. Second, these commits are mainly conducted by less experienced developers, consists of larger changes than commits that do not introduce bugs, and often fix previous bugs. Their model could predict bug-introducing commits with a precision of 61.4% and a recall of 95%.

A recent study [13] analyzed commits from the NetBeans project. Here, they developed the tool ChangeLocator, which trains a classifier to find crash-inducing changes based on historical crash reports. By means of new crash reports, they were able to predict with an average recall@10 of 74.5% a list of commits related to the crash.

These studies show that commits themselves can be leveraged to classify their suspiciousness, which we plan to use for ongoing work. In particular, we plan to investigate whether API misuses have special characteristics in their commits compared to 'ordinary' bugs. Some first insights will be given in the following section.

3 EVALUATION

We conducted a preliminary experiment on commits that introduced API misuses. First, we analyzed whether an API misuse is more likely to occur when an API is first introduced in a project, i.e., the import statement was added to the respective source file. Second, we are interested in the size of the changesets of misuse introducing commits since the smaller the size the easier we can locate the misuse and its surrounding context.

3.1 Acquisition of Experimental Data

Our evaluation uses the MUBench benchmark [2] comprising several API misuses and their respective fixing commits. For simplicity, in our evaluation, we only considered misuses originating from projects using the *git* version control system (VCS) and which are publicly available. This qualifies 48 misuses from 19 different projects[2]. Then, we extracted the respective misuse-introducing commit. A misuse-introducing commit describes this commit that added the code site which first revealed the API misuse. By means of the VCS and the knowledge of which commit eventually fixed the misuse, we manually identified these misuse-introducing commits. Note that we also took situations into account which moved or renamed files and consequently the buggy code site. In future work, we plan to automate this step by means of the SZZ algorithm [11]. In 11 cases, we were not able to find the misuse-introducing commits since the misuses are already fixed since the initial commit of the repository or we were not able to find the misuse at all. In such cases, the repository was often migrated, for instance, from an svn to a git, and therefore the respective misuse-introducing commit is no longer present in the considered git repository. Consequently, we investigated the remaining 37 misuses from 18 different projects.

[2]accessed benchmark and projects on March 14, 2018

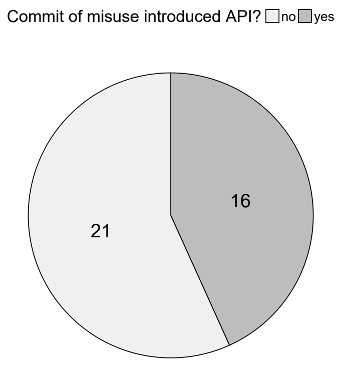

Figure 2: Number of misuses whose commits introduced/not introduced the API

3.2 Analysis of Preliminary Questions

Do misuses mainly occur when APIs are applied for the first time? We assumed that misuses are more likely when a developer adds an API to a project for the first time. However, as Figure 2 depicts, the proportion of misuses that initially introduced the API, i.e., added the respective import statement, is in the (narrow) minority with 16 cases. We found that approximately one-half of all misuses were introduced after the respective API was already present.

Additionally, we observed that in 14 of those 16 cases the entire source file was added at once. This means that we can only be sure of two cases where an API was misused upon being added to an existing file for the first time. We also observed that six of these 14 misuses were actually introduced in the initial commit of the project. Moreover, we noticed that six commits actually introduced multiple misuses (two to four misuses per commit). These results can support API misuse detection, e.g., deeply analyzing initial commits or further inspecting commits that already contain a misuse.

What is the typical size of API misuse introducing commits? Small code changes help to narrow down the search space for finding potential API misuses as well as similar and related API usages from which valid specifications can be inferred. For our analysis, we only considered the 31 non-initial commits. The rationale is that initial commits tend to add a huge set of source code files and lines, which would otherwise bias our results. For instance, the initial commit of the Closure project from our case study added 418 files and 117,414 lines with insertions. As mentioned before, we suggest a separate and deep API misuse detection for initial commits.

We considered the size of non-initial commits of that misuses on the file level (cmp. Figure 3) as well as on the code line level (cmp. Figure 4). In these violin plots, we observe that, except for a few outliers, there are more changed than added files. Note that we did not observe any file deletions in considered commits and therefore none are depicted. Typically, the median number of added files is one, while the median of changed files is three. We also analyzed the differences in lines. For that purpose, we applied the `git diff` command with the appropriate options for ignoring changes in whitespace and linebreaks on *Java* files[3]. By these means, we were able to count the number of lines containing insertions, (`insert`),

[3]`git diff --word-diff-regex=[^[:space:]] HEAD~1 HEAD -- '*.java'`

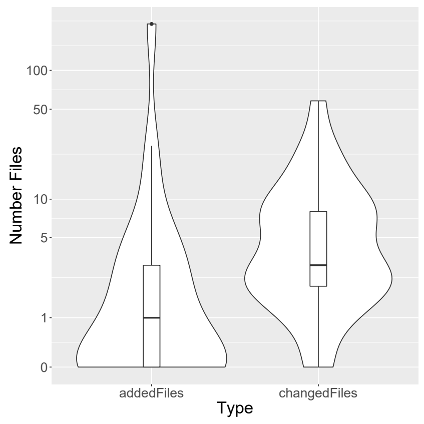

Figure 3: Added and changed files per misuse in their related non-initial, misuse-introducing commits (logarithmic scale for y-axis)

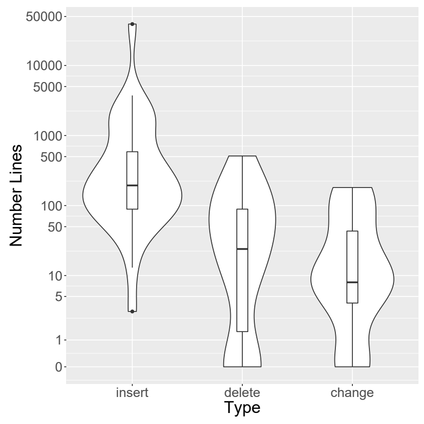

Figure 4: Inserted, deleted, and changed lines per misuse in their related non-initial, misuse-introducing commits (logarithmic scale for y-axis)

deletions (`delete`), and changes (`change`) of code elements. In Figure 4 we observe that commits with misuses usually have more lines with insertions than deletions or changes. More precisely, the median value of 194 lines contains at least one insertion, while the median values of lines containing a deletion or a change are 24 and 8, respectively.

Thus, we can conclude that in our observed data, non-initial commits with API misuses usually change existing files, while most changes appear to be insertions. All in all, our results give first insights into the characteristics of API misuse introducing commits, e.g., that many misuses are introduced after the respective API was already used in a previous version. However, our experimental corpus is rather small to derive general statements. Moreover, we did not compare these characteristics to commits that do not introduce an API misuse. This will be part of a bigger study in future work which is discussed in the subsequent section.

4 CONCLUSION AND CHALLENGES

This paper presented how API misuse detection can be enhanced by the analysis of commits. In particular, we introduced a concept by which the detection is applied at the moment when a developer commits changes. It comprises an analysis of the code changes in a commit as well as its related meta information. Based on this information, we can detect API usages as well as suspicious commits. In a preliminary analysis, we revealed that (1) API misuses not necessarily occur at the time when an API is first used in a project and (2) that typically API misuse introducing commits feature many line insertions in existing files. However, our results are based on a small experimental corpus of 37 misuses. Thus, we plan a much larger study in future work for which we will automate the search of misuse introducing commits, e.g., with the SZZ algorithm [11].

Moreover, we will compare whether the characteristics of commits introducing bugs are also true for API misuses. Here, we are also interested in specific characteristics of commits with API misuses, e.g., size of changed API statements. Additionally, we will investigate whether our concept by previously performing an API-specific search can reduce the number of false positive or non-related patterns. Finally, assuming a continuously growing pattern database, we need effective methods to easily query and retrieve related API usage patterns according to the actual suspicious usage.

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *22nd FSE*. ACM, New York, NY, USA, 472–483. https://doi.org/10.1145/2635868.2635901
[2] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *13th MSR*. https://doi.org/10.1145/2901739.2903506
[3] Le An and Foutse Khomh. 2015. An Empirical Study of Crash-inducing Commits in Mozilla Firefox. In *11th PROMISE*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/2810146.2810152
[4] Rosalva E Gallardo-Valencia and Susan Elliott Sim. 2009. Internet-Scale Code Search. In *1st SUITE*. IEEE, 49–52.
[5] Reid Holmes, Robert J Walker, and Gail C Murphy. 2006. Approximate Structural Context matching: An Approach to Recommend Relevant Examples. *TSE* 32, 12 (2006), 952–970.
[6] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *TSE* 39, 6 (June 2013), 757–773. https://doi.org/10.1109/TSE.2012.70
[7] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. 2016. How Good are the Specs? A Study of the Bug-Finding Effectiveness of existing Java API Specifications. In *31st ASE*. 602–613.
[8] Santanu Paul and Atul Prakash. 1994. A Framework for Source Code Search using Program Patterns. *TSE* 20, 6 (1994), 463–475.
[9] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *TSE* 39, 5 (2013), 613–637.
[10] Naiyana Sahavechaphan and Kajal Claypool. 2006. XSnippet: Mining For Sample Code. In *21st OOPSLA*. ACM, New York, NY, USA, 413–430. https://doi.org/10.1145/1167473.1167508
[11] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes?. In *2nd MSR*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083147
[12] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *22nd ASE*. ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/1321631.1321663
[13] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2017. ChangeLocator: Locate Crash-Inducing Changes Based on Crash Reports. *EMSE* (11 Nov 2017). https://doi.org/10.1007/s10664-017-9567-4
[14] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *28th ICSE*. ACM, 282–291.
[15] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. *23rd ECOOP* (2009), 318–343.