2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

MUBench: A Benchmark for API-Misuse Detectors

Sven Amann[†] Sarah Nadi[†] Hoan A. Nguyen[‡] Tien N. Nguyen[‡] Mira Mezini[†§]

Technische Universität Darmstadt[†] Iowa State University[‡] Lancaster University[§]

{amann, nadi, mezini}@cs.tu-darmstadt.de, {hoan, tien}@iastate.edu

ABSTRACT

Over the last few years, researchers proposed a multitude of automated bug-detection approaches that mine a class of bugs that we call *API misuses*. Evaluations on a variety of software products show both the omnipresence of such misuses and the ability of the approaches to detect them.

This work presents MuBench, a dataset of 89 API misuses that we collected from 33 real-world projects and a survey. With the dataset we empirically analyze the prevalence of API misuses compared to other types of bugs, finding that they are rare, but almost always cause crashes. Furthermore, we discuss how to use it to benchmark and compare API-misuse detectors.

CCS Concepts

•**Software and its engineering** → **Software defect analysis;** *Software post-development issues;*

Keywords

API-Misuse Detection; Bug Detection; Benchmark

1. INTRODUCTION

Over the last 15 years, researchers proposed a multitude of automated bug-detection approaches [13]. These approaches commonly mine API usage patterns from source code and find rare violations of those, assuming that they often correspond to bugs. We call such violations *API misuses*. For example, one API misuse is when a developer forgets to close a resource. Evaluations on a variety of software projects show both the omnipresence of API misuses and the ability of the detectors to find them [9, 14].

However, to the best of our knowledge, no work empirically analyzes the prevalence of API misuses compared to other types of bugs or shows which kinds of misuses a particular technique detects. This makes it hard to judge the impact of the detectors in general, to assess their capabilities, and to compare them with one another. As a first step towards these goals, we present MuBench, a dataset of API misuses that can be used to benchmark and compare API-misuse detectors. We explored existing bug datasets, mined projects from SourceForge and GitHub, and conducted a survey to collect 89 instances of API misuses. From this sample, we created a taxonomy of API misuses and a dataset with detailed metadata about each instance. We find that 61 of the misuses (69.5%) cause crashes, which stresses the importance of mitigating this kind of bug.

MuBench is publicly available on GitHub.[1] We chose this platform to enable researchers to contribute to the benchmark and to uniquely reference the particular benchmark version they use in their work. We encourage researchers to use and extend MuBench to achieve replicability of evaluations and comparability of API-misuse detectors.

Source	Total Size	Reviewed	Misuse	Crash
BugClassify	2,914	294	26	16
Defects4J	357	357	14	12
iBugs	390	390	56	?
QACrashFix	24	24	15	15
SourceForge	130	130	13	6
GitHub	2,660	78	3	2
Survey	17	17	12	5
Total	6,491	1,189	89	61

Table 1: API Misuses by Source

2. FINDING API MISUSES

An *API misuse* is an API usage that violates the API's contract, as opposed to one that does not comply with the client code's logic. For example, not closing a stream is a misuse, while querying the wrong database column is not. Based on this intuition, we used our best judgement to identify API misuses, following three independent approaches: (1) We analyzed existing bug datasets, filtering for API misuses (Section 2.1), (2) we analyzed SourceForge and GitHub for misuses of the Java Cryptography Extension (JCE) APIs (Section 2.2), and (3) we conducted a survey, asking developers for problems caused by misuse of Java APIs (Section 2.3). Table 1 summarizes the results.

2.1 Existing Bug Datasets

Many researchers created bug datasets to evaluate their approaches for problems such as defect prediction or patch generation. These datasets encompass bugs reported in the

[1] https://github.com/stg-tud/MUBench, last checked Feb 19, 2016

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2903506

respective project's issue tracker or fixed in its version-control system. Since API misuses are a subset of general software bugs, we manually reviewed three such datasets to identify instances of misuses. During this review, we assessed the prevalence of misuse-related bugs compared to other bugs.

BugClassify. This dataset by Herzig et al. [6] consists of 7,401 tickets from the issue trackers of five Open Source projects. They manually classified 2,914 of these tickets as reporting bugs. We randomly selected 10% of those tickets for each of the five projects, a total of 294 tickets, from which we identified 26 API misuses (8.8%). We found that most tickets report logic errors, such as wrong calculations or missing handling of certain cases. Other categories are mistakes in configuration files and multi-threading issues.

Defects4J. This defect dataset by Just et al. [7] consists of 357 source-code bugs that were fixed in a single commit, reported in an issue tracker, and had at least one accompanying testcase that failed before and passed after the fix. From all of these cases, we identified 14 API misuses (3.9%).

iBugs. This dataset by Dallmeier and Zimmerman [2] consists of 390 fixing commits from three Open Source projects. The commits were selected through heuristics on the commit messages. From all of these cases, we identified 56 API misuses (15.1%).[2] Many of the other issues were unrelated to API usage and often even unrelated to source code.

QACrashFix. This dataset by Gao et al. [4] consists of 24 source-code bugs from 16 GitHub projects. The bugs were selected by, first, mining the issue trackers of the projects for crash reports related to the Android API and, second, reviewing the resulting candidates manually. From all of these cases, we identified 15 API misuses (62.5%). Interestingly, a very large part of these crash bugs are API misuses.

2.2 Java Cryptography Extension APIs

Previous work shows that developers often misuse cryptographic APIs [3,5,8] and that they would welcome help using these APIs correctly [10]. To add examples of such critical misuses to MuBench, we mined bug-fixing changes from projects on SourceForge and GitHub as follows: (1) We identified projects with at least 10 stars that use the JCE API, i.e., whose latest source code contains imports from the `javax.crypto` package. (2) For each fix, we extracted the actual source-code change [12] and analyzed the changes to the abstract syntax tree [11] to find changes to the usages of a JCE type. (3) We manually reviewed the candidates.

Using this approach, we extracted 130 candidates from SourceForge, from which we identified 15 misuses, and 2660 candidates from GitHub, from which we reviewed a random sample of 78 and identified 3 misuses, so far.

2.3 Survey

All bugs we reviewed so far were committed to version-control systems. Since we find that many API misuses lead to obviously spurious behavior (such as exceptions), we hypothesize that many are already ruled out during development, e.g., through testing. Thus, developers might face many misuses that we cannot find by reviewing bug datasets.

Following our hypothesis, we conducted a survey,[3] which we promoted via colleagues, friends, and Twitter, to reach developers. Within 9 days, we collected 16 responses naming 17 distinct API misuses. We identified only 3 of these misuses in our review, which supports our hypothesis.

Interestingly, many participants pointed out multiple fixes for misuses. For example, to ensure a resource gets closed, we can call `close()`, or call `IOUtils.closeQuietly()` from Apache Commons, or use the try-with-resources statement. This suggests that while there may be multiple ways to misuse APIs, there are also multiple ways to use it correctly.

We added all examples to our dataset. For each, we provide a Java file with examples of the misuse and alternative fixes. We publish the survey responses as part of MuBench.

```yaml
source:
  name: BugClassify
  url: https://www.st.cs.uni-saarland.de/softevo//bugclassify/
project:
  name: Mozilla Rhino
  url: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
report: https://bugzilla.mozilla.org/show_bug.cgi?id=286251
description: >
  IRFactory.initFunction() is called twice along one possible execution path,
  which causes an infinite loop.
crash: yes
internal: yes
api:
  - org.mozilla.javascript.IRFactory
characteristics:
  - superfluous call
pattern:
  - single object
challenges:
  - path dependent
fix:
  description: >
    Remove duplicated call.
  commit: https://github.com/mozilla/rhino/commit/ed00a2e83de1e768918604a65def097...
  files:
    - name: src/org/mozilla/javascript/Parser.java
```

Figure 1: Meta Data of API Misuse rhino-286251

2.4 Results

In total, we reviewed 1,189 candidates from 7 sources. We identified 89 API misuses from 33 projects and the survey. From these, 61 (69.3%) cause crashes. We added the misuses from all sources to MuBench.[4]

If we consider BugClassify, Defects4J, and QACrashFix, for which we know the exact number of bugs they contain, we find that only 6.6% of all bugs are API misuses. However, many of those misuses cause crashes (95.5%), which stresses the importance of mitigating this kind of bug.

We found that many of the fixing commits resolve multiple misuses. Also many commits contain more changes than the fix itself, such as refactorings or reformatting, which makes isolating the actual misuse more difficult.

3. THE API MISUSE DATASET

For each of the misuses we identified in our data collection, we provide a file with meta data in YAML format.[5] Figure 1 shows an example of such a file for the misuse rhino-286251.

The `source` designates where we found the misuse example. The `name` either identifies one of the existing datasets we analyzed (Section 2.1), our survey (Section 2.3), or the SourceForge or GitHub project (Section 2.2). We provide a `url` to the original bug dataset, if applicable.

The `project` designates where the misuse occurred. We provide the project's `name` and `url`. For misuses from the survey, no project information is provided.

The `report` designates a ticket—usually in the project's issue tracker—reporting the misuse. From this report, the

[2]Due to problems identifying the meta data from the old dataset, we did not add these misuses to MuBench yet.
[3]http://goo.gl/forms/3hua7LOFVJ, last checked on Feb 15, 2016
[4]We still review the findings from GitHub. They will be added soon.
[5]http://yaml.org/, last checked on Feb 10, 2016

fix, and respective API documentation, we assemble a `description` of the misuse. Furthermore, we document whether the misuse caused the code to `crash` and whether the misused API is `internal` to the project (as opposed to from an external library). Finally, we list all types whose `api` is part of either the misuse or the respective correct usage.

To classify the misuse, we provide its `characteristics`. We drafted a respective taxonomy from the changes required to turn the misuse into a correct usage. Each misuse has one or multiple of the following characteristics:

`superfluous call` - the misuse contains a superfluous call.

`missing call` - the misuse misses a call.

`wrong call` - the misuse calls the wrong method. We distinguish this from both a superfluous and a missing call, if the present call generally shows the right intent. For example, when `File.mkdir()` is called instead of `File.mkdirs()`.

`missing precondition` - (part of) the misuse needs a guard. We differentiate three cases: (1) a `predicate` check on an object, e.g., `isEmpty()`, or (2) a `null` check on a reference, before calling a method on it or passing it to a method that expects a non-`null` parameter, or (3) a `value constraint` check, e.g., that the number passed to `PreparedStatement.setFetchSize()` must be smaller or equal to the number passed to `PreparedStatement.setMaxRows()`, if the number passed to `setMaxRows()` is larger than 0.

`missing catch` - the misuse does not handle a particular exception or any exception at all.

`missing finally` - (part of) the misuse should happen in a `finally` block, regardless of whether there already is such a block or even a `try` statement.

`ignored result` - the misuse ignores the return value of a call, e.g., the result of a pure method.

Since API-misuse detectors mine usage `patterns` and identify violations of those, they are limited by the usages they can extract. We currently distinguish the following kinds:

`single node` - the misuse consists of a single call, e.g., if `Files.write()` is called without specifying what to do if the file does not exist (optional parameter).

`single object` - the usage involves only a single object with multiple calls, e.g., how to open, use, and close a stream.

`multiple object` - the usage involves multiple objects, e.g., how to use an `Iterator` created from a `Collection`.

A survey by Robillard et al. [13] presents more fine-grained characteristics of the patterns approaches handle. We plan to build a taxonomy from these and add it to our benchmark.

During our reviews, we found special challenges for misuse detection that are orthogonal to the above characteristics:

`multi-method` - the misuse spreads over multiple methods, e.g., a stream is closed by a callee and later accessed by the caller. Detectors need to consider called methods.

`multiple usages` - the misuse interleaves with other usages of the same type, e.g., two buttons are configured side by side. Detectors need to separate the distinct usages.

`path dependent` - the misuse occurs only along a particular execution path, e.g., a required call happens in the `then` branch, but not in the `else` branch. Detectors need to decide whether there is such a path or not.

Finally, we describe the `fix` for the misuse. As for the misuse itself, we provide a short `description` that we compiled from available patches and API documentation and a URL to the fixing `commit`, if we could find it from the report. Since the commits sometimes change files unrelated to the misuse, we also list the related `files`, including, if possible, a direct link to the `diff` of the fix in that file.

4. USING THE BENCHMARK

With the current dataset, we assess the prevalence of API misuses compared to other bugs, as discussed in Section 2.4. We publish the scripts that generate the reported statistics and more, such as the distribution of different kinds of misuses and the frequency of detection challenges.

Our main motivation to create MuBench is the evaluation of API-misuse detectors. In the past, researchers evaluated their approaches on a collection of real-world projects. Such evaluations show that the detectors find misuses in real code and how many false positives they produce. Our taxonomy of API misuses now allows researchers to additionally assess which types of misuses a detector conceptually covers. For example, we now realize that, to the best of our knowledge, there is no detector that identifies superfluous calls. Furthermore, our benchmark allows—for the first time—measuring how many misuses a detector misses.

To use the dataset as a benchmark for API-misuse detectors, the source code of each misuse is needed, to run the respective analyses on. Ideally, we want to assess two aspects of a detector: (1) its ability to detect a certain misuse in the context of the code it occurs in and (2) its ability to detect a certain misuse in general. As code surrounding or intertwined with a usage may add considerable noise, a detector's general ability to detect a certain misuse does not imply it is able to detect that misuse in any context. The other way around is also true. Only because an approach misses a misuse in a specific context does not mean it is incapable of detecting this misuse at all. To address (1), we plan to extract the original occurrence of the misuse, i.e., the original code of all classes with at least one method contributing to the misuse. The commit URL from MuBench enables us to automate this. Subsequently, we need to ensure that the example is parseable, such that standard source-code processors can work with it. To address (2), we plan to extract a minimal example of the misuse, i.e., we strip the previous example of all code elements unrelated to the misuse.

In addition, we will extract fixed version of both examples, to add negative datapoints to the evaluation, i.e., cases in which misuses detectors should not report anything. This allows us to assess whether detectors find certain misuses only accidentally, because such a detector would likely identify the misuse also in the fixed version. For example, a detector that ignores control flow might correctly detect a misuse where a method is called twice, but also reports a fixed version where the method is called once per path.

We note that Dallmeier et al. [2] present an interesting approach to classify bugs, using the set of AST nodes changed by the fix. Future work should compare the results of such an automated approach to our manual classification.

5. EXTENDING THE BENCHMARK

Due to the high manual effort of reviewing bug reports, we evaluated only part of the BugClassify and the GitHub candidates. Now that we established the data format and the classification rules, it is straightforward to review the remaining reports. We will continually do this over time.

As we show in Section 2.3, developers face more problems with API misuse than we can find in issue trackers or code

repositories. To find more of such examples, we plan to promote our survey to a broader audience. In addition, we want to mine Q&A sites for API-misuse related questions. We believe that such a study could help to identify further common API misuses, as well as to assess how commonly developers face API misuses we already know about.

We publish a meta-data template as part of MuBench and encourage other researchers to contibute descriptions of their findings directly on GitHub. Alternatively, examples can still be submitted to our survey questionnaire.

6. LIMITATIONS

Apart from the 16 misuses of the JCE API, which where reviewed by two of the authors, the misuses in MuBench were reviewed solely by the first author. To ensure quality, we publish the dataset and encourage others to review it.

Our dataset encompasses 77 API misuses from real-world projects and 12 misuses from our survey. This number is small, compared to other benchmark datasets. However, we focus on a very specific class of bugs and provide detailed information about every misuse. Now that the concept is in place, we can continuously enlarge the dataset. We encourage other researchers to contribute to it as well.

We identified misuses from 33 projects. They may not be representative for API misuses in general. However, we found that a relatively small number of criteria suffices to characterize all these misuses. We see this as an indicator that we covered a large fraction of the different kinds of API misuses. The classification enables us, for the first time, to gather empirical data about API misuses. We expect that as the benchmark increases in size, it will also closely mirror the actual distribution of the different kinds of API misuses.

Another limitation of MuBench is that the instances we identified may not be representative for the code misuses appear in. For example, a certain misuse might often be distributed over multiple methods, while the examples we found only show occurrences in a single method or vice versa. Note that we do not intent for MuBench to replace evaluation on real-world code. We see it as an additional validation point to assess the kinds of API misuses covered by a detector and to enable comparison of different approaches.

7. RELATED DATASETS

Several datasets of software bugs have been created in the past [2, 6, 7]. These datasets contain instances of arbitrary software bugs, identified heuristically or manually. We used them as sources for the identification of API misuses.

BegBunch [1] is a bug-detection benchmark of C programs. It provides a dataset to measure tool accuracy, w.r.t finding buffer overflows, memory/pointer bugs, integer overflows, and faulty format strings. Furthermore, it provides a second dataset to measure scalability of detectors and tooling to assess detectors' performance on either. The goal of our work is a similar infrastructure for API-misuse bugs in Java programs. MuBench is a first step in this direction.

8. CONCLUSION

In this work, we present MuBench, a dataset to benchmark API-misuse detectors. The dataset provides detailed meta data about 89 API misuses from 33 projects and a survey. We draft a taxonomy of API misuses. We believe that MuBench will advance the state-of-the-art in misuse detection, providing insights about conceptual capabilities of tools and false negatives in their application.

9. ACKNOWLEDGMENTS

The work presented in this paper was partially funded by the German Federal Ministry of Education and Research (BMBF) with grant no. 01IS12054 and by the DFG, project E1 in CRC 1119 CROSSING. The authors assume responsibility for the content.

10. REFERENCES

[1] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. BegBunch: Benchmarking for C Bug Detection Tools. DEFECTS'09, pages 16–20. ACM, 2009.

[2] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. ASE'07, pages 433–436. ACM, 2007.

[3] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. CCS'13, pages 73–84. ACM, 2013.

[4] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. ASE'15, pages 307–318, 2015.

[5] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. CCS'12, pages 38–49. ACM, 2012.

[6] K. Herzig, S. Just, and A. Zeller. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction. ICSE'13, pages 392–401. IEEE Press, 2013.

[7] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. ISSTA'14, pages 437–440. ACM, 2014.

[8] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why Does Cryptographic Software Fail?: A Case Study and Open Problems. APSys'14, pages 7:1–7:7. ACM, 2014.

[9] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. ESEC/FSE'13, pages 306–315. ACM, 2005.

[10] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. "Jumping Through Hoops": Why do Developers Struggle with Cryptography APIs? ICSE'16, 2016.

[11] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone Management for Evolving Software. *IEEE Trans. Softw. Eng.*, 38(5):1008–1026, 2012.

[12] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A Graph-based Approach to API Usage Adaptation. OOPSLA'10, pages 302–321. ACM, 2010.

[13] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *IEEE Trans. Soft. Eng.*, 39:613–637, 2013.

[14] A. Wasylkowski and A. Zeller. Mining Temporal Specifications from Object Usage. *ASE*, 18(3):263–292, 2011.