Assisting Example-based API Misuse Detection via Complementary Artificial Examples

Maxime Lamothe, *Member, IEEE,* Heng Li, *Member, IEEE,* and Weiyi Shang, *Senior Member, IEEE*

Abstract—Application Programming Interfaces (APIs) allow their users to reuse existing software functionality without implementing it by themselves. However, using external functionality can come at a cost. Because developers are decoupled from the API's inner workings, they face the possibility of misunderstanding, and therefore misusing APIs. Prior research has proposed state-of-the-art example-based API misuse detectors that rely on existing API usage examples mined from existing code bases. Intuitively, without a varied dataset of API usage examples, it is challenging for the example-based API misuse detectors to differentiate between infrequent but correct API usages and API misuses. Such mistakes lead to false positives in the API misuse detection results, which was reported in a recent study as a major limitation of the state-of-the-art. To tackle this challenge, in this paper, we first undertake a qualitative study of 384 falsely detected API misuses. We find that around one third of the false-positives are due to missing alternative correct API usage examples. Based on the knowledge gained from the qualitative study, we uncover five patterns which can be followed to generate artificial examples for complementing existing API usage examples in the API misuse detection.

To evaluate the usefulness of the generated artificial examples, we apply a state-of-the-art example-based API misuse detector on 50 open source Java projects. We find that our artificial examples can complement the existing API usage examples by preventing the detection of 55 false API misuses. Furthermore, we conduct a pre-designed experiment in an automated API misuse detection benchmark (MUBench), in order to evaluate the impact of generated artificial examples on recall. We find that the API misuse detector covers the same true positive results with and without the artificial example, i.e., obtains the same recall of 94.7%. Our findings highlight the potential of improving API misuse detection by pattern-guided source code transformation techniques.

Index Terms—API-misuse detection, Mining Software Repositories, Empirical Software Engineering, Software reuse

1 Introduction

Application Programming Interfaces (APIs) offer software developers the means to interact with Software Development Kits, libraries, operating systems, frameworks, and cloud services [1]–[4]. Through their usage, software developers can simplify their work and concentrate on their novel ideas while relying on existing software to reduce the overhead of re-inventing functionality that already exists [5], [6]. However, using APIs is not free of issues or pitfalls. Because developers are decoupled from the inner workings of the APIs they use, they can misunderstand them and potentially misuse them [7], [8]. API misuses are defined as violations of the implicit usage constraints of an API [9]. Indeed, these API violations can lead to software crashes and introduce vulnerabilities.

Due to their ubiquity, determining how APIs are being used and misused is an important task in software development [10]–[13]. Indeed, prior research has investigated the usability of APIs to uncover how to improve API usability for API users [14]–[17]. However, even with the existence of various studies and tools to improve API usability [18]–[21], APIs still suffer from misuses [7]. Because APIs provide interfaces to existing functionality, such interfaces can obfuscate information and make it difficult for users to determine the correct way to invoke the underlying functionality when using an API [8], [22]. While API recommendation tools attempt to provide a prescriptive way to address the misuse problem, they cannot address cases where a misuse already exists in a code base. API misuse detectors have therefore been created to uncover cases where APIs were used in potentially incorrect ways [7].

API misuse detectors, particularly those that employ API usage examples to uncover potential misuses [7], are at the mercy of their sample sizes. A lack of usage examples was recently reported as one of the biggest challenges in API misuse detection [7]. In particular, uncommon API usages and alternative correct API usages have been found to make up 53.5% of false positive misuses [7]. An obvious and naive solution to reduce the incidence false positives is to have a greater diversity of correct API usages examples. Mining more correct examples is a time consuming, dataset dependent task that does not guarantee the quality of the mined examples. However, example mining can be automated and in the right circumstances it can indeed reduce the incidence of false positive misuses. To achieve this goal, recent API misuse detectors, such as MuDetect, mine multiple projects to collect API usage examples [9], however the resulting false positive rate still has room for improvement [23].

In this paper, we examine the challenge of missing correct API usage examples from a different perspective. Instead of mining source code from more projects to obtain more API usage examples, we propose to generate artificial examples based on the existing correct API usage examples.

- *Maxime Lamothe and Heng Li are with the Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Canada. E-mail: {maxime.lamothe, heng.li}@polymtl.ca*
- *Weiyi Shang is with the Gina Cody School of Engineering and Computer Science, Concordia University, Canada. E-mail: shang@encs.concordia.ca*

Manuscript received date; revised date.

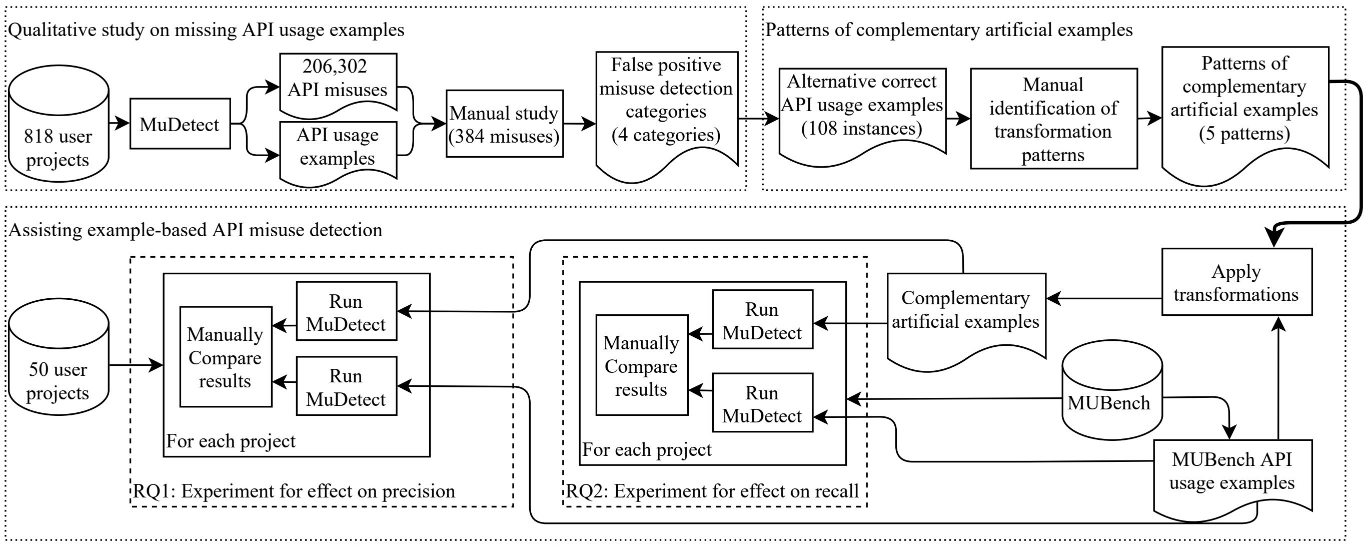

Qualitative study on missing API usage examples
818 user projects
MuDetect
206,302 API misuses
API usage examples
Manual study (384 misuses)
False positive misuse detection categories (4 categories)
Patterns of complementary artificial examples
Alternative correct API usage examples (108 instances)
Manual identification of transformation patterns
Patterns of complementary artificial examples (5 patterns)
Assisting example-based API misuse detection
50 user projects
Manually Compare results
Run MuDetect
Run MuDetect
For each project
RQ1: Experiment for effect on precision
Manually Compare results
Run MuDetect
Run MuDetect
For each project
RQ2: Experiment for effect on recall
Complementary artificial examples
Apply transformations
MUBench
MUBench API usage examples

Fig. 1. An overview of our study setup, data collection, and experiments

In this paper, we refer to these artificially generated, semantically and syntactically correct, examples as *alternative correct API usages*. We refer to incorrect API usage examples as API misuses, and correct API usage examples as API usage examples. The overview of our study is shown in Figure 1. We first undertake a qualitative study of API misuses that are identified by a state-of-the-art API misuse detector *MuDetect* on 818 open source projects. Through our study, we aim to discover patterns of opportunities where adding more alternative correct API usages may reduce the false positive detection of API misuses. In total, we identify five patterns that can be followed to generate artificial API usage examples. Such artificial examples can be used to complement the existing API usage examples used in API misuse detection. While the study was indeed conducted using MuDetect, our examples can be used on other example-based misuse detection approaches. Finally, the use of our alternate examples is light-weight and does not require the complete compilation of an application to obtain bytecode or machine code analysis, and it is thus also more explainable to developers.

We evaluate the usefulness of these artificial examples using projects from another set of 50 open source projects and the MUBench dataset. Our evaluation results show that all five of our patterns are applicable in transforming the existing API usage examples into the artificial ones. More importantly, by using the generated artificial examples to complement the existing ones, we can eliminate 55 false-positives from the API misuse detection. In addition, through a pre-designed experiment in MUBench, we find that the artificial examples do not reduce true API misuse detection by the API misuse detector, i.e., the recall remains at 94.7%. The contributions of this paper are:

- This paper tackles the challenge of missing correct API usage examples from a different direction from prior research.
- Through a qualitative study, we identify five patterns of alternative correct API usages which can be used to generate artificial API usage examples.
- The artificial API usage examples can complement existing API usage examples to reduce the false positive detection of API misuses while keeping all true positives.

Our findings highlight the potential of generalizing correct and incorrect API usages based on pattern-guided source code transformations. While our evaluation uses MuDetect, other API misuse tools also rely on examples to detect API misuses. Our examples could also help these detectors. We simply evaluate our approach using MuDetect because it is the current state-of-the-art approach.

Paper organization. Section 2 presents the background of example-based API misuse detection and a motivating example for our study. Section 3 presents our qualitative study on the false positive examples in API misuse detection. Section 4 presents the five patterns that can be used to generate artificial examples in order to complement the existing ones. Section 5 evaluates the usefulness of the generated complementary artificial examples. Section 6 discusses the threats to validity. Section 7 presents related work of this paper. Finally, Section 8 concludes the paper.

2 Background and a Motivating Example

In this section, we present the background of our study as well as an example to motivate our study.

2.1 Background: Example-based API misuse detection

Studies have demonstrated the advantages of example-based static misuse detection [7], [24]. While rule-based or constraint-based API misuse detection requires the existence of vetted knowledge of an API to codify usage rules that can then be used to detect misuses, example-based detectors can rely on existing API usages to extract the knowledge needed for their detection [1], [7], [24].

However, because example-based static API-misuse detectors, such as MuDetect [9], extract real usages of APIs to use as examples of API usages, they are dependent on their sample of examples. In a systematic evaluation of static

API-misuse detectors, it was determined that over 53.5% of false positives of misuse detectors are due to uncommon or alternative correct ways to use an API [7]. This occurs even in cases where API-misuse detectors are trained in a cross-project setting which provides more examples [9].

MuDetect is the state-of-the-art in example-based API-misuse detection, by default, it uses a minimum-pattern-support to allow examples with a minimum number of examples to qualify as potential API usage examples. Potential API usage examples are greedily explored and clustered according to isomorphic pattern candidate extensions [9]. While clustering and extending example candidates, code semantics are also observed, in a graph form, for data and control nodes that could have side-effects or are oddly linked to the API example (e.g., only linked because of usage order, but not actually linked through any control action). Only if all nodes successfully pass through its heuristics does MuDetect consider a potential API usage example. These heuristics were designed in an attempt to prevent flagging uncommon usages as misuses [9]. In spite of these safeguards, false positive detection still occurs [9].

False positives are particularly hurtful to API misuse detectors by causing an over-reporting of misuses, which in-turn can overwhelm the users of these misuse detectors. Based on the 33.0% precision of state-of-the-art approaches such as MuDetect [9], it can be understood that example-based static API-misuse detectors can stand to benefit from new ways to augment their sample of API usage examples.

2.2 A Motivating example

Prior research suggests that a majority of false positive API misuse detection is due to a lack of less frequent API usage examples [7]. However, little is known about the nature of less frequent API usage examples. By observing falsely detected API misuses perhaps it is possible to determine some patterns of less frequent API usages and find new ways to help reduce the mistakes in API misuse detection. For example, in Figure 2 we can see an example of a falsely detected API misuse and the API usage example that was used for its detection. Based on the example presented, we can see that the API misuse detector falsely detects that it is a mistake to have the *put* API method in the *if* block rather than have it in a missing *else* block as it is in the API usage example used for detection. The API misuse detection tool does not recognize that the conditional statement that determined whether the key is already contained in the map has been inverted in the wrongly detected API misuse.

This lack of knowledge stems from a lack of varied API usage examples. Indeed, other API misuse detection approaches have attempted to make use of API usage rules obtained from API documentation to circumvent this problem [1]. However, these approaches have not shown a vastly superior ability to detect API misuses, and they suffer from other pitfalls such as their requirement for high quality and up-to-date documentation. Current state-of-the-art static detection approaches attempt to remedy this problem by mining API usage examples from large inter-project data sources [9]. However, although this does improve the performance of API misuse detectors, the problem persists. If we can identify patterns within API usage examples that are prone to causing false positive API misuse detection, these patterns can potentially be used to generate complementary examples. These complementary examples, based on the patterns of real examples, could patch knowledge gaps without needing extensive source code searches to find all possible API usages. For example, to address the issue in Figure 2, if a complementary example existed in which the *put* method call was located in an *if* statement with inverted logic, i.e., *!containsKey()*, the API usage shown in Figure 2(a) would not be falsely detected. In this case the inverted logic is not the cause of the faulty misuse detection, but rather a way to obtain a different control logic while maintaining equivalent and correct functionality. While this could be a particularity of MuDetect, our examples would also cover situations where the inverted logic could be the case of a faulty misuse detection, as well as cases where the fault is due to the control logic.

Therefore, in the next section we conduct a qualitative study in an effort to uncover patterns of missing API usage examples.

3 Qualitative Study on Missing API Usage Examples

In this section, we first conduct a qualitative study to gain understanding of the missing correct API usage examples.

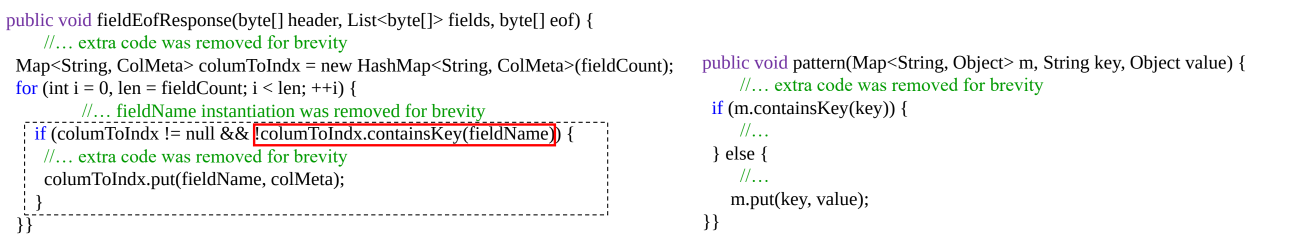
3.1 Qualitative study setup

Although work has been done to determine the caveats and problems with existing misuse detection techniques, work remains to be done to determine strategies to handle infrequent API usages and alternative usages for the same API [7]. We therefore seek to systematically determine patterns of alternative API usages and how to leverage them to reduce false-positives in API misuse detection. We first present the projects and tools used to conduct our preliminary study.

Subject projects. We use a readily available dataset [10] from a recent API research work that includes 3,099 Java projects available on Github. Although the dataset was originally assembled to study five open-source Java APIs (Guava, Hibernate-orm, Jackson, JUnit, and Log4j), the projects in this dataset are not limited to using only these five APIs and therefore present a rich source of varied API usage. Note that, we intentionally do not select an existing API misuse dataset (like MUBench) for this qualitative study in order to avoid the bias (positive or negative) of existing knowledge in the benchmarks of API usage examples.

To conduct our experiments, we selected a sub-sample of 1,000 projects, randomly selected from the original sample. Of these 1,000 projects we discovered that 132 of them were incompatible with MuDetect due to compilation errors in the projects. Our final sample size was therefore 868 projects. We reserved 50 of those projects for our final evaluation and the rest was used for our manual study. The names and download links for the projects used for this study, as well as the results of our experiments can be found in our replication package[1].

1. https://figshare.com/s/337e35f63c41251e20cb



```
public void fieldEofResponse(byte[] header, List<byte[]> fields, byte[] eof) {
    //… extra code was removed for brevity
  Map<String, ColMeta> columToIndx = new HashMap<String, ColMeta>(fieldCount);
  for (int i = 0, len = fieldCount; i < len; ++i) {
          //… fieldName instantiation was removed for brevity
   if (columToIndx != null && !columToIndx.containsKey(fieldName)) {
    //… extra code was removed for brevity
    columToIndx.put(fieldName, colMeta);
   }
  }}
```

(a) An API usage falsely detected as an API misuse

```
public void pattern(Map<String, Object> m, String key, Object value) {
      //… extra code was removed for brevity
  if (m.containsKey(key)) {
      //…
  } else {
      //…
    m.put(key, value);
  }}
```

(b) The API usage example used for API misuse detection

Fig. 2. An example of a falsely detected API misuse.

API misuse detection. In order to study the cases where valid API examples are missing, we use an automated tool to detect API misuses, and further examine the false positives in the detection results. We opt to use MuDetect [9] as an API misuse detector because it is a vetted API misuse detector that has shown state-of-the-art results. More importantly, MuDetect has the ability to uncover API usage examples against which it can measure potential API misuses [9]. This automatic mining of frequent API usage examples allows us to leverage the large scale of open-source repositories to mine usage examples from a wide variety of APIs and obtain varied samples of examples of API usages that could be qualified as odd or misused.

We ran MuDetect in its intra-project mode. We did not opt for its inter-project mode because running the inter-project mode on a sample of 818 projects is prohibitively time consuming due to the explosion of misuse patterns that occurs. Furthermore, using the intra-project mode allows us to obtain a "worst-case" real usage scenario which serves our goal of studying missing API usage examples.

MuDetect uses multiple heuristics to identify frequent API usages. The first heuristic is based on the frequency of API usage patterns. MuDetect has a minimum-pattern-support variable which allows any API with more usages than the set threshold to qualify as a potential API usage pattern. For the automatic extraction of API usage examples used in our qualitative study, we set the minimum-pattern-support to its default value to allow patterns with a minimum number of examples to qualify as potential API usage patterns. We used this value because it was successfully used in the MUBench dataset for its evaluation [9]. Furthermore, a lower threshold would allow for more false positive API usage patterns and increase the already non-trivial detection time.

MuDetect saves API call information on a per-misuse basis. The location (i.e., file, line number, calling method) of API misuses as well as the locations of API examples used for detection are recorded as part of the MuDetect tool process. We were therefore able to use this information to manually observe real instances of potential misuses.

3.2 Qualitative study process

In total, we obtain 206,302 potential API misuses. To understand the API misuses that are detected due to the lack of API usage examples, three authors of this paper acted as reviewers and conducted a manual study on a statistical sample of 384 detected API misuses (with 95% confidence level and 5% confidence interval). The overall approach used was based on prior works [25], [26]. Our final goal for the quantitative study is to uncover patterns of missing API usage examples that cause false positive API misuse detection. In this step, we first study the false positives in the detection results to identify the ones that may be caused by missing API usage examples. For each detected API misuse, the reviewers also observe five API usage examples that were leveraged by MuDetect to detect it. The five API usage examples can help the reviewers understand why an API misuse was detected. The manual study includes four steps.

Step 1. We first start by manually examining a sample of 174 detected API misuses. Each misuse was examined and categorized by two of the three reviewers. Therefore, each of the three reviewers was given 116 random potential misuses (i.e., 2/3 of the 174 detected misuses) to categorize as they saw fit (i.e., with open card sorting). This allowed each reviewer to examine 30% of the total dataset.

Step 2. Once the 174 misuses were categorized, the three reviewers discussed their categories and settled on a base classification schema. Only the classification schema is discussed at this stage, no misuse classifications are compared.

Step 3. Using the newly agreed schema, all reviewers reexamined their categorization results and relabel as required by the agreed schema categories. Once all of the 174 detected API misuses were classified according to the same schema, we measured the agreement ratio using Krippendorff's α [27]–[29]. The calculated agreement ratio was 0.744, i.e., a substantial agreement for consensus. Afterwards, all the three reviewers discuss the cases of disagreement, until final categorization of the 174 misuses were made.

Step 4. Due to the substantial agreement ratio achieved in the last step, a further sample of 210 misuses (for a total of 384 categorized potential misuses) was therefore manually classified by three reviewers without the need of overlapping, unless necessary (e.g., if an author felt unsure about the classification). Whenever a reviewer believed that a new category was identified during this step, all three reviewers discussed the particular case/API usage. The reviewers found that the codes established in Step 2. were stable during this round.

3.3 Qualitative study results

After the four steps, we put the 384 detected API misuses into a total of four categories.

Alternative correct usage: *(108 instances)* This category is used to describe API usages that are similar to the ones in the API usage examples that were used for detection. However, although similar, these falsely detected API usages use some alternative means of working with the API that could potentially have been detected by an API usage example complementary to the ones used for detection. API usages categorized in this category are used for further inquiry into potential transformation patterns to create complementary artificial examples.

Different usage scenarios: *(155 instances)* This category is used to describe API usages that were used in different scenarios. Contrarily to usages categorized as *alternative correct usages*, we could not identify how these usages could have been detected with complementary API usage examples, because these APIs are used to service different purposes. Therefore, these API usages require completely different API usage examples to the existing ones.

Correct misuse: *(13 instances)* Some of the examples that were selected for manual review were correctly identified by MuDetect as misuses, therefore we categorized these examples as such.

Not sure: *(108 instances)* If the reviewers could not agree on why an API usage was targeted as an API misuse by MuDetect or whether the API usage was a misuse or a special domain specific use case, we categorized the API usage as "Not sure". While the authors could likely classify all of these if pressed to do so, because there is an element of uncertainty, we err on the side of caution. This conservative sorting allows us to be more certain of our *alternative correct usage* classifications.

3.4 Summary of the qualitative study results.

Around one third of the overall results of our qualitative study are "alternative correct usages". These manually identified alternate API usages that were falsely identified as API misuses present opportunities for us to identify which alternate correct usages cause confusion in API misuse detection. The found prevalence of alternative correct usages provides an opportunity to identify general patterns of alternative correct API usages. Through these general patterns we can transform the existing frequent API usage examples into less frequent alternative correct examples, in order to address the challenging of missing API usage examples. Such examples would later help reduce the rate at which those API usages are falsely detected as API misuses.

4 Patterns of Complementary Artificial Examples

Section 3 shows that a considerable amount of API misuse detection results are actually due to missing correct API usage examples that represent the alternative correct usages of an API. If more usage examples of these correct API usages were available, they may significantly reduce the false positives in API misuse detection.

We aim to produce transformation patterns that could take real mined API usage examples as input (e.g., like those mined by API misuse detection approaches like MuDetect). These transformation patterns could then output alternative correct API usages that could be used to uncover API misuses with fewer false positives.

Therefore, in this section, all authors of this paper together discuss each API misuse detection results that were classified as "alternative correct usage" in the qualitative study (cf. Section 3), in order to uncover patterns of the needed complementary examples that can be used to reduce the false positive detection results. From the 108 "alternative correct usages", we identify 46 *Pipelines* cases, 9 *Alternative iterators* cases, 3 *Complementary imports* cases, 15 *Inverted conditions* cases, 35 *Intermediates* cases. For each of the patterns, the authors further discuss whether an artificial example can be automatically generated based on transforming existing API usage examples. The discovered pattern can later be used to generate artificial API usage examples in order to complement the existing examples to identify correct API usages and reduce falsely detected misuses. In total, we discover five such patterns.

In the rest of this section, we discuss each of our five manually identified patterns in the following template:

Description: Description to the pattern of complementary API usage examples.

Example: Discussion of a concrete example that is presented in Figures 3-7.

Detection strategy: Our strategy to detect possible API usage example candidates for the pattern.

Transformation approach: Our approach to generating artificial API usage examples based on the transformation of an existing API usage example.

Pattern 1: Pipelines.

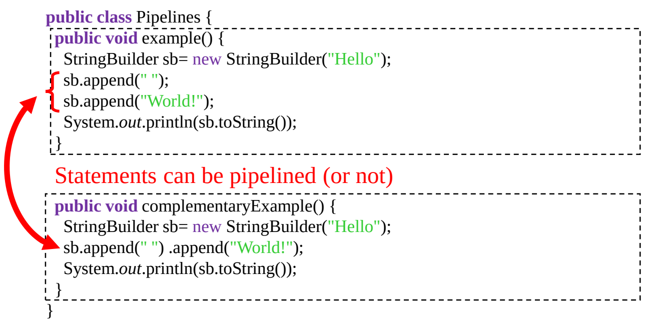

```
public class Pipelines {
  public void example() {
    StringBuilder sb= new StringBuilder("Hello");
    sb.append(" ");
    sb.append("World!");
    System.out.println(sb.toString());
  }
  public void complementaryExample() {
    StringBuilder sb= new StringBuilder("Hello");
    sb.append(" ") .append("World!");
    System.out.println(sb.toString());
  }
}
```

Fig. 3. Pattern 1: Pipelines

Description. This pattern is built primarily around APIs that can be used in stages on the same object, also known as the pipeline pattern [30]. Using the pipeline pattern requires that an API method return the same object type as the calling object type. This pattern is particularly useful because developers sometimes pipeline a few API call stages, and sometimes they use individual stages, one at a time.

Example. As shown in the example in Figure 3, a string builder can be used to append new string to the end of an existing string. Intuitively, the string builder can call the API method *append* multiple times, separately, in order to append different strings to its end. On the other hand, the API method *append* returns a reference to the original string builder to enable the method to be called in a pipeline. Hence, as shown in our example, it is semantically equivalent to either call *append* twice in separate, or in a pipeline. Therefore, it is possible to artificially generate a complementary API usage example, if such an API is called repetitively either separately or in a pipeline.

We would like to mention that the number of strings that can be appended is not defined ahead of time, and therefore cannot be inferred. In our example only two strings (" ", and "World!") are being appended for brevity. However, one could append many more, or none at all, while still being a correct API usage. However, because we cannot produce all possible numbers that an API is repetitively called, in this study we opt to only generate the complementary example with the same number of calls. However, our approach can easily be adjusted to any threshold number of calls.

Applying Algorithm 1 on the example in Fig. 3 would find that `sb.append("World!")` could be the child of `sb.append(" ")`. The two would then be joined as shown in Fig. 3.

Detection strategy. The *Pipelines* pattern requires an API that is called on, and returns, the same object. This can be ascertained via abstract syntax tree (AST) analysis with source bindings or through inference if a collapsed form pipeline is available (e.g., the *sb.append("").append("World!");* in the complementaryExample() method in Figure 3. However, there are exceptions to this rule, for example Java Streams present a pipeline like pattern, but they cannot be broken up into different stages. Java Streams stages must be done in a single pipeline, otherwise a new Stream must be created [31]. A simplified algorithm for detection and transformation can be found in Algorithm 1.

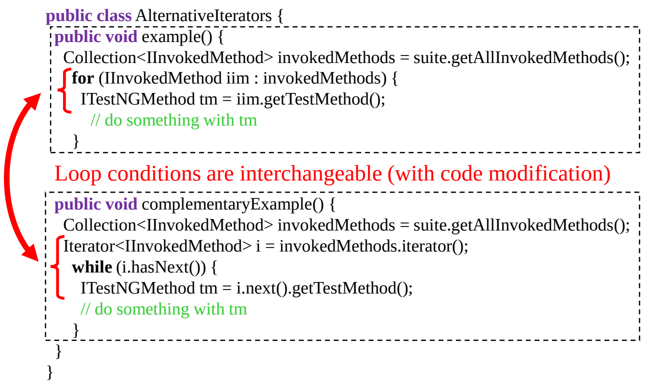
Algorithm 1: Searching and transforming pipelines.

```
Input  : Correct API example code $apiEx$
Output: Complementary artificial example
foreach $apiCall$ in $apiEx$ do
    if $apiCall$.isChildofOtherAPICall() &&
      $apiCall$.isNotStream() then
        $apiCall$.separateCallAndChild
    else if $apiCall$.CouldBeChildofOtherAPICall() then
        $apiCall$.joinCallAndChild
end
```

However, in some cases it might not be possible to fully collapse the stages of otherwise pipeline-able code. For instance, in our example of Pattern 1 in Figure 3, the string "World!" was assigned dynamically after having started the *StringBuilder* (e.g., by requesting user input). In that case, we would have to determine if the dynamic assignment could be moved outside of the pipeline pattern. In cases where intermediate instantiating of variables occurs between otherwise pipeline-able API calls, we must determine if moving those intermediate variables would break the original method and only transform code if it would not break. This analysis can be done by using control-flow and data-flow analysis. However, even in cases of unmovable intermediate variables, if there are many pipeline-able stages, it might still be possible to partially collapse some safe stages (e.g., all stages before an intermediate variable appears), which can be done to produce a possible complementary correct usage. In this study we only handle safe cases, and collapse the calls to the first occurrence of the original call.

Transformation approach. Starting from the top method (i.e., *example()*) in Figure 3, the stages of the pipeline can be collapsed after the first API call (in this case *.append("")*). Because the pipeline stages could be considered as a single statement, semi-colons must be adjusted accordingly. The reverse is also possible. It is possible to transform the pipeline form presented in *complementaryExample()* into its non-collapsed version in *example()*. This can be done by determining the return type of a pipeline stage (in this case *.append("")*). If the return type matches the type of the original object (in this case *StringBuilder*), then we can separate the stage by calling it on the original object in a new statement (semi-colons must be adjusted). Note that the original order should be preserved. Furthermore, although the example in this paper presents the same number of calls (i.e., twice) as the detected example, the pattern could be used to generate different numbers of calls when possible (e.g., break up the example string differently) or indeed no calls at all (e.g., the StringBuilder in our example would contain the whole string, no appends would be called).

Pattern 2: Alternative iterators.



```
public class AlternativeIterators {
  public void example() {
    Collection<IInvokedMethod> invokedMethods = suite.getAllInvokedMethods();
    for (IInvokedMethod iim : invokedMethods) {
      ITestNGMethod tm = iim.getTestMethod();
        // do something with tm
    }
```

Loop conditions are interchangeable (with code modification)

```
  public void complementaryExample() {
    Collection<IInvokedMethod> invokedMethods = suite.getAllInvokedMethods();
    Iterator<IInvokedMethod> i = invokedMethods.iterator();
    while (i.hasNext()) {
      ITestNGMethod tm = i.next().getTestMethod();
      // do something with tm
    }
  }
}
```

Fig. 4. Pattern 2: Alternative iterators

Description. This pattern arises from the different control flows and data flows that are induced by different loops that are allowed for a programming language (e.g., Java in our study). During our manual study we noticed that some examples of API usages were being falsely identified as API misuses because their control-flow was directed by different types of loops (e.g., *for* loop, *while* loop and *foreach* loop). We are particularly interested in loops that can be transformed into other types without changing the overall behavior of the program, or the API call under inspection.

Example. Figure 4 presents an example of a *for* loop that was transformed into an equivalent *while* loop using an iterator. In this example, the *getTestMethod()* API call can still be used to obtain the same effect. However, the syntax of the context of the API call has changed and the overall control and data flow of the API call is affected. Anecdotally different programmers appear to have different preferences for using different types of loops. Perhaps the legibility of code differs for developers based on their proficiency with various loops. The overall reasons why developers chose certain loops does not matter. However, the fact that different developers can use different loops to achieve the same effect is important. Developers seldom work alone on software projects, and therefore it is possible for different types of loops to appear with the same API call. If one type of loop is not frequent enough, it may be mistaken as a misuse when compared to examples with other loop types.

Applying Algorithm 2 on the example in Fig. 4 would find the *ForEach* loop, identify that all bindings are indeed recoverable and then create an iterator before the loop as shown in Fig. 4. The *ForEach* loop would be changed to a

While loop, the iterator would be used with `hasNext`, and a new variable (e.g., the result of `i.next()`) would be created to use in the place of the original variable (e.g., `iim`).

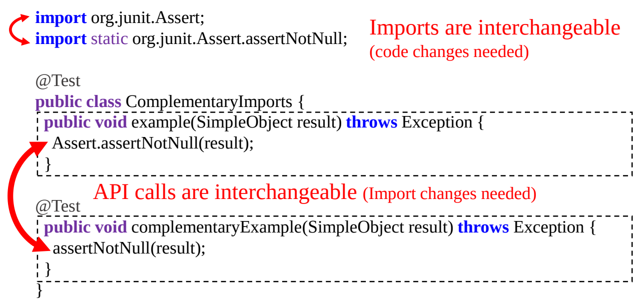
Algorithm 2: Searching and transforming iterators.

```
Input  : Correct API example code $apiEx$
Output: Complementary artificial example
1 /* Only shows one kind for brevity */
2 $iteratorStatements \leftarrow apiEx$.getIS()
3 foreach $iS$ in $iteratorStatements$ do
4     if $iS$.isForeachLoop() && $iS$.allBindingsRecoverable() then
5         $instantiateNewIteratorBeforeLoop()$
6         $iS$.changeToWhileLoop()
7         $iS$.changeIterParameterTo($hasNext$)
8         $iS$.transformIterVarsTo($next()$)
9 end
```

Detection strategy. The *Alternative iterators* pattern requires that an API call be affected by a loop (either by data or control-flow). If an API call is inside a loop but is not affected by the loop, it should be possible to construct a sub-graph of the usage that can cover any loop scenario [7]. In such cases, the loop itself does not impact the correctness of the API call. Furthermore, for us to produce a transformation it should be possible to transform a loop into another type only by referring to method calls for which we can infer all bindings. If an example requires code modification outside of inferable bindings, we do not attempt the transformation to limit the introduction of defects. A simplified algorithm for detection and transformation can be found in Algorithm 2.

Transformation approach. As shown in the pattern in Figure 4, we can easily transform a *foreach* Java loop into a *while* loop by introducing an intermediate iterator and the *hasNext* and *next* methods on this iterator. The same can be achieved in reverse. Similarly, it is also possible to transform a standard Java *for* loop into either of these types with the introduction of some intermediate variables that can be inferred from bindings in the existing code example. If we cannot infer all bindings, we do not attempt the transformation.

Pattern 3: Complementary imports.


Imports are interchangeable
(code changes needed)
API calls are interchangeable (Import changes needed)

```
import org.junit.Assert;
import static org.junit.Assert.assertNotNull;

@Test
public class ComplementaryImports {
 public void example(SimpleObject result) throws Exception {
  Assert.assertNotNull(result);
 }

@Test
 public void complementaryExample(SimpleObject result) throws Exception {
  assertNotNull(result);
 }
}
```

Fig. 5. Pattern 3: Complementary imports

Description. In our manual study, we uncovered cases where developers had imported static API methods and used them directly. In other cases, the same API was statically called from the owner class.

Example. This can be seen in Pattern 3 in Figure 5 where the *assertNotNull* API call occurs in both example methods. However, one of them is called explicitly by the *Assert* class; while the other one relies on the *import* statement. This example is simplified to show both imports in the same file, however this is not the case when we create our patterns. In cases like this, we actually create a copy of the original file and change the imports in that file instead. We then feed both of the example files to the misuse detector and force the detector to use both versions when it attempts to find a misuse that uses either file. If one of these usages is missing from the examples, it is possible to mistake the missing one as a misuse. Better type inference using partial program analysis could allow the resolution of this specific pattern. However, because partial program analysis can be computationally intensive, we believe that there is value added in computing examples ahead of time and using them as a quick lookup when the need arises. Overall, using complementary imports should allow tool makers to 'front-load' some of the computing requirements and therefore speed up the overall detection while reducing false-positives.

Applying Algorithm 3 on the example in Fig. 5 would identify `Assert.assertNotNull` as a static method call. This method would then be modified to `assertNotNull`, and the static import `org.junit.Assert.assertNotNull` would be added to replace the non-static version as shown in Fig. 5.

Detection strategy. To detect this pattern, an API call must use a static method. Furthermore, it should be possible to import this static method independently from the class itself, as shown in the *import static org.junit.Assert.assertNotNull* statement in the pattern of Figure 5. In-depth binding analysis could be used to detect this pattern. However, in-depth binding analysis is computationally expensive and could slow down already non-trivial API misuse detection times if done for every occurrence. To address this, we use a heuristic based on typical Java naming convention where the name of a class should start with a capital letter. Therefore, we can obtain a list of imported static method calls in the *import* statements and the calls to the static method calls in the source code. Although this pattern could be integrated directly into the API misuse detection approach, it is much more efficient to use this pattern for API usage examples, where the binding information only needs to be checked once for one API usage example, rather than for each potential API misuse. A simplified algorithm for detection and transformation can be found in Algorithm 3.

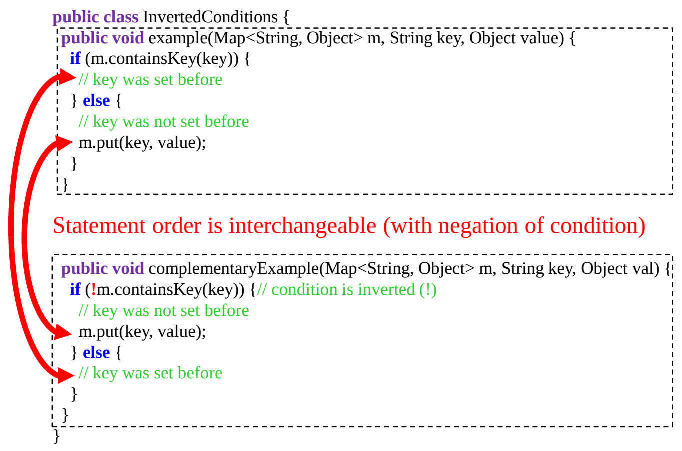
Transformation approach. As shown in the method the pattern of Figure 5, the *Assert.* class call in the *example* method can be removed as shown in the *complementaryExample* method. The corresponding static import to the target API method must be determined and added to allow the new syntax. The reverse is also possible. If a static method import is in use with a direct import, the corresponding import to the target class must be determined and added, and the static method must be modified to be called on the imported class.

Pattern 4: Inverted conditions.

Description. Similarly to our *Alternative iterators* pattern,

Algorithm 3: Searching and transforming imports.

```
Input  : Correct API example code $apiEx$
Output: Complementary artificial example
foreach $apiCall$ in $apiEx$ do
    if $apiCall$.fromStaticMethod() then
        if $apiCall$.notImportedAsStatic() then
            $apiCall$.changeImportToStatic()
            $apiCall$.changeCallSyntaxForStaticMethod()
        else
            $apiCall$.changeImportFromStatic()
            $apiCall$.changeCallSyntaxNonStaticMethod()
end
```



```
public class InvertedConditions {
 public void example(Map<String, Object> m, String key, Object value) {
  if (m.containsKey(key)) {
    // key was set before
  } else {
    // key was not set before
    m.put(key, value);
  }
 }
```

Statement order is interchangeable (with negation of condition)

```
 public void complementaryExample(Map<String, Object> m, String key, Object val) {
  if (!m.containsKey(key)) {// condition is inverted (!)
    // key was not set before
    m.put(key, value);
  } else {
    // key was set before
  }
 }
}
```

Fig. 6. Pattern 4: Inverted conditions

different developers appear to use different orders for their conditional statements. Although good practices [32] suggest using logic and naming conventions that make sense with respect to chosen names and reduce the number of double negatives (e.g., *!doesNotContain* instead of *contains*), it is still possible for situations to arise where two (or more) equivalent correct code expressions exist.

Example. In the pattern example in Figure 6, we present a simplified example of this pattern. In this example we can see that reversing the logic inside the conditional *if* statement changes the logic of the overall method. The *put* API call is therefore moved from the *if* block to the *else* block when the condition is inverted. As shown in the example, this pattern can work in either direction. Applying Algorithm 4 to the example in Fig. 6 would find `if(m.containsKey(key))`, identify that no `else if` condition exists, and proceed to invert the condition to obtain `!m.containsKey(key)`, and then invert the code blocks inside the respective condition blocks as shown in Fig. 6.

Detection Strategy. First, this pattern requires the use of a conditional statement (*if* statement) to gatekeep API usage. In the pattern of Figure 6 this API call is the *put* method. Currently we only consider cases with single *if/else* statements, we do not apply the pattern if any *else if* statements are involved. Afterwards, we investigate the possibility of switching the order of conditional statements. Similarly to Pattern 3, it is less computationally expensive to determine an equivalent loop once for a known API usage example to create a complementary usage example, than it is to determine the equivalence of every loop analyzed by an API misuse detector. A simplified algorithm for detection and transformation can be found in Algorithm 4.

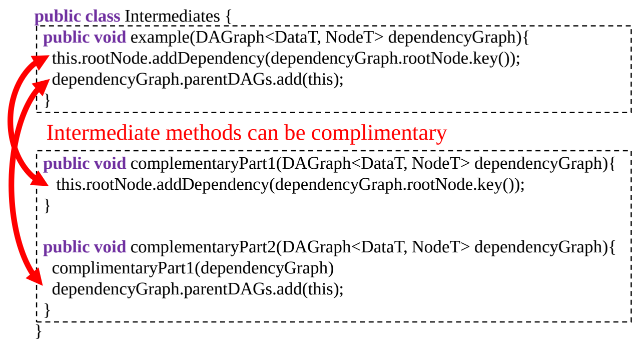
Algorithm 4: Searching and transforming conditions.

```
Input  : Correct API example code $apiEx$
Output: Complementary artificial example
$conditionalStatements \leftarrow apiEx$.getCS()
foreach $CS$ in $conditionalStatements$ do
    if $CS$.doesNotHaveElseIf() then
        $CS$.invertCondition()
        $CS$.invertIfAndElseCodeBlocks()
end
```

We have also observed cases where a complementary example could be produced by exchanging an *if/else* statement with a *try/catch* clause. However, it was not readily apparent how this particular version of the pattern could be generalized safely, while remaining certain that we would not introduce undesirable side-effects. However, we present this strategy here because we did find instances of these false positive API misuse identification in our manual study.

Transformation approach. The pattern first requires the inversion of the conditional logic in an *if* statement. After the conditional logic has been inverted, the functionality that was originally in the *if* block can be transferred to the *else* block, and vise-versa. If multiple conditions are present in the if statement care must be taken to either invert all the conditions separately without fail or invert the complete statement as one piece. The inversion of the logic inside the *if* statement in it's simplest form can stem either from the removal or addition of the "not" operator (i.e.,!).

Pattern 5: Intermediates.



```
public class Intermediates {
 public void example(DAGraph<DataT, NodeT> dependencyGraph){
  this.rootNode.addDependency(dependencyGraph.rootNode.key());
  dependencyGraph.parentDAGs.add(this);
 }
```

```
 public void complementaryPart1(DAGraph<DataT, NodeT> dependencyGraph){
  this.rootNode.addDependency(dependencyGraph.rootNode.key());
 }

 public void complementaryPart2(DAGraph<DataT, NodeT> dependencyGraph){
  complimentaryPart1(dependencyGraph)
  dependencyGraph.parentDAGs.add(this);
 }
}
```

Fig. 7. Pattern 5: Intermediates

Description. This pattern allows more flexibility in the expression of API usage examples by parameterizing intermediate functionalities. Similarly to the extract-method refactoring, the Intermediates pattern is meant to extract functionality from an existing method into a new method. Existing functionality that is related to an API usage can be extracted to a new method, and replaced with a method call. This can be done to allow different control flows to be represented by alternative examples and considered as equivalent by misuse detection tools.

Example. A simplified example is presented in Figure 7 where the *complementaryPart1* method replaces functionality originally in the *example* method. This example is trivial since it does not reduce the number of statements in the *complementaryPart2* method. However, the transformation can be expanded to more complex forms that would apply some separation of concerns. Applying Algorithm 5 to the example in Fig. 7 would loop through the two API calls (i.e., `addDependency` and `add`), and find that only the `addDependency` API call has no dependence on other variables or API calls. The algorithm then suggests that the `addDependency` call can be pulled out into its own method and that the method can be called to obtain the same result.

Detection strategy. This pattern requires the possibility of extracting functionality, abstracting this functionality to an intermediate method and either invoking it in the original method or introducing it to the original method as a parameter. We require that the control and data-flows of a method be separable before or after an API call. If the method statements are heavily coupled, this transformation cannot be applied. A simplified algorithm for detection and transformation can be found in Algorithm 5.

Algorithm 5: Searching and transforming intermediates.

```
Input  : Correct API example code $apiEx$
Output: Complementary artificial example
1 foreach $apiCall$ in $apiEx$ do
2     if $apiCall$.notDependentOnOtherVars() then
3         $apiCall$.abstractCallToVar($newVar$)
4         $newVar$.addAsMethodParameter()
5 end
```

Transformation approach. The control and data flows of the original method must then be analyzed to determine where a proper method extraction could occur. The transformation approach is similar to an *extract method* in refactoring [33]. However, there exist infinite possibilities to extract new intermediate methods from an existing program. Therefore, in our heuristics we only extract statements adjacent, i.e., right before or after, to the API call that is targeted to create an intermediate example. Once an intermediate method has been introduced, the original method must be modified to remove the functionality that was extracted, and instead call the intermediate method, with its appropriate parameters.

5 Assisting Example-based API Misuse Detection

In this section, we evaluate the usefulness of our generated artificial API examples that are based on the five patterns discovered in Section 4. In particular, we perform an experiment that detect API misuses in open source projects, with and without the use of the artificial API usage examples. In the rest of this section, we present the subjects of the experiment, the experimental process, and the experimental results.[2]

2. A prototype implementation to automate the generation of complementary artificial examples can be found online: https://figshare.com/s/337e35f63c41251e20cb

5.1 Experimental subjects

To test our complementary examples, we specifically rely on already known API usage examples that have been used to identify misuses in the MUBench dataset. The MUBench dataset comes with a prepared benchmark named *FSE18-Extension* which contains 107 known misuses, each with a single known API usage example. We use these existing API usage examples in the MUBench dataset [34] as the API usage examples for the API misuse detection tool. The MUBench dataset provides a baseline of known and vetted API misuses in a variety of real projects. We therefore use these API usage examples as input to create our alternative correct API usage examples.

There may exist other API usage scenarios that the data in the MUBench dataset does not cover. Therefore, as mentioned in Section 3, we use 50 open source Java projects from a prior study that mined these projects from Github [10]. To avoid bias, we make sure that none of the 50 projects were included in our qualitative study where the patterns were discovered (cf. Section 3 and Section 4). All of these projects are used in prior API-related research and had at least 17 months of history and had an average of 26K lines of code (minimum: 1.4K, maximum: 225K). The details of all 50 projects can be found in our replication package identified in Section 3. Because these projects have not been used in prior API misuse studies, the results obtained from running MuDetect on these projects are all manually vetted.

5.2 Experimental process

In particular, our experimental process is designed to answer two research questions:

- **RQ1:** Can artificial examples reduce false positives from the API misuse detection results?
- **RQ2:** Would artificial examples negatively impact the detection of any true API misuse?

Experimental process to answer RQ1. We extract the existing API usage examples that are provided by MUBench. We then detect API misuses using the state-of-the-art API misuse detection tool, i.e., MuDetect, on our 50 open-source projects. We do this by using the extracted existing examples for a "cross-project" detection baseline. We save these results. We then generate artificial API usage examples to complement the existing API usage examples from MUBench, based on the five patterns shown in Section 4. We allow our patterns to generate as many examples as they can (e.g., the Alternative iterators pattern could generate a new example that uses a "for" loop using an "Iterator", and also a different example that uses a "while" loop with an "Iterator"). Using these artificial examples, we rerun MuDetect on the same 50 subject projects, now with our alternative correct API usages enhancing the approach.

By default, MuDetect would consider all 'correct' examples independently to attempt to detect API misuses. In order to observe the value of the artificial examples in complementing existing examples, we therefore force MuDetect to consider our complementary artificial API usage examples immutably paired to the original API usage examples. Note that, we seek to preserve developer intent and maximize the data available to the misuse detector.

Therefore, we do not aim to use the artificial examples to replace the existing ones. Instead, the role of the artificial example is to complement existing examples. We neither aim to treat the artificial examples as distinct examples independent from the original examples, as we seek to preserve the existing predictive power of MuDetect and augment it whenever possible. Finally, we compare the results of both runs to answer the first research question. If any misuses are identified by one run but not by the other, we consider these as candidate false-positives and manually examine them.

Experimental process to answer RQ2. We would like to ensure that our complementary examples do not negatively impact the overall power of API misuse detectors. However, our experimental process for RQ1 cannot serve for this goal because there exists no ground truth on all of the true API misuses in the 50 open source projects. On the other hand, there exists a specially designed experiment, i.e., the *ex1* experiment in MUBench, which particularly serves the goal of calculating a recall upper bound for a given API misuse detectors with known API usage examples. This experiment is used in prior works as the de facto standard for evaluating recall for API misuse detection [7]. We therefore run this experiment with only the original API usage examples. We then run the experiment again, having the existing examples complemented by the artificial examples. We compare our results to determine the effect of our complementary API usage examples on the recall of an API misuse detector. In particular, for each run, we follow the same evaluation approach as Amann et al. [7] and record a positive identification of an API misuse if any detection proves to be positive.

5.3 Results

All five of our patterns are applicable to complement the existing API usage examples in MUBench. Out of all 107 API usage examples in MUBench, only one example does not meet the detection strategy of any pattern. This API usage example was created for a synthetic Java survey, and contained a single statement composed of five Java stream stages. Because we strategically do not consider streams to avoid potential mistakes (cf. Pattern 1, Pipeline in Section 4), we were unable to generate any complementary API usage examples for this particular API usage example.

In particular, we were able to improve three existing API usage examples with complementary API usage examples using the *Pipelines* pattern, eight using the *Alternative iterators* pattern, 22 using the *Complementary imports* pattern, eight using the *Inverted conditions* pattern, and 65 using the *Intermediates* pattern. Although we were able to express all five of our patterns on this dataset, the *Intermediates* pattern stands out as the most popular pattern in this case. This dataset contained methods where further separation of concerns could be introduced, which allowed for a greater expression of our *Intermediates*. A different dataset could perhaps present different pattern frequencies. However, because our patterns were generated on a completely separate dataset, and yet all five of our patterns could be used in the MUBench dataset, we are confident that our patterns present potential for general complementing of existing API usage patterns.

The artificial examples can assist API misuse detection by reducing false positives. In total, we find 55 cases of API misuse detection results that were detected with the original examples but were not detected when complemented by our artificial examples. We then manually go through each of those 55 examples to determine if they were API misuses or not, and which of our transformation patterns was used to disable their detection. We find that all 55 API usages were correctly identified when using complementary artificial examples, and that they were originally mistakenly labelled as API misuses. MuDetect identified 7,327 potential misuses without the help of artificial examples. However MuDetect presents results ordered by the likelihood of a misuse occurring. The top-20 results for each project are normally evaluated to prevent overwhelming users. Considering the top-20 results of each project, 36% of the cases removed by artificial examples directly affect the top-20 results.

Furthermore, the average ranking of the confidence of our removed cases is 16.47, with a standard deviation of 15.42, and with the largest confidence rating rated as 42nd highest. These findings indicate that the rest of our results are not far from the threshold criteria set by prior research [9]. Furthermore, using artificial examples did not add any new false positives.

We find that 43/55 (81.8%) of the mistakenly identified API misuses that were corrected by our complementary artificial examples used some kind of conditional statements. This shows that it is particularly important to have a well-rounded sample of API usage examples that contain various types of conditional statements because their detection appears sensitive to their format. The second most common mistakenly identified API usage type used iterators (6/55), this can also be used as a suggestion for future API misuse datasets to carefully consider various types of loops or use complementary artificial examples to enhance them. Finally, we report 3/55 wrongly identified API misuses with intermediate methods, 2/55 with pipelines, and 1/55 with static imports. Although these cases are less prevalent, we still encourage the use of their transformation patterns because they still provide a reduction in false positive.

Answer to RQ1: 55 falsely detected API misuses were prevented by complementary artificial examples. Complementary artificial API usage patterns can therefore successfully be used to reduce the incidence of false positive API misuse detection on real world projects.

The artificial examples do not prevent the detection of true API misuses. Due to build errors for some examples in the MUBench dataset we were able to obtain results for 95/107 API misuses in the dataset. MuDetect was able to successfully detect 90/95 of these misuses (94.7% recall) both with and without our complementary examples. All 90 correctly identified misuses were the same for both experiments (i.e., with and without our complementary examples). Therefore, we do not find any case where using complementary artificial API usage examples reduce the overall recall of API misuse detectors.

Answer to RQ2: Our findings on the MuBench dataset indicate that using complementary artificial API usage examples does not reduce the overall recall of API misuse detectors (94.7% recall with & w/o our complementary examples).

6 Threats to Validity

Construct validity. We do not claim to have found all API misuses, falsely detected API misuses, or API usage patterns pertaining to the APIs in this study. However, we believe that the projects and tools used in this work are adequate to produce results that give insight into the problem at hand. We only provide results that have been vetted by multiple individuals. Results which were labelled as "Not sure" could be API misuses, and might uncover more alternate usage patterns. However, because we could not guarantee their labelling, they were not considered for the patterns presented in this paper.

External validity. Because the API misuses, workarounds, and frequent pattern instances in this study were detected for Java APIs in Java user applications, it is possible that the findings in this paper do not generalize to other programming languages. However, although the results presented in this paper were obtained from Java APIs, the results were obtained by mining hundreds of user applications for API misuses without discriminating against any particular APIs. We therefore believe that although we cannot prove that our results generalize to other programming languages, the results presented should generalize to Java APIs. Although the results presented in this paper use MuDetect, an entire class of API misuse tools use examples to detect API misuses our examples could also help these detectors. We simply evaluate our approach using MuDetect because it is the current state-of-the-art approach. Furthermore, although the correct examples are indeed based on MuDetect in this study, it is possible to use other tools to find these (e.g., PAM to find frequent API usages). We mitigate the threat that ties our approach to specific examples by creating patterns to generate alternative correct usages that are not dependent on MuDetect or the examples that were mined for this study. We derived our patterns and algorithms based on a manual examination of the outputs of MuDetect. Potential bugs of the tool may impact some of our results. However, our methodology and main findings can generalize to other example-based API misuse detection tools.

Internal validity. The patterns presented to produce complementary artificial API usages, the suggestions presented for future API misuse detectors, and the findings from our qualitative study might not be fully indicative of API misuses and could present internal experiment bias. We attempted to mitigate these threats by having multiple reviewers for the API misuses that we presented in this work, and having these reviewers reach consensus on discussions pertaining to the patterns and suggestions that we present in this paper. Furthermore, we use completely different samples to obtain and to test the patterns presented in this work. Although the sample size of our qualitative study is statistically significant (384), it is possible that our findings only generalize to the MuDetect tool. However, MuDetect uses a published and general misuse detection approach that has been shown to be the current-state-of-the-art in example based static API misuse detection. Therefore, we believe that our results can contribute to improving the current state-of-the-art.

7 Related Work

In this section we discuss prior work related to the research presented in this paper.

7.1 API misuse detection

API misuse can stem from misunderstood or neglected constraints such as call order, or state conditions [1], [35]. Misuse detection tools and approaches exist to detect these misuses to reduce their incidence [35]–[41]. Various methods exist to detect API misuses [39], [42]–[48]. Some approaches use fine-grained API-constraint knowledge graphs to detect if API usages violate known usage constraints such as call order, condition-checking, return-conditions, and exceptions [1], [24]. In some approaches [1], these API constraints are obtained by crawling online API documentation, which is then transformed into expected declaration graphs that can then be compared against source code to detect constraint violations. Rule based API misuse detection approaches require manual evaluation of the constraints to determine their validity. While this approach has shown to be competitive in both precision and recall with other approaches such as MuDetect. We chose not to use this type of approach in our study because the manual evaluation of API constraints becomes prohibitive at scale, and state-of-the-art static detectors (e.g., MuDetect) obtain similar results without these drawbacks.

API misuse tools, like MuDetect, rely on pattern mining approaches that mine API usage examples from existing user projects to compare against potential misuses [24], [24], [34], [38], [48]–[50]. These approaches require either vetted API usage examples to compare against user code to detect misuses, or they can mine examples from user code automatically. MuDetect is a recent state-of-the-art attempt at this type of approach which uses cross-project data to improve detection of API usage examples [9]. These API usage examples can then be used to detect API misuses. The approach has shown state-of-the-art precision and recall, particularly when used in a cross-project setting [50]. We chose to use MuDetect as an API misuse detector for this work because it presented state-of-the-art results and because it can mine API usage examples from user projects which do not require manual vetting. Automatic extraction of usage examples is a great advantage when dealing with large scale detection.

7.2 API usage patterns

Learning how to use APIs can be time consuming, particularly when documentation or usage examples are sparse [18]. API mining algorithms [18], [51], [52] allow the extraction of frequently used API usage patterns that can then be used to provide relevant usage examples [51], API recommendation engines [19], [53], or insight into API usages [21]. PAM [18] is a state-of-the-art approach that mines existing frequent usage patterns from user projects.

These patterns are mined using a ″near parameter-free probabilistic algorithm″, which returns frequent API sequences that occur more than would be expected by random chance [18]. These patterns can be considered to be the ″most-interesting″ sequences of API usage in a given sample [18]. Approaches such as MARBLE [21], use these frequently used API patterns, coupled with abstract syntax trees (AST) to identify examples of boilerplate code. Boilerplate code is repetitive code that must be invoked to get some underlying functionality to work [21]. MARBLE is an approach to mine this boilerplate code from user projects. These examples of boilerplate code can then be given to API developers to uncover potential improvements to their API [21]. Although these approaches are able to mine interesting API usage sequences, they still rely on frequent patterns to obtain results. If a training dataset does not contain sufficient examples, then knowledge gaps can occur even with approaches such as PAM. Therefore, in this paper we chose not to rely on frequent pattern mining approaches because they have already been used in existing API misuse detection and have not solved the problem of false positives or low recall (e.g., MuDetect uses pattern frequency as one of it's heuristics to identify API usage examples).

Detecting semantically equivalent code could yield more API usage patterns. Indeed, research has been done in the detection of semantically equivalent code [54]–[56] and fault tolerance [57]. However, we chose not to rely on tools that generate or check for automatically generated semantically equivalent patterns [55], [58] because we believe that our approach allows benefits that semantically equivalent code generation cannot provide at this time. For example, while checking for semantically equivalent code can detect a large number of semantically equivalent examples, it would require checking every example for equivalence, a more computationally expensive task than the comparisons used in our approach. Furthermore, contrarily to our patterns, automatically generated semantically equivalent examples may not reflect real usage patterns or developer intent. Because we are simultaneously attempting to discern how to improve API misuse detection, and the different ways in which developers actually use APIs, automating the process may obscure the overall developer intent. For these reasons, at this time, we base our examples on user generated content to preserve the overall developer intent, reduce computational overhead, and to remain compatible with existing misuse detection approaches. However, we believe that coupling the automatic detection of semantically equivalent code and an overall understanding of developer intent is a very interesting topic for future research.

8 Conclusion

In this paper, we conduct a qualitative study on the falsely detected API misuses obtained by using a state-of-the-art example-based API misuse detection approach on a large sample of projects. By manually studying real examples of falsely detected API misuses, we uncover 108 cases of alternate but correct API usages that were falsely identified as API misuses. Through a manual investigation by three reviewers, we discover five patterns, which can be used to transform existing API usage examples into artificial API usage examples. Such artificial examples can cover the knowledge gaps caused by a lack of diversified existing API usage examples. We provide detailed discussions and simplified examples to explain these five patterns, as well as our strategies to detect these patterns in the source code, and approaches to transform existing API usage examples with these patterns.

We evaluate the usefulness of the complementary artificial API usage examples through the use of 50 open-source Java projects and through the MUBench misuse benchmark. We find that using the artificial examples does not reduce the recall of API misuse detection but does allow for the removal of falsely identified API misuses. Our findings highlight the potential of generalizing API usage examples through pattern-guided source code transformations and reduce the dependence of example-based API misuse detection on haphazardly mining large samples of user projects.

References

[1] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, "Api-misuse detection driven by fine-grained api-constraint knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 461–472.

[2] H. Zhong, N. Meng, Z. Li, and L. Jia, "An empirical study on api parameter rules," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 899–911.

[3] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[4] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2019.

[5] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *Proceedings of the 13th international conference on Software engineering - ICSE '08*, p. 481, 2008.

[6] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, mar 2006.

[7] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[8] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, may 2013.

[9] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating Next Steps in Static API-Misuse Detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, ser. MSR '19. Piscataway, NJ, USA: IEEE, may 2019, pp. 265–275.

[10] M. Lamothe and W. Shang, "When apis are intentionally bypassed: An exploratory study of api workarounds," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 912–924.

[11] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Engg.*, vol. 20, no. 5, pp. 1275–1317, Oct. 2015.

[12] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online Q&A forum reliable?" in *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. New York, New York, USA: ACM Press, 2018, pp. 886–896.

[13] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 183–193.

[14] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "Api designers in the field: Design practices and challenges for creating usable apis," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 249–258.

[15] J. Stylos and B. A. Myers, "The implications of method placement on api learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, p. 105–112.

[16] A. Macvean, M. Maly, and J. Daughtry, "Api design reviews at scale," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2016, p. 849–858.

[17] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 293–304.

[18] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–265.

[19] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[20] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing API Usage Examples at Scale," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. New York, New York, USA: ACM Press, 2018, pp. 1–12.

[21] D. Nam, A. Horvath, A. Macvean, B. Myers, and B. Vasilescu, "Marble: Mining for boilerplate code to identify api usability problems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 615–627.

[22] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings. 27th International Conference on Software Engineering. ICSE 2005.*, 2005, pp. 117–125.

[23] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 602–613.

[24] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, p. 383–392.

[25] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[26] N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and interrater reliability in qualitative research: Norms and guidelines for cscw and hci practice," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, Nov. 2019.

[27] R. Artstein and M. Poesio, "Inter-coder agreement for computational linguistics," *Comput. Linguist.*, vol. 34, no. 4, p. 555–596, Dec. 2008.

[28] K. Krippendorff, "Computing krippendorffs alpha reliability," *University of Pennsylvania Scholarly Commons*, Jan 2011.

[29] K. H. Krippendorff, *Content Analysis - 3rd Edition: an Introduction to Its Methodology*. SAGE Publications, Inc, 2013.

[30] "Pipeline." [Online]. Available: https://java-design-patterns.com/patterns/pipeline/

[31] "Package java.util.stream," Jul 2020. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

[32] R. C. Martin, *Clean code*. Apogeo, 2018.

[33] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, p. 1757–1782, Oct. 2011.

[34] S. Amani, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench," in *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*. New York, New York, USA: ACM Press, may 2016, pp. 464–467.

[35] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, Mar. 2013.

[36] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–84.

[37] Z. Li and Y. Zhou, "Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, p. 306–315, Sep. 2005.

[38] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 35–44.

[39] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 283–294.

[40] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 795–800.

[41] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 123–134.

[42] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 288–298.

[43] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 925–935.

[44] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *Proceedings of the 24th European Conference on Object-Oriented Programming*, ser. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 2–25.

[45] C. Lindig, "Mining patterns and violations using concept analysis," in *The Art and Science of Analyzing Software Data*. Elsevier, 2015, pp. 17–38.

[46] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, ETAPS 2009*, ser. FASE '09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 370–384.

[47] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 496–506.

[48] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE Computer Society, 2009, p. 295–306.

[49] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing Library API Misuses Via Mutation Analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, vol. 2019-May. IEEE, may 2019, pp. 866–877.

[50] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating Next Steps in Static API-Misuse Detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, vol. 2019-May. IEEE, may 2019, pp. 265–275.

[51] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[52] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, may 2013, pp. 319–328.

[53] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion," *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 233–242, sep 2011.

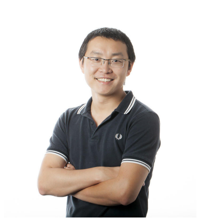
[54] T. Wang, K. Wang, X. Su, and P. Ma, "Detection of semantically similar code," *Frontiers of Computer Science*, vol. 8, no. 6, pp. 996–1011, Dec 2014.

[55] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2018.

[56] P. Liu, L. Li, Y. Yan, M. Fazzini, and J. Grundy, "Identifying and characterizing silently-evolved methods in the android api," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 308–317.

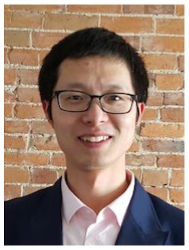
[57] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," vol. 48, no. 1, Sep. 2015.

[58] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among java neutral program variants," *Genetic Programming and Evolvable Machines*, vol. 20, no. 4, pp. 531–580, Dec 2019.



Weiyi Shang Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queen's University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-large–scale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang.

Maxime Lamothe Maxime Lamothe is an Assistant Professor at Polytechnique Montréal. Previously, he was a postdoctoral researcher in the Software REBELs lab at the University of Waterloo, Canada. He obtained his PhD and M.Eng. from Concordia University, Montreal, Canada, and his B.Eng. from McGill University. His research interests include software evolution, software APIs, software dependencies, software build systems, machine learning in software engineering, and empirical software engineering.



Heng Li Heng Li is an Assistant Professor in the Department of Computer and Software Engineering at Polytechnique Montreal, Canada, where he leads the Maintenance, Operations and Observation of Software with intelligencE (MOOSE) lab. He obtained his PhD from Queen's University (Canada), MSc from Fudan University (China), and BEng from Sun Yat-sen University (China). He also worked in industry for multiple years as a software engineer at Synopsys and as a software performance engineer at BlackBerry. His research interests include software monitoring and observability, intelligent operations of software systems, software log mining, software performance engineering, and mining software repositories. Contact him at: heng.li@polymtl.ca; https://www.hengli.org.