Mining API Constraints from Library and Client to Detect API Misuses

Hushuang Zeng, Jingxin Chen, Beijun Shen*, Hao Zhong
School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
{zenghushuang, chenjingxin, bjshen, zhonghao}@sjtu.edu.cn

***Abstract*—Calling Application Programming Interfaces (APIs) shall follow various constraints such as call orders, condition checking and exception handling. If they are incorrectly used, API misuses are introduced to code, and such misuses can cause severe bugs. To effectively detect API misuses, prior approaches mine constraints from client code, and assume that the violations of constraints are potential misuses. However, as client code only occurs a small portion of API usages and projects often call different APIs, constraints mined from client code are typically incomplete. As a result, when mined constraints are used to detect bugs, the real misuses which can be identified are limited and their violations are often false positives.**

Our research purpose is to find more misuses and reduce false positives. To achieve this goal, in this paper, we propose an approach that mines API constraints from both client and library. From client code, our approach builds API usage graphs and uses frequent subgraph mining algorithm to mine frequent usage patterns as API constraints. From library code, our approach analyzes the implementation of APIs and derives various types of constraints with our inferring strategies. After constraints are mined from both sources, they are integrated to detect API misuses with a graph matching algorithm. We compared our approach with MuDetect on its MuBench dataset. Our bi-source approach takes advantage from both the comprehensiveness and informativeness of library-based constraints and the accuracy of client-based patterns. As a result, it significantly improves the detection effectiveness of MuBench from 39.5% to 50.2% of the recall, and from 30.6% to 41.7% of the precision.

***Index Terms*—API misuse detection, API usage pattern, API constraint**

I. Introduction

Recently, more and more software systems are built upon third-party libraries and frameworks through their Application Programming Interfaces (APIs) to reduce the development cost [1]. When invoking APIs, programmers must follow their constraints to avoid API-related bugs. For example, before calling `next()` declared by the `Iterator` class, programmers must ensure that `hasNext()` returns true. If it is not checked, their code can throw `NoSuchElementException`. In this paper, we call the violations of API constraints as *API misuses*. When they occur, API misuses can cause programming errors, slow down code, or even introduce security vulnerabilities [2]–[5].

To mitigate API misuses, static analysis tools have been proposed to detect API misuses [3], [6]–[8]. Before a tool can detect these misuses, users must define API constraints in the form of the correct or incorrect usages of APIs. As there are so many APIs, it is tedious and often infeasible to manually define API constraints for all APIs. To improve the detection capability of their tools, researchers have proposed various approaches to mine API constraints from different sources.

One of the most intensively studied sources is client code (§II). Client code presents many instances of calling APIs. From client code, the prior approaches either use dynamic analysis to extract its traces [9]–[12], or use static analysis to extract its API usages [2]. Then they employ machine learning techniques [13] or frequent-itemset mining algorithms [6]–[8] to learn their API constraints. The assumption is that an API usage is likely a bug, if it is rare. Although this assumption is useful to detect many bugs [14], researchers [2] criticize that there are still many exceptions, and consequently these approaches cause a low recall and precision to detect API misuses. In addition, Zhong and Mei [15] show that it is difficult to collect sufficient client code for mining, especially for those newly released APIs and less popular APIs.

The other intensively studied source is API documents (§II). API documents define many API constraints, but they are often written in natural languages. To formalize those constraints, researchers [16] typically adopt various natural language processing techniques to analyze documents. However, in many libraries, programmers are not willing to write high-quality documents [17]. Consequently a lot of API constraints cannot be inferred correctly from their documents [16].

The final source is library code (§II). Library code covers all APIs, and presents their implementation details. Although it is informative, it is difficult to analyze library code, due to various technical limitations. As a result, the prior approaches can only infer simple constraints, *e.g.*, API call sets [18].

As each source has its own limitations, researchers start to explore the combinations of multiple sources (§II). Among them, a recent and interesting one is the combination of client code and library code [19]. In this work, Saied and Sahraoui combine the two sources to mine API call sets. For an API, its call set defines the APIs that are often called with this API. In practice, APIs have much more complicated usages than call sets. In addition, Saied and Sahraoui have not evaluated their approach to detect API misuses.

To further improve the prior approach [19], we propose a bi-source approach that mines API constraints from both client and library to detect API misuses. From client code, we extract API usage graphs (AUGs) [20] and mine frequent usage patterns. From library code, we analyze the implementation of each

* Beijun Shen is the corresponding author

API and infer various types of constraints such as conditions checking, call order and exception handling. Then, we combine the constraints from both sources, for taking advantage at the same time from the accuracy of client-based patterns and the comprehensiveness and informativeness of library-based constraints (see §III for discussion). To obtain a rich set of constraints, our combination algorithm merges the AUG sets of both-side constraints, and then extends the constraint set by generating more AUGs through modification. Our detector uses both-side constraints to detect misuses such as missing call, missing condition checking, and missing exception handling. We have released our tool and dataset on Github: https://github.com/subZHS/CL-Detector.

In summary, this paper makes the following contributions:

1) To the best of our knowledge, our approach is the first static API misuse detector whose constraints are mined from both client code and library code. With constraints mined from client, our detector is unlikely to identify correct usages as misuses, because they are mined from usages appearing in real code. With constraints mined from library, our detector can find more misuses violating uncommon constraints and reduce wrongly identified misuses with the comprehensiveness and informativeness of constraints from library.
2) In total, we design three strategies based on prior studies [21], [22] and five new strategies to infer more comprehensive constraints from library code. Our constraints contain fine semantic details, *i.e.* the precede/follow type of call-orders and the alternative relations between condition checking and exception handling.
3) We compared our approach with the state-of-the-art detector MuDetect [20] on its benchmark MuBench [23]. The results show that our approach achieves 41.7% in precision and 50.2% in recall. Both measures are better than those of MuDetect (30.6% and 39.5%, respectively).

II. Related Work

Researchers propose various approaches to mine API constraints from client code, library code, or API documents.

Mining from Client Code. Most of approaches mine API constraints from client code, in the formats of unordered call sets [24], call sequences [25] or precondtions [26]. There are many tools for detecting API misuses [6]–[8], [20], [27]–[33]. They leverage different types of API constraints mined from client code, like unordered call sets [27]–[29], call sequences [6], [8], [30], preconditions [31], [32], or program dependency graphs [7], [20], [33]. When their constraints are used to detect bugs, they assume that deviations/anomalies are API misuses. However, as client code typically only occurs a small portion of API usages [15], these approaches produce many false positives [2].

Inferring from Document. Some approaches analyze API documents using NLP techniques and adopt heuristic linguistic patterns to infer specific type of API constraints like call orders [34] or condition checking [35], [36]. A recent approach [37] detects API misuses using an API-constraint knowledge graph constructed from API documentation. Although detecting API misuses based on API documents are confirmed to be effective [37], it is restricted by the limitation of API documents. Sometimes API documents are not exhaustive or updated in time to conform to API implementation code [36], and their quality is usually not high [17].

Inferring from Library Code. Library code can explain why certain API usage occurs. Hence API constraints inferred from library are informative and have fine semantic details to accurately detect misuses. The prior approaches which infer API constraints from library code are based on dynamic analysis [38], [39], symbolic execution [40], [41], or static code analysis [18], [21], [22], [42]. However, as the task of library code analysis is very challenging, existing static approaches can only infer simple constraints, like unordered call sets [18], call order [22], preconditions [42], or API parameter constraints [21].

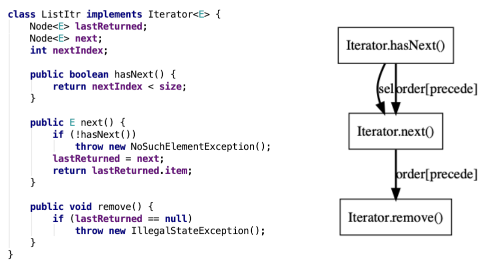
Inferring from Multiple Sources. Due to the limitation of each source, researchers start to explore the combinations of multiple sources. Zhong *et al.* [21] and Zhou *et al.* [42] mine API parameter constraints from documents and library code. Using these both-side constraints, Zhong *et al.* [21] conduct an empirical study to investigate rule types and how these rules distribute in both sources, while Zhou *et al.* [42] detect defects of API documents inconsistent with library code. Saied and Sahraoui [19] mine API call sets from both client code and library code. API call sets define APIs that are often called together, as one type of the common and simplest constraints. They have not yet made an evaluation on any application. Although the prior techniques only infer one type of API constraints (*e.g.*, parameter constraints [21], [42] and API call sets [19]) and have not use the inferred constraints to detect API misuses, the results still inspire us to mine richer constraints from multiple sources for API misuse detection.

As a comparison, our approach mines richer types of API constraints (condition checking, exception handling and call order) from both client code and library code to detect API misuses. Our combination algorithm efficiently merges both-side constraints through AUGs, and generates new alternative constraints. Using the resulting rich set of constraints, various types of API misuses are detected with the higher precision and recall, achieving the current state of the art. The effectiveness of our approach is contributed significantly by both the accuracy of client-based patterns and the comprehensiveness and informativeness of library-based constraints.

III. Motivating Examples

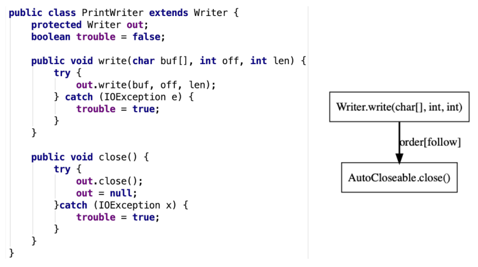
To improve the effectiveness of API misuse detection approaches, we analyze the pros and cons of constraints mined from client or library. We illustrate their limitations with some typical examples, and discuss how our approach overcome the limitations of prior approaches.

Comprehensiveness. Constraints mined from client code lack comprehensiveness, as we can only mine part of API constraints that occur frequently in client code. There are still many exceptions which they cannot cover. For example, Figure



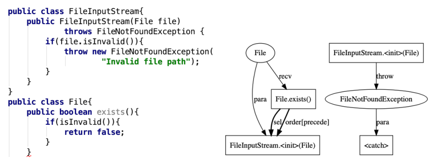
```java
class ListItr implements Iterator<E> {
    Node<E> lastReturned;
    Node<E> next;
    int nextIndex;

    public boolean hasNext() {
        return nextIndex < size;
    }

    public E next() {
        if (!hasNext())
            throw new NoSuchElementException();
        lastReturned = next;
        return lastReturned.item;
    }

    public void remove() {
        if (lastReturned == null)
            throw new IllegalStateException();
    }
}
```

Fig. 1: The Constraint of java.util.Iterator



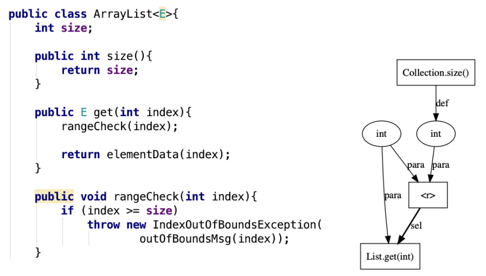
```java
public class PrintWriter extends Writer {
    protected Writer out;
    boolean trouble = false;

    public void write(char buf[], int off, int len) {
        try {
            out.write(buf, off, len);
        } catch (IOException e) {
            trouble = true;
        }
    }

    public void close() {
        try {
            out.close();
            out = null;
        }catch (IOException x) {
            trouble = true;
        }
    }
}
```

Fig. 2: The Constraint of java.io.PrintWriter



```java
public class FileInputStream{
    public FileInputStream(File file)
            throws FileNotFoundException {
        if(file.isInvalid()){
            throw new FileNotFoundException(
                    "Invalid file path");
        }
    }
}
public class File{
    public boolean exists(){
        if(isInvalid()){
            return false;
        }
    }
```

Fig. 3: The Constraint of java.io.FileInputStream/File



```java
public class ArrayList<E>{
    int size;

    public int size(){
        return size;
    }

    public E get(int index){
        rangeCheck(index);

        return elementData(index);
    }

    public void rangeCheck(int index){
        if (index >= size)
            throw new IndexOutOfBoundsException(
                    outOfBoundsMsg(index));
    }
```

Fig. 4: The Constraint of java.util.ArrayList

1(a) shows part of `java.util.Iterator` API's implementation in `java.util.LinkedList` class. Obviously, we should ensure that `hasNext()` returns true before calling `next()` or directly handle `NoSuchElementException`, and we should call `next()` before calling `remove()` to avoid `IllegalStateException` when `lastReturned` equals null. There are two call-order constraints: `hasNext()`$\rightarrow$`next()` and `next()`$\rightarrow$`remove()`, which can be represented as an AUG [20] like Figure 1(b) (§IV-A). The first one frequently occurs in client code and can be mined as a usage pattern, while `Iterator.remove()` is used less frequently and the second one may not be extracted. Besides, constraints mined from client code may be low-quality as their quality is highly dependent on the quality of client project. When there is not enough client code (*e.g.* newly released libraries, less-popular APIs, or custom APIs within projects), the corresponding constraints cannot be identified.

Our approach is a bi-source approach. Besides mining from client code, it also infers API constraints from library code no matter if APIs are frequently used or with enough client code, thus leading to relatively more comprehensive constraints. For Figure 1(a), we can infer `hasNext()`$\rightarrow$`next()` by identifying `hasNext()` in trigger condition of exception throwing statement in `next()`. We can infer `next()`$\rightarrow$`remove()` through tracking `lastReturned` field and identifying its assignment in `next()` and its nullcheck in exception trigger condition in `remove()`.

Informativeness. Constraints mined from client code are frequent usage patterns. They only represent the consequences of satisfying constraints, but they cannot reflect how to satisfy the constraints. Due to lack of sufficient information about constraints, it is possible to violate a pattern but satisfy corresponding real constraint, resulting in the misjudgment of misuses. For example, Figure 2(a) shows part of `java.io.PrintWriter` API's implementation. Empirically, the best practice is to call `close()` after calling `write()` to close the connection timely and reduce resource overhead. When detecting misuses based on a call-order pattern `write()`$\rightarrow$`close()`, only calling `close()` without calling `write()` will be wrongly regarded as misuses.

As the original source of constraints, library code has the explainability of why certain API usage occurs. Compared with patterns from client, constraints from library are informative and have fine semantic details to accurately detect API misuses, like precede/follow type of call-order and the alternative relations between condition checking and exception handling.

There are two types of call-order constraints: precede and follow. Our approach adopts different strategies to infer different call-orders from library code. In Figure 2(a), we can track the usage of `out` field and identify that the null assignment of `out` in `close()` should be executed after the method call on `out` in `write()`, which corresponds to a follow call-order: `write()`$\xrightarrow{follow}$`close()`, which means calling `write()` should be followed by calling `close()`. This is represented as an AUG like Figure 2(b). In contrast, in Figure 1(a) there are precede call-orders: `hasNext()`$\xrightarrow{precede}$`next()`$\xrightarrow{precede}$`remove()`.

Our approach can infer alternative relations of condition checking and exception handling constraints from library code. Client only needs to satisfy one of alternative constraints. Detecting misuses without considering alternative relations may lead to false positives [2]. For example, as we can handle the exception or check exception triggering condition before calling, from part of `FileInputStream` and `File`'s implementation in Figure 3(a), we can infer the alternative relation of checking `file.exists()` and handling `FileNotFoundException` when calling `FileInputStream(file)`. The AUGs are in Figure 3(b).

Accuracy. A constraint mined from client code (*i.e.*, a pattern) represents a correct usage that uses in a common way, and reflects accurate usages. The instances of such constraints are unlikely API misuses, since multiple pieces of real code use APIs in their described ways. For example, from part of `ArrayList`'s implementation in Figure 4(a), we can infer

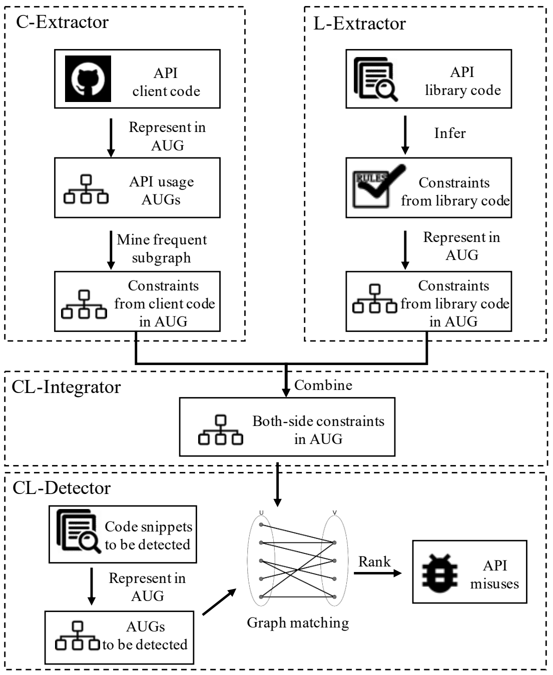


Fig. 5: An Overview of Our Approach

a condition checking constraint `list.size()>index` before calling `list.get(index)`, whose AUG is shown in Figure 4(b). In actual usage, the common practice is checking condition `list.isEmpty()` before calling `list.get(0)`, which is another way to satisfy the pre-condition of `list.get()` method.

In this sense, constraints inferred from library can lead to false positive, since there are various ways to use APIs but it is difficult to obtain all API usages from only library code. The purpose of our approach is to take advantage of both the comprehensiveness and informativeness of library-based constraints and the accuracy of client-based patterns.

IV. Our Approach

Figure 5 presents an overview of our approach, which contains four main components: 1) C-Extractor mines from client code (§IV-B); 2) L-Extractor infers from library code (§IV-C); 3) CL-Intergrator combines the results from above two components to produce API constraints (§IV-D), and 4) CL-Detector detects API misuses using above constraints (§IV-E). Mined constraints are in the format of AUGs (§IV-A).

A. Graph Representation

Compared to call sequences [43] and precondition sets [31], graphs are more informative to encode usage elements, structures, and data dependencies [2]. Meanwhile, graphs ignore some syntactical details and are less sensitive to local contexts [44]. As a result, we employ *API usage graphs (AUGs)* [20] to represent constraints. An AUG is a directed, connected multi-graph with labelled nodes and edges. Nodes represent data entities (objects, values, and literals) and usage actions (method calls, operators and instructions). Edges represent control and data flow between nodes, and are categorized into eight types, including receiver, parameters, definitions, orders, conditions, throws, handles, and synchronizes [20].

To represent API constraints more informatively, we modify AUGs, *i.e.*, adding a type attribute to each order edge to denote precede/follow orders of method calls. For example, the AUG in Figure 1(b) denotes the call-order constraints on `java.util.Iterator`. Boxes and ovals in the figure correspond to action nodes and data nodes. The "order[precede]" edge means calling `hasNext()` before calling `next()`. In contrast, the "order[follow]" edge in Figure 2(b) means calling `close()` after calling `write()`. In Figure 3(b), the recv (receiver) edge means the `exist()` method call is invoked on the `File` object, and the para (parameter) edge means the `FileInputStream()` constructor call takes the `File` object as a parameter. In Figure 4(b), the def (definition) edge means the method call `size()` returns a value to the `int` data node. The sel (condition) edge means the result of action <r> controls branching to the action `List.get()`. <r> is used to denote all equality and relational operators, and drop negation operators to abstract over different ways to express conditions.

With AUGs, our approach has two benefits: efficiently encoding API constraints from library and client, and accurately detecting the API misuses by comparing AUGs of to-be-checked code against AUGs of API constraints.

B. C-Extractor

After building AUGs from client code, C-Extractor mines frequent subgraphs of AUGs as API constraints. The key idea is that, the frequency of an API usage represents the certainty regarding the correctness of the usage. We employ MuDetect [20] to mine constraints from client code, which implements an apriori-based algorithm of frequent-subgraph mining [45].

When mining constraints, C-Extractor starts from call nodes of target APIs in AUGs, and then recursively extends to larger AUGs while ensuring enough frequent occurrences. In the extending process, it traverses all adjacent nodes of an AUG subgraph and ignores those connected only with order edges, since these nodes are irrelevant to usages of target APIs and often correspond to project-specific code. For example, an AUG subgraph, containing an order edge from a method-call node `hasNext()` to another method-call node `next()`, frequently occurs and will be mined as a call order constraint.

C. L-Extractor

For each target API, L-Extractor infers constraints from the library code. Based on the prior research work [21], [22] and our observations, we present eight strategies for inferring various constraints such as condition checking, exception handling and call orders. The inferring process requires four steps.

1) Determining analysis scopes: For each method, we first determine the analysis scope for constraint mining. If the method is declared by a class `c`, then the source code of `c` is added to the scope. When `c` is a subclass of class `s` and `c` overrides methods `M` of `s`, we add the unoverriden code (`s - M`) to the analysis scope. This is based on our observation that when programmers call methods from the instance of a class, they can call the methods that are declared by its super classes. For example, `add()` and `remove()` of `javax.swing.JPanel` are declared in its super class `java.awt.Container`, but the two methods are often call when the type of a variable is `javax.swing.JPanel`. For abstract methods, we infer their constraints from the implementations of their subclasses. For example, as shown in Figure 1, we infer the constraints of the interface `java.util.Iterator` through analyzing the implementation of its subclass `java.util.LinkedList$ListItr`.

2) Inferring Condition Checking Constraints: Condition checking constraints define that parameters of the method or status of the object shall be checked before calling a method on an object (e.g., nullchecks).

Strategy 1: Inferring from `assert`, `requireNonNull` and annotations. We scan the abstract syntax trees (ASTs) of target APIs' implementation code to locate the statements including `assert` keyword or the method call `Objects.requireNonNull()`. We extract conditions following the `assert` keyword as condition checking because they must be satisfied. For example, in the `File(String child, File parent)` constructor of the `java.io.File` class, an `assert` statement checks the condition `parent.path != null`. We extract null checks on parameters by identifying the parameters of `Objects.requireNonNull()` methods. For example, in the `removeAll(Collection<?> c)` method of the `java.util.ArrayList` class, the parameter `c` is checked against `null` through invoking `Objects.requireNonNull(c)`.

We also analyze annotations like `@param` in Javadoc. From sentences in `@param` or `@throws` annotations, we extract nullchecks on parameters by identifying keywords like `null` or elements wrapped by `<code>` tags. For example, in the `File(String parent, String child)` constructor of `java.io.File`, we derive that `child` should not be null through the sentence "NullPointerException If <code>child</code> is <code>null</code>" in the `@throws` annotation.

Strategy 2: Inferring from trigger conditions of `throw` statements. Based on the work of Zhong *et al.* [21], we infer condition checking with the following two sub-steps. We first identify all `throw` elements in target APIs' implementation code. Next, we extract trigger conditions of such `throw` statements through traversing `if` conditional branch statements that wrap them. For example, as shown in Figure 1, the condition checking of `hasNext()` is inferred through locating the `throw` statement of `NoSuchElementException`.

We then filter or convert the condition statements identified from `throw` statements and `assert` statements. Some checked conditions cannot be directly accessed from client code because they have externally inaccessible elements (private fields, private methods, or local variables). For private methods, we replace them with public methods that invoke such private methods and return the return value of corresponding private methods. For example, as shown in Figure 3, the private `isInvalid()` method is replaced with the public `exists()` method. For private fields, we convert them into their public `getter` methods if they exist. For a local variable, we attempt to simply convert it into the last assignment. If failure or the conditions still contain externally inaccessible elements, the condition checkings will be filtered. The above translations cannot be fully correct, but we make a trade off between the recall and accuracy of detecting API misuses.

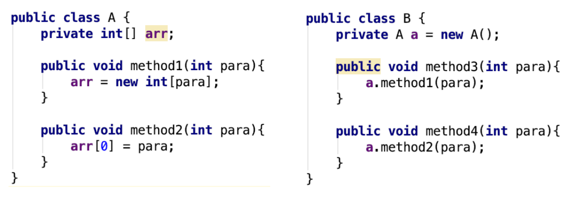
Strategy 3: Inferring from native methods. We manually define the parameter constraints for some frequent native methods, and those native methods are implemented in C and hard to analyze directly. For example, when accessing the value of an array (`arr[index]`), we should ensure that the condition, `index<arr.length`, is satisfied.

Strategy 4: Propagating condition checking through method calls. We propagate condition checking on parameters through method calls. In particular, given the `a` parameter of the `m1` method and the `b` parameter of the `m2` method, the condition checking on `b` is propagated to `a`, if `m1` calls `m2` and the value of `a` is passed to `b`. For example, in the `getLocationOnScreen(int[] location)` method of the `android.view.View` class, `location[0]` is accessed, and the condition checking `0<location.length` on the array is propagated to the parameter. For each method, we analyze methods calls within n depth, and propagate condition checking on parameters that are extracted by the previous three strategies.

3) Inferring Exception Handling Constraints: These constraints define the possible thrown exceptions of a method that need to handle while using the method.

Strategy 5: Inferring from `throw`, `throws` and annotations in Javadoc. Based on the inferring strategies in [21], we design to infer possible thrown exceptions by locating `throw` statements, `throws` keywords and annotations in Javadoc, in the ASTs of the target API methods and its calling methods within the range of calling depth n. For example in Figure 3, we can infer `FileNotFoundException` of `FileInputSteam()` method through identifying `throws` in the method declaration statement or locating the `throw` statement. We can also obtain exceptions by extracting the starting words in the annotations like `@throws, @exception` in Javadoc. For the same example about the constructor `File(String parent, String child)` of `java.io.File` API in the first strategy of condition checking constraints, we extract `NullPointerException` from the sentence "NullPointerException If <code>child</code> is <code>null</code>" in the `@throws` annotations.

Strategy 6: Inferring alternative relations with condition checking. Exception handling constraints often have condition checking constraints as their alternative constraints. Thrown exceptions always have corresponding trigger conditions, and client code using APIs can choose to handle exceptions or check exception trigger conditions to avoid the exceptions. Therefore, we establish the alternative relations of exception handling and condition checking constraints in the process of Strategy 2. For example in Figure 3, the alternative constraint of handling `FileNotFoundException` is the condition checking `file.exists()`. Considering alternative relations of constraints are conducive to reduce the false positive rate and improve the precision of misuse detection [20], [46]. We leave the other alternative constraints to our future work.



```java
public class A {
    private int[] arr;

    public void method1(int para){
        arr = new int[para];
    }

    public void method2(int para){
        arr[0] = para;
    }
}
```

```java
public class B {
    private A a = new A();

    public void method3(int para){
        a.method1(para);
    }

    public void method4(int para){
        a.method2(para);
    }
}
```

(a) (b)

Fig. 6: An Example of Call Order Propagation

4) Inferring Call Order Constraints: These constraints define orders of method calls.

Strategy 7: Inferring from call graphs. Based on the work of Zhong *et al.* [22], we infer call orders with three sub-steps: First, we build an inter-method call relation graph whose entries are the API methods of interest, and traverse methods called by them in n depth. Second, we find precede/follow call orders (initializing primitive variable before using, calling constructors to initialize an object field before using the methods in the object, and assigning an object field to `null` to reclaim the object space after using the field) for all called methods. Finally, we push the call orders back to the API methods.

For a simple and straightforward example in Figure 6, since the array `arr` should be initialized before using, we can find the call order, `method1()`$\xrightarrow{precede}$`method2()` in class `A`. Owing to `method3()` invoking `method1()` and `method4()` invoking `method2()`, this call order is propagated along the call relation graph to class `B` and becomes the call order `method3()`$\xrightarrow{precede}$`method4()`.

Strategy 8: Inferring from method names. Based on our experiences and observations, we exploit two heuristic naming rules to infer call orders. The first heuristic rule infers call orders about iterations like `hasNextLine()` $\xrightarrow{precede}$ `nextLine()`, which applies to `Iterator`, `Scanner`, `StringTokenizer` and so on. The second heuristic rule infers call orders about resource usages like `write()` $\xrightarrow{follow}$ `close()`, which applies to `DataOutputStream`, `ByteOutputStream` and so on.

D. CL-Intergrator

After mining constraints from client and library, CL-Intergrator combines them to obtain a rich set of constraints. The resulting set of constraints are comprehensive and informative. Algorithm 1 shows the major steps:

1) Representing All Constraints as AUGs: Firstly, we represent all constraints in the format of AUGs. For each constraint inferred from library, we construct an API usage as client code and then build its AUG (Line 2). For example, for the condition checking constraint `con` of an API method `method()`, we construct a usage code snippet `if(con){method();}` and then build its AUG.

2) Merging Constraints from Both Sources: If constraints are overlapped, we deliver the fine semantic details of library-based constraints to client-based constraints (Lines 3-6), such as adding the precede/follow attribute of a call order constraint to the corresponding order edge of client-based constraint AUG. Next, we directly take the union of AUG collections of constraints from client and from library (Lines 7-8).

Algorithm 1: CL-Intergrator

```
   Input: $clientConAugs_m$ /* client patterns of $m$ */
1  $libCons_m$ /* library constraints of method $m$ */
   Output: $fullConAugs_m$ /* full constraints of $m$ */
   /* Step 1: represent $libCons_m$ as AUGs */
2  AUG[] $libConAugs_m$ = buildAUGForLibCons($libCons_m$);
   /* Step 2: enhance client constraints and merge
      $libConAugs_m$ and $clientConAugs_m$ */
3  for $clientConAug_i \in clientConAugs_m$ do
4      for $libConAug_j \in libConAugs_m$ do
5          if findOverlap($clientConAug_i$, $libConAug_j$)!=null then
6              enhanceconstraint($clientConAug_i$, $libConAug_j$);
7  AUG[] $fullConAugs_m$ = Arrays.copyOf($clientConAugs_m$);
8  $fullConAugs_m$.addAll($libConAugs_m$);
   /* Step 3: build new alternative constraints */
9  for $clientConAug_i \in clientConAugs_m$ do
10     for $type \in [ConditionChecking, ExceptionHandling]$ do
11         $subAug$ = findSubAUG($clientConAug_i$, $type$);
12         $newconstraintAug$ = replaceWithLibCons($subAug$,
               $libConAugs_m$);
13         $fullConAugs_m$.add($newconstraintAug$);
```

3) Generating New Alternative Constraints: Finally, to improve the comprehensiveness and richness of constraints, we modify constraints from client according to constraints from library (Lines 9-13). In particular, if a subgraph in a constraint from client code is related to condition checking or exception handling, we replace the subgraph with the corresponding constraint from library, and generate a new constraint which has the alternative relation with the original one.

E. CL-Detector

CL-Detector detects misuses with our inferred constraints. We design and implement misuse detection based on the graph matching algorithm in MuDetect [20]. First, the code under check is represented as AUGs. A piece of code is regarded as a potential misuse, if its AUG is a violation of a constraint and matches none of its alternative constraints. Given an AUG of the code under check, a violation of a constraint is a mismatch between this AUG and the AUG of this constraint. All misuses are ranked by their suspicious scores. Considering the ranking strategy used by MuDetect [20], the suspicious score is calculated as follows:

$$score = (c_s + s_l)/v_s \cdot v_d, \quad v_d = n_m/n_c \tag{1}$$

The support c_s is the occurrence number of a constraint in the client code, which indicates the correctness of the constraint. The s_l denotes the initial weight of constraints comes that are inferred from a library. After trials, we set the weight s_l as one. If a constraint is only mined from client code, s_l is set to zero. The violation support v_s is the occurrence number of a violation, which indicates the rareness of the violation. The v_d measures the distance between a violation and the violated constraint, and is calculated through the number of missing nodes from the violated constraint n_m divided by the total number of nodes in the violated constraint n_c.

To accurately detect violations of constraints, we integrate fine semantic details of constraints from library, including precede/follow type of call orders and alternative relations between condition checkings and exception handlings. A target AUG is not a violation of $\texttt{a()} \xrightarrow{precede} \texttt{b()}$ when it only calls `a()` without calling `b()`, and is not a violation of $\texttt{a()} \xrightarrow{follow} \texttt{b()}$ when it only calls `b()` without calling `a()`. A target AUG is violation of a condition checking constraint but is not regarded as a misuse when it matches one of alternative constraints like a exception handling constraint.

V. Experiment

We have implemented an API constraint miner and an API misuse detector upon MuDetect [47] and the Eclipse JDT toolkit [48]. With our detector, we conduct several experiments with the aims of investigating three research questions:

- **RQ1:** How effectively does our approach perform compared with the state-of-the-art detector MuDetect [20]?
- **RQ2:** How is the quality of constraints inferred from library source code?
- **RQ3:** How do the constraints mined from library and client contribute to the effectiveness of our approach?

Dataset. We reuse the API misuse dataset, MuBench [49], to evaluate the effectiveness of our API misuse detection. MuBench is a state-of-the-art benchmark, and it is actively maintained. It is widely used by other API misuse detection studies [12], [20], [37], [50]. After removing some projects without available urls, the latest version of MuBench contains 223 API misuses in 57 software projects, involving the usage of 65 APIs. These misuses in MuBench consist of 110 missing condition checking, 26 missing exception handling, 81 missing call and 6 having redundant elements. In our evaluation, we select 106 Java APIs as our subjects, including the 65 APIs used in MuBench and 68 Java APIs that are analyzed in a recent study [46]. For each target API, we collect top 20 client projects that used the certain class the most on GitHub through Boa tool [51]. By employing our miner, we have mined 1,092 constraints from client code and 12,538 constraints from library code, which consist of 712 condition checking, 3,432 exception handling and 8,394 call order constraints.

A. *RQ1: The Effectiveness of API Misuse Detection*

Experimental Setup. We compare our API misuse detector against the state-of-the-art detection approach, MuDetect [20], on its MuBench [23] dataset. The latest version of MuBench contains 223 API misuse instances in 57 software projects after removing some projects with unavailable urls. We set the frequency threshold σ as five.

We evaluate MuDetect in its cross-project setting, which mines patterns from multiple client projects we collect. To mine constraints across projects, we extract client code from multiple projects that frequently call our target APIs. In particular, for each target API, we collect API usage code snippets from the top 20 Github projects that call our target APIs most frequently with the Boa tool [51]. We use the recall, precision and F1 score to measure the effectiveness of our approach, where F1 score is the harmonic mean between recall and precision.

TABLE I: Effectiveness of Detectors

Detectors	Recall	Precision				F1
		Pre1	Pre2	PreF	Kap.	
MuDetect	39.5%	28.6%	32.7%	30.6%	0.90	34.5%
Our detector	50.2%	40.0%	44.3%	41.7%	0.88	45.6%

* Pre1 and Pre2 are the precision values by the two annotators independently; PreF is the final precision value after resolving inconsistent decisions; and Kap. is the Kappa value of inter-rater agreement.

These metrics are commonly used in software bug detection researches.

The recall examines how many ground-truth misuses in MuBench are correctly detected by the detector. We manually judge whether detected ground-truth misuses are caused by violation of the mined constraints, guided by descriptions of misuses. If confirmed, these ground-truth misuses are correctly detected. The precision measures how many true positive misuses are among top-ranked detected misuses. We need to manually review all detected misuses to determine whether they are true positive, because these projects may exist other API misuses in addition to ground-truth misuses in MuBench. Reviewing all detected misuses on all projects in MuBench is practically infeasible. Therefore, we calculate the precision on the ten projects sampled by Amann *et al.* [20]. We manually examine top-20 detected misuses for each project. The ground-truth misuses in MuBench have proven to be true positive misuses with no need for manual inspection. In this experiment, all manual annotation or checking is conducted by two of authors independently, with their inconsistent results discussed to reach a consensus. We employ Cohen's Kappa [52] to measure the inter-rater agreement.

Results. Table I shows the effectiveness evaluation results of our detector and MuDetect. Our detector achieves 45.6% in F1 score, and our result is higher than 34.5% of MuDetect.

Recall Result: From 223 API misuse instances in MuBench (including 110 missing condition checking, 26 missing exception handling and 81 missing call), our detector detects 112 API misuse instances, including 36 missing condition checking, 21 missing exception handling and 55 missing call. Our detector achieves 50.2% in recall, higher than 39.5% of MuDetect. The result shows that our constraints mined from both client and library are more comprehensive than only one source. Their contributions will be analyzed in RQ3.

Those misuses in MuBench undetected by our detector can be attributed to two reasons. First, there are some misuses due to having redundant elements should be removed, but only misuses missing elements might be detected by our detector. For example, the misuse #2 of project "drftpd3-extended" in MuBench incorrectly uses `javax.crypto.Cipher` API owing to multiple calls of method `init()`. Hence, how to improve misuse detection algorithm to detect redundant elements is still a problem worth to explore. Second, rest undetected misuses are due to that corresponding violated constraints can not be mined by our approach. For example, the misuse #1 of project "yapps" in MuBench is considered a bad practice owing to using “AES” as the actual parameter of `getInstance(String)` method of `java.crypto.Cipher` API. This constraint cannot be mined because it is not defined directly in library code.

Precision Result: The Cohen’s Kappa values of two annotators are all above 0.60, which indicates that they reach substantial agreement. Our detection approach achieves 41.7% in precision, higher than 30.6% of MuDetect. The result shows that our constraints inferred from both client and library are more effective for accurate detection than only from one source, because our constraints are informative and have fine semantic details, like the precede or follow attribute of call order. For example, in project “lucene”, there are some code snippets calling only `hasNext()` without calling `next()`. They are falsely regarded as misuses by MuDetect, due to violating the pattern $hasNext() \rightarrow next()$. With the precede attribute in the corresponding constraint $hasNext() \xrightarrow{precede} next()$ of `Iterator` API, our detector would not regard them as misuses.

There are some false positive misuses roughly caused by two reasons. First, some false positive misuses are due to lack of inter-procedural analysis. The missing elements of these false positive misuses occur in other methods that target code snippets calls. For example, in project “lucene”, a code snippet is falsely regarded as a violation of the call order constraint $read() \xrightarrow{follow} close()$ of `java.io.FileInputStream` API, and `close()` method is called in another method the code snippet calls. Future work can explore how to introduce inter-procedural analysis in misuse detection process to balance the performance and computational cost. Second, some false positive misuses are due to incorrect or incomplete constraints. Some condition checking constraints and their alternative relations with exception handling constraints cannot be inferred. For example, in project “itext”, some code snippets are falsely regarded as misuses owing to violating a exception handling constraint about `NoSuchElementException` in method `nextToken()` of `java.util.StringTokenizer` API. They actually match its alternative condition checking constraint `hasMoreElements()` which could not be inferred, because the corresponding exception trigger condition has private fields difficult to convert. In future work, we will extend our strategies for a lower rate of false positive misuses.

Answer to RQ1. Our detector is more effective than MuDetect. With relatively more comprehensive and accurate API constraints mined from both client and library, it finds more API misuses and reduces wrongly identified misuses.

B. RQ2: The Quality of Constraints Inferred from Library

Experimental Setup. We evaluate the quality of constraints inferred from library source code, which reflects the effectiveness of constraint inferring strategies and affects the performance of API misuse detection using them. Due to the large number of constraints, we randomly sample API methods and manually examine their inferred constraints. We invite two programmers who have more than 3 years of Java development experience to independently examine the quality of constraints mined from library. We employ Cohen’s Kappa [52] to measure the inter-rater agreement. When the two annotators’ decisions are inconsistent, they will discuss and reach a consensus to make final decisions. The two annotators determine a constraint is true, if it appears in library code or API documentation. For example, as shown in Section III, it is feasible to determine whether a constraint is correct according to API documents and their code.

TABLE II: The Quality of Constraints Inferred from Library

Type	Recall				Accuracy			
	Rec1	Rec2	RecF	Kap.	Acc1	Acc2	AccF	Kap.
Con.	60.9%	63.8%	61.9%	0.89	92.3%	95.4%	92.3%	0.73
Exp.	100.0%	100.0%	100.0%	1.00	100.0%	100.0%	100.0%	1.00
Call.	-	-	-	-	66.7%	69.3%	68.7%	0.84
Avg.	80.5%	81.9%	81.0%	0.95	86.3%	88.2%	87.0%	0.86

* Con. refers to condition checking; Exp. refers to exception handling; Call. refers to call order.
* Rec1 and Rec2 are the recall by the two annotators independently; RecF is the final recall after resolving inconsistent decisions; and Kap. is the Kappa value of inter-rater agreement. Acc1, Acc2 and AccF of accuracy have similar meanings.

We select recall and accuracy to measure the quality of constraints. The recall examines how many actual constraints are inferred among all actual constraints. We only calculate recall of two types of constraints except call order, because all actual call-order constraints of complicated classes are difficult to determine even by manually analyzing library code. The accuracy measures how many actual constraints are inferred among all inferred constraints. We calculate recall and accuracy of each type of constraints and take the average.

Results. We evaluate 360 constraints of 100 randomly sampled API methods, including 65 condition checking, 145 exception handling and 150 out of total 324 call order constraints. Table II shows the recall and accuracy results of these constraints. The Cohen’s Kappa values are all above 0.60, which indicates that two annotators reach substantial agreement. The results demonstrate that constraints inferred from library achieve high accuracy of 87.0% in average (92.3% for condition checking, 100.0% for exception handling and 68.7% for call order) and high recall of 81.0% in average (61.9% for condition checking and 100% for exception handling). However, our constraint inferring strategies still has potential for improvement to infer more comprehensive and accurate constraints.

Recall Result: Exception handling constraints achieve a perfect recall of 100%, because the exception implementation is relatively distinguishable and easy to identify, such as through *throw* statements, *throws* keywords and annotations like *@throw* in Javadoc. On the contrary, the recall of condition checking constraints is lower due to the complexity and diversity of condition checking implementation. There are some complicated cases of condition checking, like that condition expressions might include externally inaccessible elements (private fields, private methods, or local variables) that hard to convert. For example, for method `writeUTF(String str, DataOutput out)` in `DataOutputStream` API, we cannot convert the local variable `utflen` in the precondition `utflen <= 65535`, because `utflen`, the length of utf-8 encoded string,

TABLE III: The Contributions of Constraints

	Con.	Exp.	Call.	Total	Recall	Drops by
All	36	21	55	112	50.2%	-
All - L-Extractor	31	9	48	88	39.5%	-
All - C-Extractor	18	19	47	84	37.7%	-
- Strategy 1	15	19	47	81	36.3%	3.7%
- Strategy 2	5	19	47	71	31.8%	15.6%
- Strategy 3	16	19	47	82	36.7%	2.7%
- Strategy 4	9	19	47	75	33.6%	8.2%
- Strategy 5	18	0	47	65	29.1%	22.8%
- Strategy 6	18	19	47	84	37.7%	0%
- Strategy 7	18	19	21	58	26.0%	31.0%
- Strategy 8	18	19	36	73	32.7%	13.3%

* Con. refers to condition checking; Exp. refers to exception handling; Call. refers to call order.

is obtained by the complicated calculation of traversing all characters of the parameter `str`. Although the recall of call order constraints is not evaluated, we inspect a small amount of APIs and find the main reason for undiscovered call orders. These call orders are only defined by the responsibilities or functions of APIs, and they do not necessarily have the identified forms like the objects to be used must be created first. For example, in the description of the misuse #475 of project "tbuktu-ntru" in MuBench, it violates a call order without the identified forms: `DataOutputStream.close()` $\xrightarrow{precede}$ `ByteArrayOutputStream.toByteArray()`.

Accuracy Result: Exception handling constraints achieve a perfect accuracy of 100%, because the exception implementation is relatively distinguishable and hence easy to accurately identify. Condition checking constraints also achieve a high accuracy of 92.3%, since our condition inferring strategies try to ensure the accuracy and filter out constraints unable to meet the specifications like existing tricky externally inaccessible elements in condition expressions. There are a few incorrect condition checking constraints owing to the simple extraction from Javadoc. For example, for method `removeAll(Collection<?> c)` in `ArrayList` API, we falsely extract a nullcheck of the parameter `c` from the sentence in Javadoc @*throws* "if this list contains a null element". However, the 68.7% accuracy of call order constraints is lower due to some complicated situations difficult to identify and handle. The identified order form (the objects to be used must be created first) might be invalid when the objects are checked before used, and this case is difficult to identify. For example, in `LinkedList` API the inferred call order `offer(e)`$\xrightarrow{precede}$`peekLast()` is incorrect, because in `peekLast()` it returns null when the `last` field is checked to be null.

Answer to RQ2. Our strategies infer relatively comprehensive and accurate constraints of high quality from library code, which are significant for API misuse detection.

C. RQ3: The Contributions of Constraints

Experimental Setup. To explore the contributions of our components, we conduct an ablation study. In particular, we disable our components, and calculate the recall values of the following settings: All - L-Extractor (omit the L-Extractor component), All - C-Extractor (omit the C-Extractor component), and All - C-Extractor- Strategy * (omit the C-Extractor component and a certain strategy). In this study, we analyze how many real API misuses are detected when a specific component is missing.

Results. Table III shows the contribution result of each critical component. Our detector achieves the recall values of 50.2%, 39.5% and 37.7% using all components, omitting L-Extractor and omitting C-Extractor, respectively. The result shows that C-Extractor and L-Extractor make non-negligible contributions. There are 28 and 24 API misuses in MuBench that could be detected by only client-based constraints from C-Extractor or library-based constraints from L-Extractor. The result shows that the complements complement each other. Compared with C-Extractor, L-Extractor helps detect more missing-exception-handling misuses but fewer missing-condition-checking misuses, owing to the high recall of exception handling constraints and the relatively lower recall of condition checking constraints in RQ2. Compared with other strategies, omitting Strategies 7, 5 and 2 produces poorer recall values. This result indicates that these strategies are more important and find more misuses in MuBench than other strategies.

For those misuses only detected by library-based constraints, they are violations of uncommon constraints which cannot be obtained by mining frequent usage patterns from client code. We can infer comprehensive constraints from library code no matter if they frequent occur. For example, the misuse #1 of project "itext" in MuBench is only detected by library-based constraints. It violates the exception handling constraint that usages have to handle `InvalidKeyException` when using `init()` method of `javax.crypto.Cipher` API.

For those misuses only detected by client-based constraints, they violate constraints that can be mined expediently from many client code snippets. These client-based constraints are difficult to infer as library-based constraints due to their sophisticated implementation in library code. For example, the misuse #361 of project "jodatime" in MuBench is only detected by client-based constraints. It lacks of the condition checking `countTokens()>0` before calling `nextToken()` method of `java.util.StringTokenizer` API. In the implementation of `nextToken()`, the corresponding condition contains private fields difficult to convert and thus our strategies can not infer.

Answer to RQ3. The effectiveness of our detector is attributed to both the accuracy of client-based constraints and the comprehensiveness and informativeness of library-based constraints.

D. Threats to Validity

Internal Validity. The library-based constraints may be in danger of overfitting to the target APIs. To mitigate this threat, the target APIs are commonly used and hence are sufficiently representative. Besides, we collect some strategies adopted by other studies [21], [22], [42] and summarize common strategies that are applicable to most APIs. The accuracy of manual evaluation results may be limited by the ability of annotators. To mitigate this threat, two annotators with more than 3 years of Java development experience are invited to make decisions independently and reach a consensus for inconsistencies.

External Validity. The dataset of API misuses used for evaluation may not be representative. To mitigate this threat, we use the latest version of the state-of-the-art dataset MuBench which is widely used by API misuse studies [12], [20], [37], [50]. More misuses from real projects will be taken to confirm the capability of our approach in the future.

VI. Conclusion and Future Work

In this paper, we propose a bi-source approach which mines API constraints from both client and library for API misuse detection. Using both-side constraints, it accurately detects various types of API misuses such as missing call, missing condition checking and missing exception handling. The evaluation results show that our approach significantly outperforms the state-of-the-art detector, MuDetect. The quality of constraints inferred from library is high, and their contribution has also been demonstrated to be significant. We have open-sourced our API constraint miner and API misuse detector to benefit both the user and research communities.

In the future, we will improve mining technique for more comprehensive API constraints, like introducing Stack Overflow posts as a new source and raising new constraint inferring strategies. We will explore inter-procedural analysis to enhance our misuse detection algorithm. It is also necessary to evaluate the capability of our detector on more real-world projects. In addition, although our implementation analyzes only Java code, our strategies shall work on other languages. We leave the extensions to more languages to our future work.

Acknowledgement

This research is supported by National Natural Science Foundation of China (Grant No. 62032004).

References

[1] Y. Wang and et al., "Do the dependency conflicts in my project matter?" in *Proc. ESEC/FSE*, 2018, pp. 319–330.
[2] S. Amann and et al., "A systematic evaluation of static API-misuse detectors," *Proc. ESEC/TSE*, vol. 45, no. 12, pp. 1170–1188, 2019.
[3] M. Monperrus and et al., "Detecting missing method calls as violations of the majority rule," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 7:1–7:25, 2013.
[4] J. Sunshine and et al., "Searching the state space: a qualitative study of API protocol usability," in *Proc. ICPC*, 2015, pp. 82–93.
[5] M. Egele and et al., "An empirical study of cryptographic misuse in android applications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013, pp. 73–84.
[6] A. Wasylkowski and et al., "Detecting object usage anomalies," in *Proc. ESEC/FSE*, 2007, pp. 35–44.
[7] T. T. Nguyen and et al., "Graph-based mining of multiple object usage patterns," in *Proc. ESEC/FSE*, 2009, pp. 383–392.
[8] A. Wasylkowski and et al., "Mining temporal specifications from object usage," *Proc. ASE*, vol. 18, no. 3-4, pp. 263–292, 2011.
[9] M. Pradel and et al., "Automatic generation of object usage specifications from large method traces," in *Proc. ASE*, 2009, pp. 371–382.
[10] O. Legunsen and et al., "How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications," in *Proc. ASE*, 2016, pp. 602–613.
[11] M. Pradel and et al., "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proc. ICSE*, 2012, pp. 288–298.
[12] M. Wen and et al., "Exposing library API misuses via mutation analysis," in *Proc. ICSE*, 2019, pp. 866–877.
[13] M. P. Robillard and et al., "Automated API property inference techniques," *Proc. ESEC/TSE*, vol. 39, no. 5, pp. 613–637, 2012.
[14] A. Wasylkowski and et al., "Mining temporal specifications from object usage," *Proc. ASE*, vol. 18, no. 3, pp. 263–292, 2011.
[15] H. Zhong and et al., "An empirical study on API usages," *Proc. ESEC/TSE*, vol. 45, no. 4, pp. 319–334, 2017.
[16] H. Zhong and et al, "Inferring resource specifications from natural language API documentation," in *Proc. ASE*, 2009, pp. 307–318.
[17] G. Uddin and et al., "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.
[18] M. A. Saied and et al., "Could we infer unordered API usage patterns only using the library source code?" in *Proc. ICPC*, 2015, pp. 71–81.
[19] ——, "A cooperative approach for combining client-based and library-based API usage pattern mining," in *Proc. ICPC*, 2016, pp. 1–10.
[20] S. Amann and et al., "Investigating next steps in static API-misuse detection," in *Proc. MSR*, 2019, pp. 265–275.
[21] H. Zhong and et al., "An empirical study on API parameter rules," in *Proc. ICSE*, 2020, pp. 899–911.
[22] ——, "Inferring specifications of object oriented APIs from API source code," in *Proc. APSEC*, 2008, pp. 221–228.
[23] S. Amann and et al., "Mubench: a benchmark for API-misuse detectors," in *Proc. MSR*, 2016, pp. 464–467.
[24] M. A. Saied and et al., "Mining multi-level API usage patterns," in *Proc. SANER*, 2015, pp. 23–32.
[25] H. H. Kagdi and et al., "An approach to mining call-usage patterns with syntactic context," in *Proc. ASE*, 2007, pp. 457–460.
[26] H. A. Nguyen and et al., "Mining preconditions of APIs in large-scale code corpus," in *Proc. ESEC/FSE*, 2014, pp. 166–177.
[27] Z. Li and et al., "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proc. ESEC/FSE*, 2005, pp. 306–315.
[28] C. Lindig, "Mining patterns and violations using concept analysis," in *The Art and Science of Analyzing Software Data*, 2015, pp. 17–38.
[29] M. Monperrus and et al., "Detecting missing method calls in object-oriented software," in *ECOOP 2010 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 6183, 2010, pp. 2–25.
[30] M. K. Ramanathan and et al., "Path-sensitive inference of function precedence protocols," in *Proc. ICSE*, 2007, pp. 240–250.
[31] S. Thummalapenta and et al., "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proc. ASE*, 2009, pp. 283–294.
[32] M. K. Ramanathan and et al., "Static specification inference using predicate mining," in *Proc. PLDI*, 2007, pp. 123–134.
[33] S. Thummalapenta and et al., "Mining exception-handling rules as sequence association rules," in *Proc. ICSE*, 2009, pp. 496–506.
[34] H. Zhong and et al., "Inferring resource specifications from natural language API documentation," in *Proc. ASE*, 2009, pp. 307–318.
[35] R. Pandita and et al., "Inferring method specifications from natural language API descriptions," in *Proc. ICSE*, 2012, pp. 815–825.
[36] Y. Zhou and et al., "Automatic detection and repair recommendation of directive defects in Java API documentation," *Proc. ESEC/TSE*, vol. 46, no. 9, pp. 1004–1023, 2020.
[37] X. Ren and et al., "API-misuse detection driven by fine-grained API-constraint knowledge graph," in *Proc. ASE*, 2020, pp. 461–472.
[38] C. Ghezzi and et al., "Synthesizing intensional behavior models by graph transformation," in *Proc. ICSE*, 2009, pp. 430–440.
[39] J. Henkel and et al., "Discovering documentation for Java container classes," *Proc. ESEC/FSE*, vol. 33, no. 8, pp. 526–543, 2007.
[40] R. P. L. Buse and et al., "Automatic documentation inference for exceptions," in *Proc. ISSTA*, 2008, pp. 273–282.
[41] N. Tillmann and et al., "Discovering likely method specifications," in *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods (ICFEM)*, vol. 4260, 2006, pp. 717–736.
[42] Y. Zhou and et al., "Analyzing APIs documentation and code to detect directive defects," in *Proc. ICSE*, 2017, pp. 27–37.
[43] S. Wang and et al., "Bugram: bug detection with n-gram language models," in *Proc. ASE*, 2016, pp. 708–719.
[44] X. Gu and et al., "Codekernel: A graph kernel based approach to the selection of API usage examples," in *Proc. ASE*, 2019, pp. 590–601.
[45] T. Ramraj and et al., "Frequent subgraph mining algorithms–a survey," *Procedia Computer Science*, vol. 47, pp. 197–204, 2015.
[46] T. Zhang and et al., "Are code examples on an online Q&A forum reliable? a study of API misuse on stack overflow," in *Proc. ICSE*, 2018, pp. 886–896.
[47] MUDetect. [Online]. Available: https://github.com/stg-tud/MUDetect
[48] Eclipse JDT. [Online]. Available: https://www.eclipse.org/jdt/
[49] MuBench. [Online]. Available: https://github.com/stg-tud/MUBench

[50] S. Nielebock and et al., “Cooperative API misuse detection using correction rules,” in *42nd International Conference on Software Engineering, New Ideas and Emerging Results (ICSE-NIER)*, 2020, pp. 73–76.

[51] R. Dyer and et al., “Boa: a language and infrastructure for analyzing ultra-large-scale software repositories,” in *Proc. ICSE*, 2013, pp. 422–431.

[52] J. R. Landis and et al., “An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers,” *Biometrics*, pp. 363–374, 1977.