API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph

Xiaoxue Ren[*][†]
Zhejiang University
China
xxren@zju.edu.cn

Xinyuan Ye
Australian National University
Australia
u6296255@anu.edu.au

Zhenchang Xing[‡]
Australian National University
Australia
zhenchang.Xing@anu.edu.au

Xin Xia[§]
Monash University
Australia
xin.xia@monash.edu

Xiwei Xu
Data61, CSIRO
Australia
Xiwei.Xu@data61.csiro.au

Liming Zhu[¶]
Data61, CSIRO
Australia
Liming.Zhu@data61.csiro.au

Jianling Sun
Zhejiang University
China
sunjl@zju.edu.cn

ABSTRACT

API misuses cause significant problem in software development. Existing methods detect API misuses against frequent API usage patterns mined from codebase. They make a naive assumption that API usage that deviates from the most-frequent API usage is a misuse. However, there is a big knowledge gap between API usage patterns and API usage caveats in terms of comprehensiveness, explainability and best practices. In this work, we propose a novel approach that detects API misuses directly against the API caveat knowledge, rather than API usage patterns. We develop open information extraction methods to construct a novel API-constraint knowledge graph from API reference documentation. This knowledge graph explicitly models two types of API-constraint relations (call-order and condition-checking) and enriches return and throw relations with return conditions and exception triggers. It empowers the detection of three types of frequent API misuses - missing calls, missing condition checking and missing exception handling, while existing detectors mostly focus on only missing calls. As a proof-of-concept, we apply our approach to Java SDK API Specification. Our evaluation confirms the high accuracy of the extracted API-constraint relations. Our knowledge-driven API misuse detector achieves 0.60 (68/113) precision and 0.28 (68/239) recall for detecting Java API misuses in the API misuse benchmark MuBench. This performance is significantly higher than that of existing pattern-based API misused detectors. A pilot user study with 12 developers shows that our knowledge-driven API misuse detection is very promising in helping developers avoid API misuses and debug the bugs caused by API misuses.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories.*

ACM Reference Format:
Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416551

1 INTRODUCTION

Software libraries provide reusable functionalities through Application Programming Interfaces (APIs). APIs often come with usage caveats, such as constraints on call order or value/state conditions. For example, when using the Iterator in Java, one should check that hasNext() returns true (i.e., the iteration has more elements) before calling next(), to avoid NoSuchElementExcpetion. Applications that fail to follow these caveats (i.e., misuse APIs) may suffer from bugs. [3, 9–11, 24, 37]. There are many pattern-based tools for detecting API misuses by static code analysis [2, 17, 18, 24–26, 30, 41–44]. All these methods mine API usage patterns from a codebase, and make a naive assumption that any deviations with respect to these patterns are potential misuse [4].

The systematic evaluation by Amann et al. [4] reveals that all pattern-based API-misuse detectors, no matter which form of API

[*]Also with Ningbo Research Institute.
[†]Also with PengCheng Laboratory.
[‡]Also with Data61, CSIRO.
[§]Corresponding author.
[¶]Also with University of New South Wales.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416551

patterns they adopt (call pairs/sequences [17, 44], program dependency graph [25, 29, 42], state machine [26]), suffer from low precision (0-11%) and recall (0-20%) in practice. Some approaches attempt to improve detection results by obtaining larger codebase through code search engine [41], adopting more informative usage representation [25, 38], or building more robust probabilistic models of deviation [26]. However, none of these improvements go beyond the naive assumption of pattern-based API-misuse detection.

In this paper, we propose a knowledge-driven approach, which detects API misuse against a novel API-constraint knowledge graph, rather than against API usage patterns that may not reliably manifest API usage caveats (see Section 2 for the discussion on the knowledge gap between API usage patterns and API usage caveats). Existing static code linting tools like FindBugs [28], Pylint [40] cover only general programming anti-patterns (e.g., null reference, useless control flow) that may cause program errors, but not usage caveats of hundreds or thousands of specific APIs. Compilers report only compilation errors (e.g., unhandled exception) based on API declaration, but they are unaware of API usage constraints, such as proper call order, prerequisite state, or value range.

API documentation is an important knowledge source of API usage caveats [16, 20]. Although IDEs provide direct access to API documentation, API documentation, at least in their current semi-structured document form, are insufficient to directly solve the API misuse problem [1, 16, 27]. To improve the accessibility of API-caveat knowledge, Li et al. [16] used Natural Language Processing (NLP) techniques to construct a API-caveat knowledge graph from API documentation. This knowledge graph supports API-centric search of caveat sentences. The extracted natural language caveat sentences are useful for linking API caveats with erroneous code examples or explaining controversial API usage on Stack Overflow [32, 33]. However, caveat sentences cannot be directly used to detect API misuses in source code in their natural language form.

In this paper, we propose a API-constraint knowledge graph: the entities represents API elements and value literals, and the edges represent declaration relations and four types of constraints (call-order, condition-checking, return-condition, exception-trigger) between APIs (see Section 3.1). Different from existing API knowledge graphs [16, 19] that capture only declaration relations and simply link API-caveat sentences to an API as its attributes, we develop NLP techniques to transform API-caveat sentences into specific API constraint relations. For example, by analyzing the return description "true if the iteration has more elements" of Iterator.hasNext() and the throws description "NoSuchElementExcpetion - if the iteration has no elements" of Iterator.next(), we infer a state-checking relation from next() to hasNext() with the expected state true, and the consequence NoSuchElementException of violating this expected state, as shown in Figure 2(b). Compared with existing methods that infer specifications from text [39, 48, 49], our approach infers more types of and more informative API constraints. Given a graph representation (e.g., Abstract Syntax Tree) of a program, we link the program elements with the API entities in our API-constraint knowledge graph. By analyzing the API constraints in the knowledge graph that the linked program elements violate, our approach reports API misuses and explain the detected misused by relevant API caveats (see Figure 6 for an example).

As a proof of concept, we apply our approach to Java SDK API Specification and construct a knowledge graph which contains 1,938 call-order relations and 74,207 condition-checking relations among 21,910 methods and 8,632 parameters, and 8,215 return-value conditions and 12,477 exception trigger clauses. Using the statistical sampling method [35], two developers independently annotate the accuracy of the extracted API-constrained relations. The annotation results confirm the high accuracy (>85%) of the extracted information with substantial to almost perfect agreement between the two annotators. For the 239 API misuses in the 54 Java projects in the MuBench [3], our API misuse detector achieves 60% in precision and 28% in recall. As a comparison, existing pattern-based detectors achieve about 0-11.4% in precision and 0-20.8% in recall according to the systematic evaluation of these detectors [4]. We conduct a pilot user study with 12 junior developers who are asked to find and fix the bugs in six API misuse scenarios derived from the MuBench. The developers, assisted by our API misuse warnings, find and fix bugs much faster and more correctly than those using standard IDE support. The developers rate highly (4 or 5 in 5-point likert scale) the relevance and usefulness of our API misuse warnings, not only for debugging API misuses but also for potentially avoiding them in the first place.

This paper makes the following contributions:

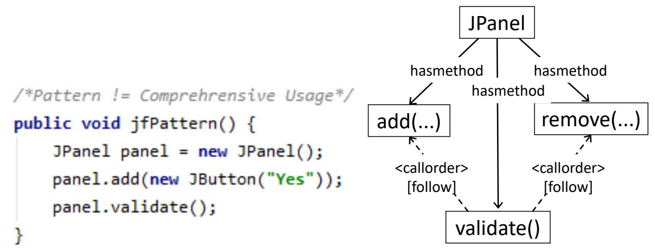
- We analyze the knowledge gap between API usage caveats and API usage patterns, and how a knowledge graph approach may bridge the gap.
- We construct the first API-constraint knowledge graph with four types of API constraint relations, and build the first knowledge-graph based API misuse detector.
- Our manual analysis confirms the high quality of the constructed knowledge graph, and our benchmark evaluation and user study demonstrate the effectiveness and usefulness of our knowledge-graph based API misuse detection.

2 MOTIVATION EXAMPLES

To overcome the limitations of pattern-based API misuse detection identified in [4], we analyze the knowledge gap between API usage caveats and API usage patterns. We focus on API usage caveats specifying call order and condition checking, as violations of these caveats represent the most frequent API misuses [4]. We illustrate the gap with typical examples and API usage patterns in the form of call sequences, but the observed knowledge gap is not restricted to these examples or specific forms of API patterns. We discuss how a API-constraint knowledge graph can bridge the gap (see Section 3 for knowledge graph schema and construction method).

2.1 Comprehensiveness

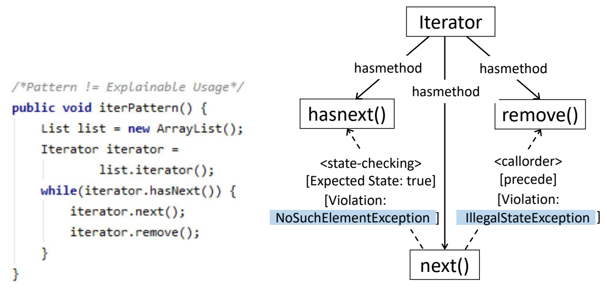
API patterns may cover only some API usage caveats, depending on API usage frequencies. Figure 1(a) shows a code example of Java Swing APIs which satisfies the call-order constraint "the add() method changes layout-related information ... the component hierarchy must be validated thereafter in order to reflect the changes". If there are enough code snippets like the one in Figure 1(a) in a codebase, *add()⟶validate()* can be mined as an API pattern, which helps to detect calling add() without validate() as a misuse. Besides add(), there are many other layout-changing methods like remove()



```
/*Pattern != Comprehrensive Usage*/
public void jfPattern() {
    JPanel panel = new JPanel();
    panel.add(new JButton("Yes"));
    panel.validate();
}
```

(a) (b)

Figure 1: Example of Java Swing APIs



```
/*Pattern != Explainable Usage*/
public void iterPattern() {
    List list = new ArrayList();
    Iterator iterator =
            list.iterator();
    while(iterator.hasNext()) {
        iterator.next();
        iterator.remove();
    }
}
```

(a) (b)

Figure 2: Example of Java Collections APIs

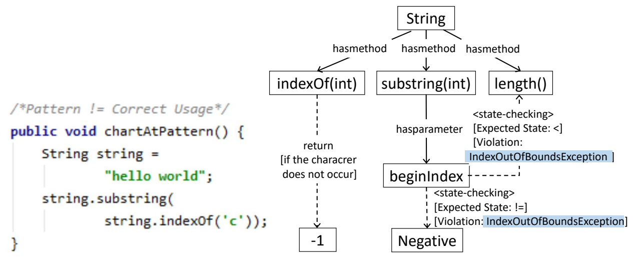
which have the same call-order constraint. However, if the API call sequence panel.remove(); panel.validate() is not frequent enough in the codebase, *remove()→validate()* will not mined as a pattern, and consequently cannot detect missing validate() after remove().

If we model API usage caveats themselves, rather than how frequent they manifest in a codebase, we can achieve more comprehensive coverage of API usage caveats. For example, by analyzing the call-order constraint of all layout-changing methods, we can construct a knowledge graph like Figure 1(b), which captures the *<call-order>[follow]* relations from the method validate() to all layout-changing methods (only add() and remove() is shown for the clarify). Based on this knowledge graph, even there is no prior use of remove() at all in the codebase, we can still detect calling remove() without validate() as a misuse.

2.2 Explainability

API patterns represent the outcomes of following API usage caveats in code, but often cannot distinguish why such outcomes emerge. Figure 2(a) shows a code example of Java Collection APIs: check that hasNext() returns true, then get the next element in the iteration and finally remove this element. Some patterns can be mined from such frequent use of collection APIs, such as *hasNext()→ next()*, *next()→remove()*, or even *hasNext()→ next()→remove()*. Superficially, they all look like call-order constraints. However, unlike *add()→validate()* discussed in Section 2.1, there are no constraints about calling next() (or remove()) after hasNext() (or next()).

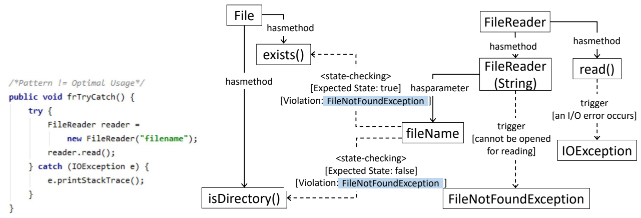
The actual constraint results in the pattern *next()→remove()* is next() must precede remove(), otherwise remove() throws IllegalStateException. The actual constraint results in the pattern *hasNext()→ next()* is that one should call hasNext() to check the state of the iteration before calling next(), because if the iteration has no elements,



```
/*Pattern != Correct Usage*/
public void chartAtPattern() {
    String string =
            "hello world";
    string.substring(
            string.indexOf('c'));
}
```

(a) (b)

Figure 3: Example of Java String APIs



```
/*Pattern != Optimal Usage*/
public void frTryCatch() {
    try {
        FileReader reader =
            new FileReader("filename");
        reader.read();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

(a) (b)

Figure 4: Example of Java IO APIs

next() throws NoSuchElementException. API patterns cannot distinguish such fine details of API caveats. Graph-based patterns [25, 38] are more informative than call sequences, but it is not straightforward to infer all fine details of API caveats from code.

In contrast, we can analyze the natural language descriptions of API caveats to distinguish different types of API constraints and extract their fine details, as shown in the knowledge graph in Figure 2(b). Due to the clear semantics of the state-checking and call-order relations, we will not report calling (hasNext()) (or next()) without the following next() (or remove()) as a misuse. More important, we can provide specific explanation of detected API misuses, such as missing state checking versus missing preceding call, as well as the expected state and violation consequences.

2.3 Best Practices

API patterns fundamentally assume frequencies reflect the rationality of API usage, but this rationality may not correspond to the best practices of handling API caveats. Figure 3(a) shows a code example of Java String APIs: substring(indexOf()). This API chain call is frequent in code, but it lacks a sanity checking of the indexOf()'s return value, because indexOf() returns -1 if the char does not occur in the string, and substring() throws IndexOutOfBoundsException if its beginIndex argument is negative. As another example, Figure 4(a) shows a code example of Java IO APIs: enclose file open and read operations in a try-catch to handle IOException, which is also very common in code. However, this way of handling IOException does not consider the specific causes of the exception: "the file does not exist, or the file is a directory or an IO error occurs".

By analyzing the description of return value conditions, parameter constraints, exception causes and API functionalities, we can model and reason about the complex constraint relations between APIs in the knowledge graphs in Figure 3(b) and Figure 4(b). For example, by examining the chain call substring(indexOf())

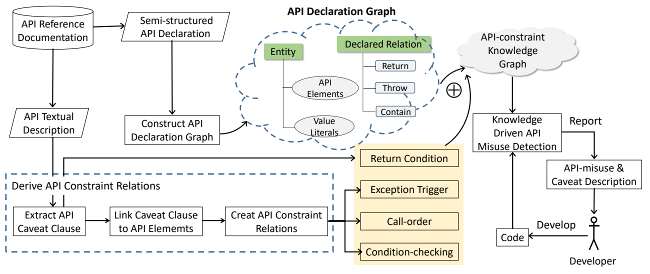

API Reference Documentation
Semi-structured API Declaration
API Declaration Graph
Entity
Declared Relation
Return
Throw
Contain
API Elements
Value Literals
API-constraint Knowledge Graph
API Textual Description
Construct API Declaration Graph
Knowledge Driven API Misuse Detection
Report
API-misuse & Caveat Description
Return Condition
Exception Trigger
Call-order
Condition-checking
Derive API Constraint Relations
Extract API Caveat Clause
Link Caveat Clause to API Elemements
Creat API Constraint Relations
Code
Develop
Developer

Figure 5: The Overall Framework of Our Approach

against the relations *[indexOf(), return, -1]* and *[beginIndex, <value-checking>[!=], negative]*, we can identify a missing-value-checking, its condition "the character does not occur" and its consequence IndexOutOfBoundsException. By examining the code in Figure 4(a) against specific parameter state-checking relations (e.g., *[filename, <state-checking>[true], File.exist()]*) and exception triggers, we can identify missing file.exists() checking and missing file.isDirectory() checking, and make specific fix suggestions (see Figure 6), as opposed to a generic unhandled exception error raised by the IDE.

3 OUR APPROACH

To support the knowledge-graph based API misuse detection presented in Section 2, we develop open information extraction methods to construct the API-constraint knowledge graph from API reference documentation, e.g. Java SDK API Specification. Figure 5 shows the overall framework of our approach. Our API-constraint knowledge graph significantly extends existing general API knowledge graphs [16, 19] (referred to as API declaration graph (see Section 3.3) in this work) with four categories of fine-grained API constraint relations derived from API caveat sentences (see Section 3.4). These API constraint relations correspond to the three most-frequent API misuse categories in the API misuse benchmark MuBench [4] (see Section 3.1). We develop a novel knowledge-driven API misuse detector that checks the program for API misuses against the API-constraint knowledge graph (see Section 3.5).

3.1 Knowledge Graph Schema

The knowledge graph entities include *API elements* (*package*, *class*, *exception*, *method*, *parameter* and *return-value*) and *value literals* (e.g., null, -1, true, or a range like negative or [0-9]). We distinguish exception from class to facilitate exception handling analysis. An entity has a *name* (null for return-value). A method or parameter entity has a *functionality description*, which is used to link method or parameter to relevant API caveat description for deriving API-constraint relations. A return-value entity has a *return-value description*. This work focuses on API-method constraints, so we do not need descriptions for packages and classes. Packages and classes are used as the declaration scope to limit the search space of API constraint relation inference.

The relations of our knowledge graph include *declaration relations* and *constraint relations*. The declaration relations include: an API element *contains* another API element (e.g., a class contains a method), a method *returns* a value-literal, and a method *throws* an exception. Different from the existing API knowledge graphs [16, 19], we attach *condition* attribute to return relations, for example the relation *[String.indexOf(char), returns, -1]* with the condition "the character does not occur". This helps identify the situation where using the return value of one method as the argument of the other method may cause program errors. For example, substring(indexOf(ch)) in Figure 3(a) may have an unhandled IndexOutOfBoundsException under this return-value condition of indexOf(). Furthermore, we attach *trigger* attribute to throw relations, which records the exception situation that cannot be prevented by certain pre-condition checking, e.g., the relation *[FileReader(String), throw, FileNotFoundException]* with the trigger "the named file for some other reason cannot be opened for reading". In contrast, the condition "the named file does not exist" can be checked by File.exists() in order to avoid FileNotFoundException.

Besides constraint-enriched return/throw relations, our knowledge graph includes two other constraint relations: *call-order* and *condition-checking* which correspond to the top-2 most frequent API misuses in the API misuse benchmark [4]: missing-call and missing-condition-checking. A call-order relation can be either *precede* (e.g., calling Iterator.next() before calling Iterator.remove()) or *follow* (e.g., calling Container.validate() after calling Container.add()). It may have an optional *condition* attribute for the preceding/following method call, e.g., the condition "if the container has already been displayed" for the relation *[Container.validate(), <call-order>[follow], Container.add()]*. Call-order is not transitive, and precede-follow are not symmetric. A condition-checking relation can be either value checking (e.g., *[beginIndex, <value-checking>[!=], negative]* in Figure 3(b)) or state-checking (e.g., *[filename, <state-check-ing>[true], File.exist()]* in Figure 4(b)). The value-checking records the *expected expression* (e.g., !=, <), and the state-checking records the *expected state* (e.g., true indicating the named file exists). Both the call-order and condition-checking relations may have a *violation* attribute which records the consequence if the call-order or the expected expression or state is not satisfied, for example, IllegalStateException for *[next(), <call-order>[precede], remove()]*, or FileNotFoundException for *[filename, <state-checking>[true], File.exists()]*.

3.2 API Documentation and Preprocessing

We construct API-constraint knowledge graph as defined above from API reference documentation, such as Java SDK API Specification. We crawl online API documentation using web crawling tool. Following the treatment in [16, 19], we keep the semi-structured API declarations and the API textual descriptions in the crawled web pages for constructing a knowledge graph. We remove other document contents, e.g., code snippets, program execution outputs and images. <code> elements in natural language sentences are kept. Following treatments in [16, 19], we use software-specific tokenizer [5, 46] to tokenize API descriptions. This retains the integrity of API tokens, e.g., an API mention Iterator.next() in text will be kept as one token rather than five tokens - Iterator, ., next, (, and). After text tokenization, we use Stanford CoreNLP [21] to split texts. A sentence from the return or throws section of a method usually omit the subject and verb. We add "this method returns" and "this method throws XXXException" to complete the sentence.

3.3 Constructing API Declaration Graph

First, we construct an API declaration graph from the semi-structured API declarations in each API document. As our API declaration graph is the same as the generic API knowledge graph in [16, 19], we adopt their tested web page parser to extract API elements, API names and descriptions, and declaration relations as required in our knowledge graph (see Section 3.1). In this work, we use the brief introduction sentences of each method in the method summary section as that method's functionality description.

To detect the API misuses related to API chain calls (e.g., substring(indexOf())), we extend the original parser in [16, 19] to extract more fine-grained return relations. The return section of a method may have more than one sentence to explain different return values in different situations, for example, String.indexOf() returns "the index of the first occurrence of the character in the character sequence" or "-1 if the character does not occur". Based on the observation of the return section of 1,000 randomly sampled methods, we define a value gazetteer and a set of Part-Of-Speech (POS) tag patterns to recognize specific values (e.g., -1, true, false) or ranges (e.g., negative, [0, 9]) mentioned in the return-value sentences. If a return-value sentence contains a specific value or range, we link the return relation to a value-literal entity (may need to be created if it is not yet in the knowledge graph). Otherwise, we create a return-value entity with that sentence as its description and link the return relation to this return-value entity.

3.4 Deriving API Constraint Relations

Different from existing API knowledge graphs [16, 19] that capture only declaration relations, our API-constraint knowledge graph contains call-order and condition-checking relations between related APIs, and constraint-enriched return and throw relations. We use the API-caveat sentence patterns developed in [16] to extract two categories of API-caveat sentences: *temporal* and *conditional*. Different from [16] that simply links API-caveat sentences to API elements as textual attributes, we develop sentence parsing and API linking techniques to derive four types of API constraint relations from API caveat sentences.

3.4.1 Extracting API-Caveat Clauses. As this work focuses on API-method usage constraints, we limit the extraction to the main description of each method (excluding the functionality description sentence of the method entity) and the description in the method's return and throws section. Each extracted caveat sentence is associated with its corresponding method or return/throw relation. We process the extracted caveat sentences into fine-grained API-caveat clauses by the following three steps, to facilitate the subsequent API linking and constraint relation inference.

Co-reference resolution. API elements are often mentioned by pronouns in API-caveat sentences, for example, "If the SecureRandomSpi() constructor is overridden in an implementation, it (refer to SecureRandomSpi()) will always be called whenever a SecureRandom is instantiated". We use co-reference resolution technique (as implemented by Stanford CoreNLP [21]) to resolve the pronouns in a API-caveat sentence to the APIs that the pronouns represent in the paragraph from which the sentence is extracted. Furthermore, the API methods being explained are commonly referred to as "this method" in its description, For example, "This method is generally called ... if a fatal error has made the connection unusable" in the description of javax.sql.PooledConnection. Co-reference resolution tools cannot resolve this type of co-reference because the corresponding API does not appear in the surrounding texts. We use the declaration-based heuristic [16] to resolve such co-reference to the API method being explained.

Splitting sentences into clauses API-caveat sentences can be rather complex. For example, "FileReader(String fileName) throws FileNotFoundException if the named file does not exist, is a directory or for some other reason cannot be opened for reading". This sentence has a result clause and a long if clause that has three condition clauses. To derive fine-grained API constraints, we use POS tagging and dependency tree analysis (as implemented by Standford CoreNLP [21]) to parse the whole sentence into several fine-grained clauses. A conditional sentence is split into a result-clause and one or more condition clauses. We also try to identify subject, verb-phrase and object in each clause using semantic role labeling [15]. For example, we can extract Subject-Verb-Object in the result-clause as follows: FileReader(String fileName) [subject] throws [verb-phrase] FileNotFoundException [object]. Note that the missing subject (e.g., "the named file" for the 2nd and 3rd conditions) can be inferred by dependency tree analysis.

Similar clause clustering. One API caveat may be mentioned in different parts of a method in the same or similar way. For example, the main description of String.indexOf() has a conditional sentence "if no such character occurs in this string, then -1 is returned", and the return section of indexOf() has another conditional sentence "indexOf() returns -1 if the character does not occur". These two sentences correspond to the same caveat. Furthermore, a class may declare several overloading or similar-functionality methods which often have the same or partially overlapping caveat sentence. For example, both String.indexOf() and String.lastIndexOf() have the caveat "return -1 if the character does not occur". The overloading methods substring(int beginIndex, int endIndex) and substring(int beginIndex) have the same caveat clause "the beginIndex is negative" but also other different clauses.

We cluster similar caveat clauses by the word-embedding based sentence similarity (see Section 3.4.2) which has been shown to be effective in matching software text [6, 13, 32, 33]. The clustering is done progressively, first within method, then within class, and finally within package. Clustering within-method considers all caveat clauses of a method (no matter their document section origin), but clustering within-class or within-package groups only the caveat clauses from the same type of document section. We select the centroid sentence in a cluster as the representative of the cluster. If the cluster has only two sentences, we select the shorter one. Clustering similar caveat clauses have two benefits. First, we can associate a caveat clause to a more specific API element/relation. For example, we can know that the conditional clause from the main description of String.indexOf() is actually related to the return relation, because a very similar clause is in the return section of indexOf(). Second, we can significantly reduce the number of caveat clauses to be analyzed in the subsequent API linking step.

3.4.2 Linking Caveat Clauses to API Elements. Given a caveat description (a clause or its subject/object phrase) associated with a method or a throw relation, we infer methods or parameters whose functionality descriptions match the caveat description. For example, for the exception trigger clause "the named file does not exist"

Table 1: Rules for Creating Call-order or Condition-checking Relations ***(p: paramter; vl: value literal; m: method; e: parameter or method; mc: method of concern; v-c: value-checking; s-c: state-checking; c-o: call-order)***

	Clause	Subject	Object	Value	Type	Relation	Example
1	-	p	-	vl	-	[p,v-c,vl]	[beginIndex, =, negative] "the beginIndex argument is negative"
2	-	p	e	-	-	[p,v-c,e]	[index, !<, String.length()] "the index argument is not less than the length of string"
3	m	p	-	-	-	[p,s-c,m]	[filename, true, File.exists()] "the named file does not exis"
4	-	m1	m2	-	-	[m1,c-o,m2]	[setSystemId(String), precede, startDocument()] "setSystemId(String) must be called before the startDocument event"
5	-	m	-	-	temporal	[m,c-o,mc]	[next(), precede, remove()] "if next() has not yet been called" in the remove()'s throws section
6	-	m	-	-	conditional	[m,s-c,mc]	[undo(), true, canUndo()] "if canUndo() returns false" in undo()'s throws section
7	m	-	-	-	temporal	[m,c-o,mc]	[validate(), follow, add()] "the container must be validated thereafter" in the main description of Container.add()
8	m	-	-	-	conditional	[m,s-c,mc]	[next(), true, hasNext()] "if the iteration has no more element" in the next()'s throws section

on the relation *[FileReader(String), throw, FileNotFoundException]*, the subject "the named file" matches the description "the name of the file to read from" of the FileReader(String)'s parameter fileName, and the whole clause matches the functionality description of File.exists() which states "tests whether the file or directory denoted by this abstract pathname exists". Based on such matches, we link the caveat clause or its subject/object to methods or parameters that are referred to by the clause or its subject/object, or whose functionality can fulfill or check the clause or its subject/object.

We perform the matching progressively, first match parameters within method, then match methods within class, and finally match methods within package. The matching is done for the caveat clause, its subject phrase and object phrase respectively. The whole clause may match a method, and the subject or object phrase may match a parameter or a method. If a subject or object phrase contains annotated code element (e.g., in <code> or <href>), we directly link the subject or object to the corresponding API element by name matching. Otherwise, we match a caveat description with the functionality description of API elements by text similarity. We select the API element whose functionality description has the highest similarity with the given caveat description within the current matching context. If this highest similarity is above the similarity threshold (0.8 in this work), a matching is found. If a matching is found within the current context, the matching stops.

Considering the sentence characteristics of caveat and functionality descriptions, we combine Jaccard coefficient and sentence-embedding similarity to match them. We denote a caveat description as cd and a functionality description as fd. Before computing Jaccard coefficient, we convert each sentence into a bag of words (BW_{cd} and BW_{fd}) using the standard text processing procedure (i.e., tokenization, stop word removal and lemmatization). Then, the Jaccard coefficient of cd and fd is: $sim_{jaccard}$=$BW_{cd} \cap BW_{fd}/BW_{cd} \cup BW_{fd}$. To compute sentence-embedding, we learn domain-specific word embeddings with the corpus of API text descriptions from API reference documentation using the continuous skip-gram model [22]. We use domain-specific word embeddings because recent studies [5, 45] show that domain-specific word embeddings outperforms general word embeddings for software text matching. We set the word embedding dimension at 200, as this setting has the best performance on similar API text corpus [6]. We compute a sentence embedding by averaging the word embeddings of all words in the sentence. Let SE_{cd} and SE_{fd} are the sentence embedding of cd and fd respectively. The sentence-embedding similarity of cd and fd is the cosine similarity of SE_{cd} and SE_{fd}, i.e., sim_{se}=$cos(SE_{cd}, SE_{fd})$. Finally, we average $sim_{jaccard}$ and sim_{se} as the similarity of a caveat description and a functionality description.

3.4.3 Creating API Constraint Relations. Given a caveat clause, we analyze its API linking results and create a call-order or condition-checking relation according to the heuristic rules in Table 1. These heuristics rules are summarized based on the observation of randomly sampled 1,000 caveat clauses. For all other API linking results, for example, none of the clause, subject phrase or object phrase is linked to an API element (e.g., "an IO error occurs" associated with the relation *[FileReader.read(), throw, IOException]*), or only the subject phrase is linked to a parameter of the method (e.g., "the named file for some other reason cannot be opened for reading" associated with the relation *[FileReader(String), throw, FileNotFoundException]*, the caveat clause remains intact with its originally associated method or throw relation. The first three rows in Table 1 cover the caveat clauses that describe the value or state validity of the parameters. Such caveat clauses are converted into value-checking or state-checking relations between a parameter and a value literal, between two parameters, or between a parameter and a method. The 4th row covers the caveat clauses that explicitly mention two methods and describe call-order constraints, e.g., "setSystemId(String) must be called before the startDocument event".

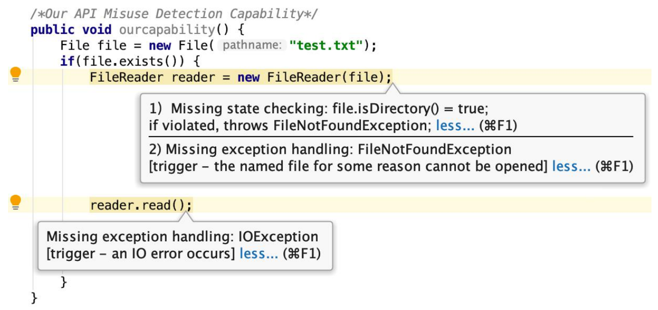
It is common that a caveat clause mentions only one method in its subject (the 5th and 6th rows) or the whole clause corresponds to one method (the 7th and 8th rows). In such cases, the other method that is implicitly referenced is the method of concern (i.e., the method that this clause is associated with), for example, the caveat "if next() has not yet been called" for the Iterator.remove() method, or "the container must be validated thereafter in order to display the added component" for the Container.add() method. Therefore, we create a constraint relation between the explicitly mentioned method and the method of concern for the cases in rows 5/6/7/8. Depending on whether the caveat clause is a temporal or conditional clause, we create either a call-order relation or a state-checking relation. Value literals are extracted by the value gazetteer and POS tag patterns we develop. Never-seen value literals will be added to the knowledge graph. We also develop POS tag patterns to extract frequently-used value-checking expressions and convert them into mathematical formula (e.g., ≥, <, =, ≠, ∈ [...]) as the expected expressions of the value-checking relations. If the caveat clause is associated with a throw exception, we use the negation of the formula as the expected expression. If the clause does not have such frequently-used value-checking expressions, we use the verb-phrase of the clause as the expected expression. For state-checking relation, if the linked method has some specific return value whose return-condition matches the caveat clause, we use this specific return value as the expected state, such as true for hasNext() checking before calling next(). Otherwise, we use the verb-phrase of the clause as the expected state. For the call-order relation, if the temporal clause has some guard condition clause, e.g., "if the container has already been displayed" for calling validate(), we use this condition clause as the condition of the call-order relation. If the caveat clause is associated with a throw relation, the violation attribute of the created constraint relation references to the exception entity of the throw relation.

3.5 API Misuse Detection

We develop a knowledge-driven API misuse detector that examines the API usage in a program against the constructed API-constraint knowledge graph. In this work, the detector performs static code analysis on the Abstract Syntax Tree (AST) of the program. For each API method used in the program (denoted as api_p), it first links api_p to an API method in the knowledge graph (denoted as api_{kg}) by matching their fully-qualified names. Then, it collects all call-order, condition-checking, and throw relations of api_{kg}.

For the call-order relation, the detector examines if the required preceding or following method is called before or after calling api_p. For the condition-checking relation, the detector examines if the required value or state checking is performed before calling api_p. If the required checking is found in the program and the expected expression or state of the condition-checking relation involves specific values/states and mathematical formulas, the detector further examines if the expected expression or state can be satisfied by the program. Let exp_p and exp_{kg} be the formula of the corresponding condition checking in the program and in the knowledge graph respectively. The detector examines if $(exp_p \wedge \neg \, \mathrm{exp}_{kg}) \vee (\neg exp_p \, \mathrm{exp}_{kg})$ is satisfiable by a SAT solver [8]. If a violation of the required call-order or condition-checking is detected, the detector reports not only an API misuse, but also the consequence of API misuse, the relevant API to fix the misuse, and the original API caveat sentence as the explanation of the API misuse. If the compiler detects an unhandled exception *ue* for api_p, our detector locates the specific throw relation for *ue* in our knowledge graph and reports the associated exception trigger.

Empowered by the API-constraint knowledge graph, our API misuse detector can perform fine-grained analysis of API usage in the program. For example, for the code in Figure 6, the compiler reports an "Unhandled exception: FileNotFoundException" for new FileReader(file), and an "Unhandled exception: IOException" for



```
/*Our API Misuse Detection Capability*/
public void ourcapability() {
    File file = new File( pathname: "test.txt");
    if(file.exists()) {
        FileReader reader = new FileReader(file);
```

1) Missing state checking: file.isDirectory() = true;
if violated, throws FileNotFoundException; less... (⌘F1)

2) Missing exception handling: FileNotFoundException
[trigger – the named file for some reason cannot be opened] less... (⌘F1)

```
        reader.read();
```

Missing exception handling: IOException
[trigger – an IO error occurs] less... (⌘F1)

```
    }
}
```

Figure 6: Our API Misuse Detection Capability

reader.read(). Reading API document may not clearly identify specific cause(s) for FileNotFoundException, because it can be caused by three conditions "the file does not exist", "the file is a directory", or "the file for some other reason cannot be opened for reading". A developer often simply encloses the API calls by try-catch like the one in Figure 4(a). Actually, the code checks if(file.exists()), which satisfies the state-checking relation *[file, true, File.exists()]*. Therefore, our detector does not report any issue for the first condition. But our detector reports the missing of the second condition based on the state-checking relation *[file, false, File.isDirectory()]*, and recommends using File.isDirectory() to perform this missing checking. As the first two conditions are either handled or reported as missing, our detector reports the third condition as a specific cause for FileNotFoundException. Such detailed API misuse reports and suggestions can enable more robust API usage.

Our detector checks the issues related to API chain calls, such as substring(indexOf()) in Figure 3(a). It analyzes the definition-use information in the program to collect the APIs api_d whose return values are used as the parameters $param$ of api_p. If api_d returns specific values under certain conditions (e.g., *[indexOf(), return, -1]* with the condition "the character does not occur"), the detector examines if this specific return value violates some value-checking relations of $param$. For example, the beginIndex parameter of substring(int) has a value-checking relation *[beginIndex, <value-checking>[!=], negative]*, which is violated by the return-1 of indexOf(). Therefore, our detector reports a missing-value-checking for substring(indexOf()), with the condition "the character does not occur" and the violation consequence IndexOutOfBoundsExcpetion.

4 TOOL IMPLEMENTATION

We construct a API-constraint knowledge graph for Java APIs. Using the web page parser developed Liu et al. [19], we extract 72,337 API elements (including 59,991 API methods, 11,334 parameters and 1,012 exception), and 64,400 API declaration relations (including 45,247 return relations and 18,999 throw relations). Using the API-caveat sentence patterns developed by Li et al. [16], we extract 97,462 conditional and temporal API-caveat sentences. From these API-caveat sentences, our approach creates 1,938 call-order relations, 74,207 condition checking relations, and enrich 8,215 return relations with return-value conditions and 12,477 throw relations with exception triggers. These API-constraint relations involve 21,910 methods, 8,632 parameters, and 8,215 return and 12,477 throw relations. We develop an IntelliJ IDE plugin which detects API misuses in Java programs based on the constructed API-constraint knowledge graph (see Figure 6).

5 QUALITY OF KNOWLEDGE GRAPH

We first want to evaluate the quality of the constructed knowledge graph. As we use tested tools [16, 19] to extract API elements and declaration relations from API documentation and to extract API-caveat sentences, we do not repeat the evaluation of these information extraction steps. We focus our evaluation on the four types of API constraint relations, which distinguish our API-constraint knowledge graph from existing general API knowledge graphs [16, 19] .

5.1 Experiment Setup

As our knowledge graph contains large numbers of API constraint relations, we use a statistical sampling method [35] to examine *MIN* [35] randomly sampled instances of each type of constraint relation. *MIN* in this work which ensures the estimated accuracy is in 0.05 error margin at 95% confidence level. For conditioned return relations, we examine two checkpoints: the accuracy of value-literals and the accuracy of condition clauses. For throw relations, we examine two checkpoints: the accuracy of trigger clauses, and if the trigger clause should be converted into specific call-order or condition-checking relations, rather than remaining as an exception trigger. For call-order and condition-checking relations, we examine three checkpoints: if the corresponding API caveat clause should be modeled as a call-order or condition-checking relation, the accuracy of API linking, and the accuracy of relevant attributes (expected expression, expected state, condition, violation). The two developers (who are not involved in this study and have more than 3 years Java development experience) independently perform the examination and all decisions are binary. We compute Cohen's Kappa [14] to evaluate the inter-rater agreement. For the data instances that the two annotators disagree, they have to discuss and come to a consensus. Based on the consensus annotations, we evaluate the quality of the created API constraint relations.

5.2 Results

Table 2 shows the examination results. See Section 5.1 for the explanation of checkpoints. The columns *Acc*1 and *Acc*2 show the accuracy by the two annotators independently. The column *AccF* is the final accuracy after resolving the disagreements. The column Kap. is the Kappa inter-rater agreement. The accuracies of all types of extracted information are above 85% The Cohen's Kappa are all above 0.60, which indicates substantial to almost-perfect agreement between the two annotators.

The trigger clauses on throw relation have 100% accuracy, which is unsurprising because these exception triggers extracted from the throws section of a method. Exception clauses describe the exception-handling knowledge in natural language sentences. In our work, we want to covert as many exception clauses as possible to call-order or condition-checking relations, because call-order and condition-checking relations represent exception-handling knowledge in a more fine-grained, structured way and supports more fine-grained API usage analysis. However, the should-not-be-trigger row shows that 15% of the examined exception triggers should be converted into call-order or condition-checking relations, but not. The primary reason is that our approach fails to link the clause or its subject/object phrases to relevant API elements. For example, the trigger clause "if the calling thread does not have permission to create a new class loader." for the relation *[checkCreateClassLoader(), throw, SecurityExceptio]* should be model as a state-checking relation *[checkCreateClassLoader(), true, checkPermission(Permission)]*. However, as the functionality description of checkPermission(Permission) is not similar to this trigger clause, our method fails to make the link.

Although our method may miss some API links, the API links it makes are very accurate (95.3% for call-order and 94.3% for condition-checking). Many API elements that are mentioned in the caveat clauses are within method or class, and these elements are easy to match within local context. For those API elements outside the class, they are usually annotated by the API hyperlink, from which API elements can be directly inferred. Our method also achieves high accuracy (>98%) for extracting value-literals, return-condition clauses and attributes of call-order and condition-checking relations. This is because these types of information are extracted by the gazetteer and sentence patterns we carefully define.

Our API-constraint knowledge graph contains highly accurate API-constraint relations, which can support practical use. The extraction of call-order and condition-checking relations can be further enhanced by more robust API linking methods.

6 EFFECTIVENESS EVALUATION

Next, we evaluate the effectiveness of our knowledge-driven API misuse detection using the API misuse benchmark MuBench [3].

6.1 Experiment Setup

MuBench has been actively maintained. The latest version of MuBench contains 269 instances of intra-method API misuses in 69 software projects. For each API misuse, MuBench identifies API(s) involved, provides a brief description of the misuse, and reference to the source code that contains the misuse. As our current knowledge graph supports only Java SDK APIs, we use Java SDK API misuses in the 54 Java projects. We download the project source code and make it compilable. We collected in total 239 instances of API misuses in these 54 projects, including 114 missing call, 107 missing condition checking and 18 missing exception handling. These API misuses involve 30 API methods, and violates 104 API usage caveats described in Java SDK API documentation.

We apply our API misuse detector to the methods that contain the API misuses, and examine how many of the 239 API misuses can be detected by our detector. We also examine if the explanation that our tool provides for the detected misuse matches the misuse description in the benchmark. As determining the precision of the detected API misuses requires in-depth project-specific knowledge, we only make an estimation of the lower bound of the detection precision. That is, we assume that only API misuses that match the ground-truth misuses are correct, but all others are incorrect.

6.2 Results

Our detector reports 113 API misuses, including 66 missing call, 42 missing condition checking and 5 missing exception handling. Among these 113 API misuses, 68 are confirmed by the MuBench, including 37 missing call, 29 missing condition checking and 2 missing exception handling. The caveat descriptions of these 68

Table 2: Accuracy of the API-Constraint Relations

Relations	Check Points	Acc1	Acc2	AccF	Kap.
Conditioned Return	*value-literal*	98.7%	99.2%	**99.0%**	0.60
	condition clause	99.7%	99.7%	**99.7%**	1.00
Trigger on Throw	*trigger clause*	100.0%	100.0%	**100.0%**	1.00
	should not be trigger?	88.0%	85.7%	85.9%	0.87
Call-order	*API linking*	95.1%	96.4%	95.3%	0.84
	attribute	97.1%	98.7%	**98.2%**	0.62
	Should be call-order?	90.4%	91.7%	90.9%	0.86
Condition-checking	*API linking*	96.4%	94.3%	94.3%	0.77
	attribute	99.5%	99.2%	**99.5%**	0.80
	Should be cond-checking?	95.3%	96.4%	95.6%	0.74

Table 3: Six API Misuse Scenarios in Our User Study

Task	Involved API	Misuse reason	Difficulty
T-1	java.util.Arrays	Condition-checking	Easy
T-2	java.util.List, java.util.ArrayList	Condition-checking	Easy
T-3	java.io.FileReader, java.io.File, java.util.Scanner	Condition-checking	Medium
T-4	javax.swing.JFrame, javax.swing.JButton, javax.swing.JPanel	Call-order	Difficult
T-5	java.util.ArrayList, java.util.Iterator, java.util.List	Missing call	Medium
T-6	javax.swing.JFrame, java.awt.Dimension	Missing call	Difficult

confirmed API misuses are consistent with the corresponding API misuse descriptions in the MuBench. Our detection precision is 60.18% overall, 56% for missing call, 69% for missing condition checking and 40% for missing exception handling. Note that these precision are lower bound estimations, as we assume that all non-confirmed 45 (113−68) API missuses are incorrect. According to our observation, some non-confirmed API misuses reported by our detector, for example some missing value or state checkings, could be API misuses. MuBench does not consider them, because they do not yet cause any bugs in the program. But adding these condition checkings could make the program more robust.

The lower bound precision of our detector is still much higher than the precisions achieved by the best pattern-based API misuse detector, which is only 11% according to the evaluation of all popular pattern-based detectors [4]. Such low precision is the primary barrier for adopting these detectors in practice. According to [4], there are two correlated reasons for such low precisions. First, the API usage patterns mined by these detectors cover only the most frequent usage, but they miss many uncommon and alternative usage. Second, these detectors make a naive assumption that a deviation from the mined patterns correspond to a misuse. In Section 2, we illustrated these two reasons with four examples. Readers are referred to [4] for the detailed analysis of these two reasons. In contrast, our detection is based on the API caveat knowledge extracted from API reference documentation, which has nothing to do with whether an API is used and how frequent it is used in code.

The recall of our detection is 28.45% (68/239), which is on-par with the conceptual recall upper bound that all existing pattern-based API misuse detector could achieve (by feeding them sufficiently examples of correct usage corresponding to the misuses in question) [4]. The actual recalls of pattern-based API misuse detectors are much lower (0-20%). Our detector does not need to learn from code examples, but it is limited by the knowledge coverage of the underlying knowledge graph. In fact, this is the primary reason for the API misuses that our detector fails to detect, accounting for 86.55% (148/171) misses. Our knowledge graph currently covers four types of API-constraint relations, but there are other types of API usage knowledge, for example, multiplicity (e.g., Iterator.remove() can only be called once after Iterator.next()), API equivalence (e.g., isEmpty(), size()=0 and hasNext() for collection APIs), no-effect API calls (e.g., the call to some API is ignored under certain condition), or class-level usage (e.g., The CharsetEncoder class should be used when more control over the encoding process is required). We leave the extension of our knowledge graph to accommodate these types of API usage knowledge as future work. The rest of missed API misuses (13.45%) are due to the limitation of our current programming analysis and detection heuristics, as these API misuses demand advanced pointer analysis, data flow analysis or expression evaluation.

Our knowledge-graph based API misuse detection can detect missing calls, missing condition checking and missing exception handling with good precision. To improve the recall of our detection, more types of API usage knowledge should be extracted and added to the underlying knowledge graph, and some advanced programming analysis should be supported.

7 USEFULNESS EVALUATION

7.1 User Study Design

We select six API misuse scenarios from the webtend project in MuBench [3]. Note that similar scenarios also occur in other projects in MuBench. As summarized in Table 3, these six scenarios involve misuses of Java Swing, IO and Collections APIs: two about missing call, one about erroneous call order, and three about missing condition checking. For each misuse scenario, we create a method to simulate the core program logic and API usage (not just the misused API) in the original method involved in the scenario. We do not use the original method because they have project-specific code elements which demand certain project-specific knowledge, while our study focuses on finding and fixing API misuses. However, the created methods are still realistic, executable programs. Our tool reports potential API misuses for all APIs in a method, among which developers have to identify the API misuse leading to the bug.

We recruit 12 master students from our school. None of these students are involved in our work. These students have basic knowledge of Java SDK APIs, but do not use Java in their daily work. As our API misuse scenarios do not involve advanced Java APIs, we believe these students are qualified for our study. Furthermore, they also simulate the target audience that our tool aims to assist, i.e., developers who may lack relevant knowledge in finding and fix API misuses. Based on a pre-study survey of these students' Java programming experience, we randomly allocate them into two

Table 4: Results of User Study

Metric		T-1	T-2	T-3	T-4	T-5	T-6	Aveg.
Accuracy	G1	1.00	1.00	1.00	0.33	0.67	0.50	0.75
	G2	1.00	1.00	1.00	1.00	0.83	0.83	0.94
	Imp.	0.00%	0.00%	0.00%	**203%**	24%	66%	49%
Time (min)	G1	2.92	2.53	5.08	9.33	4.37	7.50	5.29
	G2	1.58	1.92	2.83	3.58	2.33	4.17	2.74
	Imp.	-46%	-24%	-44%	-62%	-47%	-44%	-45%
Error Proneness	G1	2.00	2.50	4.17	4.17	3.33	4.17	3.39
	G2	1.67	2.17	3.17	3.50	2.33	3.33	2.70
	Imp.	-17%	-13%	-24%	-16%	-30%	-20%	-20%
Warning Relevance	G1	1.00	1.00	3.33	1.00	1.00	1.50	1.47
	G2	4.33	4.33	4.33	4.17	4.17	3.00	4.06
	Imp.	**333%**	**333%**	30%	**317%**	**317%**	**100%**	**238%**
Warning Usefulness	G1	1.00	1.00	3.00	1.00	1.00	1.50	1.42
	G2	4.00	4.17	4.50	4.33	4.33	3.67	4.17
	Imp.	**300%**	**317%**	50%	**333%**	**333%**	**144%**	**194%**

equivalent groups: the control group (G-1) uses the standard IntelliJ IDE to complete the tasks, while the experimental group (G-2) uses the IntelliJ IDE with our API misuse detection plugin.

The participants are given 10 minutes for each task. Each task includes a buggy method and describes the expected program output. The participants can use any IDE features like accessing API document and debugging. They can also search the Internet and read any information they feel useful for the task. If the participants believe they fix the API misuse in the task method, they submit the fixed code which signal the task completion. The participants fill in a short survey for each task. The survey asks the participants to rate: the error proneness of the API misuses if developers were given our API misuse warnings (or standard IDE warnings if any), the relevance of our detected API misuses (or standard IDE warnings if any) to the task method, and the usefulness of our API misuse warnings (or standard IDE warnings if any) for fixing the misuse. All ratings are 5-point liked scale (1 being least and 5 being most). After the experiment, we collect the task completion time and examine if the submitted code actually fixes the API misuses.

7.2 Results

Our tool can report more than one API misuse warnings during the completion of the each tasks. However, by examining the description of API misuse warnings (see the examples in Figure 6), participants can quickly filter out warning irrelevant to the bug. So their task completion does not seem to be interfered with by multiple API misuse warnings. Furthermore, the irrelevance to the bug does not necessarily mean the API misuse warnings are false-positive warnings. As discussed in Section 6.2, some warnings, if adopted, could improve the overall robustness of the code. That is why such "irrelevant" API misuses does not lower the ratings of warning relevance and usefulness.

Table 4 shows our study results with five metrics: answer correctness, task completion time, error proneness, warning relevance, and warning usefulness. We average the metrics for the control and experimental group respectively. For the three missing-condition-checking tasks Task-1/2/3, both groups achieve 1.0 answer correctness. That is, all the participants successfully find and fix the bugs in these three tasks. This is because missing condition checkings are relatively easy to spot, when analyzing the unexpected program outputs. In Task-1/2/3, the standard IntelliJ IDE does not provide any warnings for the buggy methods. In fact, IntelliJ provide very little support for the concerned API misuses in all tasks except Task-3. That is why the control group generally give the least rating (1) to warning relevance and usefulness. In contrast, the experimental group highly rate (mostly 4 or 5) the relevance and usefulness of our API misuse warnings. With the guidance of our API misuse warnings, the experimental group find and fix the bugs in Task-1/2/3 in a much shorter time than the control group. Furthermore, the experimental group rate the bugs in Task-1/2/3 less error prone if developers were given our API misuse warnings, compared with the error-prone ratings by the control group. While the participants modify the code in Task-3, IntelliJ reports an unhandled `FileNotFoundException` and suggest to use try-catch to fix the problem. Therefore, Task-3 obtains the best warning relevance and usefulness scores among the six tasks for the control group. Task-4 has the largest difference in answer correctness between the two groups. Only two participants in the control group find and fix the bug in Task-4 in the given 10 minutes time slot. Even these two participants spend about 8 minutes to complete the task. In contrast, all six participants in the experimental group successfully complete the task, with the average completion time 4 minutes. Task-4 is a erroneous call order problem (`validate()` should be called after `add()`). It is much harder to find the root cause of this call-order problem than missing condition-checking. Task-5 and Task-6 are missing-call problems. The difference of the answer correctness between the two groups are in between that of the easy Task-1/2/3 and the difficult Task-4. The experimental group still complete the Task-5/6 in shorter time than the control group, and rate the error proneness lower and the warning relevance/usefulness higher.

Our pilot user study demonstrates that our knowledge-driven API miuse detection is promising in assist developers in avoiding potential API misuses and debugging bugs caused by API misuses.

8 THREATS TO VALIDITY

As our approach extracts API caveat knowledge from API documentation, the quality of API documentation affects the effectiveness of our approach. Our approach is inspired by the studies [16, 19, 36] showing that high-quality documentation exist, especially for major SDKs, and they are an important source of information referenced for resolving API misuses [16, 32, 33].

Our approach has been tested on Java SDK 13. Other SDKs may describe the API knowledge in different document structures and sentence variants. Our experiments identify other types of API usage knowledge that should also be covered. As the schema of our API-constraint knowledge graph and our open information extraction pipeline are generic, we can adapt, extend and test our knowledge graph on other libraries and other types of API usage knowledge. We can also extend our knowledge graph to model API caveats from multiple versions of a SDK, which may support novel evolution analysis of API caveats/misuses.

Our approach achieves the start-of-the-art detection accuracy on MuBench. However, further experiments are required to confirm the generalizability of our approach on more APIs and their misuses, and unknown bugs beyond the benchmark. Furthermore, some advanced program analysis could be integrated in our tool to boost the detection recall. Our user study demonstrates the promising usefulness of our knowledge-driven API misuse detection. However, the scale of user study is small and in a control setting. In future, we would also like to use existing projects to evaluate our tool. additionally, we will work with industry partners to test the practicality of our approach and tool in real-world software development context, especially the usefulness of "non-bug" related API misuse warnings, regarded as false positives in this work.

9 RELATED WORK

API misuses are inevitable not only in source code [3, 31] but also in programming discussion forums [1, 7, 32, 33]. To reduce the risk of API misuses, many pattern-based API misuse detectors have been proposed [23, 25, 43, 44]. According to the systematic evaluation of the 12 detectors [4], most of them focus on detecting missing calls. Only four [2, 25, 30, 41] can detect missing condition checking or missing exception handling under special conditions. In contrast, our knowledge graph based detector can detect all three types of API misuses. Furthermore, Amann et al. [4] show that existing detectors achieve very low precision and recall. A recent detector MUDETECT [38] mines API usage graphs from cross-project code examples, which improves significantly the performance of pattern-based API misuse detectors. But its performance is still lower than our knowledge-graph based detection, except for the recall achieved by mining cross-project code examples. The recall of our approach can be improved by adding more types of API constraints in the knowledge graph. Readers are referred to Section 2 for the gap analysis of pattern-based and our knowledge-based methods.

Knowledge graph has emerged as a novel way of representing and reasoning software engineering knowledge [12, 47]. Two general API knowledge graphs [16, 19] have been constructed from API reference documentation and one task-oriented knowledge graph [36] has been constructed from programming tutorials. These knowledge graphs support entity-centric search of API caveats and programming tasks. Ren et al. [32, 33] use the extracted API-caveat sentences to distill erroneous code examples or explain controversial API usage on Stack Overflow. All these existing works treat API caveats as natural language sentences. In contrast, we develop sentence parsing and API linking techniques to infer fine-grained API-constraint relations from API-caveat sentences.

Robillard et al. [34] provides a survey on API property inference techniques, including Doc2Spec [48] which infers rules from text. Doc2Spec [48] and a similar work iComment [39] focus on call-order rules, while our approach infers not only call order but also condition checking, return condition and exception trigger. A recent work by Zhou et al. [49] infer simple parameter value and type restrictions from method comments. Our method infers much more sophisticated parameter and exception constraints from API caveat sentences (see Figure 3(b) and Figure 4(b)). Finally, different from the tools FindBugs [28], Pylint [40] that detect general programming anti-patterns, our tool detects misuses of specific APIs.

10 CONCLUSION AND FUTURE WORK

This paper presents the first knowledge-graph based API misuse detection method. Unlike existing pattern-based API misuse detectors, our method does not infer API misuses against API patterns in code. Instead, it detects API misuses against four types of API-constraint relations in a novel knowledge graph, derived from API reference documentation using NLP techniques. This knowledge graph advances the start-of-the-art in API misuse detection, and outperform existing pattern-based detectors by a large margin in precision and recall. The usefulness of our knowledge-graph based API misuse detection has also been demonstrated. In the future, we will enrich our knowledge graph with more types of API usage knowledge, support the chain effect analysis of API caveats, and support advanced program analysis to boost its recall. We will also evaluate the ability of our approach to battle the issues of API misuses, from prevention to detection to fixing.

11 ACKNOWLEDGEMENTS

This research was partially supported by the National Key R&D Program of China (No.2019YFB1600700), NSFC Program (No. 61972339), the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021), ANU-Data61 Collaborative Research Project(CO19314), and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–305.

[2] Mithun Acharya and Tao Xie. 2009. Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 370–384.

[3] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: a benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 464–467.

[4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[5] Chunyang Chen, Zhenchang Xing, and Ximing Wang. 2017. Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 450–461.

[6] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. 2016. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 744–755.

[7] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. 2019. How reliable is the crowdsourced knowledge of security implementation?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 536–547.

[8] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 502–518.

[9] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.

[10] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 50–61.

[11] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 38–49.

[12] Zhuobing Han, Xiaohong Li, Hongtao Liu, Zhenchang Xing, and Zhiyong Feng. 2018. Deepweak: Reasoning common software weaknesses via knowledge graph embedding. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 456–466.

[13] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 293–304.

[14] J Richard Landis and Gary G Koch. 1977. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics* (1977), 363–374.

[15] Woong Ki Lee, Yeon Su Lee, Hyoung-Gyu Lee, Won Ho Ryu, and Hae Chang Rim. 2012. Open Information Extraction for SOV Language Based on Entity-Predicate Pair Detection. In *Proceedings of COLING 2012: Demonstration Papers*. 305–312.

[16] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[17] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[18] Christian Lindig. 2015. Mining patterns and violations using concept analysis. In *The Art and Science of Analyzing Software Data*. Elsevier, 17–38.

[19] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 120–130.

[20] Walid Maalej and Martin P Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282.

[21] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *arXiv: Computation and Language* (2013).

[23] Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*. Springer, 2–25.

[24] Martin Monperrus and Mira Mezini. 2013. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 1–25.

[25] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. 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*. 383–392.

[26] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2015. Recommending API usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 795–800.

[27] "Artifact Page". 2017. [Online]. Available: http://www.st.informatik.tu-darmstadt.de/artifacts/mustudy/.

[28] Bill Pugh and David Hovemeye. 2015. FindBugs. http://findbugs.sourceforge.net/.

[29] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-sensitive inference of function precedence protocols. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 240–250.

[30] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. *ACM SIGPLAN Notices* 42, 6 (2007), 123–134.

[31] Anastasia Reinhardt, Tianyi Zhang, Mihir Mathur, and Miryung Kim. 2018. Augmenting stack overflow with API usage patterns mined from GitHub. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 880–883.

[32] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. [n.d.]. Demystify Official API Usage Directives with Crowdsourced API Misuse Scenarios, Erroneous Code Examples and Patches. ([n. d.]).

[33] Xiaoxue Ren, Zhenchang Xing, Xin Xia, Guoqiang Li, and Jianling Sun. 2019. Discovering, Explaining and Summarizing Controversial Discussions in Community Q&A Sites. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 151–162.

[34] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2012. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2012), 613–637.

[35] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling*. Vol. 15. Springer Science & Business Media.

[36] Jiamou Sun, Zhenchang Xing, Rui Chu, Heilai Bai, Jinshui Wang, and Xin Peng. [n.d.]. Know-How in Programming Tasks: From Textual Tutorials to Task-Oriented Knowledge Graph. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 257–268.

[37] Joshua Sushine, James D Herbsleb, and Jonathan Aldrich. 2015. Searching the state space: A qualitative study of API protocol usability. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 82–93.

[38] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275.

[39] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[40] Sylvain Thenault et al. 2006. PylintÂ¡ÂªCode Analysis for Python.

[41] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 283–294.

[42] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 496–506.

[43] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3-4 (2011), 263–292.

[44] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. 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*. 35–44.

[45] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 51–62.

[46] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*. ACM, 404–415.

[47] Xuejiao Zhao, Zhenchang Xing, Muhammad Ashad Kabir, Naoya Sawada, Jing Li, and Shangwei Lin. 2017. HDSKG: Harvesting domain specific knowledge graph from content of webpages. (2017), 56–67.

[48] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

[49] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.