# KGAMD: An API-Misuse Detector Driven by Fine-Grained API-Constraint Knowledge Graph

Xiaoxue Ren
xxren@zju.edu.cn
Zhejiang University
China

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

Zhenchang Xing
zhenchang.Xing@anu.edu
Australian National University
Australia

Xin Xia
xin.xia@acm.org
Monash University
Australia

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

Liming Zhu
Liming.Zhu@data61.csiro.au
CSIRO
Australia

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

## ABSTRACT

Application Programming Interfaces (APIs) typically come with usage constraints. The violations of these constraints (i.e. API misuses) can cause significant problems in software development. Existing methods mine frequent API usage patterns from codebase to detect API misuses. 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 constraints in terms of comprehensiveness, explainability and best practices. Inspired by this, we propose a novel approach named KGAMD (API-Misuse Detector Driven by Fine-Grained API-Constraint Knowledge Graph) that detects API misuses directly against the API constraint knowledge, rather than API usage patterns. We first construct a novel API-constraint knowledge graph from API reference documentation with open information extraction methods. 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. Then, we develop the KGAMD tool that utilizes the knowledge graph to detect API misuses. There are three types of frequent API misuses we can detect - missing calls, missing condition checking and missing exception handling, while existing detectors mostly focus on only missing calls. Our quantitative evaluation and user study demonstrate that our KGAMD is promising in helping developers avoid and debug API misuses.
Demo Video: *https://www.youtube.com/watch?v=TN4LtHJ-494*
IntelliJ plug-in: *https://github.com/goodchar/KGAMD*

## CCS CONCEPTS

• **Software and its engineering** → *Data types and structures.*

## KEYWORDS

API Misuse, Knowledge Graph, Java Documentation

**ACM Reference Format:**
Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2021. KGAMD: An API-Misuse Detector Driven by Fine-Grained API-Constraint Knowledge Graph. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473112

## 1 INTRODUCTION

Application Programming Interfaces (APIs) often have usage caveats, such as constraints on call order or value/state conditions. For instance, 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. Existing work has developed some pattern-based tools to detect API misuses via static code analysis. These pattern-based methods mine frequent API usage patterns from codebase, and make a naive assumption that any deviations with respect to these patterns are potential misuses [2]. Amann et al. [2] conduct a systematic evaluation and reveal that all pattern-based API-misuse detectors suffer from low precision (0-11%) and recall (0-20%) in practice.

To improve detection results, some methods attempt to mining from bigger codebase through code search engine [8], which can obtain more API usage patterns [4], or set up more robust probabilistic models of deviation [5]. However, none of these improvements go beyond the naive assumption of pattern-based API-misuse detection. In this case, to abandon such naive assumption, we propose a knowledge-driven approach named API-Misuse Detector Driven by Fine-Grained API-Constraint Knowledge Graph (KGAMD), which detects API misuses against a novel API-constraint knowledge graph derived from API documentation, rather than API usage patterns.

API reference documentation provides rich knowledge of API usage caveats [3]. Although IDEs provide direct access to API documentation, the current semi-structured API documentation cannot help solve the API misuse problems directly [3]. To improve the accessibility of API-caveat knowledge, Li et al. [3] use Natural Language Processing (NLP) techniques to construct an API-caveat knowledge graph from API documentation. This knowledge graph supports API-centric search of caveat sentences. However, caveat sentences cannot be directly used to detect API misuses in source code in natural language form.

In this paper, we propose an API-constraint knowledge graph by integrating API constraint relations into current semi-structured API knowledge graph: the entities include API elements and value literals, and the edges include two categories: declaration relations and constraints relations (i.e., call-order, condition-checking, return-condition, exception-trigger) (see Section 2.1). Different from existing API knowledge graphs [3] that capture only declaration relations and simply link API-caveat sentences to an API as its attributes, we utilize NLP techniques to extract API constraint relations from API-caveat sentences. Compared with existing methods that infer specifications from text [7], our approach infers more types of and more informative API constraints. Given a program, after representing it into a graph, such as Abstract Syntax Tree, we link the program elements with the API entities in our API-constraint knowledge graph. By analyzing target API constraints of the linked entities in our knowledge, our approach reports violates as API misuses and explains the detected misused by relevant API caveats.

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 [6], 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 [1], 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 [2]. 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.

## 2 APPROACH

Figure 1 shows the overall framework of our approach of constructing API-constraint knowledge graph and detecting API misuses. Moreover, it also contains an example of "filereader" method, which shows how our approach works to detect API misuses and report warnings to developers. Our API-constraint knowledge graph significantly extends existing general API knowledge graphs [3] (referred to as API declaration graph (see Section 2.3) in this work) with four types of fine-grained API constraint relations, which are derived from API caveat sentences (see Section 2.4). These API constraint relations correspond to the three most-frequent API misuse categories in the API misuse benchmark MuBench [2] (see Section 2.1). We develop a novel knowledge-driven API misuse detector (KGAMD) that checks the program for API misuses against the API-constraint knowledge graph (see Section 2.5).

### 2.1 Knowledge Graph Schema

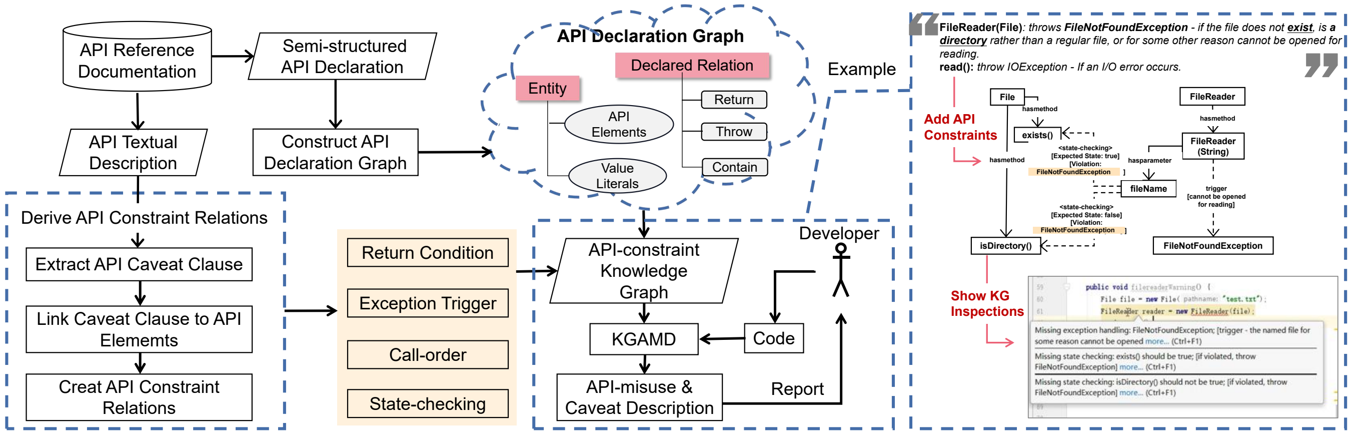
The knowledge graph entities include *API elements* (*package*, *class*, *exception*, *method*, *parameter* and *return-value*) and *value literals*. We distinguish exception from class to facilitate exception handling analysis. An entity has a *name* (null for return-value). A method or parameter entity also 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.

Relations of our knowledge graph include *declaration relations* and *constraint relations*. The declaration relations include an API element *contains* another API element, a method *returns* a value-literal, and a method *throws* an exception. Different from the existing API knowledge graphs [3], we attach *condition* attribute to return relations. It helps identify the situation where using the return value of one method as the argument of the other method may cause program errors. Furthermore, we attach *trigger* attribute to throw relations, which records the exception situation that cannot be prevented by certain pre-condition checking.

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 [2]: missing-call and missing-condition-checking. A call-order relation can be either *precede* or *follow*. It may have an optional *condition* attribute for the preceding/following method call. Call-order is not transitive, and precede-follow are not symmetric. A condition-checking relation can be either *value checking* or *state-checking*. The value-checking records the *expected expression* (e.g., !=, <), and the state-checking records the *expected state*(e.g., true). 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.

### 2.2 API Documentation and Preprocessing

To construct API-constraint knowledge graph as defined above from API reference documentation, We crawl online API documentation using web crawling tool and keep semi-structured API declarations and API textual descriptions following the treatment in [3].


API Reference Documentation
Semi-structured API Declaration
API Textual Description
Construct API Declaration Graph
API Declaration Graph
Declared Relation
Entity
API Elements
Value Literals
Return
Throw
Contain
Example
Derive API Constraint Relations
Extract API Caveat Clause
Link Caveat Clause to API Elements
Creat API Constraint Relations
Return Condition
Exception Trigger
Call-order
State-checking
API-constraint Knowledge Graph
Developer
KGAMD
Code
API-misuse & Caveat Description
Report
Add API Constraints
File
hasmethod
exists()
<state-checking> [Expected State: true] [Violation: FileNotFoundException ]
hasmethod
isDirectory()
<state-checking> [Expected State: false] [Violation: FileNotFoundException ]
fileName
hasparameter
FileReader
hasmethod
FileReader (String)
trigger [cannot be opened for reading]
FileNotFoundException
Show KG Inspections

**FileReader(File)**: *throws* ***FileNotFoundException*** *- if the file does not* ***exist****, is* ***a directory*** *rather than a regular file, or for some other reason cannot be opened for reading.*
**read():** *throw IOException - If an I/O error occurs.*

```
public void filereaderWarning() {
    File file = new File( pathname: "test.txt");
    FileReader reader = new FileReader(file);
```

Missing exception handling: FileNotFoundException; [trigger - the named file for some reason cannot be opened more... (Ctrl+F1)

Missing state checking: exists() should be true; [if violated, throw FileNotFoundException] more... (Ctrl+F1)

Missing state checking: isDirectory() should not be true; [if violated, throw FileNotFoundException] more... (Ctrl+F1)

**Figure 1: The Overall Framework of Our Approach**

## 2.3 Constructing API Declaration Graph

First, we construct an API declaration graph from the semi-structured API declarations of official API documentation. As our API declaration graph is the same as the generic API knowledge graph in [3], 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 2.1). We use brief introduction sentences of each method in the method summary section as that method’s functionality description. To detect API misuses related to API chain calls (e.g., substring(indexOf())), we extend the original parser in [3] to extract more fine-grained return relations.

## 2.4 Deriving API Constraint Relations

Our API-constraint knowledge graph also contains call-order and condition-checking relations between related APIs, and constraint-enriched return and throw relations, compared with existing API knowledge graphs [3].

### 2.4.1 *Extracting API-Caveat Clauses.*

As we focus on API-method usage constraints, we limit the extraction to the main description of each method 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 translate the extracted caveat sentences into fine-grained API-caveat clauses by resolving co-reference, splitting sentences into clauses and clustering similar clauses, to facilitate the subsequent API linking and constraint relation inference.

### 2.4.2 *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 can infer methods or parameters, whose functionality descriptions match the caveat description. 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.

### 2.4.3 *Creating API Constraint Relations.*

After obtaining a caveat clause, we analyze its API linking results and create constraint relations according to some heuristic rules (please refer to our paper for more details about heuristic rules).

## 2.5 API Misuse Detection

We develop a knowledge-driven API misuse detector named KGAMD, 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 $api_{kg}$.

## 3 IMPLEMENTATION DETAILS

## 3.1 Knowledge Base

We construct an API-constraint knowledge graph for Java SDK APIs. Using the web page parser, 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 API-caveat patterns developed by Li et al. [3], 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 plug-in which detects API misuses in Java programs based on the constructed API-constraint knowledge graph.

## 3.2 Tool Implementation and Usage

As our API-constraint knowledge graph focus on four constraint relations, which can refer to three types of API misuses: exception trigger, call order and condition checking, exception trigger and return condition, our KGAMD detects them and gives three types of warnings respectively: missing call, missing condition checking and missing exception handling. For the call-order relation, KGAMD 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 $(exp_p \land \neg \mathrm{exp}_{kg}) \lor (\neg exp_p\ \mathrm{exp}_{kg})$ is satisfied by a SAT solver. 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.

The example in Figure 1 also shows the user interface of IntelliJ IDE plugin. In the example, we assume a developer want to use "*FileReader*" to read the content of "test.txt". Our knowledge-driven API misuse detector gives three warnings like the picture shows: The first warning is about exception handling, which is caused by the constraint clause "*FileReader(File): throws FileNotFoundException - if the file for some other reason cannot be opened for reading.*" The second is condition checking caused by "*FileReader(File): throws FileNotFoundException - if the file does not exist.*" And the last one is also condition checking, which is extracted from the constraint "*FileReader(File): throws FileNotFoundException - if the file is a directory rather than a regular file.*" Compared to our KGAMD, IntelliJ IDE can also give a warning, but just tell the developer there may be an exception and please use "*try-catch*" around the code snippets. This warning is quite general and cannot give us a better solution. As the three constraint clauses, where we generate the warnings, are split from an API usage constraint, our KGAMD can give a better solution to deal with "*FileNotFoundException*". And with the warnings, we can fully utilize API documentation to tell the developer why we should make the condition checking and exception handling.

## 4 EVALUATION

### 4.1 Quality and Effectiveness

To evaluate the quality of KGAMD, we first evaluate the quality of the constructed knowledge graph. We focus on the four types of API constraint relations, which distinguish our API-constraint knowledge graph from existing general API knowledge graphs [3]. We use a statistical sampling method to examine *MIN* randomly sampled instances of each type of constraint relation. For each API constraint relations, we define several checkpoints to evaluate. The two developers independently perform the examination, and all decisions are binary. From results, our method achieves high accuracy (>98%) for extracting value-literals, return-condition clauses and attributes of call-order and condition-checking relations.
Next, we evaluate the effectiveness of our KGAMD using the API misuse benchmark MuBench [1]. From MuBench, 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. We apply our tool to examine how many of the 239 API misuses can be detected. We also examine if the explanation that our tool provides for the detected misuse matches the misuse description in the benchmark. From results, our KGAMD can detect missing calls, missing condition checking and missing exception handling with good precision, which is 60.18%, and better recall, which is 28.45%. To improve the recall of KGAMD, more types of API usage knowledge should be extracted and added to the underlying knowledge graph, and some advanced programming analysis should be supported.

### 4.2 User Study

To investigate how developers can interact with our KGAMD, we conduct a user study. We select 6 API misuse scenarios from the webtend project in MuBench [1] as user tasks. Our KGAMD 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. We randomly allocate them into two 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. Readers can check our research papers for more details about the user study. From the results of the pilot user study, our KGAMD is promising to assist developers in avoiding potential API misuses and debugging bugs caused by API misuses.

## 5 CONCLUSION AND FUTURE WORK

We propose the first knowledge-graph based API misuse detector (KGAMD). Unlike existing pattern-based API misuse detectors, KGAMD does not infer API misuses against API patterns in code. It detects API misuses against four types of API-constraint relations in a novel knowledge graph, which are 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 KGAMD 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.

## ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation of China (No. U20A20173) and Key Research and Development Program of Zhejiang Province (No.2021C01014).

## REFERENCES

[1] 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.
[2] 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.
[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.