# Detecting API-Misuse based on Pattern Mining via API Usage Graph with Parameters

Yulin Wu[1], Zhiwu Xu[1,*], and Shengchao Qin[2]

[1] College of Computer Science and Software Engineering, Shenzhen University
[2] School of Computer Science and Technology, Xidian University
yulin5@foxmail.com, xuzhiwu@szu.edu.cn, shengchao.qin@gmail.com

**Abstract.** API misuse is a common issue that can trigger software crashes, bugs, and vulnerabilities. To address this problem, researchers have proposed pattern-based violation detectors that automatically extract patterns from code. However, these detectors have demonstrated low precision in detecting API misuses. In this paper, we propose a novel API misuse detector. Our proposed detector initially extracts API usages from the code and represents them as API Usage Graphs with Parameters (AUGPs). Utilizing the association rule algorithm, it then mines the binary rules, which are subsequently employed to detect the possible violations. The experimental results show that, comparing against five state-of-the-art detectors on the public dataset MuBench, our detector achieves the highest precision (1x more precise than the second-best one) and the highest F1-score (50% higher than the second-best one).

**Keywords:** API-Misuse Detection · API Pattern · Static Analysis.

## 1 Introduction

Modern software development relies heavily on Application Programming Interfaces (APIs) that enable developers to expedite project development. However, APIs often have usage limitations, such as call ordering and conditions. For instance, in Java cryptography APIs, cipher objects must be initialized by using the API *init()* before invoking the API *doFinal()* for encryption. Deviation from API usage restrictions are referred to as API misuses. Unfortunately, API misuses are prevalent in software development, and may yield software crashes, bugs, and vulnerabilities. Zhang et al. [25] conducted a thorough analysis of $217k$ Stack Overflow posts, revealing potential API misuses in 31% of them.

To address this problem, researchers have proposed pattern-based violation detectors. CrySL [10], for instance, is a API misuse detector relying on hand-crafted patterns. However, it poses a challenge for users to manually draft API usage patterns. Moreover, utilizing these patterns for detection frequently renders an excessive amount of false positives (incorrect patterns) or false negatives (incomplete patterns) [11]. Therefore, automated detectors [23, 15, 22, 19, 2] are proposed, where patterns are first extracted from the code and then utilized to detect violations. Nevertheless, these detectors currently demonstrate a high rate of false positives. Thus, more effective detectors are required.

In this paper, we propose a novel approach to detect API misuses. Specifically, we utilize data flow analysis to extract API usages from code and represent them as our proposed graph model called API usage graph with parameters (AUGP). To enrich the API usages, we also perform the

* Corresponding author.

inter-procedural analysis to capture the relevant APIs that are encapsulated in some functions from the client code, which are referred to as *client functions* in this paper. Then we use the association rule algorithm FP-growth to mine the binary relationships between APIs, yielding API usage rules. Guided by these usage rules, we propose the violation detection and the order detection to detect API misuses. Finally, we also provide a scoring mechanism to filter and rank the reported violations.

We have implemented our approach as a tool APDetect, based on WALA. To evaluate APDetect, we conducted experiments on the publicly available dataset MuBench [1] as well as to compare against five state-of-the-art detectors. The results show that APDetect achieves the highest precision rate 56.00% (31.5% larger than the second-best one) and the highest F1-score 0.54 (0.18 larger than the second-best one) on the 10 projects selected for manual analysis. Furthermore, APDetect reports the fewest number 102 of violations (114 fewer than the second-fewest one and 20276 fewer than the most one) on all the 31 projects. These findings demonstrate that our approach is highly effective and outperforms several existing detectors.

The remainder of this paper is organized as follows. Section 2 introduces some preliminaries. Section 3 describes our approach, followed by the experimental results in Section 4. Section 5 presents the related work, followed by some concluding remarks in Section 6.

## 2 Preliminaries

In this section, we present data flow analysis and association rule learning used in the paper.

### 2.1 Data Flow Analysis

Data flow analysis is an analysis technique used to capture the flow of relevant data along a program. A classic way for data flow analysis, given in Algorithm 1, is to generate data flow equations for the nodes of a Control Flow Graph (CFG) and then solve them by iterative computation until the fixed point is reached. In this paper, we use reachable definitions to trace the data flow of relevant data.

### 2.2 Association Rule Learning

Let $I = \{i_1, i_2, ..., i_m\}$ be a non-empty set of items and $D = \{t_1, t_2, ...t_n\}$ be a non-empty set of transactions, where a transaction $t$ is a non-empty subset of $I$, that is, $t \subseteq I$. An association rule [7] is an implication of the form $X \rightarrow Y$, where $\emptyset \subset X, Y \subseteq I$, $X \cap Y = \emptyset$, and $X$ and $Y$ are called the antecedent (or left-hand-side, LHS) and consequent (or right-hand-side, RHS) of the association rule, respectively. The support of the association rule $X \rightarrow Y$ in $D$ is the percentage of transactions in $D$ that contain both $X$ and $Y$ (i.e., the probability $P(X \cup Y|D)$), and the confidence is the percentage of transactions that contain both $X$ and $Y$ in the ones that contain $X$ (i.e., the conditional probability $P(Y|X)$). An association rule is considered favorable or useful if both the minimum support threshold and the minimum confidence threshold are satisfied.

Association rule learning (ARL) is a machine learning technique to discover interesting relationships between variables in large databases. It aims to identify strong association rules found in a database using some interesting measures. In this paper, we use the association rule algorithm FP-growth [8] to mine the relationships between APIs.

**Algorithm 1:** Data flow analysis

**Input:** CFG
**Output:** INPUT[B] and OUTPUT[B] for each basic block B in CFG

```
for basic block B in CFG do
    OUTPUT[B] ← ∅;
end
while OUTPUT has any changes do
    for basic block B in CFG do
        INPUT[B] = ⋃_P OUTPUT[P] // P is a predecessor of B
        OUTPUT[B] = gen_B ∪ (INPUT[B] − kill_B) // gen_B and kill_B denotes the
            variable sets generated or killed by B, respectively
    end
end
```

## 3 Methodology

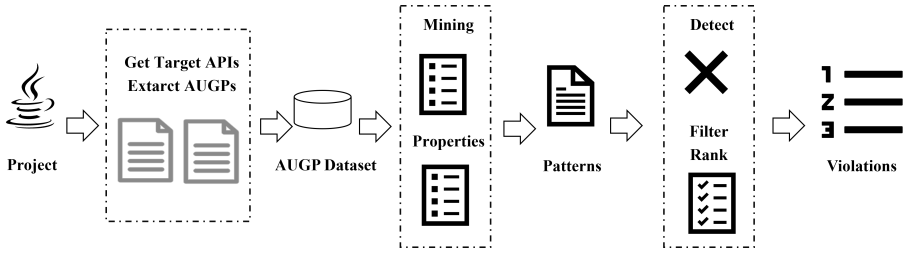


**Fig. 1.** Framework of Our Approach

In this section, we will present our approach to detect API misuses. Figure 1 shows the framework of our approach, which consists of three steps: AUGP extraction, pattern mining, violation detection.

### 3.1 AUGP Model

Graph-based Object Usage Models (GROUM) [19] and API Usage Graphs (AUG) [2] are graph models commonly used to represent API usages. However, both models are generated based on individual functions, which may result in irrelevant information or lack of critical information, undermining the effectiveness of pattern mining and violation detection. To address this issue, we propose a new graph model, called API Usage Graphs with Parameters (AUGP). AUGP focuses specifically on the APIs of interest (referred to as target APIs in this paper), their associated parameters, and the data relations between them. This enables us to effectively capture relevant APIs by filtering out irrelevant information to achieve better performance in pattern mining and violation detection.

AUGP are directed graphs with labels. Nodes in AUGP are classified into two types: instruction nodes and data nodes. Instruction nodes are used to represent the IR instructions of WALA as well

as some generated PHI nodes during the construction of AUGP. According to the instruction kinds of WALA, instruction nodes can be further classified into method call nodes, field access nodes, PHI nodes, and so on. Data nodes, on the other hand, represent parameters, variables or input/output data used in the IR instructions. Adding data nodes to AUGP ensures the logical integrity of the API usages and facilitates the inter-procedural construction.

Edges in AUGP are classified into two types: use edges and parameter edges. Use edges are used to capture the arguments used by an instruction node. An instruction may have several arguments, which will been marked by different numbers starting from 1, indicating their different orders (i.e., positions). In particular, we use *object* to denote the object of the function (i.e., functions that require a special argument *self*) if the instruction is a function call. Parameter edges are generated to connect graphs of functions during the inter-procedural construction. Likewise, numbers are marked for different parameters.

### 3.2 AUGP Extraction

The key idea to construct AUGP for target APIs is to trace the data flow for each non-trivial parameters of the APIs, which are non-constant and whose types are non-primitive, such as user-defined classes.

Given a (Java) project and some target APIs (such as from a library), prior to constructing AUGPs, we build a API location tree to record the client functions in the project that invoke at least one of the target APIs and their invoking orders of APIs. The API location tree allows a quick and efficient check to determine the presence of a client function invoking a target API or another API usage after some AUGPs have been extracted.

**Listing 1.1.** Java encryption code examples

```
public byte[] encrypt(String content, String slatKey, String vectorKey) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    initWithAES(cipher, slatKey, vectorKey);
    byte[] encrypted = cipher.doFinal(content.getBytes());
    return encrypted;
}

public void initWithAES(Cipher cipher, String slatKey, String vectorKey)
        throws InvalidKeyException, InvalidAlgorithmParameterException {
    SecretKey secretKey = new SecretKeySpec(slatKey.getBytes(), "AES");
    IvParameterSpec iv = new IvParameterSpec(vectorKey.getBytes());
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);
}
```

To build the API location tree, we just simply perform a text search on the given project function by function and line by line. We also consider the inter-procedural analysis, as some programmers may wrap APIs with inter-procedural as a client function of their projects. For example, as shown in Listing 1.1, the initialization of cipher with AES is encapsulated as the client function *initWithAES*. And to avoid the recursive functions as well as too much information leading to the interference, we limit the depth of the function call. The location tree is easily extended to support inter-procedural analysis: to mark each API with its depth and order. Take the code in Listing 1.1 for example. Assume 'javax.crypto' is the target library (all its APIs are of interesting). The API location tree

starting from the function *encrypt* (with depth 2) is shown in Figure 2, where the numbers in the blue circles and in the red circles respectively denote the depths and the orders of the corresponding APIs (and functions).

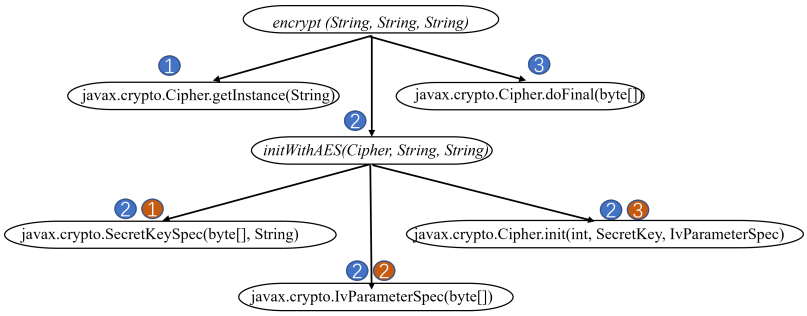


**Fig. 2.** API Location Tree Starting from Function *encrypt*

Then we construct AUGPs for each non-empty API location tree starting a function in the given project. Different from GROUM and AUG, we focus on the data flows between the target APIs. So we perform data flow analysis on the corresponding functions, guided by the API location tree. The algorithm for our AUGP construction is given in Algorithm 2, which takes an API location tree $T$ as input, and returns a set of AUGPs $GS$.

Generally, the rightmost leaf of an API location tree starting from function $f$ is the last target API called by the function $f$. From this API, we are more likely to be able to extract the possibly full usage of the target APIs. So we start with an empty graph $G$ (line 3) and the rightmost (unvisited) leaf of the given location tree (line 4), which is added into the current graph $G$ (line 5). Then we perform an intra-procedural (backward) data flow analysis on the client functions where the starting API locates (lines 6-12) and expand the graph $G$ accordingly, yielding an intra-procedural AUGP. In detail, we start with an instruction set $IS$ containing only the stating API. For each instruction $I$ in $IS$, we resolve its parameters (line 9) and add them into the graph $G$ (line 10). Moreover, these parameters can be instructions, variables or constants and some instructions can be further traced (line 11). In particular, if the instruction parameter involves the target APIs or an object is typed of some specific class of the target library (referred to as a target object), then it will be added in the instruction set $IS$ for further analysis. During the analysis, for function call instructions, it is difficult to know exactly whether the called function modifies the target objects or not. So we conservatively consider all the functions would modify the target objects. Besides, if a target object is created by this instruction parameter, we will stop the trace of this target object, that is, the parameter would not be added into the instruction set $IS$.

Afterwards, we consider the inter-procedural analysis. For that, we check whether there are some client functions in $T$ occurring in $G$ as well (line 13). If so, we resolve the information of all the client functions (line 14), that is, the starting points (*i.e.*, calling sites) of the client functions, the target object considered by the analysis, and the IR instructions of the client functions. Then for each client function, we expand the current graph with its program slice with respect to the target object (lines 15-16), which can be done via an intra-procedural analysis on the client function as well.

We continue on the client function checking, until no client functions are found (lines 13-18). In that case, the graph $G$ is a possibly full usage, as no more statements within the location tree $T$ can be added into $G$. And, apparently, the graph $G$ is an inter-procedural AUGP. We add $G$ into $GS$ (line 19) and mark the target API that occur in both $T$ and $G$ as visited (line 20). Finally, we continue on checking there are still some unvisited target APIs in $T$, and if so, we repeat to construct AUGPs for the unvisited APIs (lines 2-20).

**Algorithm 2:** BuildAUGP

**Input:** API location tree $T$
**Output:** AUGP set $GS$

```
1  GS ← ∅;
2  while T contains some unvisited leaves do
3      G ← an empty graph;
4      APICall ← the rightmost unvisited leaf of T;
5      add(G.nodes, APICall);
6      IS ← {APICall};
7      while IS is not empty do
8          I ← pop IS;
           // get parameters from data flow analysis results
9          paras ← getParameters(I);
10         add(G, paras);
11         add_nontrivial(IS, paras);
12     end
13     while G contains some client functions in T do
14         funs ← getClientFuns(G, T);
15         for fun ∈ funs do
               // building AUGP for fun and expend G
16             expand(G, fun.start, fun.object, fun.body);
17         end
18     end
19     GS ← GS ∪ {G};
20     mark(T, G);
21 end
```

Consider the code in Listing 1.1 again and one of its API location trees is given in Figure 2. We start with the rightmost API *doFinal* (line 4) and perform a data flow analysis on the client function *encrypt()* containing *doFinal*.

We first resolve two parameters *cipher* and *content.getBytes()*[3] from *doFinal*. Then we trace backward with respect to *cipher* and get the function call instruction (line 3). As we conservatively consider the function call instruction is a write for *cipher*, so *cipher* is "killed" (*i.e.*, the value of the object has been modified). But we will continue on the data flow analysis with the parameters of this function call instruction. And similar to *content.getBytes()*. The top-level graph is shown in the left part of Figure 3, where the square nodes represent the instruction ones (such as function/API call), the oval nodes represent the data ones (such as the arguments of functions), edges labelled by *object* indicate the parameter is used as the instance object calling the corresponding function, edges

[3] In WALA, there would be a temporary variable for this expression.

labelled by *used_n* indicate the n-th argument of a target API or a client function not occurring in the location tree, edges labelled by *parameter_n* indicate the n-th argument of a client function occurring in the location tree.

After that, we found a client function *initWithAES()* related to the target object *cipher*. So we expand the graph with this function, and get the final graph, which is shown in Figure 3. Note that, with inter-procedural analysis, we are able to extract the data relation between *Cipher.doFinal* and *Cipher.init*. AUGP retains the package name, class name, parameter type, and return type. For presentation convenience, the functions in Figure 3 omit the parameter type and return type.

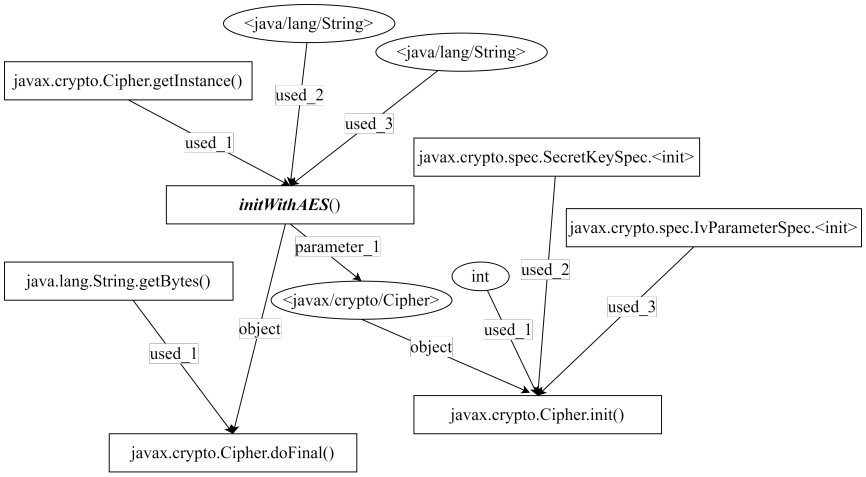


**Fig. 3.** AUGP for code in Listing 1.1

Concerning the target APIs, they can be provided by the users. But we also present a method to obtain the target APIs automatically for a given project: we scan the given project function by function and record the number of the classes of the APIs that are called in the project, then we take the classes whose numbers are larger than a given threshold and generate their corresponding APIs as the target ones. The more occurrences of (the classes of) APIs, the more probability of mining valid patterns.

### 3.3 Pattern Mining

AUGPs can capture API usages. We can use graph mining on AUGPs directly, but the result seems not good: only a small number of API usage patterns are obtained, this is because API usages are flexible, such as an API usage may have different conditional branches or wrap in different function with different inter-procedural, making the different usages interfere with each other. Moreover, applying graph patterns on violation detection prones to yield more false positives.

Therefore, we employ a looser data structure (i.e., set) and focus on binary relations between APIs. We convert each graph to a set via simply removing the edges and preserving only non-client functions. Next, we use the association rule algorithm FP-growth [8] to mine the relationships between two APIs. Our rule is an implication expression of the form $API_1 \rightarrow API_2$,

which indicates that the antecedent $API_1$ occurring in a code snippet, then the consequent $API_s$ should occur as well, no matter the orders between them. For example, a rule for the example code in Listing 1.1 may be *javax.crypto.Cipher.doFinal* $\rightarrow$ *javax.crypto.Cipher.init*, stating that if *javax.crypto.Cipher.doFinal* is called, then *javax.crypto.Cipher.init* should be called as well.

Prior to pattern mining, some graphs are filtered: (1) graphs with too fewer nodes or edges (which contains little information); (2) graphs with fields of some classes (it is not easy to trace all statements that involve this field); (3) graphs with depth larger than 3 (which are found to be expandable and more likely to be incomplete).

To facilitate violation detection, we summarize several properties for each association rule:

- **Relevance.** A function is said to be relevant if one of its class name, parameters or return type is consistent with the target API class. A rule $API_1 \rightarrow API_2$ is said to be high relevant (or low relevant or irrelevant, resp.) if both (or one or none, resp.) of $API_1$ and $API_2$ are relevant.
- **Order.** A rule $API_1 \rightarrow API_2$ is said to be in positive (or negative, resp.) order if $API_1$ should appear before (or after, resp.) $API_2$. If the order can not be determined, we said the rule is a unordered one.
- **Conditional Rule.** A rule $API_1 \rightarrow API_2$ is said to be conditional if one of $API_1$ and $API_2$ returns a Boolean value.

The relevance of a rule is easy to determine. The conditional property is helpful, as several APIs need to pass some conditional check before calling, such as the APIs *hasNext* and *next* in the class *Iterator*. We conservatively consider the rules wherein one of their APIs returns a Boolean value are conditional ones. Finally, we estimate the orders of APIs in terms of reachability. For each AUGP, we first remove the loops in it via disconnecting the end of loop statement and its corresponding begin, and calculate the reachabilities (i.e., transitive closure) between all non-client functions. Then for $f_1$ and $f_2$, we count the number $n_{f_1,f_2}$ ($n_{f_2,f_1}$, resp.) of graphs wherein $f_1$ can reach $f_2$ ($f_2$ can reach $f_1$ resp.). If $n_{f_1,f_2} > 0$ and $n_{f_1,f_2} > 4 * n_{f_1,f_2}$, then the order of $f_1$ and $f_2$ is $f_1, f_2$. Similar for the order $f_2, f_1$. Otherwise, they are unordered.

### 3.4 Violation Detection

In this section, we present two kinds of violation detection, that is, co-occurrence detection and order detection, and the ranking of violations.

**Co-Occurrence Detection.** As mentioned above, if the antecedent of a rule occurs, then does the consequent. In other words, once the antecedent is called in a code snippet but the consequent is not, then the code snippet is considered as a suspicious violation. For the conditional rules, we perform a further check on a suspicious violation: whether there exists another condition check that involves an API sharing the same object/class with the missing one. This is because there may be some other APIs that can achieve the equivalent conditional check. For example, *list.size() == 0* and *list.isEmpty() == true* achieve the equivalent conditional check.

**Order Detection.** The order detection is a complement to the co-occurrence detection. When both APIs of an association rule occur and the association rule is ordered, we will perform the order detection. Generally, there are four possible orders in (the AUGP of) the code snippet: $API_1, API_1$, $API_1, API_2$, $API_2, API_1$, and $API_2, API_2$, assuming that either $API_1$ or $API_2$ may occur multiple times. The order that matches the rule is called consistent, while the other cases are inconsistent. Clearly, according to the estimation of order in Section 3.3, the consistent order should occur many times. But the inconsistent ones may not be a misuse. Consequently, we set a minimum threshold.

If the frequency of an inconsistent order is smaller than this threshold, we consider it as a suspicious violation.

Nevertheless, after analyzing a large amount of code, we found that two cases are more likely to lead to a false positive:

- The frequency of the antecedent in the code snippet and the one of the consequent are almost the same;
- When the rule is conditional, the frequency of the conditional API in the code snippet is greater than the one of the other API of the rule.

So we exclude them. One can also think that the order check is unnecessary for these two cases.

**Filtering and Ranking.** After the co-occurrence detection and order detection, we get a list of suspicious violations. To evaluate these suspicious violations, we score the rules they violate, yielding a ranking for them.

**Table 1.** Scores for Association Rules

| **Properties** | **Score** |
|---|---|
| Positive Ordered | 5 |
| Negative Ordered, Conditional | 4 |
| Positive Ordered, Conditional | 3 |
| Negative Ordered | 2 |
| Unordered | 1 |
| High Relevant | 5 |
| Low Relevant | 2 |
| Irrelevant | -5 |
| Too Many Violations | -5 |

Table 1 shows the scores for the association rules. We focus on the positive ordered rules and high-relevant rules, so they are assigned the highest score. While the irrelevant rules and the ones that are violated too many times should be rejected by one vote (i.e., assigned the lowest score), so as not to lead to a false positive. The unordered rules are more likely to have little effect, so they would have lower scores. The conditional rules are a bit special: the negative order have higher score than the positive one, as the conditional API always appears as a consequent of a association rule.

A violation may violate several rules, we score it by the highest score among its violated rules. A violation is considered as a misuse if its score is not smaller than 6, in other words, all suspicious violations with scores smaller than 6 will be filtered out. After filtering, we can rank the list of violations.

## 4 Evaluation

In this section, we will introduce the data set used in the experiments as well as the experimental results.

### 4.1 Dataset

We implemented our approach as a detector *APDetect*, based on WALA[4]. So APDetect requires the projects in Java compiled bytecode. APDetect targets at the detection of a single project. Existing detectors of the same type in Java are DMMC [15], JADET [23] , TIKANGE [22], GROUMINER [19], and MUDDETECT [2]. However, some of these detectors require Java source code as input. Therefore, for comparison, we need projects that have both source code and bytecode. MuBench [1] is a dataset for evaluating the API misuse detector. We take 31 projects in total from MuBench that can be successfully compiled into bytecode as our experimental dataset.

**Table 2.** Type and number of violations

| Violation | Number |
|---|---|
| missing/call | 68 |
| missing/condition/null_check | 5 |
| missing/condition/synchronization | 1 |
| missing/condition/value_or_state | 35 |
| missing/violation_handling | 20 |
| redundant/call | 10 |
| redundant/violation_handling | 1 |
| Total | 140 |

Table 2 shows the statistics of the dataset, which contains a total of 140 known violations. The classification of violation types follows the one of Mubench. The API missing/call violation has the highest number, accounting for 48.5% of the total.

### 4.2 Experiments

For evaluating violation detection, soundness and completeness are two important criteria. Soundness indicates the accuracy of the detector, i.e., the number of false positives should be as low as possible; completeness requires that the detector can find as many violations as possible. Therefore, in this section, we design the precision experiment and the recall experiment. To evaluate the effectiveness of inter-procedural analysis, we also design the ablation experiment.

**Precision Experiment.** In this experiment, we need to check whether a reported violation is a true misuse or not. Moreover, some detectors may report thousands of violations. However, it is impractical to manually verify the correctness of all the reported violations on too many projects. Therefore, following the experiments of Amann et al.'s work [2], we selected 10 projects from the dataset for this experiment, on which all detectors were run on to detect violations, and took the TOP-20 reports as output for each project for manual verification.

**Recall Experiment.** This experiment aims to find the known violations. It is easy to manually check whether a known violation is reported or not. So we performed this experiment on all the 31 projects and checked whether the 140 known violations are contained in the output list reported by each detector. We also considered the 25 newly found violations in the above precision experiment, yielding a list of known violations with a total 165. In addition, we also performed the recall

[4] https://wala.sourceforge.net/wiki/index.php/UserGuide:Technical_Overview

experiment with TOP-20 reports on the 10 projects selected for the precision experiment, so as to get the F1-scores.

**Ablation Experiment.** In order to evaluate the effectiveness of inter-procedural AUGP, we performed the precision experiment with the intra-procedural AUGP (i.e., without the inter-procedural analysis).

### 4.3 Results of Precision Experiment

**Table 3.** Precision and Recall Results on 10 selected projects

| Detector | Correct | Report | Precision | Recall | F1-score |
|---|---|---|---|---|---|
| **JADET** | 8 | 111 | 7.21% | 16.48% | 0.10 |
| **TIKANGA** | 12 | 105 | 11.43% | 24.18% | 0.16 |
| **DMMC** | 12 | 141 | 8.51% | 34.07% | 0.14 |
| **GROUMINER** | 4 | 150 | 2.67% | 7.69% | 0.04 |
| **MUDETECT** | 38 | 151 | 25.00% | 62.64% | 0.36 |
| **APDetect** | 42 | 75 | 56.00% | 51.65% | 0.54 |

The results of precision experiment are shown in Table 3, where **Correct** indicates the number of violations that are reported correctly for each detector.

Table 3 reveals that APDetect achieves the highest precision rate 56.00%, which is 31.00% more than the second-best detector. Furthermore, APDetect reports the highest number of true violations, correctly identifying a total of 42, which is 4 more than the second-best detector. Notably, the detectors also detected 25 new violations that are not previously recorded in the dataset MuBench, out of which 11 were identified by APDetect. These findings demonstrate that our approach is highly effective and outperforms several existing detectors in terms of precision.

**Table 4.** Reasons for false positives

| Reason | Number | Ratio |
|---|---|---|
| Uncommon Usage | 20 | 60.61% |
| Alternative Call | 10 | 30.30% |
| Inter-procedural Disruption | 2 | 6.06% |
| Insufficient Control Flow Analysis | 1 | 3.03% |

However, there are 33 false positives in this experiment reported by APDetect. After a manual analysis on these reports as well as the rules, we summarize the reasons in Table 4 and discuss them in the following:

– The uncommon usage in this section can be interpreted as that the mined rule is not very strict such that both APIs much appear at the same time. In other words, one of them can be appear separately. Take the iterator *Iterator* for example. *hashNext()* can be called separately to determine whether there are still some elements in the iterator without necessarily calling

*next()* to get them. This reason causes the most false positives for APDetect, with a total of 20 cases, accounting for 60.61%.
- Alternative calls means that the functionality of the API can be replaced by other APIs. For example, in the class *java.util.StringTokenizer*, *nextElement()* and *nextToken()* have the same functionality and can be used interchangeably. This would make one usage being considered a rule while the others being considered as a violation. There are 10 cases due to this reason, accounting for 30.30%.
- A client function that is detected by APDetect with a violation may make its client callers to be detected with a violation, leading to a redundant false positive. We call this reason as inter-procedural disruption, which causes 2 cases.
- Insufficient control flow analysis, such as the infeasible paths, can also lead APDetect to some false positives, which is left for future work.

Similar to the analysis result reported in Amann et al. [2, 3], our study found that the main reasons behind the low precision rates in the precision experiment for the other detectors were uncommon usage, alternative call and inadequate program analysis.

**Table 5.** Recall Results on 31 projects

| Detector | Hits | Report | Recall |
|---|---|---|---|
| **JADET** | 15 | 355 | 9.09% |
| **TIKANGA** | 22 | 216 | 13.33% |
| **DMMC** | 33 | 8,097 | 20.00% |
| **GROUMINER** | 7 | 710 | 4.24% |
| **MUDETECT** | 60 | 20,378 | 36.36% |
| **APDetect** | 48 | 102 | 29.09% |

### 4.4 Results of Recall Experiment

Table 5 gives the statistical results of the recall experiment, where **hits** indicates the number of known violations that are reported by each detector. The detail results are given in Table 6.

Among the tested detectors, MuDetect identified a total of 60 violations and achieved the highest recall rate 36.36%. APDetect, on the other hand, detected a total of 48 violations and ranked second, achieving a recall rate of 29.70%, only 7.27% smaller than the top detector MuDetect. DMMC demonstrated a recall rate of 20.00% and ranked third. Notably, APDetect reported the fewest number of violations at 102, while MuDetect and DMMC reported 20,378 and 8,097 violations, respectively, which are considerably higher than that reported by our detector APDetect.

We also have a quick check on the rankings of the known violations reported by the detectors and found that some known violations reported by MuDetect and DMMC are ranked far behind. For example, the violation named by *tikanga11-4* is ranked 324-th, 58-th and 12-th by MuDetect, DMMC and APDetect, respectively. Clearly, the too-low ranking will make the programmer miss the violation.

In addition, our co-occurrence detection reports 71 violations, among which 34 ones are correct, with a precision rate 47.88%; and our order detection reports 29 violations, among which 12 ones are

**Table 6.** Detail Results of Recall Experiment

| Project | Vresion | Number of Findings | | | | | |
|---|---|---|---|---|---|---|---|
| | | DMMC | GROUMINER | JADET | TIKANGA | MUDETECT | APDetect |
| aclang | 587 | 157 | 13 | 0 | 0 | 103 | 0 |
| acmath | 998 | 685 | - | 17 | 17 | 2666 | 0 |
| alibaba-druid | e10f28 | 517 | - | 17 | 5 | 798 | 7 |
| apache-gora | bb09d89 | 0 | 15 | 0 | 1 | 193 | 1 |
| argouml | 026 | 1669 | - | 73 | 48 | 9128 | 33 |
| asterisk-java | 304421c | 114 | 10 | 0 | 1 | 6 | 2 |
| battleforge | 878 | 171 | 3 | 0 | 0 | 76 | 2 |
| bcel | 24014e5 | 322 | 87 | 21 | 3 | 315 | 2 |
| chensun | cf23b99 | 49 | 50 | 8 | 2 | 66 | 21 |
| closure | 319 | 1944 | 95 | 176 | 45 | 2567 | 7 |
| corona-old | 0d0d18b | 146 | 8 | 0 | 0 | 45 | 1 |
| hoverruan-weiboclient4j | 6ca0c73 | 29 | 0 | 0 | 0 | 6 | 0 |
| itext | 5091 | 1173 | 137 | 17 | 55 | 1304 | 13 |
| ivantrendafilov-confucius | 2c30287 | 4 | 0 | 0 | 0 | 0 | 0 |
| jigsaw | 205 | 0 | 130 | 12 | 20 | 1525 | 0 |
| jodatime | 1231 | 1 | 0 | 0 | 0 | 9 | 0 |
| jriecken-gae-java-mini-profiler | 80f3a59 | 4 | 0 | 0 | 0 | 0 | 0 |
| lucene | 1918 | - | 72 | 2 | 4 | 334 | 5 |
| minecraft-launcher | e62d1bb | 95 | 0 | 0 | 0 | 2 | 1 |
| mqtt | f438425 | 43 | 4 | 0 | 0 | 1 | 0 |
| rhino | 286251 | 258 | 43 | - | - | 156 | 0 |
| saavn | e576758 | - | 0 | - | - | 0 | 0 |
| secure-tcp | aeba19a | 1 | 0 | 0 | 0 | 0 | 0 |
| synthetic_java8-misuses | 96d0ccb | - | 0 | - | - | 0 | 0 |
| synthetic_jca | jsl | 0 | 0 | 0 | 0 | 0 | 0 |
| synthetic_survey | jsl | 5 | 0 | 0 | 0 | 0 | 0 |
| tbuktu-ntru | 8126929 | 60 | 1 | 0 | 0 | 0 | 0 |
| technic-launcher-sp | 7809682 | 138 | 4 | 0 | 0 | 0 | 1 |
| testng | 677302c | 473 | 37 | 12 | 15 | 1075 | 6 |
| thomas-s-b-visualee | 410a80f | 30 | 1 | 0 | 0 | 3 | 0 |
| yapps | 1ae52b0 | 9 | 0 | 0 | 0 | 0 | 0 |
| **Total** | | 8097 | 710 | 355 | 216 | **20378** | **102** |

correct, with a a precision rate 41.38%. The above results show that both co-occurrence detection and order detection are effective.

As the projects and the reported numbers are different from the precision experiment, we cannot get the F1-scores from these two experiments. For that, we also performed the recall experiment with TOP-20 reports on the 10 projects selected for the precision experiment, so as to get the F1-scores. The results are also shown in Table 3, which are similar to the ones for the 31 projects shown in Table 5. Table 3 demonstrates that APDetect achieves the best F1-score.

Finally, we study the violations involving API calls that are missed by APDetect for the 31 projects, and summarize the following possible reasons:

– Little data. Little data can be reflected in too few code files in a project, too few occurrences of the target classes, or too few occurrences of the target API. Any of these may cause the pattern mining unable to mine a valid rule, and then make the violation detection fail to the violations.

- Client-side functions. Client-side functions are written by the project developers and are generally more flexible to use in the projects. Our detector did not consider these client-side functions. However, there are some violations involving client-side functions in MuBench, which could be missed by our detector.
- Graph filtering. Graphs with only one target API are filtered by our approach. But in MuBench, there are some violations involving only one API. For example, a StringBuilder object is constructed by two redundant calls, wherein only one (i.e., the last one) call is useful.

Similar to the analysis result reported in Amann et al. [2, 3], our study found that the low recall rates for the other detectors were mainly due to two factors: insufficient API usage cases for mining (*i.e.*, litter data) and inadequate detector functionality.

### 4.5 Results of Ablation Experiment

**Table 7.** Results for Intra-Detection and Inter-Detection

| | **Intra-Detection** | | **Inter-Detection** | |
|---|---|---|---|---|
| **Project** | **Correct** | **Report** | **Correct** | **Report** |
| **closure** | 5 | 7 | 5 | 7 |
| **itext** | 2 | 3 | 6 | 13 |
| **jodatime** | 0 | 0 | 0 | 0 |
| **lucene** | 3 | 4 | 4 | 5 |
| **asterisk-java** | 0 | 2 | 0 | 2 |
| **bcel** | 0 | 0 | 0 | 2 |
| **chensun** | 0 | 3 | 12 | 20 |
| **jigsaw** | 0 | 0 | 0 | 0 |
| **testng** | 1 | 3 | 1 | 6 |
| **argouml** | 14 | 20 | 14 | 20 |
| **Total** | 25 | 42 | 42 | 75 |

Table 7 shows the results of the intar-procedural detection and the inter-procedural detection in the precision experiments. There are 17 more violations found by the inter-procedural detection then the one found by the intra-procedural detection, and their precisions are pretty close. This demonstrates the effectiveness and the necessary of the inter-procedural analysis.

There is no difference between the results on some projects, such as *closure* and *argouml*, for the intra-procedural detection and the inter-procedural detection. This is because these target APIs, such as *hasNext()* and *next()*, are called in the same client functions.

While on the project *chensun*, the results are significant different for these two detections. The reason is that *chensun* provides several client functions to encapsulate java database APIs, such as *java.sql.PreparedStatement.close()*. Without the inter-procedural analysis, it is unable to capture the usages of these APIs. In other words, building AUGP with the inter-procedural analysis can enrich the API usages, leading to a higher probability of mining (more) patterns.

# 5 Related Work

## 5.1 API Pattern Mining

An API Pattern defines a type of legal API usages [26] and API pattern mining is a technique for mining API usages from existing codes. Researchers have proposed various API pattern mining techniques. In this section, we briefly introduce existing mining techniques according to the following categories:

**Frequent Itemset Mining.** Li and Zhou [12] proposed PR-Miner to automatically extract the implied patterns from the code by hashing the functions in the source codes into integer itemsets, and then using the frequent subitem algorithm FPclose in data mining techniques to automatically extract the patterns of function calls from them. Bin Liang et al. [13] proposed a mining method based on frequent itemsets dependent on the quality of the dataset. As the presence of irrelevant interfering items in the code during the mining of API patterns can lead to incorrect API patterns, they proposed a novel mining tool, AntMiner.

**Sequential Pattern Mining.** Xie and Pei [27] proposed MAPO, a tool for extracting API usage patterns from source code. MAPO first extracts API call sequences from code fragments and then performs hierarchical clustering by similarity of API names, class names, etc., and the most frequent API call sequence obtained in each cluster is considered as an API usage pattern. UP-Miner [21] improved MAPO in extracting API invocation sequences from source code. UP-Miner uses a similar strategy as MAPO, which improves MAPO by reducing redundancy and interference in API invocation sequences by clustering twice, and it uses the BIDE nearest frequent sequence mining algorithm to obtain only API sequences without more frequent substrings, and finally presents the API statute using probabilistic graphs.

**Frequent Subgraph Mining.** Nguyen et al [19] proposed a method to construct a graph model GROUM from source code. The GROUM graph mainly retains function calls, WHILE loops, IF branches, etc. as nodes, and establishes basic directed edges first by the order of nodes occurring in the code, and then by the proximity and usage relationships of nodes, and finally obtains a graph model for code fragments. When the variable nodes are included in the graph, the graph model can represent the usage of some parameters, but some parameters for specific fetching values are still ignored. In the usage pattern mining, they determine whether a subgraph is a legitimate API statute based on the frequency of its occurrence in the graph. In 2018, Mover et al [16] proposed BIGGROUM based on GROUM. They explicitly define the types of nodes and edges in the graph, where the data node represents the type of the parameter being used, again ignoring the specific fetching of the parameter. In mining pattern, they classify graphs by clustering frequent itemsets in order to reduce the computational space of isomorphic subgraphs, using a similar approach to GROUM, which identifies graph models that occur more frequently than a threshold as pattern.

Besides the above techniques, researchers have proposed other techniques to mine API patterns, such as probabilistic model mining [5, 18], grammatical inference [4], template mining [6], etc.

## 5.2 Pattern-Based Detectors

Pattern-based detectors commonly mine usage patterns (*i.e.*, equivalent API usages that occur frequently) and then reports deviations from these patterns as potential misuses [2]. Amann et al. [3] have presented a detailed survey and comparison of detectors and their capabilities, so we briefly discuss the related detectors here.

JADET [23] keeps the function names, call orders, and instance objects in the code, extracts the directed graph from codes, and then extracts the relationship pairs of APIs from the graph. The relationship pairs that satisfy the minimum support are considered as the API usage patterns. A usage is considered as a violation if it misses at least 2 attributes of the pattern being violated as well as it occurs at least 10 times less than the pattern. Violations are ranked by $u \times s \div v$, where $s$ is the violated pattern's support, $v$ is the number of violations of the pattern, and $u$ is a uniqueness factor of the pattern.

TIKANGA [22] builds on JADET and it replaces API call-order pair with Computation Tree Logic (CTL). TIKANGA looks through all the objects used as actual arguments and identifies those that violate the operational preconditions. Violations are ranked by elevation in association rules.

DMMC [15] is a missing method call detector for Java. It extracts usages from codes then compares their similarity. Two usages are exactly similar if their respective sets match and are almost similar if one of them contains exactly one additional method. If a usage has high strangeness score, then it is considered a violation and ranked by the score.

GROUMINER [19] creates a graph-based object-usage representation (GROUM) for each target function. It performs frequent subgraph mining on GROUMs to mine usage patterns and then uses them to detect violations. When at least 90% of, but not, all occurrences of a subgraph can be extended to a larger graph, these rare non-extendable graphs are considered as violations and are ranked by their rareness.

MUDETECT [2] represents each target (client) function as an API Usage Graph (AUG), and then uses a frequent subgraph mining algorithm to mine usage patterns with a minimum support of 6. A subgraph is considered abnormal if it does not match any usage pattern and the confidence score is less than the threshold. Violations are ranked by their confidence scores, which are computed by $p \div v \times o$, where $p$ is the support of the violated pattern, $v$ is the number of times that the violation occurs, $o$ is violation-overlap.

The above detectors take a single Java project as input to mine patterns and detect violations. Besides, PR-Miner, AntMiner, CHRONICLER [20], COLIBRI/ML [14] are designed for C programming language. DROIDASSIST [17] is designed for Android Java Bytecode. MuDetectXP, CL-Detector [24], ALP [9] add additional inputs to enhance the detection.

## 6 Conclusion

In this paper, we have proposed a graph model AUGP for intermediate representation of API usages and an approach to extract AUGPs from code. Based on AUGP, we have proposed a detector APDetect that mines the API usage rules first and then detects the possible violations guided by the rules. We have performed experiments to evaluate our detector, and the results shows that, comparing against five state-of-the-art detectors on the public dataset MuBench, our detector achieves the highest precision and the highest F1-score. In future work, we will add exception handling and conditional branching to AUGP and explore more experiments to evaluate our detector.

## Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (Nos. 61836005 and 61972260).

## References


1. Amann, S., Nadi, S., Nguyen, H.A., Nguyen, T.N., Mezini, M.: Mubench: a benchmark for api-misuse detectors. In: MSR. pp. 464–467. ACM (2016)
2. Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: Investigating next steps in static api-misuse detection. In: Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada. pp. 265–275. IEEE / ACM (2019)
3. Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M.: A systematic evaluation of static api-misuse detectors. IEEE Trans. Software Eng. **45**(12), 1170–1188 (2019)
4. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL. pp. 4–16. ACM (2002)
5. Fowkes, J.M., Sutton, C.: Parameter-free probabilistic API mining across github. In: SIGSOFT FSE. pp. 254–265. ACM (2016)
6. Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: ICSE (1). pp. 15–24. ACM (2010)
7. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2000)
8. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: SIGMOD Conference. pp. 1–12. ACM (2000)
9. Kang, H.J., Lo, D.: Active learning of discriminative subgraph patterns for api misuse detection. IEEE Transactions on Software Engineering **48**(8), 2761–2783 (2022)
10. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: Crysl: An extensible approach to validating the correct usage of cryptographic apis. IEEE Trans. Software Eng. **47**(11), 2382–2400 (2021)
11. Legunsen, O., Hassan, W.U., Xu, X., Rosu, G., Marinov, D.: How good are the specs? a study of the bug-finding effectiveness of existing java API specifications. In: ASE. pp. 602–613. ACM (2016)
12. Li, Z., Zhou, Y.: Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In: ESEC/SIGSOFT FSE. pp. 306–315. ACM (2005)
13. Liang, B., Bian, P., Zhang, Y., Shi, W., You, W., Cai, Y.: Antminer: mining more bugs by reducing noise interference. In: ICSE. pp. 333–344. ACM (2016)
14. Lindig, C.: Mining patterns and violations using concept analysis. In: The Art and Science of Analyzing Software Data, pp. 17–38. Morgan Kaufmann / Elsevier (2015)
15. Monperrus, M., Bruch, M., Mezini, M.: Detecting missing method calls in object-oriented software. In: ECOOP. Lecture Notes in Computer Science, vol. 6183, pp. 2–25. Springer (2010)
16. Mover, S., Sankaranarayanan, S., Olsen, R.B.P., Chang, B.E.: Mining framework usage graphs from app corpora. In: SANER. pp. 277–289. IEEE Computer Society (2018)
17. Nguyen, T.T., Pham, H.V., Vu, P.M., Nguyen, T.T.: Recommending API usages for mobile apps with hidden markov model. In: ASE. pp. 795–800. IEEE Computer Society (2015)
18. Nguyen, T.T., Pham, H.V., Vu, P.M., Nguyen, T.T.: Learning API usages from bytecode: a statistical approach. In: ICSE. pp. 416–427. ACM (2016)
19. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/SIGSOFT FSE. pp. 383–392. ACM (2009)
20. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: 29th International Conference on Software Engineering (ICSE'07). pp. 240–250 (2007). https://doi.org/10.1109/ICSE.2007.63
21. Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage API usage patterns from source code. In: MSR. pp. 319–328. IEEE Computer Society (2013)
22. Wasylkowski, A., Zeller, A.: Mining temporal specifications from object usage. Autom. Softw. Eng. **18**(3-4), 263–292 (2011)
23. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ESEC/SIGSOFT FSE. pp. 35–44. ACM (2007)
24. Zeng, H., Chen, J., Shen, B., Zhong, H.: Mining api constraints from library and client to detect api misuses. In: 2021 28th Asia-Pacific Software Engineering Conference (APSEC). pp. 161–170 (2021)

25. Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M.: Are code examples on an online q&a forum reliable?: a study of API misuse on stack overflow. In: ICSE. pp. 886–896. ACM (2018)
26. Zhong, H., Mei, H.: An empirical study on API usages. IEEE Trans. Software Eng. **45**(4), 319–334 (2019)
27. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending API usage patterns. In: ECOOP. Lecture Notes in Computer Science, vol. 5653, pp. 318–343. Springer (2009)