API Misuse Detection via Probabilistic Graphical Model

Yunlong Ma*
Beihang University
Beijing, China
yunlong_ma@buaa.edu.cn

Wentong Tian*
Beihang University
Beijing, China
tianwentong2000@buaa.edu.cn

Xiang Gao†
Beihang University
Beijing, China
xiang_gao@buaa.edu.cn

Hailong Sun
Beihang University
Beijing, China
sunhl@buaa.edu.cn

Li Li
Beihang University
Beijing, China
lilicoding@ieee.org

ABSTRACT

API misuses can cause a range of issues in software development, including program crashes, bugs, and vulnerabilities. Different approaches have been developed to automatically detect API misuses by checking the program against usage rules extracted from extensive codebase or API documents. However, these mined rules may not be precise or complete, leading to high false positive/negative rates. In this paper, we propose a novel solution to this problem by representing the mined API usage rules as a probabilistic graphical model, where each rule's probability value represents its trustworthiness of being correct. Our approach automatically constructs probabilistic usage rules by mining codebase and documents, and aggregating knowledge from different sources. Here, the usage rules obtained from the codebase initialize the probabilistic model, while the knowledge from the documents serves as a supplement for adjusting and complementing the probabilities accordingly. We evaluate our approach on the MuBench benchmark. Experimental results show that our approach achieves 42.0% precision and 54.5% recall, significantly outperforming state-of-the-art approaches.

CCS CONCEPTS

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

KEYWORDS

API misuse detection, Mining Software Repository, Document Mining, Probabilistic Graphical Model

ACM Reference Format:
Yunlong Ma, Wentong Tian, Xiang Gao, Hailong Sun, and Li Li. 2024. API Misuse Detection via Probabilistic Graphical Model. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652112

*Equal contribution, listed in alphabetical order
†Corresponding author

ISSTA 24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09…$15.00
https://doi.org/10.1145/3650212.3652112

1 INTRODUCTION

The use of third-party libraries is extremely common in application software development. Typically, these libraries offer various functions through public Application Programming Interfaces (APIs) that developers can use. However, using these APIs requires following certain contracts, such as passing appropriate arguments, invoking APIs in the correct order, and adhering to pre/post-conditions of API calls. It is essential for a client application that depends on a particular library to follow these API contracts. Unfortunately, due to the complexity of API designs, inadequate documentation, and insufficient training, developers may misuse APIs, leading to program crashes, vulnerabilities, unexpected behaviors, and other issues. API misuse is a primary cause of program bugs, with over 50% of more than one million bug-fix commits attributed to API misuse [20]. Particularly, as software development increasingly relies on third-party libraries, forming software supply chains [7], detecting API misuse is vital to improving software quality.

Automatically detecting API misuses requires formal definitions of correct API usage rules. However, the usage directives that describe the contracts, constraints, and guidelines for using APIs are often provided in natural languages as a part of the API documents. The ambiguous nature of these natural language descriptions poses significant challenges to the automated detection of API misuse.

To tackle the aforementioned challenges, researchers have developed methods to automatically mine API usage rules from various sources, such as API documents [24, 40], large codebase [42, 48, 53, 55], online forum, e.g., Stack Overflow [39, 45], or a combination of these sources [23, 47]. Although these methods have shown promising results, they still have certain limitations. Firstly, while API documents can accurately describe API usage rules, they are often incomplete, loosely formatted, and diverse in writing styles, making it challenging to infer complete and accurate rules [2, 41]. Secondly, mining rules by analyzing frequent API usages from codebase is limited by the availability of high-quality API usages. Moreover, the concrete API usages in codebase and online forums may not always be reliable, i.e., frequent API usages are not guaranteed to be correct. In fact, Zhang et al. [54] have shown that 31% of posts on Stack Overflow have potential API usage violations. Thirdly, while combining rules from codebase and API documents increases the accuracy of mined API usage rules [23, 47], their performance

is still far from being satisfactory, with state-of-the-art techniques achieving only 36.3% precision and 47% recall [47].

The impreciseness and incompleteness of API usage rules mined from different sources is the main reason causing low detection accuracy. For instance, before popping up an element from Java Stack, according to JavaDoc, users need to check `!stack.empty()`, while the mined rule from codebase may check `stack.size()>0` or even no check (which may not be wrong, as a certain stack may never be empty). The usage rules mined from neither codebase nor documents are considered precise and complete, and they may even conflict with each other. How to extract valuable knowledge from imprecise and incomplete sources has not been fully investigated.

To address this issue, our key idea is to utilize a probabilistic model to quantify the trustworthiness of a mined rule. The higher the probability assigned to a rule, the more likely it is to be correct. With this approach, we can aggregate the API usage rules extracted from different sources by computing multi-source probabilities. For instance, in the example mentioned earlier, we can determine that verifying `!stack.empty()` before popping elements is not mandatory. Instead, there is a certain probability that the `pop()` operation is guarded by `stack.size()>0`, or there is even no check. By assessing API misuse in this manner, we can report detected API misuses according to trustworthiness of corresponding rule, thereby reducing the number of false positives/negatives.

In this paper, we present a formalization of probabilistic API usage rules utilizing a probabilistic graphical model, specifically Bayesian Network [13]. The Bayesian Network is used to represent the conditional dependencies among a set of nodes through a directed acyclic graph (DAG). In our context, the nodes in the DAG represent API invocations, predicates or data entities, while the edges denote the conditional dependencies between nodes. For instance, the conditional dependency between two method invocation nodes $m_a \xrightarrow{p} m_b$ means that "after invoking API m_a, API m_b needs to be invoked with a probability p". We formalize this approach and build a probabilistic graphical API usage model based on the codebase and documents. Specifically, we determine the prior probability between different nodes based on the frequency of concrete API usage in the large codebase. Additionally, we aggregate the API usage rules extracted from the documents into the probabilistic model. We then use the constructed probabilistic model for API misuse detection. For a given API invocation, we calculate the posterior probabilities of whether another API should be first invoked, whether a certain argument should be verified, or whether the return value should be validated.

To realize this idea, we implement a tool called GraphiMuse (**Graphi**cal **M**odel based API **Mis**use Detection) for learning the probabilistic API usage rules and detecting API misuses. Similar to existing work [42], GraphiMuse encodes concrete API usages as API-Usage Graphs (AUGs), a comprehensive usage representation that captures different types of API usage rules. Then, it employs a frequent subgraph-mining algorithm to mine rules and initialize probabilities, a large language model (LLM) enhanced algorithm for interpreting API documents, and a graph-matching strategy to identify rule violations. GraphiMuse reports misuses with a confidence score, indicating the confidence level of GraphiMuse on the detection results. GraphiMuse is then evaluated on MuBench [3], a widely used benchmark for detecting Java API misuses. Evaluation results show that GraphiMuse achieves 42.0% precision and 54.5% recall, significantly outperforming state-of-the-art approaches.

The contributions of this paper are summarized as follows:

- We present a novel idea to represent API usage rules using a probabilistic graphical model, where the probability indicates the trustworthiness of mined API usage rules.
- We propose an approach to constructing the probabilistic API usage rules by mining the codebase and documents, and a strategy to aggregate the knowledge from different sources.
- We implement a tool called GraphiMuse, and evaluation on MuBench shows GraphiMuse outperforms existing techniques.
- We make our implementation open-source available at https://github.com/18373637myl/GraphiMuse.

2 BACKGROUND

Before presenting the technical details, let us first provide some background on Bayesian network and API misuse detection.

2.1 Bayesian Network

A Bayesian network [13] is a probabilistic graphical model that depicts the conditional dependencies among a group of variables via a directed acyclic graph. In a Bayesian network, the nodes represent variables in the Bayesian sense: they may be observable quantities, latent variables, or hypotheses. Edges represent conditional dependencies, while nodes that are connected indicate conditional dependence between variables, otherwise, two nodes are independent. Formally, given a set of nodes $x_1, x_2, \ldots, x_n$, the joint probability satisfies

$$P[x_1, x_2, \ldots x_n] = \prod_{i=1}^{n} P[x_i|\text{pa}(x_i)]$$

where $\text{pa}(x_i)$ is the set of parents of node x_i. In other words, the joint distribution factors into a product of conditional distributions. With the joint distribution, the model can answer questions about the presence of a cause (e.g., x_j) given the presence of an effect (e.g., x_i). This refers to the conditional probability of node x_j given that node x_i takes on a specific value. To compute this conditional probability, we can use Bayes' rule, which states that

$$P(x_j|x_i) = \frac{P(x_i, x_j)}{P(x_i)}$$

Here, $P(x_i, x_j)$ is the joint probability of x_i and x_j, and $P(x_i)$ is the marginal probability of x_i.

2.2 API Misuse Detection

Existing methods detect API misuse by statically checking the code against API usage rules mined from codebase or documents.

Mining API usage rules from codebase. To mine API usage rules from the codebase, the prevalent approach in the field is the frequent itemset mining technology [30, 38]. This is based on a simple assumption that the frequently used patterns are correct. This approach typically begins by obtaining high-quality source code datasets from the codebase. The source code is then preprocessed using various techniques, including control flow analysis, data flow analysis, and abstract syntax tree generation, which aims

to transform the source code into a more standardized representation. For instance, MuDetect [42] transforms programs into AUG, a graph specifically designed for API usage. Subsequently, frequent itemset mining algorithms such as the Apriori algorithm [37] are applied to mine API usage rules from standardized representations. Although existing approaches have achieved promising results [42], false positives and negatives are still prevalent.

Mining API usage rules from API documents. Mining API documents is typically achieved through *document crawling*, *directive sentence identification*, and *rule extraction*. The API documents are usually crawled from official library websites. Within the crawled documents, only a portion of the sentences contains directive information, which includes constraints and guidelines about API usage. Directive sentences can be identified either through regular expression matching [28] (example keywords include 'must', 'require', 'should', etc.) or learning-based classification [24, 40]. Once the directive sentences are identified, API usage rules can be extracted through the following steps:

(1) recognizing name entities (such as API calls and predicates) embedded within sentences (e.g., "if there are no more tokens in this tokenizer's string" of `nextToken` is a predicate);
(2) matching recognized entities with real program elements (e.g. parameter, method, value literal). For instance, the above predicate can be matched to `hasMoreTokens` because its functionality description "Tests if there are more tokens available from this tokenizer's string" is highly similar to the predicate entity.
(3) identifying the types of rule by utilizing keyword analysis, e.g., by analyzing the directive sentence "if neither next nor previous have been called" of `ListIterator.set()`, the rule is inferred as a call order relation according to keywords 'have been called'.

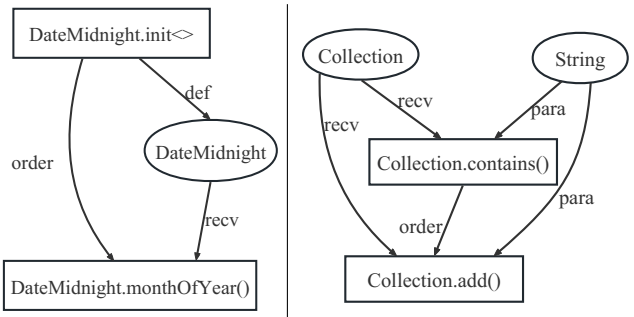
The above workflow may result in false positives, since it fails to take into account the abundant semantic information encapsulated in the API description statements.

3 MOTIVATION

Mining API usage rules from codebase or documents seems straightforward, but it is based on the simplistic assumption that the rules derived from frequent API usage or documents are necessarily correct. However, just because an API is frequently used in a particular way, it does not guarantee this API usage is correct [42]. Usage rules in large codebase are often influenced by individual developers' coding preferences, which may not align with the best practices. In contrast, constraints extracted from documents tend to be oversimplified. The knowledge gap between usage caveats and correct usage rules makes it challenging to guarantee the accuracy of mined rules [25, 40]. In this section, we present several mined API usage rules and demonstrate the limitations of existing techniques.

3.1 Imprecise Usage Rules

Figure 1a shows an API usage rule mined by MuDetect [42] from the Jodatime project, which is a library that provides support for dates, times, time zones, durations, intervals, and partials. The rule is about the usage of the `DateMidnight` class. This rule requires that (1) *subrule-1:* a `DateMidnight` object is defined via `DateMidnight.init`and (3) *subrule-2:* after creating a `DateMidnight` object, `monthOfYear()` method must be invoked.



(a) An imprecise rule **(b) An incomplete rule**

Figure 1: API usage rule examples mined from (a) Apache Jodatime [14] and (b) Testng [6]

However, the API usage rule mentioned above is not entirely correct. The *subrule-1* regarding the process of creating an object is accurate. But, *subrule-2* is wrong since the `monthOfYear()` method is not necessarily called after the creation of the object. This error happens because calling `monthOfYear()` after `DateMidnight.init` has reached a pre-defined frequency threshold. As a result, any `DateMidnight` object that not calls `monthOfYear()` will be marked as API misuse, which is apparently incorrect.

Representing usage rules as probabilistic models can effectively address these issues. Although the frequency of `monthOfYear()` surpasses the frequency threshold, it is still significantly lower compared to the frequency of `DateMidnight.init`. It is necessary to create a `DateMidnight` object before invoking `monthOfYear()`, however, after the creation of the `DateMidnight` object, other methods instead of `monthOfYear()` may be called. Specifically, in the Jodatime project, invoking `monthOfYear()` method occurs 27 times, whereas `DateMidnight.init` method occurs 274 times. Based on this observation, we establish a lower dependency probability between `DateMidnight.init` and `monthOfYear()`. This reduces the number of false positives caused by the absence of `monthOfYear()` method and increases detection accuracy.

Additionally, mining API usage rules from documents can also be error-prone. Natural language can be highly ambiguous, particularly in API reference documents which include complex logical relations and conditions. For instance, the `orElseThrow()` function from the `java.util.Optional` class contains a directive sentence "throws X - if there is no value present", where X is the exception defined by the developer. Developers may use the customized exception X to implement certain functionalities by catching and handling X in an intended way. Therefore, it is unclear whether exception X should be triggered on purpose or prevented from being triggered. Based on this directive sentence, existing tool [40] simply extract rule '`[isPresent()`, if true, `orElseThrow()]`' to prevent triggering X. By referring to the large codebase, however, this usage rule is hardly used by developers, making it highly improbable. By utilizing a probabilistic model, we can ignore this constraint, thereby reducing the possibility of generating false positives.

3.2 Incomplete Knowledge Coverage

Apart from being incorrect, the mined rule may be overly simplistic and miss the necessary dependencies among API elements. For

example, the document of function `RandomAccessFileOrArray()` from `itextpdf.text`, includes only one directive sentence "This method throws IOException", which means IOException should be caught. However, one should also call method `close()` to close the `RandomAccessFileOrArray` object to prevent resource leakage. However, closing this object as a postcondition has not been mentioned in the document, resulting in the incomplete rule. Fortunately, the above constraint is a frequently used rule in the codebase. `RandomAccessFileOrArray()` and `close()` are used together for 18 times. With this information, we can infer a more complete rule.

Similarly, rules mined from a codebase may be incomplete, particularly in handling API parameters. Object-oriented programming languages like Java are empowered with inheritance and polymorphism, which may not be fully reflected in the concrete API usages within the codebase. Figure 1b illustrates a rule where the method `Collection.contains()` is expected to receive an argument of type `Object` and also check whether the object is `null`. However, as shown in this rule, it requires a `String` as a parameter, and the process of checking the input parameter for `null` is not reflected. To address this issue, we noticed that the Javadoc provides a clear description of `Collection.contains()`, which includes not only the expected parameter type but also covers the requirement to check whether the input `Object` is `null`. This effectively addresses the problem of incomplete information mined from the codebase, particularly concerning API parameter handling.

4 METHODOLOGY

In this section, we present the definition of probabilistic API usage rules, describe the detailed approach for constructing the probabilistic model, and demonstrate how to detect API misuse.

Figure 2 shows the overall workflow of the proposed technique. Specifically, GraphiMuse first mines rules from the codebase by encoding the program into API usage graphs (AUGs) [42] and extracting API usage rules based on frequency obtained from the AUGs. Then, the tool employs an open information retrieval technique to extract API constraints from documents. The knowledge mined from both the codebase and documents is then aggregated to construct a probabilistic usage model. Here, the usage rules obtained from the codebase initialize the probabilistic model, while the constraints from the documents serve as a supplement for adjusting the probabilities accordingly. Based on the learned probabilistic rules, GraphiMuse performs misuse detection and reports potential misuses to developers with a confidence score, indicating its confidence level on the detection results.

4.1 Probabilistic API Usage Rules

As we mentioned above, the probabilistic API usage rules could be represented as a Bayesian network. Formally, the probabilistic API usage rules are defined as follows:

Definition 4.1 (Probabilistic API usage rule). A rule is defined as a graph $R = (N, E, Pr)$, which is a set of nodes $N = \{n_1, n_2, \ldots, n_n\}$ together with a set of edges $E = \{e_1, e_2, \ldots, e_m\}$ that connect pairs of nodes. Each node n is associated with a function $pr_n : N' \rightarrow$ `probability`, where $pr_n \in Pr$ and N' is the set of parent nodes of n, representing the probability of n appears based to the occurrence of N'.

In our context, node n_i represents data entities, such as variables or literals, or actions such as method invocations, predicates, or catch blocks, while edges represent control and data flow between data entities and actions. Two conditional dependent nodes (n_s, n_t) means n_s should appear before n_t in a correct API usage with a certain probability. The function pr_n takes n's parent nodes as inputs and calculates the joint conditional probability of n. For example, Figure 3 presents the probabilistic API usage rule derived from Apache httpclient project. In this rule, the node set N consists of four nodes, with two representing actions: one is the API invocation node `Integer.parseInt()`, and the other is the `<catch>` node, representing the code block for exception handling. The remaining two nodes are data nodes, one being a common class `String`, and the other an exception node representing `NumberFormatException`. The edge set E consists of four edges, where the order edge denotes the order relationship between action nodes, the para edge represents the relationship between action nodes and parameter nodes, and the throw edge signifies the relationship between action nodes and exception nodes. The joint conditional probabilities indicate the probability of a specific node based on the presence of its parent nodes. The top-left part of Figure 3 shows the joint probability of node `Integer.parseInt()`, while the bottom-right part shows that of node `<catch>`. As an example, the probability of `<catch>` occurring is 0.46 if the node `Integer.parseInt()` exists and an `NumberFormatException` object is not created (third row of the right matrix). This indicates that in 46% of cases, after invoking `Integer.parseInt()`, despite encountering an exception, the thrown exception is not an `NumberFormatException`.

4.2 Initializing Probabilistic Rules via Codebase Mining

We first present the details of mining API usage rules from the codebase and initialize the probabilistic graphical model.

4.2.1 Mining probabilistic API usage rules from codebase. Just as existing work, e.g., MuDetect [42], GraphiMuse also first applies pre-processing to encode the program as a set of API usage graphs (AUGs). Then, it extracts frequent AUGs as the ingredients for initializing the probabilistic API usage rules. Algorithm 1 presents the algorithm for mining probabilistic rules from the codebase, which takes a set of AUGs as inputs and produces a set of frequent AUGs as rules. Given a set of AUGs, the algorithm first extracts the set of API call nodes whose frequencies are greater than threshold τ (line 2). The node n's frequency `freq`(n) is defined as the number of occurrences of n in the codebase, and the default value of τ is set as 10 (following the setting of MuDetect [42]). For each API call node, GraphiMuse first creates an initial rule (line 6), and recursively extends it by expanding their incoming and outgoing edges (lines 10-16). Similarly, GraphiMuse only considers the edges with frequency of $\geq \tau$. This process ends until there are no incoming or outgoing edges to expand (line 18). On top of the general workflow, we further designed three heuristic strategies to increase the rule accuracy.

• *Generalize nodes.* The nodes and edges in the API usage rules may be too specific, which limits their applicability to a wide range of API usage. To address this issue, we propose a node abstraction approach that is reflected in the expandable node selection at line 11 in Algorithm 1. During the recursive mining process, expandable

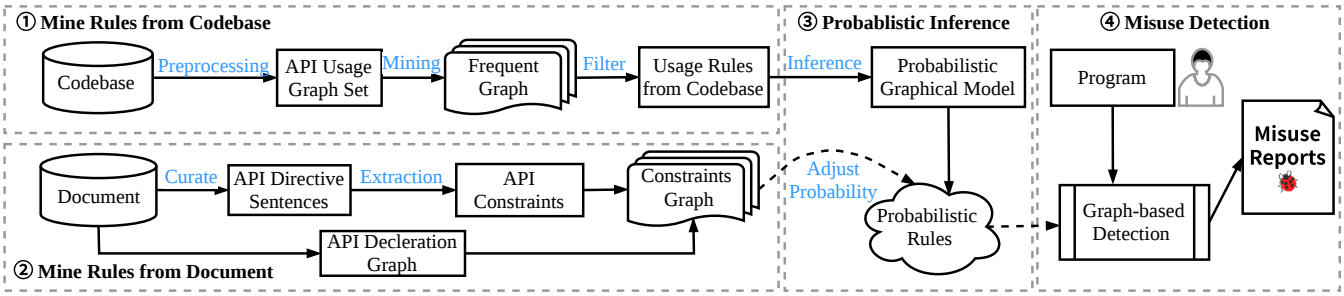


Figure 2: Overall framework of GraphiMuse

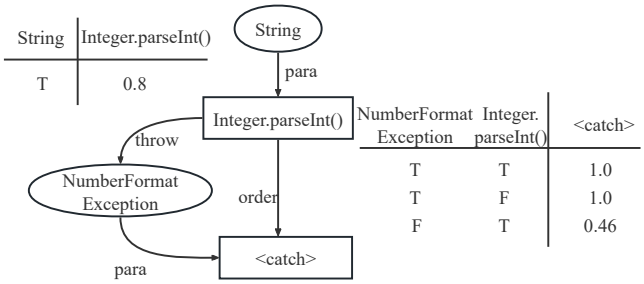


Figure 3: An example of probabilistic model mined from Apache httpclient [1]

Algorithm 1: Codebase Mining Algorithm

Input: A set of AUGs: *AUGs*; Frequency threshold: τ;
Output: A set of probabilistic Rules: *R*;

```
1  def mine(AUGs, τ):
2      N_apicall ← {n | n ∈ nodeSet(AUGs) ∧
3                      isAPICall(n) ∧ freq(n) ≥ τ};
4      Rules ← ∅;
5      foreach n ∈ N_apicall do
6          r ← new Rule(n);
7          Rules ← Rules ∪ extend(n, r, τ, AUGs);
8      return generateProbabiliticModel(Rules);
9  def extend(n, r, τ, AUGs):
10     R ← ∅;
11     N_e ← {n' | e ∈ edgeSet(AUGs) ∧ freq(e) ≥ τ ∧
12             (e = n → n' ∨ e = n' → n)};
13     if N_e ≠ ∅ then
14         for n_e ∈ N_e do
15             r_i = extension(r, n_e);
16             R = R ∪ extend(n_e, r_i, τ, AUGs) ;
17     else
18         R ← R ∪ {r};
19     return R;
```

nodes are calculated by referring to the edges connected to node *n*. In each step, there could be multiple but conflicting choices of expandable nodes. For example, when expanding the `group()` method invocation (from class `java.util.regex.Matcher`), there are two choices: `find()` → `group()` or `matches()` → `group()`, and both of them exceed the given threshold τ. Keeping both expanding options will result in many false positives, because rule `find()` → `group()` will report usage "`matches(); group()`" as a misuse since it does not invoke `find()` before `group()`, which is clearly a false positive. If only one of them is kept, similarly, the kept rule will also cause false positives. To solve this problem, GraphiMuse abstracts specific nodes into an abstracted form. In the above example, GraphiMuse treats [`find()` ∨ `matches()`] → `group()` as the correct rule, which can significantly improve the tool's precision.

• *Remove redundant information.* Many nodes that are irrelevant to API usage rules can also affect GraphiMuse's accuracy. We have removed certain nodes, including the *return* node and nodes related to exception handling.

• *Merging similar rules.* When two rules contain overlapping sections, it usually suggests that one of them is flawed or neither of them is correct. To tackle this problem, we implemented a mechanism that utilize logical operations to cluster rules based on their relationships. For instance, if $\texttt{rule}_1$ and $\texttt{rule}_2$ are considerably similar, they can be grouped as one rule by integrating them with logical operators like $\texttt{rule}_1$ or/and $\texttt{rule}_2$. The method for assessing the similarity of rules involves counting the common nodes between two rules and calculating the proportion of these common nodes relative to both rules, based on different weights assigned to action nodes and data nodes. The decision to merge two rules is made by determining whether the similarity reaches a predefined threshold. For instance, if the different nodes connected to the same API represent two types of parameters, we infer that the two rules might have a relationship of API rewriting, and thus we employ the `or` operator to merge them. Merging similar rules could enhance the flexibility and precision of rules.

4.2.2 Generating probabilistic model. After generating API usage rules, Algorithm 2 demonstrates how to convert them into probabilistic graphical models. For each rule in the input set, GraphiMuse iterates over all its nodes in the topologic order. For each non-root API call node *n*, all its parent nodes N' are first identified by referring to its incoming edges from the rule (line 7). Next, the influence of parent nodes on the child node is then calculated at lines 6 to 12. Specifically, the impact of N' on *n* can be represented as the probability that *n* exists under the existence of each node in N'. There are $2^{\wedge}|N'|$ possible combinations of parent nodes, with each combination representing whether certain parent nodes exist or not. To generate each combination, the algorithm uses a loop that iterates over all possible integers from 0 to $2^{\wedge}|N'|$ - 1. For each

Algorithm 2: Probabilistic Model Generation Algorithm

```
Input: Aggregated line feature
Output:
def generateProbabiliticModel(R):
    foreach r ∈ R do
        foreach n : TopoSort(nodeSet(r)) do
            if !isAPICall(n) ∨ isRootNode(n) then
                continue;
            DepDegree ← { };
            N′ ← {n′ | (n′ → n) ∈ edgeSet(r)};
            for i from 0 to (2^|N′| − 1) do
                combination = binaryRepresentation(i);
                N′_i ← mask(N′, combination);
                DepDegree[i] ← freq(n, N′_i) / freq(N′_i);
            n = n ~DepDegree;
    return R;
```

integer i in the loop, the algorithm generates a binary representation of i, which is a string of 0s and 1s of the same length as N'. For example, if N' has length 3 and i is 5, the binary representation of i will be "101". The binary representation is then used to create a certain combination of parent nodes by applying a mask to N', where each node exists if the corresponding binary digit is 1, and does not exist if it is 0. For each combination, GraphiMuse calculates the dependency degree between node n and its parent nodes N'_i by calculating $\frac{\texttt{freq}(N'_i,n)}{\texttt{freq}(N'_i)}$. To calculate $\texttt{freq}(N'_i, n)$, GraphiMuse tracks where n originated from and counts the frequencies of their parents and n. Conceptually, this formula measures the probability of $Pr(n, n'_1, n'_2 \dots)/Pr(n'_1, n'_2 \dots)$, where $n'_1, n'_2 \dots$ are set of parent nodes of n. Let us revisit the example shown in Figure 3. For this example, the *DepDegree[1]* of node <catch> is $Pr(<\texttt{catch}>, \overline{\texttt{InterruptedException}}, \texttt{Thread.sleep()})$ / $Pr(\overline{\texttt{InterruptedException}}, \texttt{Thread.sleep()})$, where $\overline{n}$ indicates node n does not exist. The produced value of this formula in this case is 0.26. Each node is associated with a *DepDegree* matrix, which represents its dependency degree on its parent nodes. The *DepDegree* will be used in the following misuse detection process.

4.3 Adjusting Probabilistic Rules via Document Mining

In this section, we explain our approach to mine rules from API reference documents and then adjust the probabilistic model.

4.3.1 Mining API usage rules from documents. To mine rules from documents, we develop an open information retrieval technique to automatically build API usage rules. Given an API document as input, our approach mainly follows the workflow explained in Section 2 to construct usage rules. Differently, to improve the accuracy and completeness of the constructed usage rules, we further design the following three strategies.

• *Construct declaration graph.* Object-oriented programming languages, such as Java, possess intricate inheritance and polymorphism features, which have been extensively studied in code analysis. However, when processing documents, previous research has overlooked the inheritance relationships between classes. In some cases, API usage rules may only be specified by the superclass, leaving out descriptions in subclasses. The absence of knowledge about these inheritance relationships makes it challenging to infer accurate and comprehensive API usage constraints. To mitigate this issue, GraphiMuse constructs an API declaration graph that records the complex relationships between various classes and APIs. When building API usage rules, it refers to the document of the current class and also its superclasses. By utilizing declaration graphs, GraphiMuse can create more precise and complete API usage rules.

• *Extract parameter constraints with LLM.* Parameter constraint denotes a condition that solely pertains to the parameters within a single method. For example, `StringBuffer.insert(int offset)` method has description: *"The offset argument must be greater than or equal to 0"*, from which GraphiMuse can extract a parameter constraint: [offset,>=,0]. Existing approaches extract such constraints using linguistic patterns. However, parameters and constraints in directive sentences are expressed in various forms, such as "offset is not negative", "offset >= 0", or "offset is greater than or equals to zero", which makes it difficult to support all kinds of patterns. To solve the problem, we implement an approach to extract parameter constraints by leveraging the state-of-the-art pre-trained large language model (LLM), ChatGPT, which has shown a strong ability to comprehend diverse and complex descriptions. To obtain constraints using the ChatGPT, we input the system message "I want you to act as an API knowledge miner" and prompt to instruct ChatGPT for mining constraints from the documents and provide examples of mining examples manually annotated by us. Then we continuously ask ChatGPT to generate mining results. If the result is correct, we will make it "think step by step", which is considered to be useful in improving the reliability of LLM [50]. Otherwise, we offer feedback against the error and query ChatGPT to regenerate the result excluding previous fault information. Through multiple iterations, ChatGPT can extract the constraints automatically using its advanced language model capabilities.

• *Extract intra-method constraints with LLM.* As discussed in Section 2, matching entities with concrete API is key for constraint generation. However, relying solely on text similarity between an entity and an API description may lead to imprecise matches. For instance, the method `Scanner.hasNextLine()` has a directive sentence "*throws IllegalStateException, if this scanner is closed*". Current approaches match the entity "*this scanner is closed*" with the method `close()` because its functionality description "*Closes this scanner*" has high textual similarity with this entity. However, this is an incorrect matching since method `close()` directly closes a scanner instead of checking whether it is closed or not. To address this issue, we also leverage ChatGPT to generate more precise constraints. By feeding ChatGPT a snippet of the method function and the directive description, it can output constraints as a triple with higher accuracy. With the help of ChatGPT, GraphiMuse significantly reduces the number of false positive matches.

4.3.2 Integrating document rules into probabilistic model. The API usage rules mined from the documents will be then integrated into the probabilistic model obtained from Section 4.2. The integration is conducted in the following two ways.

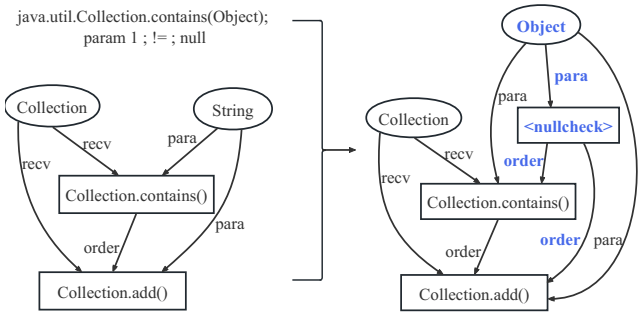


Figure 4: An example of codebase and document integration

• *Integrating new rules from documents.* As discussed in Section 3.2, the rules obtained from either the codebase or documents can be incomplete. While the rules mined from the codebase typically capture the relative positional relationships between different nodes, they may miss important logical relationships, particularly in relation to parameter usage rules. In contrast, the rules obtained from documents often exhibit higher accuracy and stronger logical relationships. They explicitly specify the causal relationships between API invocations and provide clear guidelines for parameter usage. Therefore, GraphiMuse incorporates the rules from the documents into the probabilistic model by adding extra nodes. By doing so, GraphiMuse can leverage the strengths of both sources to create a more accurate and comprehensive model.

Figure 4 illustrates an example of integration. The left graph depicts codebase-mined rules, while the rule extracted from documents of `Collection.contains()`[1] is [*param_1*; !=; `null`]. Both of them focus on the API `Collection.contains()`. The information from the documents emphasizes the need to check whether the first parameter of `Collection.contains(obj)` is `null`, which is not reflected in the graph. Additionally, the documents define the parameter of `contains()` as *Object*, while the graph accepts a *String* as an argument. Therefore, GraphiMuse makes modifications by changing the parameter type from *String* to *Object*, and inserts a `nullcheck` node into the graph. The right part of Figure 4 shows the final API usage rule after the adjustment. GraphiMuse assigns the biggest probability for newly added or modified nodes, as the information from the documents is more reliable. For example, Pr(`nullcheck` | `contains()`) will be assigned a value of 1, indicating that a `nullcheck` is mandatory before calling `contains()`.

• *Adjusting the value of probabilities.* As the document information is considered more reliable, the purpose of adjusting the probability model is to reflect the high accuracy of the rules extracted from documents. To achieve this, GraphiMuse increases the dependency strength between corresponding nodes in the rules mined from the documents. To illustrate this adjustment, let us revisit the example shown in Figure 3. From the documents, GraphiMuse mines the following rule: [`hasNext()`, `pred`, `next()`], which highlights that `hasNext()` is a prerequisite for `next()`. In such cases, GraphiMuse decreases the probabilities of `hasNext()` being false and `next()` being true by a predefined degree $\mathcal{A}$, while ensuring the probability after adjustment remains valid. At the same time, it increases the probabilities of both `hasNext()` and `next()` being true by $\mathcal{A}$. This adjustment aims to emphasize the scenarios where a missing `hasNext()` node is more likely to be misuse.

[1] docs.oracle.com/javase/8/docs/api/java/util/Collection.html

4.4 Detecting API Misuse via Probabilistic Model

The previous subsections describe how GraphiMuse extracts API usage rules from large codebase and documents, and construct probabilistic models. This subsection describes how GraphiMuse utilizes these models for detecting API misuse. The detection method consists of three parts: *graph matching*, *confidence score calculation based probability*, and *API misuse report*.

4.4.1 Graph matching. Given a target program, GraphiMuse first transforms it into a series of AUGs denoted as T_{AUG}. Following this, for each target AUG $t_{AUG} \in T_{AUG}$, GraphiMuse performs graph matching between t_{AUG} and each rule constructed in the previous subsections. The matching process starts with identifying common API call nodes shared between t_{AUG} and *rule*. If any common API call node(s) exist and t_{AUG} lacks some nodes defined in *rule*, the pair <t_{AUG}, *rule*> is considered a potential API misuse candidate.

4.4.2 Confidence score calculation. Since the API usage rule may not always be accurate, leading to possible false positives. Therefore, GraphiMuse evaluates each candidate's confidence score with a probabilistic model. For every misuse candidate <t_{AUG}, *rule*>, the confidence score is defined as the probability of it being a true positive. To accomplish this, GraphiMuse utilizes the probability model to convert the accuracy of rules into a confidence score. Specifically, suppose the node set of *rule* is N=$n_1, n_2, \ldots$, the API call node with t_{AUG} is n_i, and the missing nodes in t_{AUG} is N_m, where $N_m \subset N \wedge n_i \notin N_m$. For instance, for the rule shown in Figure 3 and a given API usage $\texttt{Usage}_{\texttt{sleep}}$ `Thread.sleep()` without `catch` and `InterruptedException`, n_i would be `Thread.sleep()`, N includes the three nodes in Figure 3, $\overline{N_m}$ are the missing `catch` and `InterruptedException` nodes. The possibility of <t_{AUG}, *rule*> being a true positive is calculated as the possibility of $\Pr(\overline{N_m}, N - N_m)$ being a correct API usage rule, i.e., the possibility that all the node from N_m does **not** appear while all the nodes from $N - N_m$ existing. This means that missing N_m will not lead to misuse. Hence, for API call n_i, the confidence score of <t_{AUG}, *rule*> is calculated as $\Pr((\overline{N_m}, N - N_m) \mid n_i)$. The probabilistic model is constructed in Algorithm 2, where all the prior probabilities over n_i (e.g., $\Pr(n_i \mid n_1, n_2 \ldots n_j)$) have been calculated. According to Bayes' theorem and Bayesian Network, GraphiMuse can easily calculate the posterior probability of $\Pr((\overline{N_m}, N - N_m) \mid n_i)$. For the above example, the possibility of usage $\texttt{Usage}_{\texttt{sleep}}$ being correct would be $\Pr((\overline{\texttt{catch}}, \overline{\texttt{InterruptedException}}) \mid \texttt{Thread.sleep()})$, which is very low according to the probabilistic model from Figure 3.

4.4.3 API misuse report. Using the confidence score, GraphiMuse could filter out the detected API misuses with a low confidence score, i.e., ignoring the less trustable detection results. GraphiMuse allows the user to set the threshold θ to filter out the misuses whose confidence score is less than θ. This strategy enables retaining a limited number of results, allowing users to focus on the more trustable

results. Moreover, based on the confidence score, GraphiMuse can also rank the detection results to prioritize detected misuses that are likely to be true positives. Depending on the user's preference, GraphiMuse reports the top N misuses to users.

5 IMPLEMENTATION AND EVALUATION

In this section, we aim to evaluate GraphiMuse and answer the following research questions:

- **RQ1:** Compared to state-of-the-art approaches, how effective is GraphiMuse in detecting API misuses?
- **RQ2:** What is the impact of hyperparameters on the API misuse detection precision?
- **RQ3:** What are contributions of each component to GraphiMuse's effectiveness?

5.1 Implementation

We implement GraphiMuse on top of MuDetect [42]. Our extension over MuDetect utilizes approximately 6,000 lines of Java code and around 1,000 lines of Python code. The document rule mining is implemented in Python, while the codebase rule mining, rule integration functionality, and probability model generation were implemented in Java. When conducting AUG mining, we set the frequency threshold (τ in Algorithm 1) to 10 according to previous studies,which means that nodes appearing with a frequency exceeding 10 will be expanded and further included in the mining process. For document mining, our targets include widely used classes in Javadoc and the documents of projects that are required for precision evaluation in the experiments.

5.2 Experimental Dataset

To evaluate the effectiveness of GraphiMuse, we employ the extended MuBench [3, 42] benchmark as our dataset. MuBench is a collection of API misuse data derived from 69 real open-source software projects, and it is actively maintained. MuBench details each misuse instance, its exact location, and the originating projects. The original MuBench dataset [3] contains 84 API misuse instances. Sven et al. [42] further expanded MuBench to include 141 additional misuses, resulting in an extended benchmark comprising 225 misuses. In addition, during the experimental process, we identified 17 new instances of API misuse. As a result, the API misuses identified in the ground truth include a total of 242 instances. After setting up the dataset, we perform detection at an individual project level and evaluate how many misuses can be detected by GraphiMuse. We use precision and recall as evaluation metrics.

All the experiments are conducted on a device with 64 cores of 2.3GHz CPU, 128GB RAM, NVIDIA Ampere A100 GPUs, and 40 GB memory. The operating system is Ubuntu 20.04. A detailed exploration of our experimental procedures and results is as follows.

5.3 RQ1: Effectiveness of GraphiMuse

To assess GraphiMuse's effectiveness, we measure its precision and recall in detecting API misuses and compare with existing tools.

5.3.1 Evaluate GraphiMuse on the MuBench dataset. We choose GROUMiner [31], DMMC [29] and MuDetect [42] as our comparison tools. GROUMiner represents API usages in the form of directed acyclic graphs, and detects API misuse based on mined graph patterns. DMMC detects missing method calls as violations of the majority rules in the codebase. MuDetect also uses a graph representation of API usages to mine and detect API misuse and achieves promising results. Furthermore, MuDetect has implemented a cross-project version *MuDetectXP*, mining API usage patterns from multiple projects, aiming to increase the accuracy of mined patterns. We also conducted a comparative study with MuDetectXP. Note that, we did not implement a cross-project version of our tool mainly because cross-project mining is inefficient, which takes much time to mine one certain pattern. Moreover, APICAD [47] is a relevant tool that achieves state-of-the-art results, however, it is designed for C/C++, disabling the direct experimental comparison with it.

Table 1: Experimental Results: *Experiment P* and *R* measures precision and recall, respectively. CM: confirmed misuses, KS: kappa score, Hits: detected misuses from ground truth.

Detector	Experiment P			Experiment R		
	CM	P	KS	Hits	R	KS
GROUMiner	6	2.0%	0.64	7	2.9%	1.00
DMMC	19	6.2%	0.52	24	9.9%	0.72
MuDetect	51	13.9%	0.84	49	20.2%	0.87
MuDetectXP	47	27.0%	0.90	95	39.2%	0.84
GraphiMuse	113	42.0%	0.92	132	54.5%	0.93

To measure the effectiveness of the above tool, we follow the experimental setting of MuDetect. Precision is calculated by the ratio of correctly identified API misuses to the total reported by these tools. In contrast, recall is defined as the number of correctly detected API misuses, which are referred to as *hits*, divided by the total number of misuses specified in the benchmark. The correctness of detected misuses is manually validated. To avoid bias of manual inspection, two authors independently reviewed the experimental results and subsequently engaged in discussions. We utilized Cohen's Kappa score to quantify the level of agreement.

In terms of API documents, we take as inputs the Java JDK API reference[2] and the API documents of the projects in MuBench. We use the projects' specific API documents since they describe the public APIs whose usage rules may also be mined from the project itself (e.g., the usage of public API in unit tests).

Results. Table 1 summarizes the results of GROUMiner, DMMC, MuDetect, MuDetectXP and GraphiMuse. Following MuDetect, we scrutinized the top 20 detection outcomes for each project, and GraphiMuse identified a total of 269 instances. Among these, 113 were true positives, culminating in a precision of 42.0%. In comparison, GROUMiner detected 6 true positives out of 300 results, resulting in a precision of 2.0%. DMMC achieved 19 true positives out of 307 results (6.2%). MuDetect obtained 51 true positives out of 367 results (13.9%). MuDetectXP achieved 47 true positives out of 174 (27.0%). The experimental results show that our probabilistic model can effectively detect API misuse. This is because, by representing API usage rules in a probabilistic manner, we significantly reduce false positives, thus improving precision. Moreover, the use of high-quality rule sources mining from reference API

[2] docs.oracle.com/javase/8/docs/api/index.html

documents effectively increases the tool's coverage of API knowledge, enabling it to detect a greater variety of API misuse types. In contrast, GROUMiner, DMMC and MuDetect generated a significant number of false positives due to the issues of nodes under generalization, redundant information, and similar rules. These concerns have been considerably improved in GraphiMuse through the implementation of heuristic strategies and document mining.

Table 1 also presents the results of recall. GraphiMuse identified a total of 132 out of 242 API misuses provided by MuBench, resulting in a recall rate of 54.5%. In comparison, GROUMiner, MuDetect, and MuDetectXP found 7, 49, and 95 misuses, achieving a recall rate of 2.9%, 20.2%, and 39.2%, respectively. It can be observed that GraphiMuse shows a slight improvement in recall rate, although not significantly. GraphiMuse outperforms existing tools mainly because the document knowledge integrates more API usage patterns into the probabilistic model, enabling the detection of more API misuses. This might be attributed to the dataset containing some APIs that are not widely used, making it challenging for codebase mining methods to achieve substantial results.

Although GraphiMuse performs well in terms of averaged precision, it fails to detect certain API misuses. For instance, within the Jackrabbit [4] project, an API misuse of `Map.get()` is caused by the absence of a `int` parameter. However, the widespread use of `int` creates a loosely connected dependency between `int` value creation and API call `Map.get()`. Although this misuse is detected by GraphiMuse, it does not appear among the top 20 detection results due to the low probability. In contrast, MuDetect includes this particular misuse in its top 20 detection results.

5.3.2 Evaluate GraphiMuse in real-world open-source project. In addition to MuBench, we selected 5 high-quality Java open-source projects for the evaluation of GraphiMuse. Due to the absence of ground truth, the detection outcomes were assessed for potential API misuses through manual verification. Furthermore, we evaluated the time required to conduct analyses on open-source projects. The experimental setup we employed is identical to the configuration used in MuBench, with the probability threshold for filtering results set at 80%.

Results. table 2 presents the detection results for real-world open-source projects. Based on the results, GraphiMuse is indeed capable of identifying potential API misuses in real-world open-source projects, with an average accuracy rate of 25.8%. The accuracy is lower compared to the results from the dataset experiments. Upon analyzing the experimental outcomes, we discovered that the reduced effectiveness could be attributed to the challenge of obtaining comprehensive documentation for some open-source projects, which hindered the effective application of methods that integrate codebase and documentation information. Additionally, the strong coding conventions of some open-source project contributors led to the generation of patterns in the code mining process that were not necessarily correct. This is a common problem that is inherently difficult to avoid in pattern-mining-based approaches.

5.4 RQ2: Impact of Hyperparameter

We then measure the impact of the hyperparameter on GraphiMuse's effectiveness.

Table 2: Experimental Results in real-world open-source projects. LOC: Lines of Code, PAM: Potential API Misuses, TF: Total Findings ,KS: Kappa Score, Time: Detection Time.

Project	LOC	PAM	TF	KS	Time(s)
StreamSpinner	43554	15	57	0.90	31.22
OX-Framework	116772	10	27	0.64	75.41
htmlunit-2.9	174832	8	37	0.86	62.25
heritrix3	217446	12	52	0.72	138.36
rhino	238674	10	40	0.92	178.71

5.4.1 Impact of top N. When reporting the detected API misuses at Section 4.4.3, the number of reported misuses affects the precision of GraphiMuse. Beyond the top-20 results shown in Table 1, we also evaluate the precision in top 10, 15, 20, 25, and 30, respectively. Due to the limited number of detection results from MuDetectXP, variations in top n are minor, therefore, Table 2 does not include it. Other experimental settings remain consistent with RQ1.

Table 3: Precision of different tools @Top-n

Tool	P@10	P@15	P@20	P@25	P@30
GROUMiner	2.6%	2.8%	2.0%	1.6%	1.2%
DMMC	7.4%	7.6%	6.2%	6.0%	6.5%
MuDetect	11.6%	10.8%	13.9%	12.6%	12.3%
GraphiMuse	47.9%	45.2%	42.0%	37.8%	35.8%

Results. Table 3 presents the precision of GraphiMuse with N as 10, 15, 20, 25, and 30, where P@N shows the precision with the top-N results. As N decreases, the precision of GraphiMuse steadily improves, increasing from 35.8% at $N = 30$ to 47.9% at $N = 10$. These experimental results demonstrate that GraphiMuse effectively prioritizes more reliable detection results. This outcome is attributed to our ranking metric, which is based on probability models that reflect the reliability of the detection outcomes.

5.4.2 Impact of threshold θ. Moreover, instead of reporting the top-N result, GraphiMuse could report misuse results based on a user-defined threshold θ (Section 4.4.3). The setting of θ also affects the quality of the final rules. Intuitively, if θ is set too low, too many erroneous API misuses will be introduced, leading to a high number of false positives. Conversely, if θ is set too high, many correct misuses will be discarded, resulting in false negatives. To determine a reasonable threshold θ, we conducted experiments to formally analyze the effects of different thresholds.

Table 4: Precision and recall of GraphiMuse with different threshold θ

θ	0.2	0.3	0.4	0.5	0.6	0.7	0.8
Precision	6.2%	10.5%	12.8%	16.8%	33.5%	47.2%	63.1%
Recall	54.5%	54.5%	54.1%	53.7%	52.9%	52.1%	49.6%

Results. Table 4 presents the precision and recall of GraphiMuse with different threshold θ. The experimental results show that as θ gradually increases, the precision continuously improves while the recall declines. As mentioned earlier, despite our probability

model reflecting the reliability of the detection results, there are still instances where true positives have relatively low confidence. These true positives are filtered out as θ exceeds their confidence level, leading to a decrease in recall. However, experimental results indicate that the decrease in recall is relatively gradual, and even when θ increased to 0.8, there is still approximately 50% recall. With this threshold, a substantial number of false positives are filtered out, leading to significant improvement in the detection outcomes. These experimental findings demonstrate that GraphiMuse effectively provides higher confidence levels for true positives.

5.5 RQ3: Ablation Study

Since our work involves building a probabilistic model based on rules mined from both the codebase and the documents, the contribution of each component significantly impacts the effectiveness of GraphiMuse. Therefore, in this section, we perform an ablation experiment to evaluate the contribution of each component.

5.5.1 Effect of heuristic strategies in codebase mining. In Section 4.2, we proposed three heuristic strategies to enhance the accuracy of mined rules from the codebase. We evaluate whether the three heuristic strategies designed in Section 4.2.1 are helpful in increasing GraphiMuse's accuracy. To do that, we simply disable those strategies and execute GraphiMuse using the same setting at RQ1.

Table 5: Effectiveness of codebase mining heuristics and integrating documents

Tool	P@10	P@15	P@20	P@25	P@30	Recall
GraphiMuse	47.9%	45.2%	42.0%	37.8%	35.8%	54.5%
GraphiMuse^{w}	44.5%	37.1%	32.8%	27.2%	23.3%	54.5%
GraphiMuse^{d}	31.1%	28.3%	27.2%	27.0%	27.0%	42.1%

Results. Row GraphiMuse^{w} in Table 5 shows the experimental results with the heuristic strategies in codebase mining disabled. Without these strategies, detection precision falls by 5% to 15% compared to results with heuristics. This improvement can be attributed to the heuristic strategies providing a higher level of generalization for the mined rules from the codebase, enhancing rule abstraction. The experimental results also demonstrate that the heuristic strategy did not alter the tool's recall rate. This is attributed to the fact that our heuristic strategy is primarily designed to unearth more accurate rules and streamline redundant rules, rather than generating new rules. The effectiveness of rule optimization is also evidenced in RQ1, where MuDetect detects 367 results, while GraphiMuse detects only 269, significantly reducing false positives.

5.5.2 Effect of Integrating Documents. In Section 4.3.2, we present techniques for integrating document rules into probabilistic model. We then evaluate whether integrating the knowledge from documents into the probabilistic model is helpful. To measure the effect of integrating documents, we compare the precision of GraphiMuse with and without the knowledge from documents. The remaining experimental settings are the same as RQ1.

Results. Table 5 presents the experimental results, where row GraphiMuse^{d} shows the results without documents. The experimental results reveal that the precision of detection results when incorporating document information, is on average 15% higher compared to the precision without incorporating document information. This improvement can be attributed to the higher priority given to document information during the ranking process, leading to detection rules that rely on documents having greater reliability. In fact, the top 10 detection results are often supported by document information, further emphasizing the high accuracy of such information and its significant contribution to API misuse detection.

5.5.3 Effect of heuristic strategies in document mining. In Section 4.3, we introduced several heuristic strategies in document mining. To evaluate their usefulness, we use the knowledge extractor developed by Ren et al. [40] as a baseline (without those strategies). Since we did not find the open-source tool for this paper, we re-implemented this tool by ourselves. We compare our document mining with this approach using two metrics: recall^{doc} and precision^{doc}. Since the dataset used by Ren et al. is also not available, we randomly sampled 150 API descriptions from the Java JDK API reference as our dataset. We manually analyze the rules contained in each API fragment and take them as ground truth.

Table 6: Effectiveness of document mining heuristics

Type \ Method	Baseline		Ours	
	precision^{doc}	recall^{doc}	precision^{doc}	recall^{doc}
Parameter	89%	72%	94%	89%
Intra-method	57%	88%	89%	88%

Results. Table 6 illustrates the results with and without the heuristic strategies. Our tool achieves a 5% and 17% improvement in precision and recall for parameter rules, respectively, and a 32% improvement in precision for intra-method rule mining. These increases can be attributed to LLM's excellent comprehension of complex natural language in documents, and its ability to discern subtle differences in context. These results validate that our heuristic rules can enhance the quality of patterns mined from documents.

5.6 Discussion

In this section, we discuss the limitation of GraphiMuse and the threats that may affect the validity of our evaluation.

Limitations. First, since our probabilistic model is initially inferred from the codebase, it's influenced by the quality of the code. Although our probabilistic model can alleviate this problem to some extent, the tool might still learn erroneous patterns from improperly written code by developers. Second, due to the inherent limitations of NLP technology, the rules mined from the documents are not guaranteed to be entirely accurate. Moreover, our tool currently supports a portion of misuse categories, but misuse scenarios in real-world projects are diverse, such as repeated invocations (`ServerSocket.bind()` can only be called once), thread synchronization errors (`wait()` and `notify()` should always be used within a synchronized code block), etc. We plan to expand document mining to support a wider variety of misuse types.

Threats to Validity. GraphiMuse outperforms existing approaches on the MuBench dataset. However, it still needs to be tested on more real-world projects and collected feedback to assess its generalizability. Moreover, GraphiMuse currently only supports Java

programs, disabling the experimental comparison with API misuse detector designed for C/C++ programs. We plan to support more programming languages in the near future.

6 RELATED WORK

In this section, we will introduce the relevant work on API misuse detection and the mining of patterns from documents and codebase.

API misuse detection. In work related to API misuse detection, some research uses manually designed templates to represent API patterns. IMChecker [11] employs domain-specific language to transcribe API specifications, enabling static detection of API misuses. Similar to IMChecker, CrySL [17] is designed to ensure proper usage of Cryptographic APIs. Many others use templates, e.g. CogniCrypt [16] and CodeQL [5]. Besides manually designing templates, existing approaches detect API misuses via mutation analysis [51], stacked LSTM [34], active learning and interactive approaches [21]. Different from these approaches, we represent API usage as a graph and detect misuses via graph matching. Moreover, we focus on misuse detection in Java, and there are many tools that detect misuse in C [15], C++ [26, 27], Python [8, 12, 46] and Go [19]. We will consider the support for misuse detection in these languages as part of our future work.

Mining rules from codebase. To automatically generate API usage rules, the most common approach is mining from the codebase. Nielebock et al. [33] extract API information from commits, subsequently mining API patterns from similar codes within the codebase. APISAN [53], MUDetect [42] and DMMC [29] determine patterns based on the frequency of occurrence of API usages. Similar approaches are also used by PR-MINER [22], CHRONICLER [52], GROUMiner [31], and etc. However, these studies rely on frequency-based mining, overlooking the confidence information embedded within the patterns, which leads to low accuracy. Our approach can mitigate this issue by leveraging the probabilistic model. The most relevant work is HAPIs [32] which similarly learns API usage rules via a statistical approach. Differently, HAPIs only learns the method call sequence patterns using a hidden markov model [36], while GraphiMuse models general API usage patterns based on probabilistic graphical model. Other work [9, 10] mine code rules from codebase, but for different purpose.

Mining rules from documents. Besides, researchers have suggested various methods for automated document mining. Pandita et al. [35] infer temporal constraints from natural language API descriptions with NLP and ML techniques. Liu et al. [24] proposed LeadFOL to automatically transform API reference documents into formalized first-order logic formulas using a joint learning method. Several works [18, 25, 40] utilize knowledge graphs to construct fine-grained dependencies and constraint relations of API. In contrast, we use LLM to enhance knowledge retrieval ability and leverage mined rules to make modifications to the probabilistic model, thereby enhancing detection performance.

Combined mining codebase and documents. Wang et al. [47] proposed a method for constructing API usage patterns by incorporating information from both documents and code repositories. They extracted API usage patterns separately from documents and code repositories and then merged them using logical operations to create new patterns. Although this approach is intuitive, it does not fully exploit the advantages of the high accuracy of document information and the wide coverage of code repository information. To address this limitation, we aggregate the knowledge from two sources using the probabilistic model and combine them deeply.

Ranking. Past research has used different sorting techniques, like comparing the similarity between the code under test and the rules [31, 48, 49], or utilizing the frequency of occurrence of the particular misuse as a criterion for sorting [22, 43, 44]. These methods consider the mined rules as correct rules and place high demands on the accuracy of the mining process. In contrast, GraphiMuse does not assume complete rule accuracy, instead, it utilizes the probability model to represent the possibility of a rule being correct.

7 CONCLUSION

In this paper, we proposed GraphiMuse, a novel probabilistic API misuse detector, based on a probabilistic graphical model. It extracts API usage rules from both codebase and documents, and aggregates them to generate a probability model. We employed heuristic strategies in mining codebases and documents to enhance detection accuracy. Evaluation results on MuBench showed that GraphiMuse achieves a precision of 42.0% and a recall of 54.5%, outperforming existing approaches. Furthermore, we assessed the impact of documentation information, hyperparameters, and heuristic strategies. Our implemented tool and dataset are publicly available at https://github.com/18373637myl/GraphiMuse.

8 ACKNOWLEDGEMENT

This work was supported by the National Natural Science Foundation of China under Grant Nos (62202026 and 62141209), and partly by the Guangxi Collaborative Innovation Center of Multisource Information Integration and Intelligent Processing.

REFERENCES

[1] n.d.. httpclient. http://svn.apache.org/repos/asf/jakarta/commons/proper/httpclient/trunk/.
[2] 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.
[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] asf. n.d.. jackrabbit. http://svn.apache.org/repos/asf/jackrabbit/trunk/.
[5] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[6] Cédric Beust. n.d.. TestNG. https://github.com/cbeust/testng.
[7] Robert J Ellison, John B Goodenough, Charles B Weinstock, and Carol Woody. 2010. Evaluating and mitigating software supply chain security risks. *Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TN-016* (2010).
[8] Miles Frantz, Ya Xiao, Tanmoy Sarkar Pias, and Danfeng Daphne Yao. 2022. POSTER: Precise Detection of Unprecedented Python Cryptographic Misuses Using On-Demand Analysis. In *The Network and Distributed System Security (NDSS) Symposium*.
[9] Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[10] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.
[11] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, and Jiaguang Sun. 2019. Vetting api usages in c programs with imchecker. In *2019 IEEE/ACM*

41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 91–94.
[12] Xincheng He, Xiaojin Liu, and Lei Xu. 2023. Python API Misuse Mining and Classification Based on Hybrid Analysis and Attention Mechanism. *International Journal of Software Engineering and Knowledge Engineering* (2023).
[13] Finn V Jensen et al. 1996. *An introduction to Bayesian networks*. Vol. 210. UCL press London.
[14] JodaOrg. n.d.. jodatime. https://github.com/JodaOrg/joda-time.git.
[15] Yuan Kang, Baishakhi Ray, and Suman Jana. 2016. Apex: Automated inference of error specifications for c apis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 472–482.
[16] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–936.
[17] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2019. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2382–2400.
[18] 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.
[19] Wenqing Li, Shijie Jia, Limin Liu, Fangyu Zheng, Yuan Ma, and Jingqiang Lin. 2022. CryptoGo: Automatic Detection of Go Cryptographic API Misuses. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 318–331.
[20] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 241–252.
[21] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1400–1415.
[22] 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.
[23] Qingmi Liang, Zhirui Kuai, Yangqi Zhang, Zhiyang Zhang, Li Kuang, and Lingyan Zhang. 2022. MisuseHint: A Service for API Misuse Detection Based on Building Knowledge Graph from Documentation and Codebase. In *2022 IEEE International Conference on Web Services (ICWS)*. IEEE, 246–255.
[24] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-based extraction of first-order logic representations of API directives. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 491–502.
[25] 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.
[26] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting {Missing-Check} Bugs via Semantic-and {Context-Aware} Criticalness and Constraints Inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. 1769–1786.
[27] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. 2020. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1837–1852.
[28] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17 (2012), 703–737.
[29] 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.
[30] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg.
[31] 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.
[32] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016. Learning API usages from bytecode: A statistical approach. In *Proceedings of the 38th International Conference on Software Engineering*. 416–427.
[33] Sebastian Nielebock, Robert Heumüller, Kevin Michael Schott, and Frank Ortmeier. 2021. Guided pattern mining for API misuse detection by change-based code analysis. *Automated Software Engineering* 28, 2 (2021), 15.
[34] Shuyin OuYang, Fan Ge, Li Kuang, and Yuyu Yin. 2021. API Misuse Detection based on Stacked LSTM. In *Collaborative Computing: Networking, Applications and Worksharing: 16th EAI International Conference, CollaborateCom 2020, Shanghai, China, October 16–18, 2020, Proceedings, Part I 16*. Springer, 421–438.
[35] Rahul Pandita, Kunal Taneja, Laurie Williams, and Teresa Tung. 2016. ICON: Inferring temporal constraints from natural language api descriptions. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 378–388.
[36] Lawrence Rabiner and Biinghwang Juang. 1986. An introduction to hidden Markov models. *ieee assp magazine* 3, 1 (1986), 4–16.
[37] Ti Ramraj and Ri Prabhakar. 2015. Frequent subgraph mining algorithms–a survey. *Procedia Computer Science* 47 (2015), 197–204.
[38] T. Ramraj and R. Prabhakar. 2015. Frequent Subgraph Mining Algorithms – A Survey. *Procedia Computer Science* 47 (2015), 197–204. https://doi.org/10.1016/j.procs.2015.03.198 Graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014).
[39] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. 2020. Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 925–936.
[40] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2021. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. ACM, 461–472. https://doi.org/10.1145/3324884.3416551
[41] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th international conference on software engineering*. 643–652.
[42] 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)*. 265–275. https://doi.org/10.1109/MSR.2019.00053
[43] 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.
[44] 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.
[45] Gias Uddin, Foutse Khomh, and Chanchal K Roy. 2020. Mining API usage scenarios from stack overflow. *Information and Software Technology* 122 (2020), 106277.
[46] Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda. 2021. APIScanner-Towards Automated Detection of Deprecated APIs in Python Libraries. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 5–8.
[47] Xiaoke Wang and Lei Zhao. 2023. APICAD: Augmenting API Misuse Detection Through Specifications From Code And Documents. In *45th International Conference on Software Engineering (ICSE)*.
[48] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18 (2011), 263–292.
[49] 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.
[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]
[51] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 866–877.
[52] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. 2016. Chronicler: Interactive exploration of source code history. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 3522–3532.
[53] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. {APISan}: Sanitizing {API} Usages through Semantic {Cross-Checking}. In *25th USENIX Security Symposium (USENIX Security 16)*. 363–378.
[54] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow. In *Proceedings of the 40th international conference on software engineering*. 886–896.
[55] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer, 318–343.

Received 16-DEC-2023; accepted 2024-03-02