Detecting Defects Based on Mining and Confirming Programming Patterns From Different Versions of Projects

ZHANQI CUI\textsuperscript{1}, (Member, IEEE), XIANG CHEN\textsuperscript{2}, (Member, IEEE), LIWEI ZHENG\textsuperscript{1}, ZHIHUA ZHANG\textsuperscript{1}, AND YONGMIN MU\textsuperscript{1}

\textsuperscript{1}Computer School, Beijing Information Science and Technology University, Beijing 100101, China
\textsuperscript{2}School of Computer Science and Technology, Nantong University, Nantong 226019, China

Corresponding author: Zhanqi Cui (czq@bistu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61702041, in part by the Science and Technology Project of the Beijing Municipal Education Commission under Grant KM201811232016, and in part by the Qin Xin Talents Cultivation Program of Beijing Information Science Technology University under Grant QXTCP C201906.

ABSTRACT Many implied programming patterns are contained in large-scale software. Most of the implied programming patterns are missing proper documentation. Defects would be brought into the software, if any of the patterns is violated by programmers. To alleviate this problem, many works are proposed to find defects by mining programming patterns from the software. However, a great many of candidate patterns and defects are reported by these approaches. These patterns and defects need to be manually confirmed, and the applicability and scalability of these approaches are restricted by this problem. In view of this problem, this paper proposes an approach to automated mining, confirming, filtering function call sequence patterns (FCSPs), and detecting defects which violate the patterns. At first, FCSPs are mined from a previous stable version and an update version under analyzing, respectively; then, the FCSPs are confirmed by analyzing correlations; after that, useful FCSPs are filtered with respect to the FCSPs mined from the previous version; finally, the version under analyzing is scanned for suspicious defects against the filtered FCSPs. 3 open source projects are selected as the experimental subjects to evaluate the approach. As the experimental result shows, the efficiency of defect detection is improved by the proposed approach. It confirms programming candidate FCSPs with 82% F1-measure and 77% accuracy, and eliminates 55% suspicious defects without sacrificing the performance.

INDEX TERMS Programming patterns, mining patterns, confirming patterns, version history, defect detection.

I. INTRODUCTION

Large-scale software implied many programming patterns. Since the time and schedule of software development are limited, many of these implied patterns lack proper specification documents. For example, an observational study are carried out by Saied et al. \cite{1} on API usage constraints and the documentation of API. They found that from 79% to 88% usage constraints are not specified in documents at all. Moreover, some implied programming patterns hide deeply, and their existence are not recognized by software engineers. Therefore, the defects, which violate these patterns, cannot be found by traditional approaches, such as code review.

However, once these implied patterns are violated when programming, such as wrong API invocation sequences or passing arguments with incorrectly types, defects would be brought into the software. To alleviate this kind of defects, a large body of researches are carried out on defect mining \cite{2}, \cite{3}. In defect mining, code and documents of software are converted to datasets, and defect patterns or programming rules are mined from the datasets. Recently, many successfully application of mining programming patterns for defect detection are reported, because defect mining techniques can be automated to a large extent \cite{4}, \cite{5}.

Many techniques have been carried out by researchers to automatically mine implied programming patterns. However, the number of mined candidate patterns and reported defects are usually massive. The applicability and scalability of these
defect mining approaches are severely influenced by this problem. For example, 32283 candidate programming patterns are mined by PR-Miner [6], and 1447 suspicious defects are reported even after inter-procedural analysis; 200 open-source projects are used as the experimental subjects by Legunsen et al. [7], as the experimental results shown, in the reported suspicious bugs, which violate the mined programming patterns, 97.89% of them are false positives. Furthermore, the process of validating candidate programming patterns and suspicious defects will consume substantial manual efforts and error prone. Moreover, the experiences and skills of the software engineers play important roles in this process. In addition, the validation process is difficult to automate and error prone. By analyzing mined programming patterns, we find that useful information is implied in the semantics of patterns in the form of natural language, which can be further used to confirm them. For example, the elements between real patterns like (fileOpen(), fileClose()) are strong correlated, whereas the correlations between the elements of false positive patterns are weak.

Besides, with the rapid development of new software process methodologies and the prevalence of mobile Internet, like agile processes and mobile applications, these new features lead to releasing update versions frequently. Many historical versions are accumulated during these processes. For example, according to the version history on Anzhi App store, the Today’s Headline and the Tik Tok, which are the most popular mobile applications of news and short videos in China, released 4 Android platform versions in April 2019, respectively. Many applications release updates once or twice a week. Intuitively, these historical versions of software imply considerable information.

In this paper, a novel approach is proposed to mine FCSPs for detecting defects. The mined FCSPs are confirmed and filtered by using the information of the programming patterns in the form of natural language and the historical version information of the project. At first, it mines candidate FCSPs from two different versions of a program; then, it confirms candidate FCSPs by embedding them into vectors and analyzes correlation between elements of the pattern; after that, it compares the consistence between the two sets of confirmed FCSPs, filters out confirmed FCSPs with respect to the change of the confidence of FCSPs; finally, it scans the program under analyzing for suspicious defects against the filtered FCSPs. In the experiments and evaluation, we find that candidate programming patterns can be automatically confirmed by correlation analysis and a large portion of FCSPs are contained across different versions. In addition, without sacrificing the defect detection ability, the suspicious defects are reduced by using the historical version information.

The contributions of the paper can be summarized as follows:

- We propose an approach of automatically confirming programming patterns by embedding patterns based on the natural language semantics implied in the patterns. The number of candidate patterns is reduced by correlation analysis.
- We propose an approach of finding defects with respect to the FCSPs mined from different versions of a project. By using the information of the historical version to filter FCSPs, the suspicious defects are reduced.
- To evaluate the effectiveness of the approach, we implement a prototype tool CV-Miner and carry out an experimental study on 3 open source projects.

The remainder of the paper is organized as follows. The related work of mining and confirming programming patterns are reviewed in Section II. Section III introduces how to confirm FCSPs, and how to filter FCSPs from two different versions of a project for detecting defects. Section IV discusses the design and results of experiments. Finally, we conclude the paper in Section V.

II. RELATED WORK

Mining implied programming patterns from the source code of programs is one of the popular intercross fields of software engineering and machine learning, the mined patterns are widely used for defect detection [4], API recommendation [16], etc. Engler et al. [17] proposed to extract a set of rule templates, to improve the accuracy of static analysis. The rule templates are classified as MUST and MAY rules, which are used to check against the program under test to find violations. Kremene et al. [18] proposed another template based approach, which used a group of probabilistic graphical models to infer specification from program. A large body of research works [5] are carried out to mine various kinds of programming rules for detecting defects in programs after then, such as association rules, sequence patterns, state machines, frequent subgraphs, etc.

Program elements which tend to occur together frequently are mined by PR-Miner [6] as association rules. Programming rules which composed by variable and function elements both can be found by PR-Miner. The AntMiner [19] tool employ program slicing techniques to improve the accuracy of mining specification. The original source repository is decomposed into independent sub-repositories by AntMiner. In sub-repositories, statements not relevant to critical operations are excluded. Alattin is developed by Thummalapenta and Xie [20] for mining alternative patterns. The alternative patterns are like “P1 or P2”. In this kind of patterns, the P1 and P2 are alternative rules. The false positive rates are reduced by Alattin. To improve the precision of defect detection, PSP-Finder is proposed by Cui et al. [15] to mine path-sensitive function call association patterns for detecting defects. Yun et al. [21] proposed APISan to automatically infer correct API usages by relaxed symbolic execution.

To reduce the false programming rules reported, the syntactical context of the source code is considered by Kagdi et al. [22] for mining sequential patterns. Perracotta [23] proposed a dynamic approximate inference algorithm to mine sequence patterns in large programs. A large

---

1Anzhi App store: http://www.anzhi.com/
set of inferred temporal properties are reduced to a small set of interesting temporal properties by Perracotta. In order to produce effective specifications for finding defects, Wu et al. [24] proposed MineHEAD to exploit heterogeneous data, including client programs, library source code, and comments. Three types temporal specification patterns are mined by MineHEAD: precedence, sequence, and pairing.

ADABU [25] collected program execution information to alleviate the problem that mined state machines are too large and many states are unlabeled. The execution information is observed by ADABU to construct object behavior models. Then, object behavior models are summarized in the form of state machines. As the result shown, ADABU generated smaller and more easily understandable state machine models. To improve the performance and scalability, the collaborations of selected objects are focused by Pradel and Gross [26] to mine programming rules in the form of state machines.

To represent the rules that need to be complied, program rule graphs (PRG) are mined by Zhong et al. [27]. The PRGs are in the form of directed graphs. A prototype tool Java Rule Finder (JRF) is implemented to infer specifications from Java API libraries. To mine usage patterns of multiple objects, Nguyen et al. [28] proposed a graph-based method GrouMiner. In GrouMiner, the labeled directed acyclic graphs (DAG) are used to model the usage of objects in scenarios, and the subgraphs which are frequently emerges in the DAGs are mined as usage patterns in the form of subgraphs.

An automatic tool AutoInfer is presented by Wei et al. [29] to infer sophisticated postconditions of commands as contracts from the simple contracts in the code, which are written by programmers. Relevant, non-trivial invariant contracts are extracted by AutoInfer. By analyzing control dependency and mining source code, Nguyen et al. [30] proposed to find the preconditions of APIs of large-scale open source projects, in order to help software engineers using APIs correctly.

Studies are carried out to address the problem of high false positive rates of the mined patterns and defects. PR-Miner [6] performed an inter-procedural checking, including check the callee and the caller for each function that contains violations, to prune false positive violations. In order to improve the accuracy of mining temporal safety policies, WM-Miner proposed to use trustworthiness metrics to weight the contribution of each trace for mining specifications. To reduce the false positive patterns reported, NAR-Miner [31] only reported rules between program elements with strong semantic relationships. A strong semantic correlation between two program elements often implies a data share or data dependence relationship between them.

To improve the accuracy of mining defects, we proposed to detect defects based on mining FCSPs in previous work [14]. In the approach, the constraints implied in function call sequences are utilized and the false positive defects are reduced. Different from the previous work, in this paper, we confirm candidate FCSPs by embedding them to vectors and filter confirmed patterns by using historical information of the program, to further reduce false positive candidate FCSPs and suspicious defects. As we know, this paper is the first attempt to confirm and filter mined candidate programming patterns by embedding and to mine implied programming patterns from different versions of programs for detecting defects.

This paper focus on implied programming patterns in the form of the function call sequences. However, the approach proposed in this paper can accommodate to implied programming patterns in other forms, like frequent subgraphs and sate machines, and it can accommodate to implied programming patterns which are consisted by other kinds of program elements, like variables and conditions.

### III. OUR APPROACH

As Figure 1 show, a novel approach is proposed to confirm and filter mined FCSPs in different versions of a project for detecting defects. Let’s assume $V_\alpha$ is a previous version of a software, $V_\beta$ is a update version under analyzing. At first, it extracts function call information from both $V_\alpha$ and $V_\beta$ of the software and mines for candidate FCSPs, respectively; then, it confirms candidate FCSPs based on correlation analysis; after that, it filters out confirmed FCSPs of $V_\beta$ by the confidence change; finally, it scans $V_\beta$ for defects which go against the filtered FCSPs.

#### A. MINING FCSPS FROM PROGRAMS

A program is composed by functions, which are basic functional modules. A set of $n$ functions compose a version of a program can be denoted as $V = \{FD_1, FD_2, \ldots, FD_n\}$. A set of $m$ statements which compose a function $FD$, can be expressed as $FD = \{st_1, st_2, \ldots, st_m\}$. A statement in a program could be either variable define, variable use, conditional, function declaration, and function call etc. Among them, the function call statements are used to call other functions to complete the expected features. To simplify the analysis, the function call statements in a function are extracted to a sequence. In the body of a function definition $FD$, the function call statements in $FD$ can be expressed as a sequence $FC = \{fc_1, fc_2, \ldots, fc_q\}$ ($fc_i \in FD$ and $fc_i$ is a function call statement), according to the appearance order in $FD$. This sequence can be used to represent the relationship of functions in some way. Although the control flow, data flow, point-to information are not analyzed in this paper, they can be obtained by static analysis [8], [9] for more precise results.

The set of all the items are represented as $I = \{i_{t1}, i_{t2}, \ldots, i_{tn}\}$, which are the function call statements in the program under analysis. A sequence $(is_1, is_2, \ldots, is_p)$ is an ordered list of itemsets, $D$ denotes the dataset of the sequences. An element $is_i$ in the sequence is a sub-set of $I$, which can be denoted as $(i_{t1}, i_{t2}, \ldots, i_{tq})$. The items in an itemset is an unordered set, whereas the itemsets in a sequence are ordered. For two itemsets $is_u$ and $is_v$, in a sequence, if $u < v$, $is_u$ emerges before $is_v$ in the sequence. For two sequences, $s_u = \langle a_1, a_2, \ldots, a_k \rangle$ and
Mining sequential pattern is to mine frequent sequences in the dataset of sequences with minimum supportcount, which is specified by users [10]. For a program under analyzing, \( V = \{FC_1, FC_2, \ldots, FC_n\} \) is used to represent the dataset of function call sequences. \( FC_i \in V \) is a function call statements sequence in the function definition of \( FD_i \). Equation (1) defines the support of a function call sequence in a program. The supCount of sequence \( s \) represents the count of sequences in \( V \) which contain \( s \), whereas \( |V| \) is the count of function definitions. The support is used to denote the rate of sequences in \( V \) that contain \( s \).

\[
\text{support}(s, V) = \frac{\text{supCount}(s, V)}{|V|},
\]

The commonness of a sequence pattern is measured by support. minSup is the lower bound value of support. The frequent patterns are patterns with support greater than or equal to minSup. For a program under analyzing, the frequent FCSPs are represented as a set \( SP \), which is used as candidate FCSPs for following analysis.

Based on AprioriAll [11], the GSP algorithm [12] is used in this paper to mine frequent FCSPs in the programs. In the GSP algorithm, the mined sequential pattern are restricted by maximum and minimum gaps. In the process of mining a FCSP, if the distance of two adjacent elements in the FCSP is far, it is probably a false FCSP, since the possibility of existing correlations between the FCSP elements is relatively low. In this approach, we mine frequent FCSPs in both version \( V_a \) and \( V_b \) to generate two pattern sets \( SP_a \) and \( SP_b \), respectively.

**B. CONFIRMING FUNCTION CALL SEQUENCE PATTERNS**

Programming patterns are typically composed by function calls, variable declaration, variable usage, conditions. All these patterns are composed by function names and variable names. By analyzing mined programming patterns, we find that the elements between real patterns are strong correlated, whereas the correlations between the elements of false positive patterns are weak. For example, the function names of a candidate pattern \( \langle \text{dictGetIterator}(), \text{dictClose}() \rangle \) imply both of the two functions will operate on object \( \text{dict} \). The correlation between \( \text{dictGetIterator}() \) and \( \text{dictClose}() \) is strong, and it is a real FCSP after manually checked. On the contrary, we cannot find strong correlations between the two elements of a candidate pattern \( \langle \text{lookupKeyReadOrReply}(), \text{checkType}() \rangle \), and it is a false positive FCSP after manually checked.

Previous programming pattern mining works suffer the problem that a large number of candidate patterns, which will consume huge manual efforts to validate. To alleviate this problem, we propose to confirm candidate patterns by utilizing the natural language semantics implied in candidate patterns and embedding programming patterns to vectors.

Firstly, the function names in the candidate FCSPs are obtained and converted into sets of words. A FCSP \( sp \) which is composed by \( n \) function call statements can be represented as \( sp = \{fc_1, fc_2, \ldots, fc_n\} \). \( fc \) is the name of a function called in the statement, which is typically composed by a set of words. The name of a function statement can be represented as \( fc = \{w_1, w_2, \ldots, w_m\} \), \( w_i \) is a word included in the name. The name of functions generally comply with specific standards, such as camel case and underscore case. For example, a phrase in the camel case capitalize the first letter of words in the middle of the phrase with no intervening spaces, underling or punctuation. In contrast, the underscore case split each word with a underline. As we all know, abbreviations are often used in the name of functions. For example, “f” is used to replace “file” and “2” is used to replace “to” for short. These abbreviations need to be restored to the original case before analysis. Furthermore, a stop words list is constructed to improve the accuracy of analysis. The words “to” and “for”, which are widely used in the names of functions without specific meanings, are included in this list. According to above instructions, an element of a FCSP can be converted to a set of words.

Secondly, each element of a FCSP is embedded to a vector. For a function call statement \( fc = \{w_1, w_2, \ldots, w_m\} \),
which is an element in pattern \( sp \), each word in \( fc \) can be embedded to a \( k \)-dimensional vector as \( \overrightarrow{w} = [z_1, z_2, \ldots, z_k] \) by word embedding approaches, such as word2vec \([13]\). After all the words of an element are converted to vectors, the element is embedded to a vector according to Equation (2).

\[
\overrightarrow{fc} = \sum_{i=1}^{m} \overrightarrow{w_i} = [z_{fc,1}, z_{fc,2}, \ldots, z_{fc,k}],
\]

Thirdly, the FCSPs are confirmed by the average distance of pattern elements. The cosine distance of two function statement \( fc_u \) and \( fc_v \) in a pattern \( sp \) is calculated by using Equation (3). Then, the distances of elements in a pattern are calculated pair by pair and the average distance between elements of the pattern are calculated by using Equation (4). Based on our observation, the elements of real patterns are usually very similar, such as \((\text{openFile}()), \text{closeFile}())\), whereas false positive patterns are usually dissimilar. As a result, \( \text{min}_{dis} \) is set as the lower bound value for the average distance of real patterns. For a pattern \( sp \), distance\((sp)\) is the average cosine distance of its elements. If distance\((sp)\) is greater than \( \text{min}_{dis} \), \( sp \) is reported as a real pattern, otherwise, \( sp \) is reported as a false positive pattern.

\[
distance(\overrightarrow{fc_u}, \overrightarrow{fc_v}) = \frac{\sum_{i=1}^{k}(z_{fc,u,i} \times z_{fc,v,i})}{\sqrt{\sum_{i=1}^{k}(z_{fc,u,i})^2 \times \sum_{i=1}^{k}(z_{fc,v,i})^2}},
\]

\[
\text{distance}(sp) = \frac{\sum_{u=1}^{n} \sum_{v=1}^{n} \text{distance}(\overrightarrow{fc_u}, \overrightarrow{fc_v})}{C_n^2}, \quad (fc_u, fc_v \in sp, \text{ and } v \neq u),
\]

Algorithm 1 describes the procedure of confirming candidate FCSPs. First, the elements of candidate FCSPs are embedded to vectors in line 2-7. Then, the average distance of elements of a FCSP is calculated in line 8-16. Finally, the candidate FCSPs with average distances which are greater than \( \text{min}_{dis} \) are ruled out in line 17-19.

C. FILTERING FCSPS BY ANALYZING THE HISTORICAL VERSION

The programming patterns across different versions are utilized to improve the efficiency of defect detections. A large number of implied programming patterns can be found in both of the two versions as frequent patterns, whereas other implied programming patterns are only frequent in one version or even only emerge in one version. As Figure 2 shows, for the FCSPs of \( V_\beta \), six cases can be classified with respect to the function call sequences of \( V_\alpha \). Case 1, 2, 3 and 4 represent FCSPs both contained in \( V_\alpha \) and \( V_\beta \), whereas case 5 and 6 represent FCSPs only exist in \( V_\beta \), which are updates like new methods defined or libraries introduced in \( V_\beta \). In our approach, FCSPs in case 2, 4 and 6 are taken out from the set of candidate FCSPs.

![Algorithm 1 Confirming Candidate FCSPs](image)

**Algorithm 1 Confirming Candidate FCSPs**

**Input:**
- Set: \( SP /\text{candidate FCSPs} \)

**Output:**
- Set: \( SP /\text{confirmed FCSPs} \)

1: for each \( sp_i \) in \( SP \) do
2: for each element \( fc_j \) in \( sp_i \) do
3: for each word \( w_k \) in \( fc_j \) do
4: \( \overrightarrow{w_k} \leftarrow \text{embedding the word} \ w_k \text{ to a vector} \)
5: \( fc_j \leftarrow \text{embedding elements to a vector} \)
6: end for
7: end for
8: for each element \( fc_u \) in \( sp_i \) do
9: for each element \( fc_v \) in \( sp_i \) do
10: if \( fc_u \neq fc_v \) then
11: \( \text{sum} \leftarrow \text{sum} + \text{distance}(\overrightarrow{fc_u}, \overrightarrow{fc_v}) / \text{the sum of distances between elements} \)
12: end if
13: end for
14: end for
15: if \( \text{distance}(sp_i) \geq \text{min}_{dis} \) then
16: \( SP'.add(sp_i) / \text{rule out candidate FCSPs with long average distances} \)
17: end if
18: end if
19: end if
20: end for
21: return \( SP' \)

![Figure 2. Different cases of FCSPs between two versions.](image)

**FIGURE 2. Different cases of FCSPs between two versions.**

We define confidence to evaluate the certainty degree of a FCSP. For a FCSP and a program, \( \text{supCount} \) is the count of function call sequences in the program which contain the FCSP, and \( \text{vioCount} \) is the count of function call sequences in the program which violate the FCSP. The confidence of a FCSP is the value of dividing \( \text{supCount} \) by the sum of \( \text{supCount} \) and \( \text{vioCount} \). The definition of violate will be further discussed in subsection 2.4. For a FCSP \( sp \) and a program \( V \), Equation (5), as shown at the bottom of this page, defines the confidence of \( sp \) with respect to \( V \). MinConf is the lower bound value of confidence for a real FCSP. A FCSP
could be a false FCSP, if its confidence is smaller than minConf. The confirmed FCSPs are measured by confidence to evaluate the degrees of certainty.

For a stable version \( V_\alpha \) and a update version \( V_\beta \), the confidence change of a FCSP \( sp \) is defined as Equation (6).

\[
\delta_{\text{conf}}(sp, V_\alpha, V_\beta) = \text{confidence}(sp, V_\beta) - \text{confidence}(sp, V_\alpha).
\]

(6)

We have more confidence on \( V_\alpha \) than \( V_\beta \), because \( V_\alpha \) was thoroughly tested before releasing and has been used for a while. In \( V_\beta \), if the confidence of a FCSP decreases, it gives a clue that new places which violates the FCSP are introduced in this version. In other words, new defects are probably brought into the \( V_\beta \) version. In other cases, if the confidence of a FCSP remains the same or increases, it means that no new defects violating the FCSP are brought into \( V_\beta \). We suppose that the software engineers obey FCSPs for the majority of the time and only makes mistakes accidentally. Therefore, a FCSP may be a false pattern if the confidence of the FCSP significantly increases or decreases. We set \( \delta_{\text{max}} \) as a upper bound value of \( \delta_{\text{conf}} \), FCSPs with \( |\delta_{\text{conf}}| \) greater than \( \delta_{\text{max}} \) are ruled out as false positives.

Algorithm 2 describes the process of filtering confirmed FCSPs. According to the above heuristic insights, the confirmed FCSPs could be filtered by following rules: 1) exclude FCSPs which have confidences less than \( \text{minConf} \) in line 3; 2) exclude FCSPs which have \( \delta_{\text{conf}} \) greater than or equal to \( 0 \) in line 7; 3) exclude FCSPs which have \( |\delta_{\text{conf}}| \) greater than \( \delta_{\text{max}} \) in line 8. Besides, the FCSPs which only contained in \( V_\beta \) are directly add into \( SP_\beta'' \) in line 14.

D. CHECKING THE PROGRAM AGAINST THE FCSPS FOR DEFECTS

After the FCSPs are confirmed and filtered, the update version of the program under analyzing is checked against the FCSPs to find violation. These function definitions which not comply with the FCSPs are reported as suspicious defects.

A violation of a FCSP \( sp = \{fc_1, fc_2, \cdots, fc_n\} \) with respect to a function call sequence \( s \) is defined as: a subsequence \( sp' \) of \( sp, sp' = \{fc_1, \cdots, fc_{i-1}, fc_{i+1}, \cdots, fc_n\} \) is included in \( s \), but \( fc_i(1 < i < n) \), which is a function call statement, is not included in \( s \). A violation of \( sp \) with respect to \( s \) means the function definition of \( s \) violates \( sp \), and a suspicious defect is contained in the function. To quantitatively measure the certainty degree of a suspicious defect, the suspicious of a defect is defined as Equation (7), as shown at the top of the next page.

\[\text{confidence}(sp, V) = \begin{cases} \frac{\text{supCount}(sp, V)}{\text{supCount}(sp, V) + \text{vioCount}(sp, V)}, & \text{supCount}(sp, V) > 0, \\ 0, & \text{supCount}(sp, V) = 0. \end{cases}\]

(5)

Algorithm 3 presents the process of checking program against FCSPs. It should be noted that the reported defects still need to be validated manually, because the mined FCSPs are potential implied usage patterns from the statistic point of view. In order to save the costs of manually validating reported suspicious defects, the FCSPs with low suspicious and corresponding defects are further reduced by suspicious in line 3. minSus is a threshold value of suspicious, because we suppose software engineers only make mistakes occasionally. The violations with suspicious value greater than or equal to \( \text{minSus} \) are reported as suspicious defects in line 7.

IV. EXPERIMENTS AND EVALUATION

A. PROTOTYPE TOOL IMPLEMENTATION

To evaluate the proposed approach, a prototype tool named CV-Miner is implemented. All the experiments are conducted on a PC with 8GB memory and Intel i7 2.3 GHz CPU, running Ubuntu 16.04 with pyinstaller 2.14 and Python 3.5.

The CV-Miner is consist of three main modules. The function call sequences in programs are extracted by the program analysis module, then the Rapidminer\(^2\) is used to mine
candidate FCSPs of the two versions, respectively. The pattern confirmation module confirms mined candidate FCSPs by embedding them to vectors and calculates the distance between elements of the pattern. The word2vec\(^3\) tool is used to convert words to vectors. The confirmed FCSPs are filtered by the defect detection module, then the filtered patterns are used to check the version under analyzing for detecting code segments which violate the FCSPs. The suspicious defects are reported according to the violations.

### B. EXPERIMENTAL DESIGN

In the experiments and evaluation, we plan to answer following research questions.

RQ1. Can the candidate programming patterns be confirmed by using the semantics information of them in the form of natural language?

RQ2. Is it common that different versions of a project contain a large portion of same programming patterns?

RQ3. Can the historical versions be effectively used to promote the defects detection efficiency?

RQ4. Can the defects detection capability of the approach be sacrificed by automated confirming and filtering programming patterns?

To answer above research questions, the CV-Miner is compared with our previous work \([14]\). In this paper, the approach in \([14]\) is called Seq-Miner. To facilitate the comparison, the three open source projects used in \([14]\) are chosen as the experimental subjects to compare CV-Miner and Seq-Miner from following aspects, include the time costs, the number of FCSPs, the number of suspicious defects reported. The three open source projects include the memory database Redis,\(^4\) the cross-platform scripting language Lua,\(^5\) and the Sqlite\(^6\) which is a embedded database.

Table 1 lists the statistics of the experimental subjects. The “Versions” column gives the different versions of the projects, one is the update version under analyzing, and the other is an older version which has been used. Different projects have different practices in versioning. Redis uses major.minor:patchlevel for its versioning. For Redis, release with an even minor number means it is a stable version, whereas releases with odd minor numbers are unstable versions. For instance, Redis 2.9. x is unstable version, Redis 3.0 is a stable version. According to the version history of Redis, the latest version is 4.0.10, and the last stable version is 3.2.12. Lua use x.y.z for its versioning. x.y is the version number and z is the release number of Lua. According to the version history, the latest version is 5.4.0, and the last stable version is 5.3.5. According to the version history, the latest version of Sqlite is 3.24.0, and the last stable version of Sqlite is 3.23.1.

In Table 1, the number of C files are listed in the “.c files” column. The lines of code are listed in the “LOC” columns (without blank lines and comments). As Table 1 shows, the size of the three projects varies from 2 to 75 files and 12k to 142 k lines of code. The number of functions definitions in the three projects are listed in the “Function definitions” column. This is also the number of sequences which are used to mine FCSPs. The number of function call statements of the three projects are listed in the “Function calls” column.

### C. EXPERIMENTAL EVALUATION

By following the values of parameters used in \([14]\) and \([15]\), we set the values of minSup, minConf, maxGap, and minSus to 0.01, 0.8, 10, and 0.9 in the experiments, respectively. With respect to minSus, δ\(_{\text{max}}\) is set to 0.1.

The time costs of the CV-Miner and the Seq-Miner are compared in Table 2. The name of the two methods are listed in the “Methods” column. The time costs to generate FCSPs, time costs to check for violations of FCSPs, and the total time costs are listed in “Time to generate FCSPs”, “Time to check for suspicious defects”, and “Total time” columns, respectively. In the total time, the time costs of mining FCSPs by Rapidminer are not counted. Besides, the manual efforts

---

\(^3\)Word2vec: https://code.google.com/archive/p/word2vec/

\(^4\)Redis: https://redis.io/

\(^5\)Lua: http://www.lua.org/

\(^6\)Sqlite: https://www.sqlite.org/
for validating the suspicious defects are not counted either. As Table 2 shows, the time costs of CV-Miner is 2.1 times of the Seq-Miner in total. The reason is that the CV-Miner mines FCSPs from an additional historical version, and confirming and filtering candidate FCSPs of the historical version cost additional times. However, the procedure of extracting information from source code, mining FCSPs, confirming candidate FCSPs, filtering confirmed FCSPs and detecting defects by checking programs against the filtered FCSPs are fully automated. No extra manual efforts will be introduced by the CV-Miner.

Experiments are carried out on Redis 3.2.12, Lua 5.3.5, and Sqlite 3.2.3.1 to evaluate the effectiveness of automatically confirming programming patterns. In the experiment, true positive (TP) is used to denote the number of real patterns that are correctly reported, true negative (TN) is used to denote the number of false patterns that are correctly reported, false positive (FP) is used to denote the number of false patterns that are incorrectly reported as real, false negative (FN) is used to denote the number of real patterns that are incorrectly reported as false. Base on above four indicators, precision, recall, $F_1$ - measure, and accuracy are used as the metrics to evaluate CV-miner. Precision is a ratio of correctly reported patterns (TP) and the total number of reported real patterns (TP + FP). Recall is the total number of correctly reported real patterns (TP)/total number of real patterns (TP + FN). $F_1$ - measure is the harmonic mean of precision and recall, which is calculated as $2 \times$ precision $\times$ recall/(precision + recall). Accuracy is calculated as number of correctly reported patterns (TP + TN)/total number of candidate patterns (TP + TN + FP + FN).

As Figure 3(a) shows, the precision increases as the $min_{dis}$ increases. As Figure 3(b) shows, the recall decrease as the $min_{dis}$ increases. As Figure 3(c) and (d) shows, both the values of average $F_1$ - measure and accuracy achieve max when $min_{dis} = 0.6$. Thus, we choose $min_{dis} = 0.6$ as the min threshold of cosine distances between pattern elements in following experiments.

Table 3 shows the detail values of the performance metrics when $min_{dis} = 0.6$. The column “Mined FCSPs” and “Confirmed FCSPs” are the number of FCSPs mined as candidate candidates and confirmed as real patterns by CV-miner, respectively. As Table 3 shows, 9 out of 13 pattern candidates are confirmed as real patterns which means 31% candidates do not need to be further validated manually. In the project Redis, the cosine distance of two elements in candidate candidate $\langle$getDecodedObject(), decrRefCount$\rangle$ is 0.25 and reported as a false pattern. But it is a real pattern indeed, which are used to increase and decrease the reference count of a object, and cause a false negative. Generally, 7 out of 8 real patterns and 3 out of 5 false patterns are correctly recognized. As Table 3 shows, when $min_{dis} = 0.6$, the average precision, recall, $F_1$ - measure, and accuracy values are $78\%$, $88\%$, $82\%$, and $77\%$, respectively. As the result shows, our answer for RQ1 is that “the natural language information of candidate programming patterns can be used to confirm them effectively and reduce the manual efforts for validating candidate patterns”.

Table 4 compares FCSPs generated between version $V_\alpha$ and $V_\beta$, which are the last stable version has been used and the update version under analyzing, respectively. As the column “FCSPs” shows, “Candidates” list the mined candidate
FIGURE 3. The precision, recall, F1-measure and accuracy of confirming programming patterns with different $\min_{dis}$.

FCSPs, and “Confirmed” list the FCSPs confirmed by correlation analysis. The column “Only in $V_\alpha$” and “Only in $V_\beta$” are the number of FCSPs only contained in $V_\alpha$ and $V_\beta$, respectively. The number of FCSPs contained in both of the two versions are listed in the middle column is. According to the confidence change, the FCSPs can be further categorized into three types. As Table 4 shows, 11 out of 13 (85%) candidate FCSPs mined from the $\alpha$ versions can also be found in the $\beta$ versions of the three projects. Besides, all the 11 candidate FCSPs mined from the $\beta$ versions can also be found in the $\alpha$ versions. Among these 11 common candidate FCSPs, 5 of them increase the confidence, 2 of them decrease confidence, the confidence of the rest 4 FCSPs remains the same. In addition, all the confirmed patterns are contained in both of the two versions. Therefore, our answer for RQ2 is that “different versions of a project contains a large number of same FCSPs”. Moreover, a large percentage of the common FCSPs keep the same confidence unchanged.

Defects are injected into the projects by fault injection techniques to evaluate the effectiveness of the approach. For FCSPs, we use two mutation operators to inject defects: one mutation operator is to remove a function call statement in a function definition, the other is to exchange the order of two function call statements in a function definition. For each project, the “FCSPs” column lists the FCSP randomly selected, the “Mutations” column is the mutation operator applied, the “Files” and “Functions” columns are the location where the fault injected. Six faults are injected by applying the two mutation operators as Table 5 presented.

The results of detecting injected defects of the three projects by CV-Miner and Seq-Miner are compared in Table 6. The number of FCSPs, which are used for checking violation by CV-Miner and Seq-Miner, are listed in column “FCSPs”. The number of reported suspicious defects are listed in the “Suspicious defects reported” column. The “Defects” column lists the number of injected faults, which are found by the tools.

As Table 6 describes, the CV-Miner reports 17 suspicious defects in total, whereas 38 suspicious defects are reported by the Seq-Miner in total. For the 3 projects, the CV-Miner method reduces 55% of suspicious defects. Furthermore, 11 FCSPs are used by the Seq-Miner, which is 3.7 times of the FCSPs used by the CV-Miner. Fewer FCSPs and suspicious defects will cost less human efforts, because the FCSPs and suspicious defects need to be manually validated. The fewer FCSPs and suspicious defects reported, the less human efforts will be cost. Therefore, our answer for RQ3 is that “by reducing the candidate FCSPs and the suspicious defects, the information of historical versions of programs can be used to promote the defect detection efficiency”.

As the result shows, all the injected 6 bugs are detected by both of the two methods. Since the ability of detecting defects of CV-Miner is based on Seq-Miner, CV-Miner cannot detect more defects. However, the experiment results show that CV-Miner can detect as many defects as Seq-Miner after reducing the number of candidate FCSPs and suspicious defects. Thus, our answer for RQ4 is that “in comparison with Seq-Miner, the CV-Miner method reduces the candidate FCSPs and suspicious defects without sacrifice the capability of detecting defects, which violate programming patterns”. CV-Miner can only detect new defects introduced into the new version by updates. Defects already
TABLE 4. The distribution of FCSPs across different versions of the projects.

| Projects | FCSPs | Only in $V_\alpha^d$ | Confidence in $V_\beta - \text{Confidence in } V_\alpha$ (Exist in both $V_\alpha$ and $V_\beta$) | Only in $V_\beta$ | Total in $V_\alpha$ | Total in $V_\beta$ |
|----------|-------|---------------------|---------------------------------------------------------------------------------|-----------------|-------------------|-----------------|
| Redis    | Candidates | 2 | 2 | 0 | 0 | 2 | 7 |
|          | Confirmed  | 0 | 0 | 0 | 0 | 0 | 0 |
| Lua      | Candidates | 1 | 1 | 0 | 0 | 1 | 1 |
|          | Confirmed  | 0 | 0 | 0 | 0 | 0 | 0 |
| Sqlite   | Candidates | 0 | 0 | 0 | 0 | 0 | 0 |
|          | Confirmed  | 0 | 0 | 0 | 0 | 0 | 0 |
| Total    | Candidates | 2 | 2 | 4 | 5 | 13 | 11 |
|          | Confirmed  | 0 | 0 | 2 | 5 | 9 | 9 |

TABLE 5. The faults injected in Redis, Lua, and Sqlite.

| Projects | FCSPs | Fault ID | Files | Functions | Mutations |
|----------|-------|----------|-------|-----------|-----------|
| Redis    | (dictNext(), dictRelea- | 1 | server.c | resetCommandTaleStats() | remove |
|          | seltator()) | 2 | pubsub.c | pubsubUnsubscribeAllChannels() | exchange |
| Lua      | (luaL_buftail(), luaL_ | 3 | lstlib.c | str_pack() | remove |
|          | pushresult()) | 4 | liblu.b | | |
| Sqlite   | (sqlite3_mutex_enter(), | 5 | sqlite3.c | sqlite3.db_status() | exchange |
|          | sqlite3_mutex_unlock()) | 6 | sqlite3.c | str_pack() | remove |

V. CONCLUSION AND FUTURE WORK

In order to improve the efficiency of defects detection based on mining programming patterns, this paper proposes to mine, confirm, and filter FCSPs in different versions of a project to find defects. At first, the approach mines candidate FCSPs which are implied in different versions of a project, respectively; then confirms candidate FCSPs based on correlation analysis; finally, filters out confirmed FCSPs based on confidence changes and checks the program for suspicious defects. As the experimental results shows, the natural language information implied the candidate FCSPs can be used to confirm them, and the historical version information can be used to filter FCSPs and reduce suspicious defects reported.
repository, and explore the possibility of checking programs against mined programming patterns across different projects.

**ACKNOWLEDGMENT**

The authors would like to thank Dr. Yingfei Xiong at Peking University and the anonymous reviewers for their insightful comments. Part of this work was carried out while the first author was visiting Peking University. The preliminary version of this article was presented at the Annual Conference on Software Analysis, Testing, and Evolution (SATE), in 2018 [32].

**REFERENCES**

[1] M. A. Saied, H. Sahraoui, and B. Dufour, “An observational study on API usage constraints and their documentation,” in Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol., Reeng. (SANER), Mar. 2015, pp. 33–42.

[2] J. Wang, Y. Deng, H. Zhang, K. Chen, T. Xie, and D. Zhang, “Mining succinct and high-coverage API usage patterns from source code,” in Proc. 10th Working Conf. Mining Softw. Repositories (MSR), May 2013, pp. 319–328.

[3] P. Yu, F. Yang, C. Cao, H. Xu, and X. Ma, “Mining API usage change rules for software framework evolution,” Sci. China Inf. Sci., vol. 61, no. 5, May 2018, Art. no. 050108.

[4] Z. Li, J. Wu, and M. Li, “Sidudy on key issues about API usage,” J. Softw., vol. 29, no. 6, pp. 1716–1738, 2018.

[5] M. P. Robillard, E. Bodden, D. Kowrykow, M. Mezini, and T. Ratchford, “Automated API property inference techniques,” IEEE Trans. Softw. Eng., vol. 39, no. 5, pp. 613–637, May 2013.

[6] Z. Li and Y. Zhou, “PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code,” in Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng., Sep. 2005, pp. 306–315.

[7] O. Legunsen, W. U. Hassan, X. Xu, G. Rouy, and D. Marinov, “How good are the specs? A study of the bug-finding effectiveness of existing java API specifications,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng., Sep. 2016, pp. 602–613.

[8] H. Mei, Q. Wang, L. Zhang, and J. Wang, “Software analysis: A road map,” Chin. J. Comput., vol. 32, no. 9, pp. 1697–1710, 2009.

[9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the state of static analysis: A large-scale evaluation in open source software,” in Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER), Mar. 2016, pp. 470–481.

[10] J. C.-W. Lin, W. Gan, and T.-P. Hong, “Efficiently maintaining the fast updated sequential pattern trees with sequence deletion,” IEEE Access, vol. 2, pp. 1374–1383, 2014.

[11] R. Agrawal and R. Srikant, “Mining sequential patterns,” in Proc. 11th Int. Conf. Data Eng., Mar. 1995, pp. 3–14.

[12] R. Srikant and R. Agrawal, “Mining sequential patterns: Generalizations and performance improvements,” in Proc. Int. Conf. Extending Database Technol., vol. 31, no. 6, 1996, pp. 1–17.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in Proc. Workshop 1st Int. Conf. Learn. Represent. (ICLR), Jan. 2013, pp. 1–12.

[14] Z. Cui, Y. Mu, Z. Zhang, and W. Wang, “Defects detection based on mining function call sequence patterns,” Comput. Sci., vol. 44, no. 11, pp. 226–231, 2017.

[15] Z. Cui, X. Chen, Y. Mu, M. Pan, and R. Wang, “PSP-finder: A defect detection method based on API mining correlations from function call paths,” Chin. J. Electron., vol. 27, no. 04, pp. 776–782, Jul. 2018.

[16] X. Sun, C. Xu, B. Li, Y. Duan, and X. Lu, “Enabling feature location for API method recommendation and usage location,” IEEE Access, vol. 7, pp. 49872–49881, 2019.

[17] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, “Bug as deviant behavior: A general approach to inferring errors in systems code,” in Proc. 18th ACM Symp. Operating Syst. Princ. (SOSP), Oct. 2001, pp. 57–72.

[18] T. Kremenek, P. Twomey, G. Back, A. Ng, and D. Engler, “From uncertainty to belief: Inferring the specification within,” in Proc. 7th Symp. Operating Syst. Des. Implement. (OSDI), Nov. 2006, pp. 161–176.

[19] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, “AntMiner: Mining More Bugs by Reducing Noise Interference,” in Proc. 38th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), 2016, pp. 333–344.

[20] S. Thummalapenta and T. Xie, “Alattin: Mining alternative patterns for detecting neglected conditions,” in Proc. 24th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), 2009, pp. 283–294.

[21] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, “APISan: Sanitizing API usage through semantic cross-checking,” in Proc. 25th USENIX Secur. Symp., 2016, pp. 363–378.

[22] H. Kagdi, M. L. Collard, and J. I. Maletic, “An approach to mining call-usage patternswith syntactic context,” in Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Nov. 2007, pp. 457–460.

[23] I. Yang, D. Evans, D. Bhadwaj, T. Bhat, and M. Das, “Perracotta: Mining temporal API rules from imperfect traces,” in Proc. 28th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), May 2006, pp. 282–291.

[24] Q. Wu, G.-T. Liang, Q.-X. Wang, and H. Mei, “Mining effective temporal specifications from heterogeneous API data,” J. Comput. Sci. Technol., vol. 26, no. 6, pp. 1061–1075, 2011.

[25] V. Dallmeier, C. Lindig, A. Wasylykowski, and A. Zeller, “Mining object behavior with ADABU,” in Proc. Int. Workshop Dyn. Syst. Anal., 2006, pp. 17–24.

[26] M. Pradel and T. R. Gross, “Automatic generation of object usage specifications from large method traces,” in Proc. 24th IEEE/ACM Int. Conf. Automat. Softw. Eng. (ASE), Nov. 2009, pp. 371–382.

[27] H. Zhong, L. Zhang, and H. Mei, “Inferring specifications of object oriented APIs from API source code,” in Proc. 15th Asia–Pacific Softw. Eng. Conf. (APSEC), Dec. 2008, pp. 221–228.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Graph-based mining of multiple object usage patterns,” in Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC/FSE), 2009, pp. 383–392.

[29] Y. Wei, C. A. Furia, N. Kazimian, and B. Meyer, “Inferring better contracts,” in Proc. 31st IEEE/ACM Int. Conf. Softw. Eng. (ICSE), May 2011, pp. 191–200.

[30] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, “Mining preconditions of APIs in large-scale code corpus,” in Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE), 2014, pp. 166–177.

[31] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, “NAR-miner: Discovering negative association rules from code for bug detection,” in Proc. 26th ACM Joint Eur. Softw. Eng. Conf. Found. Softw. Eng. (ESEC/FSE), Nov. 2018, pp. 411–422.

[32] Z. Cui, X. Chen, Y. Mu, Z. Zhang, and X. Ma, “Mining function call sequence patterns across different versions of the project for defect detection,” in Proc. Int. Conf. Softw. Anal., Test., Evol., 2018, pp. 154–169.

ZHAOQI CUI received the B.S. and Ph.D. degrees in software engineering and computer science from Nanjing University, in 2005 and 2011, respectively. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interest includes software analysis and testing.

XIANG CHEN received the Ph.D. degree in computer software and theory from Nanjing University, in 2011. He is currently an Associate Professor with the School of Information Science and Technology, Nantong University. His research interests include software defect prediction, regression testing, and combinatorial testing.
LIWEI ZHENG received the Ph.D. degree in computer software and theory from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, in 2009. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include requirement engineering and trusted computing.

ZHIZHUA ZHANG received the M.S. degree in computer application from the Harbin University of Science and Technology, in 1996. She is currently an Associate Professor of computer science with Beijing Information Science and Technology University. Her research interest includes software testing.

YONGMIN MU received the Ph.D. degree from the China University of Mining and Technology, Beijing, in 1997. He is currently a Professor with the Computer School, Beijing Information Science and Technology University. His research interests include automated software testing, and software theory and application.