计算机科学
COMPUTER SCIENCE

基于决策树算法的 API 误用检测

李康乐，任志磊，周志德，江贺

引用本文

李康乐，任志磊，周志德，江贺. 基于决策树算法的 API 误用检测[J]. 计算机科学, 2022, 49(11): 30-38.

LI Kang-le, REN Zhi-lei, ZHOU Zhi-de, JIANG He. Decision Tree Algorithm-based API Misuse Detection[J]. Computer Science, 2022, 49(11): 30-38.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于改进 CenterNet 的航拍绝缘子缺陷实时检测模型

Real-time Detection Model of Insulator Defect Based on Improved CenterNet

计算机科学, 2022, 49(5): 84-91. https://doi.org/10.11896/jsjkx.210400142

基于集成回归决策树的 lncRNA-疾病关联预测方法

Ensemble Regression Decision Trees-based lncRNA-disease Association Prediction

计算机科学, 2022, 49(2): 265-271. https://doi.org/10.11896/jsjkx.201100132

一种可用于分类型属性数据的多变量回归森林

Multivariate Regression Forest for Categorical Attribute Data

计算机科学, 2022, 49(1): 108-114. https://doi.org/10.11896/jsjkx.201200189

基于决策树的车联网安全态势预测模型研究

Research on Forecasting Model of Internet of Vehicles Security Situation Based on Decision Tree

计算机科学, 2021, 48(6A): 514-517. https://doi.org/10.11896/jsjkx.200700158

基于随机森林的入侵检测分类研究

Research on Intrusion Detection Classification Based on Random Forest

计算机科学, 2021, 48(6A): 459-463. https://doi.org/10.11896/jsjkx.200600161

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.211100177

基于决策树算法的 API 误用检测

李康乐[1]　任志磊[1,2]　周志德[1]　江　贺[1]
1 大连理工大学软件学院　辽宁 大连 116600
2 高安全系统的软件开发与验证技术工业和信息化部重点实验室(南京航空航天大学)　南京 211106
(kangleli@mail.dlut.edu.cn)

摘　要　通过应用程序编程接口(Application Programming Interface,API)复用已有的软件框架或类库,可有效地提高软件开发效率。然而,正确使用 API 须遵守很多规约,如调用顺序、异常处理等。若违反了这些规约就会造成 API 误用,进而可能导致软件崩溃、产生错误或漏洞。尽管很多 API 误用检测技术已经被提出,但是这些技术仍面临两个方面的挑战:1)难以获取 API 使用规约;2)难以同时检测多种不同类型的 API 误用。为了应对上述挑战,提出了一种基于决策树算法的 API 误用检测方法。首先,将 API 使用源代码转换为 API 使用图,从图中挖掘 API 使用规约,有效地应对了第一个挑战。其次,在获取的 API 规约信息的基础上构建 API 使用决策树,并通过融入剪枝策略来提高 API 使用决策树的泛化能力。最后,在检测阶段提出了粗粒度和细粒度相结合的检测方式,来提高 API 使用决策树的检测能力,有效地应对了第二个挑战。实验结果表明,该方法能够在一定程度上发现 API 误用缺陷。

关键词:API 误用;决策树;规约挖掘;缺陷检测
中图法分类号　TP311

Decision Tree Algorithm-based API Misuse Detection

LI Kang-le[1],REN Zhi-lei[1,2],ZHOU Zhi-de[1] and JIANG He[1]
1 School of Software Technology,Dalian University of Technology,Dalian,Liaoning 116600,China
2 Key Laboratory of Software Development and Verification Technology of High Security System Ministry of Industry and Information Technology (Nanjing University of Aeronautics and Astronautics),Nanjing 211106,China

Abstract　Application programming interface(API) benefits to effectively improve software development efficiency by reusing existing software frameworks or libraries. However,many constraints must be satisfied to correctly use APIs,such as call order,exception handling. Violation of these constraints will cause API misuse,which may result in software crashes,errors,or vulnerabilities. Although many API misuse detection techniques have been proposed,these techniques still face two challenges:1) the acquisition of API usage specification is difficult,and 2) the detection of many different types of API misuse at the same time is difficult. To address the above challenges,a decision tree algorithm-based API misuse detection method is proposed. First,the API usage source code is converted into an API usage graph,and the API usage specification is mined from the graph to effectively solve the first challenge. Second,an API usage decision tree is constructed based on the obtained API specification information,and the generalization ability of the API usage decision tree is improved by incorporating pruning strategies. Finally,a combination of coarse-grained and fine-grained detection is proposed in the detection phase to improve the detection capability of the API usage decision tree,which effectively solves the second challenge. Experimental results show that the proposed approach can realize detection of API misuse defects to a certain extent.

Keywords　API Misuse,Decision tree,Specification mining,Bug detection

1　引言

在软件开发过程中,为了节省软件开发时间和提高软件开发效率,开发人员经常需要使用 API 来复用已有的软件框架或类库。利用 API 信息隐藏机制,开发人员无需访问源码或理解被调用 API 内部工作机制的细节,就可以完成相应的功能。但是正确地使用 API 需要遵循一定的规约,比如在使用文件读写流读写文件时,要进行异常处理并在使用完后

到稿日期:2021-11-17　返修日期:2022-06-01
基金项目:南京航空航天大学科研基地创新(理工类)项目(NJ2020022);国家自然科学基金(62032004,62072068);国家重点研发计划(2018YFB1003900)
This work was supported by the Fundamental Research Funds for the Central Universities(NJ2020022),National Natural Science Foundation of China(62032004,62072068) and National Key Research and Development Program of China(2018YF-B1003900).
通信作者:任志磊(zren@dlut.edu.cn)

及时关闭流等。如果 API 在使用过程中出现了违反规约的情况，就可能导致软件崩溃、产生错误或漏洞。然而，由于 API 种类众多、文档信息不够完善、更新维护不及时等原因，开发人员在学习使用 API 的过程中面临着严峻的挑战，导致在软件开发过程中经常存在一些潜在的 API 误用，严重威胁软件的安全。

虽然各种程序分析技术已经被提出用于检查 API 的误用，但是 API 误用情况仍然很普遍[1-2]。现有的 API 误用检测方法大致可以分为两类。第一类是检查是否违反给定 API 规约的工具，如 IMChecker[3] 和 Semmle[4] 等。这些工具的有效性取决于人工编写规约的质量。然而，编写精确的规约需要丰富的专业知识，即使是对有经验的开发人员来说也有很大的挑战性。此外，编写规约必须使用领域特定语言（Domain Specific Languages，DSL）编写，这些领域特定语言随工具的不同存在较大差异，如 IMChecker 的 Yaml、用于 Semmle 的 CodeQL 等，这会进一步增加 API 使用的负担。第二类是以 APISAN[5] 和 JADET[6] 为代表的从代码中自动挖掘使用规约的方法。为了自动检测并最终修复此类误用，从实际代码中推断 API 使用规约，待检测代码中出现与已识别的使用规约不符即为误用。但是最近的工作表明此类方法仍然有很多不足[7]，即其能捕获常见的 API 使用规约，但是检测的精度不高，而且会出现一些漏报。

为了有效地检测 API 误用，需要解决如下问题。

（1）API 使用规约的获取。在文档不全或缺失的情况下，获取 API 的使用规约是一件比较困难的事情[8]。受制于手工编写规约的巨大代价，大部分软件都不提供完整的使用规约，或根本不提供使用规约，而期望开发人员编写精确的规约也是不合理的[9]。

（2）同时检测多种不同类型的 API 误用。一个源程序文件可能包含多种 API 误用类型，对于这种情况，想要较为全面地检测 API 误用是一件比较困难的事情，经常会有漏报的情况[10]。

本文提出了一种基于决策树算法的 API 误用检测方法。具体来说，本文首先分析了各类 API 误用的根本原因，将 API 使用源代码转换为 API 使用图（API Usage Graph[1)]，AUG），基于这些误用因素从 AUG 中针对性地挖掘 API 使用规约，采用广度优先遍历 AUG 中的节点，根据节点之间数据流边和控制流边的传入传出关系，挖掘目标 API 使用上下文规约。主要挖掘包括：目标 API 的参数值信息、前后 API 调用信息、异常处理信息、前置条件信息以及后置条件信息[11]。违反这些规约，在实际开发中容易导致软件的功能性错误、性能问题、安全漏洞等代码缺陷[8]，本文专注于挖掘这些群体的 API 使用规约，从而有效解决第一个问题。

其次，本文将 API 误用检测建模为一个分类预测问题来解决第二个问题。利用挖掘到的 API 使用规约提出了一种 API 误用检测模型，即 API 使用决策树（API Usage Decision Tree，AUDT）。AUDT 是在 ID3 算法的基础上通过融入剪枝策略构建而成的。该算法以信息论为基础，采用信息增益来度量根节点的选择，从而生成决策树。为了提高 AUDT 的泛化能力，分别加入预剪枝和后剪枝策略。预剪枝在对决策树的每个节点划分前先进行估计，若当前节点的划分不能带来决策树的泛化性能的提升，则停止划分并将当前节点标记为叶节点。后剪枝对于生成的一棵完整的决策树，自底向上对非叶子节点进行考查，若该节点对应的子树用叶子节点能带来决策树泛化性能的提升，则将该子树替换为叶子节点。在误用检测阶段，提出了粗粒度和细粒度两种检测机制。具体来说，粗粒度检测用于检测 5 种关键信息是否缺失的误用情况，避免了重复执行决策树；细粒度检测是在 API 使用规约信息不缺失的基础上，利用决策树检测目标代码中这些规约信息是否使用正确。

为了评估提出的方法，本文选取了典型的 Java API 误用示例。实验结果表明，在当前训练数据的情况下，AUDT 具有一定的误用检测能力，能够检测出多类型的 API 误用。综上所述，本文的贡献如下：

（1）提出了一种基于误用根本原因为导向的 API 使用规约挖掘策略。从 API 误用的根本原因出发，挖掘出与 API 使用相关的详细信息。

（2）提出了一种 API 使用决策树方法 AUDT，用于检测目标 API 使用过程中包含的误用。

本文第 2 节介绍了 API 误用检测的背景和动机；第 3 节介绍了本文方法的框架；第 4 节进行了实验评估；第 5 节介绍了相关的工作；最后总结全文并展望未来。

2 背景和动机

2.1 API 误用背景

开发人员在使用 Java API 进行软件开发的过程中，由于自身的疏忽，比如在遍历集合中的元素之前忘记进行空检查，或是在遍历集合的同时修改了集合中的元素等，会导致一些非预期的行为[8]。这些 API 误用是导致软件错误、崩溃和漏洞的普遍原因。API 的使用通常需要满足特定的使用规约，这些规约由一些基本程序元素组合而成，如方法调用顺序、异常处理等[12]。API 使用过程中这些元素的组合受到约束，API 使用规约描述了调用软件库应该满足的要求，违反这些使用规约则会造成 API 误用。不幸的是，API 文档并没有充分指明这样的规约[13]。因此，开发人员也会参考论坛中的答案，比如 Stack Overflow，来理解 API 的用法[14]。但最近的一项研究表明，Stack Overflow 上的代码片段可能是不可靠的，即使是那些被接受和支持的答案[12,15]。

2.2 动机

Amann 等提供了一个公开的 API 误用数据集，该数据集包含了很多跨多个 Java 项目的 API 误用[8]。Nguyen 等[11] 研究了 Amann 提供的部分误用数据集，总结了造成 API 误用的五大原因：方法调用的顺序不正确（或因果调用关系不正确）、异常处理不正确、缺少前置条件、缺少后置条件，以及参数值不正确。

本文还研究了剩余数据集以及其他误用数据集，如图 1 所示。

1) https://github.com/stg-tud/MUDetect

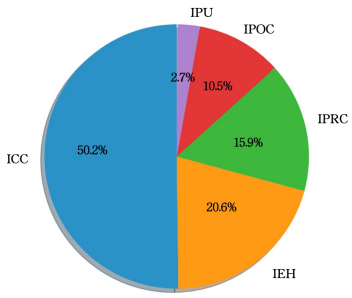

图 1 各类型误用所占比例

Fig. 1 Percentage of each type of misuse

其中约有 50.2%的 API 误用是违反方法调用之间的顺序造成的，这类误用被称为不正确的因果关系调用/缺失因果调用(Incorrect /Missing Causal Calling，ICC)[3]。约有20.6%的 API 误用是由于异常处理不正确造成的，本文将这类误用称为不正确的异常处理/缺失异常处理(Incorrect/Missing Error Handling，IEH)。异常处理不正确包括：缺失异常处理，即对于应该处理的异常未进行捕获；错误的异常处理，比如应该捕获 RuntimeException 而捕获了其他的异常。约 15.9%的 API 误用是由于不合适的前置条件/缺失前置条件造成的(Incorrect/Missing Pre_Condition，IPRC)，比如，在调用 next()之前没有进行 hasNext()检查，一个函数的返回值作为另一个函数的参数时，需要检测该返回值是不是满足条件。约 10.5% 的 API 误用是由于不合适的后置条件/缺失后置条件造成的(Incorrect/Missing Post_condition，IPOC)。约 2.7%的误用是由于使用不正确的参数值而导致的 API 误用(Improper Parameter Using，IPU)。

这些误用原因基本上涵盖了所有类型的 API 误用。分析导致误用的具体原因，从误用本身出发，在大量优质项目中挖掘出 API 规约信息，即目标 API 的参数值信息、前后 API 调用信息、异常处理信息、前置条件信息以及后置条件信息，利用这些信息构建 API 使用决策树，进而检测目标代码中是否出现误用情况。现有的误用检测器搜索范围大，不能聚焦到造成误用的根本原因上，这激励了本文以误用的根本原因为方向去挖掘 API 使用规约。通过分析这 5 种 API 误用类型，本文发现造成 API 误用的情形中都包括使用不正确和缺失，这激励本文在构造 API 使用决策树时对不正确和缺失分开检测，避免在规约缺失的情况下重复执行决策树而降低检测效率。在误用检测阶段，首先检查是否存在缺失的这种误用，排除这种误用后进一步检查是否存在违反具体规约的使用。

3 方法框架

本文企图从源代码中获取有关目标 API 相关的 5 类规约信息，这些信息具体描述了 API 在使用过程中需要注意的事项。在此基础上建立和完善 API 使用决策树并对目标代码进行检测，判断是否存在 API 误用的情况。

如图 2 所示，本文方法的框架分为 4 个部分。第一部分为代码示例搜集阶段。首先从 GitHub 上搜寻包含目标 API 使用的优质项目，通过 API 示例提取器搜集有关目标 API 使用相关的示例。第二部分为源代码解析与规约信息提取阶段。首先需要对源代码进行解析，本文将源代码转换为 AUG，启发式地寻找 API 使用的 5 类规约信息。第三部分为 API 使用决策树的构建阶段。利用第二部分挖掘到的目标 API 的规约信息，构造并完善 API 使用决策树。第四部分为 API 误用检测阶段。利用上一步构造的 API 使用决策树检测目标代码中是否包含 API 误用以及具体的误用情况。

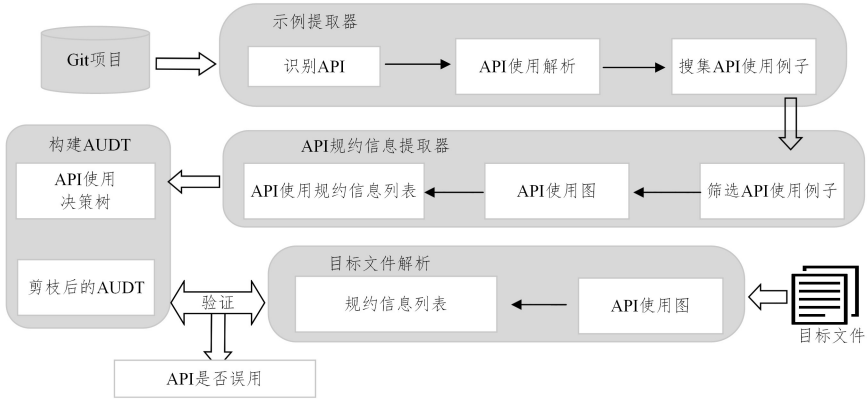

图 2 API 误用检测框架

Fig. 2 API misuse detection framework

3.1 搜集目标 API 使用相关的源文件

在 API 使用相关源文件搜集阶段，本文主要搜集 Github 托管平台上的开源 Java 项目。通过源文件提取器，过滤项目中的配置等文件，得到目标 API 使用相关的 Java 源文件。

3.2 从源代码中挖掘 API 使用规约

源代码是高度结构化的程序，对于各种方法之间的调用关系以及 API 的具体使用方法是比较隐晦的，直接从源程序中提取规约信息是非常困难的一件事。这里借鉴 Sven 等[16]的思想，将源程序转换为抽象语法树(Abstract Syntax Tree，AST)，并在 AST 的基础上进一步解析构造，经过静态代码分析，将源代码转化为 AUG 的形式，在此基础上更加细粒度地拆分和描述每个节点的信息。AUG 是一个有向连通图，由标记的节点集和边集组成。节点代表数据实体，边表示节点所代表的实体之间的流向。图 3 展示了本文中一个 API 使用图抽取的例子。节点分为 Action 节点和 Data 节点。使用

Action 节点（图 3 中的方框）表示 API 使用中的方法调用、操作符。使用 Data 节点（图 3 中的椭圆框）来表示在 API 使用中的对象、参数值。边分为控制流边（图 3 中的虚线）和数据流边（图 3 中的实线）[16]。

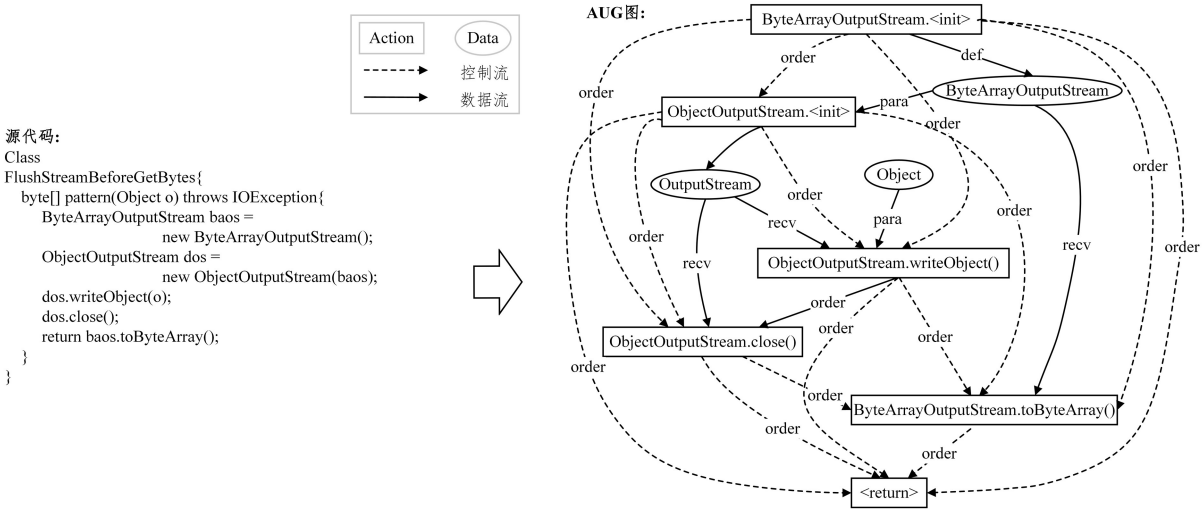

图 3　源程序转换为 AUG 图

Fig. 3　Source program conversion to AUG

本文启发式地从 AUG 中挖掘出 5 类规约信息。采用广度优先策略，在 AUG 中搜寻有关目标 API 的参数值信息、前后 API 调用信息、异常处理信息、前置条件信息以及后置条件信息，这些信息详细地描述了该 API 方法的具体使用方式。如果某条规约信息自身也需要满足使用规约，则递归地挖掘出其自身的规约。具体流程如图 4 所示。

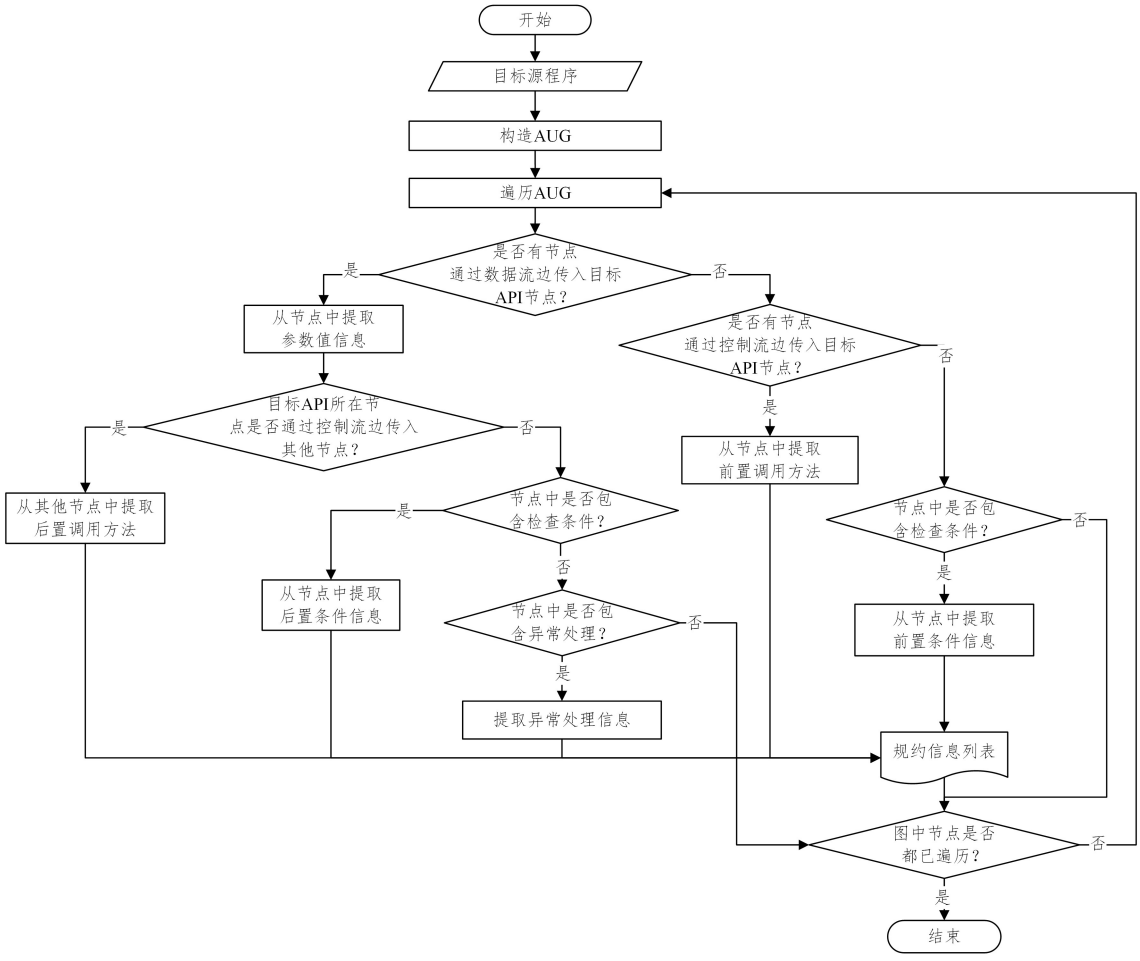

图 4　从 AUG 中提取规约信息流程图

Fig. 4　Flowchart of extracting constraint information from AUG

首先，将目标源程序转换为 AUG。然后，遍历 AUG，判断是否有节点 $Node_i$ 通过数据流边传入目标 API 节点 $Node_t$，

如果有则从节点中提取参数值信息加入到规约信息列表中。接着，判断目标 API 所在的节点 $Node_i$ 是否通过控制流边传入其他节点 $Node_i$，如果是则从 $Node_i$ 节点中提取后置方法调用加入到规约信息列表中，否则继续判断节点 $Node_i$ 中是否包含检查条件。如果 $Node_i$ 包含检查条件，则从中提取后置条件信息加入到规约信息列表中；如果不包含，则进入是否包含异常处理的判断。如果包含异常处理，则提取异常处理信息加入规约信息列表中；如果不包含，则判断图中节点是否都已遍历。如果没有节点通过数据流边传入目标 API 节点，那么就接着判断是否有节点 $Node_i$ 通过控制流边传入目标 API 节点 $Node_i$，若有，则从 $Node_i$ 节点中提取前置调用方法加入到规约信息列表中；若没有，接着判断 $Node_i$ 节点中是否包含检查条件，若包含则从 $Node_i$ 节点中提取前置条件信息加入到规约信息列表中，否则判断图中节点是否都已遍历。如果图中节点都已遍历，结束该流程；否则对没有遍历的节点重复以上操作，直至所有 $Node_i$ 节点都已遍历。

具体挖掘 API 规约信息的步骤如算法 1 所示。

算法 1 提取目标 API 的使用规约

输入：TargetAPI，Set[AUG]

输出：包含规约信息的 JSON 文件

```
1. for each AUG do
2.     if TargetAPI. parameter ≠ null then
3.         Map←{Parameter←this. value}
4.     end if
5.     while NodehasEdge⋃Visited≠AUG. AllNode do
6.         for each node do
7.             Visited←CurrentNode
8.             Query←Currentnode. next
9.             Query. remove(CurrentNode)
10.            if 前置节点不为空 then
11.                Map←{Pre_api←PreNode. api}
12.                Map←{Pre_condition←Condition}
13.            end if
14.            if 后置节点不为空 then
15.                Map←{Post_api←PostNode. api}
16.                Map←{Post_condition←Condition}
17.            end if
18.            if 前置节点或后置节点中包含异常 then
19.                Map←{Exception←exception}
20.            end if
21.            if 前置节点或后置节点中包含异常 then
22.                HashMap←{Exception←exception}
23.            end if
24.        end for
25.        更新 Map 中各个类型的频率值
26.    end while
27. end for
28. 将 Map 持久化到 JSON 文件中
```

首先，确定传入目标 API 的参数节点是否为空，如果不为空，则收集当前的参数值信息并将其以〈*Parameter*，*value*〉键值对的形式添加到 *Map* 集合中(第 2—4 行)。对一个连通有向图进行广度优先搜索，将已经访问过的节点 *Current-Node* 加入到访问过的节点集合 *Visited* 中。对于与当前节点直接相连的节点，如果没有访问过，则将其加入到队列 *Query* 中。提取当前 API 节点的上一个节点中的调用方法并将其加入到 *Pre_api*，以〈*Pre_api*，*value*〉键值对的形式存入 *Map* 集合(第 11 行)。

如果当前节点的上一个节点中有包含检查条件信息，则将该条件加入到 *Pre_condition* 并以〈*Pre_condition*，*value*〉键值对的形式存放到 *Map* 集合中(第 12 行)。提取当前节点的下一个节点中的 *API* 调用并将其加入到 *Post_api* 中，以〈*Post_api*，*value*〉键值对的形式存入 *Map* 集合。如果当前节点的下一个节点有包含检查条件的信息，则将该条件加入到 *Post_condition* 中，并以〈*Post_condition*，*value*〉键值对的形式存放到 *Map* 集合中(第 15—16 行)。如果当前节点的前一个节点或后一个节点中包含异常处理的信息，将异常处理信息加入到 *Exception* 中，并以〈*Exception*，*value*〉键值对的形式存放到 *Map* 集合中(第 18—20 行)。每遍历一个 AUG，更新一下 *Map* 以及每种规约信息的频率值。频率值是每种规约信息对应出现的次数与目标 API 使用图总数之比。最后，将 *Map* 中的内容持久化到 JSON 文件中。

3.3 构建 AUDT

在挖掘到规约信息的基础上，本文把 API 误用检测问题建模为一个分类预测问题。使用 ID3 算法构建 AUDT[17-18]，决策树的生成依赖于信息增益的变化。接下来介绍信息熵和信息增益。

3.3.1 信息熵

熵表示随机变量的不确定性，熵值越大则变量的不确定性就越大。设 x 是一个有限值的离散随机变量，其概率分布如下：

$$P(X=x_i)=p_i, i=1,2,\cdots,n \tag{1}$$

则随机变量 X 的熵定义为：

$$Ent(X)=-\sum_{i=1}^{m} p_i \log(p_i) \tag{2}$$

3.3.2 信息增益

特征 A 对训练数据集 D 的信息增益如式(3)所示，其中 D^v 表示接下来用的特征有 V 个选择，以参数值为例，有参数值的个数、参数值的类型以及参数值的范围。每次选取使得信息熵减少最多的特征作为节点。

$$Gain(D,a)=Ent(D)-\sum_{i=1}^{V}\frac{|D^v|}{|D|}Ent(D^v) \tag{3}$$

3.2 节中已挖掘出目标 API 的 5 类规约信息以及这些规约信息对应规约内容出现的频率，这一节将利用这些信息，在 ID3 算法的基础上构建 API 使用决策树。构建 AUDT 的过程如算法 2 所示。

算法 2 构建决策树

输入：训练集 D；属性值 A；频繁阈值σ

输出：决策树

```
1. def TreeGenerate(D,σ)
2. DecisionTree←null
3.     while JSONFile. readline()≠null do
4.         if constraintType≠null do
5.             DecisionTree←ChildDecisionTree(constraint Type,σ)
```

```
6.      end if
7.    end while
8. return DecisionTree
/ * 创建子决策树 * /
9. def ChildDecisionTree(param←String,σ←float)
10. if D 中样本全属于同一类别 C then
11.       将 node 标记为 C 类叶节点；return
12. end if
13. if A=Ø or D 中样本在 A 上取值相同 then
14.       将 node 标记为叶节点，其类别标记为 D 中样本数最多的类；
          return
15. childTree←null
16. ParentNode←Node(param)
17. ConstraintInfo←curInfo(param)
18. for ConstraintInfo do
19.       //选取信息熵减少最多的规约信息作为根节点
20.       parentNode←Max(Gain(D,A))
21.         if currentNode.frequency > σthen
22.           childLeftTree←createNode(Y)
23.         else
24.         childRightTree←createNode(getType(param))
25.         end if
26. end for
27. return childTree
```

对于目标 API 的 5 种规约信息，如果存在，则相应地对其创建子决策树（第 1－7 行）。在构建子决策树的过程中，首先判断当前数据是否需要分类（第 10－12 行），以及决策树生成到当前节点时节点是否需要划分（第 13－14 行）。然后从 A 中选择最优划分属性，初始化子树（第 15 行），使用当前访问的规约类型作为子树的根节点（第 16 行）。根据当前遍历的规约类型来截取文件对应规约内容（第 17 行），然后遍历规约信息，每次选取信息熵减少最多的规约信息作为根节点来生成子树。接着设定阈值 σ 来判断当前规约信息的正确性，若频繁阈值大于设定的阈值，则代表该用法正确，相应地创建左子树（第 21－22 行）；若频繁阈值小于规定的阈值，代表误用情况，根据当前访问的规约类型创建对应的误用叶子节点（第 24 行）。

3.3.3 决策树的剪枝策略

为了提高 AUDT 的检测性能，本文融入了剪枝策略[19-20]，防止过拟合造成 API 使用决策树的泛化能力较差。

预剪枝在生成决策树的同时进行剪枝，只有在信息增益大于预先规定的阈值 g 时才进行分类，若信息增益达不到规定的阈值 g，则不会产生新的分支。

后剪枝在决策树生成的过程中不考虑结构复杂度，而是在决策树构造完成后进行剪枝以减小复杂度，从而提高模型的预测能力。其损失函数的定义如下：

$$C_\alpha(T)=\sum_{t=1}^{|T|}N_tH_t(T)+\alpha|T| \tag{4}$$

其中，经验熵为：

$$H_t(T)=-\sum_{k=1}^{K}\frac{N_tk}{Nt}\log\frac{N_tk}{Nt} \tag{5}$$

其中，树 T 的叶子节点的个数为 $|T|$，t 是树 T 的叶子节点，该节点有 N_t 个规约特征，其中 k 类规约特征有 N_{tk} 个，$H_t(T)$ 为叶子节点上的熵，α 为超参数。式(4)右边第一项为每一个叶子节点内的规约特征总数关于其类别分布的熵乘以该节点样本数，然后再对全部叶子节点的该值求和。该式子可以表示规约树模型对规约信息数据的误差，即拟合度。式(4)右边第二项为超参数 α 与树的叶子总数量之和，可以表示对树结构复杂度的惩罚项，其中 α 可以控制拟合度和惩罚项两者之间的影响，较大的 α 促使生产更简单的树，较小的 α 促使生产较为复杂的树，$\alpha=0$ 则意味着不考虑模型复杂度，只考虑模型对训练数据的拟合程度。对决策树进行剪枝，就是当 α 确定时，选择损失函数最小的模型，即损失函数最小的子树。

决策树剪枝的具体步骤如算法 3 所示。

算法 3 决策树的剪枝算法

```
输入：生成的决策树，正则化参数 α
输出：损失函数最小的决策树
1. for each leaf node do
2.    // 整棵树损失函数 α
3.   α←Cα(T)
4.   T1←ExchangeLeafToParent(T)
5.   b←Cα(T1)
6.      if b<a then
7.          / * 减掉父节点 * /
8.          parentNode←leafNode
9.          T←update
10.     end if
11. end for
12. return T
```

在算法 3 中，对于一个叶子节点而言，计算当前状态下整棵树的损失函数 a 以及代替其父节点后（即剪枝后的决策树）的整棵树的损失函数 b。如果 b 比 a 小，则进行剪枝操作，即剪掉父节点，自己代替父节点生成新的叶子节点。在新位置上重复上述步骤，直到所有叶子都计算过。最终得到一棵损失函数最小的决策树 T。

3.4 误用检测阶段

在 API 误用检测阶段，利用 3.3 节和 3.4 节构造并进行剪枝过的 AUDT 对目标代码进行 API 误用检测。首先，利用算法 1 从目标源代码中提取规约文本信息持久化到本地文件中。

在 API 误用检测阶段，相关步骤如算法 4 所示。

算法 4 API 误用检测

```
输入：目标 API 规约信息的 JSON 文件，AUDT
输出：误用报告
1. / * 标记阶段 * /
2. vectorTapi←null
3. vectorDT←null
4. for each constraint type information：Type do
   / * 标记提取的文本信息 * /
5.      vectorTapi←(information_type==null)? 0：1
6. end for
7. for five types of decision subtrees：sub do
       / * 标记决策树中五类规约类型 * /
```

```
8.    vectorDT←(constraintSubTree==null)? 0:1
9. end for
10. /* 粗粒度检测 */
11. result←vectorT_api&&vectorDT
12. if result ∈ExceptionVec then
13.       reportMisuse←checkMiss
14. else /* 细粒度检测 */
15.       reportMisuse←DecisionTreeCheck(T_api)
16. end if
17. return misuseReport
```

首先对提取的文本信息进行标记(第4—6行)。对于这5类目标规约信息,如果规约信息不为null,则标记为1;如果为空,则标记为0。通过标记得到关于目标API自身的这5类规约信息的向量。对决策树中5类规约类型进行向量标记(第7—9行),如果规约类型决策子树存在,则将该规约类型标记为1,相反标记为0。用向量的与运算进行粗粒度检测(第11—13行),若二者向量运算结果的相似度满足预期值,说明待检测文件中API在规约类型上符合决策树的规约类型,通过决策子树细粒度检测目标代码中这些规约信息是否使用正确,最后返回误用检测报告。

4 实验评估

为了充分挖掘目标API的使用规约,首先需要给定大量包含目标API的使用示例,利用GitHub上有关目标API的大量使用示例。本节使用AUDT对API误用的相关代码进行缺陷检测实验。

4.1 实验对象

实验选取了一些常见的误用,包括相关文献中Stack Overflow[2]真实误用数据、数据库的连接操作的真实误用数据[8]、以及其他论文中用到的误用数据[21-22]的部分数据。这些误用数据包括有关Java Cryptography APIs的误用代码片段、与Java数据库连接的误用代码片段、文件流读取的误用片段,以及迭代器使用相关的误用代码片段。这些测试用例中多数包含多种类型的API误用,且它们之间不互斥。使用的项目数据集主要来自最近的API研究工作[23],选取了其中可用的Java项目以及其他工作中的项目作为补充,这些项目提供了丰富的API使用资源,是挖掘API使用规约的重要数据来源。

4.2 研究问题

这一节通过以下研究问题来说明本文方法的有效性。

(1)RQ1:AUDT在不同类型的API误用上的检测效果如何?

(2)RQ2:本文方法在检测包含多种误用类型的情况下表现如何?

(3)RQ3:剪枝策略对API误用检测有什么影响?

4.3 实验分析

这一节通过具体的实验结果来分析上述提到的3个研究问题。

4.3.1 RQ1:检测到的API误用类型的分布

第一个研究问题主要讨论使用API决策树的方式,对于这5种API误用类型的检查结果如何,这有助于分析本文工具的优势和不足。

AUDT检测不同类型的API误用的数量如图5所示。ICC类型的API误用总数为15个,其中AUDT找到了真实的API误用为13个,召回率达到了86.7%。IEH类型的API误用总数为8个,其中AUDT找到了真实的API误用为7个,召回率达到了87.5%。IPRC类型的API误用总数为7个,其中AUDT找到了真实误用为5个,召回率达到了71.4%。IPOC类型的API误用总数为6个,其中AUDT找到了真实误用为5个,召回率达到了83.3%。IPU类型的API误用总数为5个,其中AUDT找到了真实误用3个,其召回率仅为60%。通过分析发现,对于参数值使用不当这种误用,由于上下文语义不充分,检测这类误用比较困难,仅依靠已有代码示例还是不能充分挖掘到符合目标的使用方式。

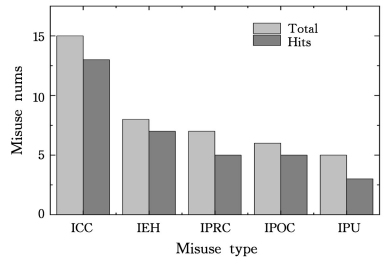

图5 不同类型API误用总数和检测到的误用数量

Fig. 5 Total number of misuses and number of detected misuses for different types of APIs

4.3.2 RQ2:检测多类型误用

第二个研究问题主要探讨了不同方法之间在检测包含多种误用类型的实例时的有效性。前面提到,目前存在的检测器在检测API误用的过程中,有时会出现对程序中的API误用存在检查不全面的情况。这种漏报不利于开发人员全面排查存在的API误用。针对这种情况,本文评估了AUDT在包含多种误用类型的测试用例中的表现。

本文手动过滤只包含一种误用类型的测试用例,通过交叉组合得到包含5种误用类型的测试用例3个、4种误用类型的测试用例共6个、3种误用类型的测试用例8个、2种误用类型测试用例10个。包含多种误用测试用例的检测结果如下,对于包含2种误用类型的测试用例,AUDT检测出9个;对于包含3种误用类型的测试用例,AUDT检测出6个;对于包含4种误用类型的测试用例,AUDT检测出5个;对于包含5种误用类型的测试用例,AUDT检测出1个。通过分析发现,这种直接从造成API误用的根本原因出发,去挖掘目标API的使用规约能够涵盖多种误用类型的检测。

4.3.3 RQ3:是否融入剪枝策略对AUDT的影响

第三个研究问题分析了剪枝策略对AUDT的影响,以及不同剪枝策略之间的差异。表1的前两行显示了不同剪枝策略下AUDT的性能。其中,第一行为实验的最佳配置,采用后剪枝的策略;第二行表示预剪枝的策略;第三行表示不使用剪枝策略下AUDT的表现。

表1中的第三行表示在不使用剪枝策略的情况下,AUDT的精度为15.8%,比后剪枝(最优配置)低了2.1%,召回率方面比最优配置低了2.4%,总的来说F1值比最优配置

低了 3%。但在不剪枝的情况下，AUDT 比预剪枝情况下在 F1 方面还高出了 4%，由此可见预剪枝在精度方面并没有提高决策树的性能，反而在某些指标上有所下降。通过后剪枝，AUDT 整体性能（即 F1 值）提升了 3%。分析可知，不使用剪枝策略的 AUDT 容易出现过拟合现象，无论是对决策树的分类精度，还是对其规模以及可理解性，都产生了不利的影响，而预剪枝的行为可能会带来欠拟合的风险。这是因为：一方面，叶子节点随分割不断增多，极端情况下每个叶子节点中只包含一个实例，而这对于未见的样例是没有意义的；另一方面，决策树不断生长，树的深度越大对应的规则越长，后剪枝下的 AUDT 效果比预剪枝好。经过分析发现，预剪枝策略产生的高欠拟合问题对 AUDT 整体性能的影响较大，这也促使在后剪枝过程中采用自底向上的方式对非叶子节点进行考查，这样虽然消耗了更多的时间，但保留了更多的分支，在降低了欠拟合风险的同时解决了过拟合问题。

表 1 融入剪枝策略前后的实验结果

Table 1 Experimental results before and after incorporating pruning strategy

（单位：%）

Detector	Precision	Recall	F1
AUDT(post-pruning)	17.9	80.5	29.3
AUDT(pre-pruning)	13.6	82.3	22.3
AUDT(w/o pre-pruning and w/o post-pruning)	15.8	78.1	26.3

5 相关工作

本节回顾了 API 误用检测相关的工作并进行归类总结。

现有的检测技术有基于人工编写规约来检测 API 误用的方法，如 IMChecker[3] 和 Semmle[4] 等使用基于 YAML 的 DSL 来指定 API 函数的行为。Semmle[4] 可以通过使用 CodeQL[25]（一种基于 Datalog 的声明性逻辑编程语言）在程序的关系表示上指定的正确或错误行为语义模式来发现 API 的误用。类似地，MOPS[26] 使用基于有限自动机的规范，利用违规使用模型检查器进行检查。

然而，制定一个正式的 API 规约是困难的，需要有经验的开发人员花费大量的精力去编写[9]，因此也出现了一些利用数据挖掘得到频繁使用模式进而检测 API 误用的方法，代表性的有 DMMC[10]，PR-MINER[27] 和 TIKANGA[28] 等利用频繁的项集挖掘来识别至少支持预先规定好的阈值，偏离绝大多数的使用方式被认为很有可能存在缺陷。这种基于频率的方式没有利用对 API 的顺序、语义等信息，在实际表现中误报率较高。

为了利用 API 顺序以及语义信息，出现了使用统计语言模型、循环网络等方法从代码中推断 API 使用规约并用于 API 误用检测的技术。Zhong 等[29] 研究了自然语言的 API 使用规约提取技术，并研发了一个 API 使用规约提取工具 Doc2Spec，利用提取的规约检测目标代码中的 API 误用缺陷。Wang 等[24] 将深度学习中的循环神经网络模型应用于 API 使用规约的学习及 API 误用缺陷的检测，通过预测结果与实际代码进行比较来发现潜在的 API 误用缺陷。这些方法虽然利用上下文信息，在一定程度上利用了语义信息，但只是针对单个位置的预测，检测能力稍有不足。

除了以上方法，还有利用一些特殊技术的检测方式。例如，Nielebock 等[21] 提出了基于修正规则的协同 API 误用检测技术，通过使用相应的方法重用已知的 API 误用修复，并使用误用的 API 自动推断一个修正规则。与之类似，Zeng 等[30] 根据已经发现的 API 误用检测实例，结合补丁文件中修复前后的代码刻画 API 误用模式，然后搜索符合误用模式的 API 调用序列并报告相似缺陷。通过变异分析的误用检测技术，Wen 等[31] 利用突变分析检测新发布的 API 的误用模式。基于修正规则的方法严重依赖已有的修复信息，基于突变技术的误用检测严重依赖于突变因子的设计，这些方法在检测的全面性以及对于其他类型检测的推广上性能不足。

结束语 API 误用检测的核心任务是获取 API 的正确使用规约以及优秀的检测算法。API 使用规约描述了调用软件或类库的 API 时应该满足的规则，在软件开发的多个环节都扮演着重要的角色。无法获取完整使用规约的情况下，需要从源代码中挖掘 API 的使用规约，利用相关技术分析判别 API 的使用正确与否。

本文针对性地挖掘有关 API 使用规约，使用规约中的规约类型涵盖了目标 API 的详细使用，违反了这些规约类型即代表存在 API 误用。本文在挖掘到的 API 使用规约的基础上，构造 AUDT，进而检测目标 API 的使用是否存在误用。但是受限于误用数据集中的部分用例，该模型的泛化能力还具有不确定性，并且检测精度有待提高。

未来计划将 AUDT 应用在不常用的 API 上，以研究 AUDT 是否可以检测更多的 API 误用，并尝试使用其他算法来构建 AUDT。

参 考 文 献

[1] LEGUNSEN O，HASSAN W U，XU X，et al. How good are the specs? a study of the bug-finding effectiveness of existing java api specifications[C]// 2016 31st IEEE/ACM International Conference on Automated Software Engineering(ASE). 2016：602-613.

[2] ZHANG T，UPADHYAYA G，REINHARDT A，et al. Are code examples on an online q&a forum reliable?：a study of api misuse on stack overflow[C]// 2018 IEEE/ACM 40th International Conference on Software Engineering(ICSE). 2018：886-896.

[3] GU Z，WU J，LI C，et al. Vetting api usages in c programs with imchecker[C]// 2019 IEEE/ACM 41st International Conference on Software Engineering：Companion Proceedings(ICSE-Companion). 2019：91-94.

[4] GITHUB. Semmle-a code analysis platform for finding zero-days and automating variant analysis[Z]. 2017.

[5] YUN I，MIN C，SI X，et al. Apisan：Sanitizing API usages through semantic cross-checking[C]// 25th USENIX Security Symposium(USENIX Security 16). Austin，TX：USENIX Association，2016：363-378.

[6] WASYLKOWSKI A，ZELLER A，LINDIG C. Detecting object usage anomalies[C]// Proceedings of the 6thJoint Meeting of the European Software Engineering Conference and the ACM

SIGSOFT Symposium on the Foundations of Software Engineering. 2007:35-44.
[7] GU Z, WU J, LIU J, et al. An empirical study onapi-misuse bugs in open-source c programs[C]// 2019 IEEE 43rd Annual Computer Software and Applications Conference(COMPSAC). 2019:11-20.
[8] AMANN S, NADI S, NGUYEN H A, et al. Mubench: A benchmark for api-misuse detectors[C]// 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories(MSR). 2016:464-467.
[9] LI Z, MACHIRY A, CHEN B, et al. Arbitrar: User-guided api misuse detection[C]// 2021 IEEE Symposium on Security and Privacy(SP). 2021:1400-1415.
[10] MONPERRUS M, MEZINI M. Detecting missing method calls as violations of the majorityrule[J]. ACM Transactions on Software Engineering and Methodology, 2013, 22(1):1-25.
[11] NGUYEN T T, VU P M, NGUYEN T T. Api misuse correction: A fuzzy logic approach[C]// Proceedings of the 2020 ACM Southeast Conference. 2020:288-291.
[12] AMANN S, NGUYEN H A, NADI S, et al. A systematic evaluation of staticapi-misuse detectors[J]. IEEE Transactions on Software Engineering, 2019, 45(12):1170-1188.
[13] DEKEL U, HERBSLEB J D. Improvingapi documentation usability with knowledge pushing[C]// 2009 IEEE 31st International Conference on Software Engineering. 2009:320-330.
[14] WEN M, CHEN J, WU R, et al. Context-aware patch generation for better automated program repair[C]// 2018 IEEE/ACM 40th International Conference on Software Engineering(ICSE). 2018:1-11.
[15] GEORGIEV M, IYENGAR S, JANA S, et al. The most dangerous code in the world: validatingssl certificates in non-browser software[C]// Proceedings of the 2012 ACM Conference on Computer and Communications Security. 2012:38-49.
[16] SVEN A, NGUYEN H A, NADI S, et al. Investigating next steps in staticapi-misuse detection[C]// 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). 2019:265-275.
[17] JIN C, LUO D L, MU F X. An improved id3 decision tree algorithm[C]// 2009 4th International Conference on Computer Science Education. 2009:127-130.
[18] HSSINA B, MERBOUHA A, EZZIKOURI H, et al. A comparative study of decision tree id3 and c4.5[J]. International Journal of Advanced Computer Science and Applications, 2014, 4(2):13-19.
[19] ESPOSITO F, MALERBA D, SEMERARO G, et al. A comparative analysis of methods for pruning decision trees[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1997, 19(5):476-491.
[20] RASTOGI R, SHIM K. Public: A decision tree classifier that integrates building and pruning[J]. Data Mining and Knowledge Discovery, 2000, 4(4):315-344.
[21] NIELEBOCK S, HEUMÜLLER R, KRÜGER J, et al. Cooperative api misuse detection using correction rules[C]// 2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results(ICSE-NIER). 2020:73-76.
[22] WICKERT A K, REIF M, EICHBERG M, et al. A dataset of parametric cryptographic misuses[C]// 2019 IEEE/ACM 16th International Conference on Mining Software Repositories(MSR). 2019:96-100.
[23] LAMOTHE M, LI H, SHANG W. Assisting example-based api misuse detection via complementary artificial examples[J]. IEEE Transactions on Software Engineering, 2021.
[24] WANG X, CHEN C, ZHAO Y F, et al. API Misuse Bug Detection Based on Deep Learning[J]. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5):1342-1358.
[25] AVGUSTINOV P, DE MOOR O, JONES M P, et al. QL: Object-oriented queries on relational ! data[C]// 30th European Conference on Object-Oriented Programming(ECOOP 2016). 2016.
[26] CHEN H, WAGNER D. Mops: an infrastructure for examining security properties of software[C]// Proceedings of the 9th ACM Conference on Computer and Communications Security. 2002:235-244.
[27] LI Z, ZHOU Y. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code [J]. ACM SIGSOFT Software Engineering Notes, 2005, 30(5):306-315.
[28] WASYLKOWSKI A, ZELLER A. Mining temporal specifications from object usage[C]// 2009 IEEE/ACM International Conference on Automated Software Engineering. 2009:295-306.
[29] ZHONG H, ZHANG L, XIE T, et al. Inferring resource specifications from natural languageapi documentation[C]// 2009 IEEE/ACM International Conference on Automated Software Engineering. 2009:307-318.
[30] ZENG J, BEN K, ZHANG X, et al. API misuse bug detection based on sequence pattern matching[J]. Huazhong Keji Daxue Xuebao(Ziran Kexue Ban)/Journal of Huazhong University of Science and Technology(Natural Science Edition), 2021, 49(2):108-114, 132.
[31] WEN M, LIU Y, WU R, et al. Exposing library api misuses via mutation analysis[C]// 2019 IEEE/ACM 41st International Conference on Software Engineering(ICSE). 2019:866-877.

LI Kang-le, born in 1997, postgraduate, is a student member of China Computer Federation. His main research interests include intelligent software engineering and data mining.

REN Zhi-lei, born in 1984, Ph.D, associate professor, is a member of China Computer Federation. His main research interests include evolutionary computation, automatic algorithm configuration, and mining software repositories.

(责任编辑:柯颖)