软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(6):2432−2452 [doi: 10.13328/j.cnki.jos.007327] [CSTR: 32375.14.jos.007327]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Java 程序资源泄露缺陷检测: 传统模型和语言模型的有效性分析*

刘天阳[1], 叶嘉威[1], 计卫星[2], 刘 辉[1]

[1](北京理工大学 计算机学院, 北京 100081)

[2](北京师范大学 人工智能学院, 北京 100875)

通信作者: 计卫星, E-mail: jwx@bnu.edu.cn

摘 要: 资源泄露是由于有限的系统资源未能及时正确关闭所导致的缺陷, 广泛存在于各种语言程序中, 且具有一定的隐蔽性. 传统的缺陷检测方法通常基于规则和启发式搜索预测软件中的资源泄露. 近年来, 基于深度学习的缺陷检测方法通过不同的代码表征形式并使用循环神经网络、图神经网络等技术捕获代码中的语义信息. 最近的研究显示, 语言模型在代码理解和生成等任务中表现出色. 然而语言模型针对资源泄露检测这一特定任务上的优势和局限性尚未得到充分评估. 研究基于传统模型、小模型和大模型的检测方法在资源泄露检测任务中的有效性, 并探究小样本学习、微调以及静态分析与大模型结合的多种改进方式. 具体而言, 以 JLeaks 和 DroidLeaks 数据集为实验对象, 从资源泄露根本原因、资源种类、代码复杂度等多个维度分析不同模型的表现. 实验结果表明, 微调技术能够显著提升大模型在资源泄露检测领域的检测效果. 然而, 大部分模型在识别第三方库引发的资源泄露上仍需改进. 此外, 代码复杂度对基于传统模型的检测方法的影响更大.

关键词: 资源泄露; 缺陷检测; 实证研究; 大模型

中图法分类号: TP311

中文引用格式: 刘天阳, 叶嘉威, 计卫星, 刘辉. Java程序资源泄露缺陷检测: 传统模型和语言模型的有效性分析. 软件学报, 2025, 36(6): 2432–2452. http://www.jos.org.cn/1000-9825/7327.htm

英文引用格式: Liu TY, Ye JW, Ji WX, Liu H. Detection of Resource Leaks in Java Programs: Effectiveness Analysis of Traditional Models and Language Models. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2432–2452 (in Chinese). http://www.jos.org.cn/1000-9825/7327.htm

Detection of Resource Leaks in Java Programs: Effectiveness Analysis of Traditional Models and Language Models

LIU Tian-Yang[1], YE Jia-Wei[1], JI Wei-Xing[2], LIU Hui[1]

[1](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

[2](School of Artificial Intelligence, Beijing Normal University, Beijing 100875, China)

Abstract: Resource leaks, which are defects caused by the failure to timely and properly close the limited system resources, are widely present in programs of various languages and possess a certain degree of concealment. The traditional defect detection methods usually predict the resource leaks in software based on rules and heuristic search. In recent years, defect detection methods based on deep learning have captured the semantic information in the code through different code representation forms and by using techniques such as recurrent neural networks and graph neural networks. Recent studies show that language models have performed outstandingly in tasks such as code understanding and generation. However, the advantages and limitations of large language models (LLMs) in the specific task of resource leak detection have not been fully evaluated. The effectiveness of the detection methods based on traditional models, small models, and LLMs in the task of resource leak detection is studied, and various improvement methods such as few-shot learning, fine-tuning and the

* 基金项目: 国家自然科学基金重点项目 (62232003)
本文由“大模型下的软件质量保障”专题特约编辑王赞教授、王莹副教授、陈碧欢副教授、姚远副教授、张敏灵教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-03-06

combination of static analysis and LLMs are explored. Specifically, taking the JLeaks and DroidLeaks datasets as the experimental objects, the performance of different models is analyzed from multiple dimensions such as the root causes of resource leaks, resource types and code complexity. The experimental results show that the fine-tuning technique can significantly improve the detection effect of LLMs in the field of resource leak detection. However, most models still need to be improved in identifying the resource leaks caused by third-party libraries. In addition, the code complexity has a greater influence on the detection methods based on traditional models for resource leak detection.

Key words: resource leak; defect detection; empirical study; large language model (LLM)

资源泄露是由于有限的系统资源未能被及时或正确关闭而引发的软件缺陷[1−3]. 尽管 Java 语言提供了自动内存管理机制, 但程序员仍需要在代码中显式关闭系统中的有限资源 (如文件句柄、套接字和线程等), 而不能依赖垃圾回收器进行管理[1]. 与其他类型的代码缺陷相比, 资源泄露通常具有更高的隐蔽性. 一般情况下, 当资源泄露大量累积并导致资源耗尽时可能会引发系统崩溃. 因此, 通过常规测试通常难以发现资源泄露[4].

资源泄露在应用程序中普遍存在[1,5], 被认为是最具有挑战性的问题之一[3,6]. 目前, 静态分析方法仍然是识别资源泄露的主流手段[4,7,8]. 例如, 基于规则和启发式搜索方式构建的工具 PMD[9]. 近年来, 基于深度学习的技术逐渐成为缺陷检测研究的热点领域[10−15]. 研究人员将代码表示为 Token 序列、抽象语法树 (abstract syntax tree, AST) 和代码属性图 (code property graph, CPG) 后, 将其转化为向量表示, 并进一步利用循环神经网络 (recurrent neural network, RNN)、长短时记忆网络 (long short-term memory, LSTM) 和门控循环单元 (gated recurrent unit, GRU) 等模型来学习代码中的语义信息, 从而预测代码缺陷[16]. 这些传统的基于深度学习的缺陷检测方法, 其检测效果往往受到训练数据集规模的限制并可能存在梯度消失的问题. 大规模预训练语言模型在自然语言处理领域取得的成就启发了研究人员探索新的缺陷检测方法[17−20]. 一方面, 研究人员通过结合语言模型和静态分析提升了缺陷检测器的检测效果. 例如, Li 等人[21]结合大模型和静态分析技术检测 Linux 内核中的变量未初始化的问题. Sun 等人[22]使用大模型匹配预先定义的场景和漏洞类型从而识别智能合约中的逻辑错误. 目前, 已有研究尝试利用大语言模型的代码理解能力获取资源释放操作对等信息, 并结合轻量级可达性分析判别代码是否存在资源泄露[5]. 另一方面, 研究人员对大模型的缺陷检测能力进行了综合评估. 例如, Yu 等人[23]从定性和定量两个角度探究了模型在代码安全审计中的应用, LLM4Vulm 框架[24]探究了不同模块对缺陷推理能力的影响. 语言模型分为大语言模型 (简称大模型或 LLM) 和小语言模型 (简称小模型或 SLM), 大语言模型擅长处理复杂任务, 而小语言模型则能更加高效的训练和运行 (下文中将小语言模型和大语言模型统称为语言模型). 与大模型相比, 小模型的参数量一般不超过 10 亿个, 如 Phi3、CodeT5 等. 然而, 尽管基于传统模型的方法、小模型和以 GPT-4 为代表的大模型在与代码相关的任务中展示出了极大潜力, 目前仍缺乏系统深入的研究来全面评估其在资源泄露检测任务中的表现.

本文旨在评估基于传统模型、小模型与大模型的缺陷检测方法在资源泄露检测任务中的有效性, 并探讨检测过程中的薄弱环节, 为今后资源泄露检测工具的设计与评估提供新思路. 本研究系统地比较了多种传统模型与语言模型在资源泄露检测中的表现. 具体来说, 本文评估了 4 种传统模型方法以及 7 种具有代表性的语言模型, 如以 GPT-3.5、GPT-4 为代表的大模型和以 Phi3、CodeT5 等为代表的小模型. 此外, 本文还考察了结合大模型与静态分析的资源泄露检测方法 INFERROI[5], 以及通过词法、语法和语义相似度设置提示词样例的方法 GRACE[19]. 本文还对微调后的 GPT-3.5 模型进行了深入评估. 所有实验均在 JLeaks[2]和 DroidLeaks[25]数据集上进行.

本文通过研究以下问题, 深入探讨了传统模型、小模型和大模型在资源泄露检测中的应用效果.

RQ1: 不同模型在资源泄露任务上的整体能力如何?

RQ2: 不同模型在检测由不同原因导致的资源泄露时, 是否存在差异?

RQ3: 不同模型在识别不同资源类型的资源泄露时, 是否存在差异?

RQ4: 不同模型在处理不同复杂度的代码时, 是否存在差异?

实验结果表明, 微调后的 GPT-3.5 在资源泄露检测任务中显著提升了检测效果. 此外, 本文还观察到以下几个主要结果: 首先, 不同资源泄露原因下的各个模型的效果差异不大. 此外, 静态分析与大模型结合的方式能够提高缺陷检测召回率. 其次, 进一步分析发现, 无论是语言模型还是传统模型, 在识别文件资源泄露、局部变量导致的资源泄露以及涉及标准库的资源泄露时均显示出更高的检测能力. 再者, 本文发现语言模型在处理复杂代码时表

现更好, 相对而言, 传统模型的缺陷检测能力随着代码行数和圈复杂度的增加而有所下降. 另外, 基于实验和评估结果, 本文提出了以下建议: 首先, 微调能够显著提升语言模型针对特定类型缺陷的检测效果, 构建特定类型高质量缺陷库在微调过程中至关重要. 其次, 大模型和传统分析结合初见成效, 未来应综合考虑如何提升大模型的能力以及静态分析的精度, 以优化资源泄露检测的整体效果. 此外, 尽管通过多模型结合可以达到较高的真阳率, 但仍然无法避免高假阳率, 因此, 如何在提高真阳率的同时控制假阳率值得进一步研究.

本文的主要贡献如下.

• 深入分析了基于传统模型、基于小模型和基于大模型的方法在资源泄露检测任务中的差异性, 首次系统地对 4 种传统模型与 7 种具有代表性的语言模型在资源泄露检测中的效果进行了比较.

• 对比了不同调整方式对语言模型在资源泄露检测中的应用效果, 对比了优化提示词、小样本学习、模型微调以及大模型与静态分析方法结合对资源泄露检测的影响.

• 挖掘了传统模型和语言模型针对不同缺陷原因、不同资源类型以及不同复杂度代码上的优势和薄弱环节, 为未来研究提供了参考.

本文第 1 节阐述背景及相关工作, 包括资源泄露的基本概念、基于传统模型的缺陷检测方法以及基于语言模型的缺陷检测方法. 第 2 节详细描述本文使用的资源泄露数据集及其处理方式, 并刻画数据集的特征. 第 3 节提出 4 个研究问题及其意义, 并对评估的模型进行详细介绍. 第 4 节通过实验, 对比基于传统模型与基于语言模型的资源泄露检测方法, 针对 4 个研究问题进行了分析和探讨. 第 5 节分析有效性威胁. 第 6 节提出对未来工作的建议. 最后, 第 7 节总结全文.

1 背景与相关工作

1.1 资源泄露基本概念

资源泄露是一种常见的且难以预测的软件缺陷, 可能导致系统崩溃或漏洞[4]. 在 Java 中, 虽然内存管理机制能够自动回收不再使用的对象, 但无法有效管理操作系统分配的有限资源 (如文件流、网络连接等). 如果这些资源未能及时显式释放, 就会引发资源泄露问题.

资源泄露的原因主要归结于未提供关闭方法 (notProClose)、在常规路径下缺失关闭方法 (noCloseRPath)、发生异常时无法关闭资源 (noCloseEPath) 以及存在缺失关闭方法的分支 (noCloseCPath)[2]. 从传播距离来看, 资源泄露可分为跨库资源泄露和库内资源泄露. 本文的跨库资源泄露, 特指某个类提供了资源获取接口, 但是未提供相应的资源关闭接口. 例如, 图 1(a) 展示了 apache/HBase 项目中的代码片段. 根据 GitHub 中的提交信息[26]和问题讨论[27], 该代码中资源泄露发生的原因在于无法关闭 htable 对象. 代码显示, this.scanner 和 this.htable 均为私有成员对象, 无法被其他类直接访问. 因此只能通过调用该类的 close 方法释放资源. 然而, 图 1(a) 的代码表明, 在 close 方法中仅调用了 this.scanner 的关闭方法, 却未对 this.htable 执行相同操作, 导致通过该类获取 this.htable 资源的其他类无法正确释放资源.

```
/**
https://github.com/apache/hbase/commit/e04009c9894b1ace20759c5f97f30126f3129aa3
*/
public class TableRecordReaderImpl {
  private ResultScanner scanner;
  private HTable htable;
  ...
  public void restart(byte[] firstRow) throws IOException{
    ...
    this.scanner = this.htable.getScanner(scan);
  }
  public void setHTable(HTable htable){
    ...
    this.htable = htable;
  }
  public void close(){
    this.scanner.close();}}
```

(a) HBase 项目中未提供资源关闭接口示例

```
/**https://github.com/bitcoinj/bitcoinj/commit/759156c71b4114fb537d065981d60573562481e9*/
private static void sanityCheck(File file, int expectedSize) throws IOException {  …
  CheckpointManager manager = new CheckpointManager(params, new FileInputStream(file));
  …}
```

(b) bitcoinj 项目中 noCloseRPath 示例

```
/**
https://github.com/apache/hadoop/commit/d7f712cd4262f51ea2972596ce0a48cde623ecf9
*/
public InputStream serialize() throws IOException {
  …
  DataOutputStream out = new DataOutputStream(bytes);
  out.writeByte(fileType.ordinal());
  …
  out.close();
}
```

(c) Hadoop 项目中 noCloseEPath 示例

```
/**
https://github.com/AnySoftKeyboard/AnySoftKeyboard/commit/cc8c1c8fa4e34baf1e2b23b0abb3897a331f2619
*/
private void addWords(Cursor cursor){
  …
  if(newCount == mContactsCount  && newHash == mContactsHash ){
    return;
  }
  …
  cursor.close();
}
```

(d) AnySoftKeyboard 项目中 noCloseCPath 示例

图 1 资源泄露原因示例

其次, 针对库内资源泄露的情况, 可以根据不同的场景进一步细分. 如果资源获取后未调用资源释放方法, 则缺陷原因可归类为在常规路径下缺失关闭方法 (noCloseRPath); 如果资源获取后虽然存在释放方法的调用, 但在异常情况下无法执行该释放操作, 则归因于发生异常时无法关闭资源 (noCloseEPath); 如果资源获取后在分支语句 (如 if、switch) 中某个分支未调用资源释放方法, 则属于存在缺失关闭方法的分支 (noCloseCPath). 图 1(b) 展示了 bitcoinj/bitcoinj 项目中的代码片段, 其中 new FileInputStream(file) 为匿名变量获取了系统资源. 然而, 在 sanityCheck 方法中并未调用该变量的关闭方法, 致使系统资源无法关闭, 因此该代码片段的资源泄露问题归因于 noCloseRPath. 图 1(c) 展示了来自 apache/hadoop 项目的代码片段, 该代码定义了资源变量 out, 并在常规路径下调用了资源关闭方法. 然而, writeByte 方法可能抛出异常, 导致当异常发生时, out.close() 代码无法执行, 进而引发资源泄露. 图 1(d) 则展示了 noCloseCPath 导致的资源泄露. 当条件 newCount == mContactsCount && newHash == mContactsHash 成立时, 方法直接返回. 尽管在条件不成立的情况下资源可能会被关闭, 但是在该条件成立的分支上资源未被释放, 从而导致泄露.

1.2 基于传统模型的软件缺陷预测

传统模型在软件缺陷预测中的应用核心在于如何有效地表征代码. 目前, 代码表征的方法主要分为 4 类: 基于代码度量的表征、基于序列的表征 (又称基于 Token 的表征)、基于抽象语法树的表征, 以及基于图的表征. 代码度量 (如代码行数、圈复杂度、继承深度等) 是一组用于评估软件质量的指标[28]. Younis 等人[29]选择了源代码行数、圈复杂度和路径个数等 8 项度量值识别可利用的漏洞. 然而, 尽管代码度量能够有效地表征代码的整体特性, 但在捕获代码细节方面却存在局限性[30]. 因此, 基于传统模型的软件缺陷预测方法更倾向于使用其他 3 种代码表征形式.

基于序列的表征方式将源代码视为自然语言进行词法分析. Li 等人[31]通过方法名称利用深度神经网络预测代码的脆弱性. 为了充分挖掘变量和方法名称中的有用信息, DeepBugs[11]基于语义表示对名称进行推理, 从而实现缺陷检测. 然而, 这种方式容易受到命名准确性和规范性的影响, 并且忽略了代码符号序列 (Token) 之间的关联以及代码的结构信息. Bugram[10]利用缺陷代码出现频率相对较低的规律, 采用 n-gram 模型来构建符号之间的关联, 并利用统计信息将项目中出现频率低的序列视为潜在的缺陷代码.

抽象语法树是源代码的抽象语法结构的树状表示. 基于语法树的表征方式通常借助于开源工具 (如 JavaParser[32]、ANTLR[33]等) 生成抽象语法树, 然后通过特定的遍历算法遍历并编码语法树上的每个节点. 然而, 由于语法树的规模可能较大, 直接编码整个语法树容易导致长期依赖等问题. 因此, 研究人员通常会对语法树进行剪枝或过滤. 例如, Anbiya 等人[34]采用宽度优先算法遍历语法树, 并使用 TF-IDF 算法对节点进行编码, 在此过程中会过滤掉不可用的节点或子树. ASTNN[35]在构建语法树后, 将其拆分为小语法树形成多路语法树, 并通过递归编码器捕获语句的词法和语法信息, 最终利用门控递归单元获取整个代码片段的向量表示. 基于抽象语法树的代码表征方式结合了代码的语法信息和结构信息, 相较于基于序列的表征方式更加全面.

基于图的代码表征方式能够更加详尽地呈现代码的复杂结构, 例如控制依赖和数据依赖关系等. Yamaguchi 等人[36]通过结合抽象语法树、控制流图和程序依赖图构造了代码属性图. Devign[14]则利用图神经网络对代码属性图进行编码, 从而实现了较好的缺陷检测效果. Wang 等人[37]通过人工定义的 8 种关系 (如数据依赖关系、控制依赖关系、跳转关系等), 将抽象语法树扩展为有向图, 并将该图的邻接矩阵输入到门控神经网络中, 实现代码缺陷检测. 为了去除图中的冗余信息, Cheng 等人[38]提出了 DeepWukong 方法, 该方法通过对系统 API 调用处进行代码切片并生成切片图, 然后使用 3 种图卷积网络来检测代码缺陷. 与其他代码表征方式相比, 基于图的表征通常能够捕捉更多代码语义信息.

1.3 基于语言模型的软件缺陷预测

语言模型通过大规模数据训练得到, 具有强大的学习能力, 采用注意力机制, 能够有效捕获输入序列中的长距离依赖, 解决了传统神经网络中梯度消失的问题[39]. 其中, 自注意机制允许模型调整输入序列中不同位置的重要性, 能够计算位置之间的相关性. 这使得语言模型在自然语言处理和代码理解等任务中表现出色.

在软件缺陷检测领域, 针对语言模型的研究主要分为两大方向: 基于语言模型的缺陷检测工具的设计和模型缺陷检测能力的综合评估[40]. 语言模型在程序理解、代码生成等方面展现了强大的能力[41,42]. 为了解决静态分析在处理复杂程序时误报率高的问题, 研究人员提出将静态分析与大模型相结合的方法. Li 等人[21]提出了 LLift 框架, 该框架结合了大模型和静态分析技术, 用于检测 Linux 内核中的变量未初始化的问题. LLift 框架引入了后约束指导路径分析的概念, 以减少漏洞探索路径, 从而提高检测效率, 并提升后续大模型的应用效果. Sun 等人[22]开发了 GPTScan 工具, 该工具将大模型和静态分析结合, 用于检测智能合约中的逻辑漏洞. GPTScan 首先通过可达性分析筛选出候选函数, 接着利用 GPT 模型匹配预先定义的场景和漏洞类型, 并进一步识别关键变量和语句, 为后续的静态分析提供输入.

为了全面了解语言模型在缺陷检测方面的能力, Yu 等人[23]设计了 5 种不同的提示词, 从定性和定量两个角度探究了大模型在代码安全审计中的应用. 研究表明, 大模型在代码安全审计中相较于最先进的静态分析工具表现更优, 尤其是 GPT-4 在识别代码中潜在的缺陷时表现出色. 然而, 研究也发现这些模型存在产生冗长且不符合要求的输出的问题. Khare 等人[43]进一步使用 5 个不同的安全基准测试集, 对包括 GPT-3.5、GPT-4 和 Code LLaMA 等模型在内的 5 种语言模型进行了评估, 发现语言模型在需要局部推理的简单漏洞检测方面表现更佳. LLM4Vuln[24]作为一个评估框架, 不仅解耦和增强了大模型对代码缺陷的推理能力, 还提供了一个缺陷知识数据库和不同的提示方案. Ding 等人[40]指出现有漏洞数据集存在数据质量差、标签准确率低和重复率高等问题, 并提出了 PrimeVul 数据集用于训练和评估大语言模型. 他们的研究表明, 现有测试基准严重高估了大模型的性能.

在现有的研究中, 语言模型在软件缺陷预测领域展示了强大的潜力. 研究者们主要从两个方面进行了探索: 一方面, 通过设计结合大模型和静态分析的方法, 提升了缺陷检测工具的检测效果, 如 LLift 框架[21]、GPTScan[22]等工具在各自领域的应用; 另一方面, 研究者对大模型的漏洞检测能力进行了综合评估, 展示了其在代码安全审计[23]、局部推理[43]和使用特定提示词时的优越性[40]. 尽管这些工作表明语言模型在代码理解和漏洞检测方面有明显优势, 仍存在一些研究空白.

(1) 缺少针对语言模型在资源泄露检测任务下的研究: 资源泄露问题在软件开发中是一个常见的安全隐患, 影响着软件的稳定性和安全性. 尽管语言模型在通用的缺陷检测任务中表现突出, 但是关于资源泄露检测研究相对较少. 这主要受限于特定缺陷的样本数量, 然而大规模高质量资源泄露数据集 JLeaks[2]的出现为研究资源泄露检测提供了强有力的数据支撑. 资源泄露检测的关键在于资源获取和释放操作的识别以及可达性分析[5]. 缺少针对性研究可能无法充分发挥语言模型在资源泄露方面的潜力.

(2) 缺乏对检测工具在缺陷原因、代码属性等多维度的评测: 当前针对基于语言模型的缺陷工具评估通常集中在单一维度上, 而对这些工具在不同缺陷原因、代码属性 (如复杂度、代码行数、隐蔽程度) 等多维度下的检测效果研究不足, 仅依赖单一的评估可能无法全面反映工具在是实际应用中的有效性. 深入的多维度评测可以帮助研究人员更好地理解缺陷检测工具的优缺点, 并提供有针对性的改进和建议.

为了填补上述研究中的空白, 本文选择了大规模资源泄露数据集来评估传统模型、小模型和大模型在资源泄露检测方面的能力. 从多个维度展开实证研究, 本文不仅评估了模型的整体检测效果, 比较了基于传统模型的缺陷检测方法、基于小模型的缺陷检测方法和基于大模型的缺陷检测方法的差异, 还深入探究了不同模型在多个维度上的优势和劣势. 这些研究结果为未来资源泄露缺陷检测研究提供了新的思路和指导.

2 资源泄露数据集处理与分析

本文选取了 JLeaks[2]和 DroidLeaks[25]两个 Java 资源泄露数据集作为基础数据集. DroidLeaks 主要聚焦于移动应用中的资源泄露问题, 从 32 个流行的移动应用中共挖掘出 298 个资源泄露实例. JLeaks 从 GitHub 上流行且维护良好的 Java 项目中收集了 1 150 个资源泄露实例 (截至 2024 年 10 月), 成为目前已知规模最大的资源泄露数据集. 两个数据集均详细标注了资源泄露的原因和涉及的资源类型等. 由于本文主要关注方法粒度的资源泄露检测, 对于因 notProClose 而引发的跨库资源泄露问题不在本文研究范围之内. 为确保数据集的平衡性, 本文将资源泄露

代码对应的修复方法视为无资源泄露的代码片段. 此外, 本文排除了 4 个无法被 javalang 解析的代码片段. 最终, 本文获取了 2 778 个代码片段, 其中有 1 390 个资源泄露实例, 包括 DroidLeaks 中的 264 个缺陷和 JLeaks 中的 1 126 个缺陷. 随后, 本文按照 7:1:2 的比例对数据集进行了划分, 最终如表 1 所示, 训练集中包含 1 944 个样本, 验证集中包含 278 个样本, 测试集中包含 556 个样本.

表 1　数据集划分及特征分布

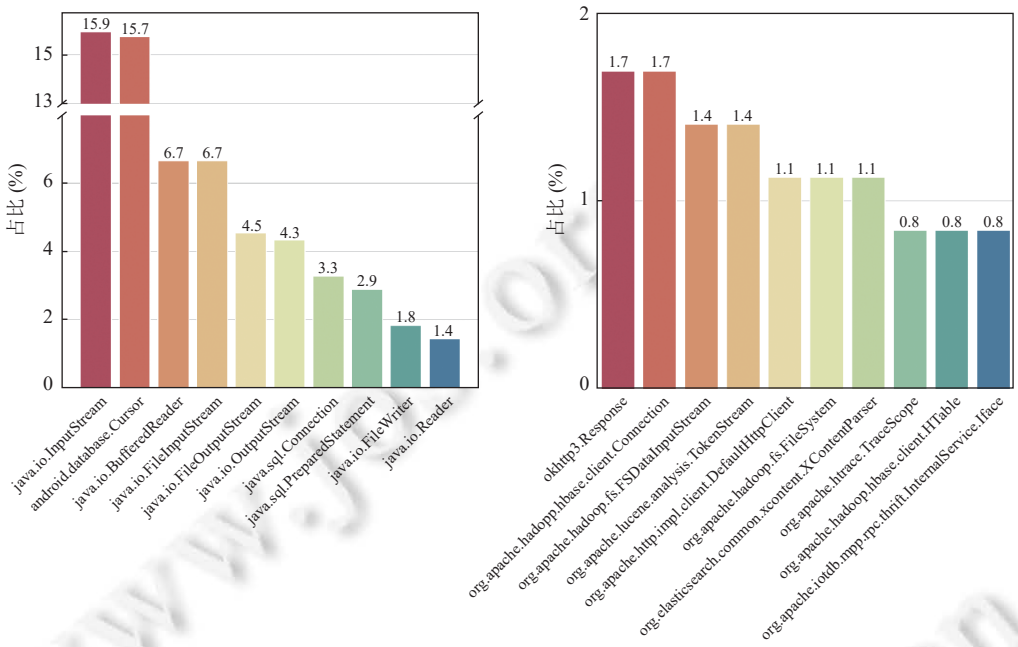
划分	全部		缺陷原因 (仅缺陷)			关键变量作用域/使用场景 (仅缺陷)				资源类别 (仅缺陷)		
	#缺陷	#非缺陷	#noCloseRPath	#noCloseEPath	#noCloseCPath	#全局变量	#参数	#局部变量	#匿名变量	#文件	#套接字	#线程
训练集	953	991	451	480	22	81	33	753	86	635	302	16
验证集	151	127	72	74	5	12	7	121	11	96	52	3
测试集	286	270	141	133	12	28	15	218	25	183	96	7
总计	1 390	1 388	664	687	39	121	55	1 092	122	914	450	26

在类型方面, 本文对 JLeaks 以及 DroidLeaks 数据集中的数据从资源数据类型、资源类别和资源变量作用域这 3 个方面进行了详细标注. 首先, 关于资源数据类型, JLeaks 明确标识了资源所涉及的第三方库或标准库, DroidLeaks 也标注了资源获取时所使用的库类型. 图 2 展示了基准数据集中涉及的前 10 个标准库和第三方库中类名的统计数据. 从图中可以看出, Stream 和 Cursor 相关的资源占比最高. 其中, java.io.InputStream 在所使用的标准库中占比约为 16%. 这是因为 java.io.InputStream 作为 java.io 库中用于文件读写的基础类, 被广泛应用于各种软件项目中. 需要说明的是, 在统计过程中, 对于样本中涉及多个资源的代码片段, 本文仅选取了其中的第 1 个资源类型作为代表进行研究和讨论. 其次, 本文对系统资源类别进行了更为细致和全面的分类, 涵盖了文件、套接字和线程等关键资源类别, 如表 1 所示, 基础数据集中包含文件资源、套接字资源和线程资源的数量分别为 914、450 和 26 个, 占比分别为 65.76%、32.37% 和 1.87%. 最后, 本文对数据集中关键变量作用域进行了分类. 所谓关键变量是指持有资源的变量. Java 中变量一般分为局部变量和全局变量. 全局变量包括类变量和实例变量. 局部变量中包含两类特殊变量: 参数变量和匿名变量. 参数变量是在方法签名中定义的, 通常由外部方法传入, 因其作用范围受限且具备输入特性. 匿名变量则是那些未显式定义的变量, 尽管使用方便, 但可能导致难以追踪和管理的隐患. 因此本文将参数和匿名变量从局部变量中抽离出来. 如表 1 所示, 在基础数据集中, 全局变量共 121 个, 占比 8.71%, 局部变量 (不包含参数和匿名变量) 共 1 092 个, 占比 78.56%, 参数共 55 个, 占比 3.96%, 匿名变量共 122 个, 占比 8.78%.

针对资源泄露原因的分类, DroidLeaks 和 JLeaks 的分类方式各有侧重但大体相似. 总体上, DroidLeaks 将其分为完全泄露、异常路径下泄露和特定常规路径下泄露. 完全泄露指的是程序员完全忘记释放系统资源; 异常路径下的泄露则是指资源在正常执行路径中能够被正确释放, 但在异常情况下却无法正常释放; 特定常规路径下的泄露是指在某些特定的常规执行路径中, 资源能够正常释放, 但在其他路径中资源却未能正确释放. JLeaks 的分类方式与 DroidLeaks 基本一致, 但略有差异. JLeaks 将资源泄露的原因划分为 4 类: 常规路径下未关闭资源 (对应 DroidLeaks 的完全泄露)、异常路径下未关闭资源 (对应 DroidLeaks 的异常路径下泄露)、分支路径下未关闭资源 (对应 DroidLeaks 的特定常规路径下泄露) 以及未提供关闭方法 (本文暂不考虑). 如表 1 中的统计结果, 在 1 390 个资源泄露实例中, 常规路径下未关闭资源的情况共 664 个, 占比 47.77%; 异常路径下未关闭资源的共 687 个, 占比 49.42%; 而在分支路径下未关闭资源的情况相对较少, 共 39 个, 占比 2.81%.

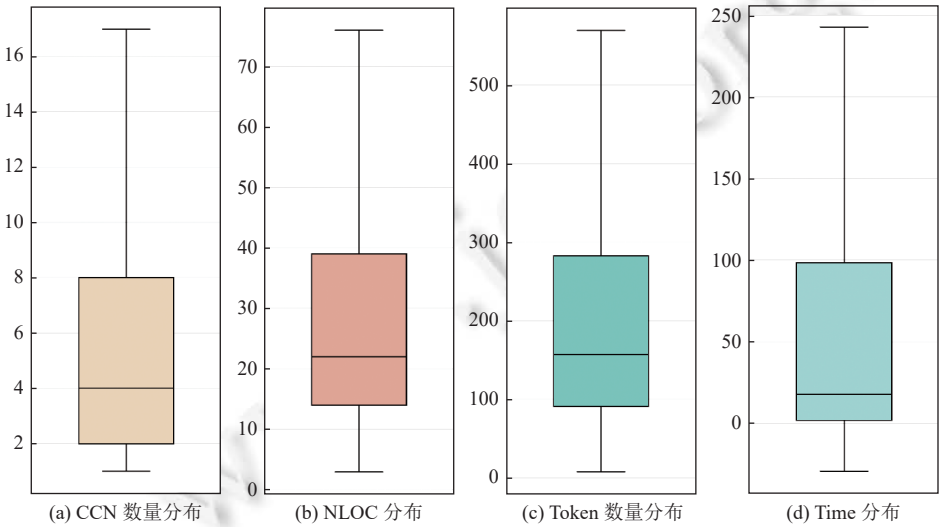
为了更全面地刻画资源泄露数据集的特征, 本文使用了 lizard 工具对 2 778 个资源泄露方法和非资源泄露方法进行了统计分析, 主要分析指标包括非注释行数 (lines of code without comments, NLOC)、圈复杂度 (cyclomatic complexity, CCN) 以及 Token 数量等. 这些指标不仅能够揭示代码的基本特性, 还能够为评估缺陷检测工具的效果提供支持. 圈复杂度是一种衡量代码控制流图中独立路径数量的重要指标, 反映了代码的逻辑复杂度. 较高的圈复杂度通常意味着代码中包含更多的条件判断、循环和分支结构, 这增加了代码的维护难度, 在一定程度上提高了产生缺陷的风险. Token 是指代码中的基本组成部分, 包括标识符、关键词、操作符等. Token 数量

的多少直接反映了代码的语法复杂度和逻辑结构. 通常情况下, Token 数量的增加意味着代码逻辑更加复杂, 从而可能导致开发者在编写和维护代码时更容易引入缺陷. 通过这些指标的分析, 本文试图进一步探索代码复杂度与资源泄露缺陷之间的关联性, 从而为改进检测工具的精度和效果提供理论支持. 图 3 展示了数据集中圈复杂度、非注释行数和 Token 数量的分布情况. 在本文使用的基准数据集中, CCN 的最高值为 209, 但在 80% 的情况下, CCN 低于 9. NLOC 的最大值超过 600 行, 其中 80% 的方法行数低于 46 行. Token 数量的最大值超过 6000 个, 但 80% 的方法 Token 数量低于 343 个.



(a) 数据集中 Top-10 标准库占比　　(b) 数据集中 Top-10 第三方库占比

图 2　频率 Top-10 标准库和 Top-10 第三方类统计



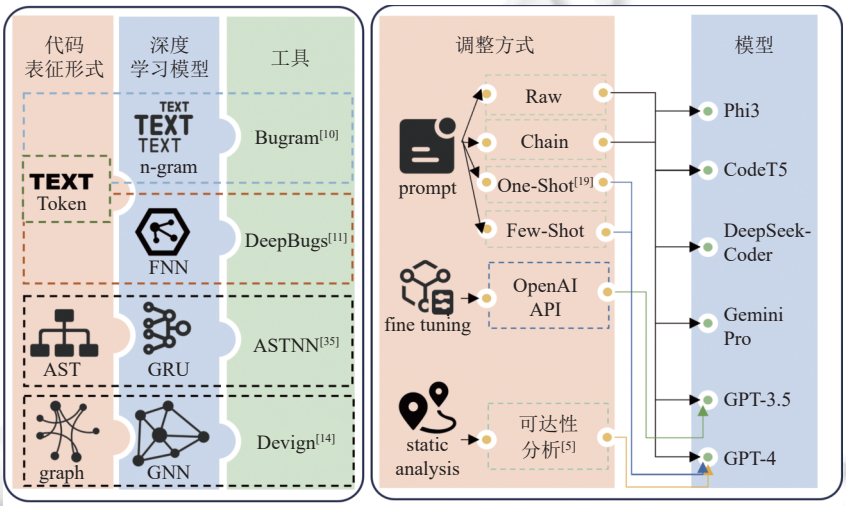
(a) CCN 数量分布　(b) NLOC 分布　(c) Token 数量分布　(d) Time 分布

图 3　CCN、NLOC、Token 和缺陷持续时间分布图

资源泄露通常具有较高的隐蔽性, 难以被及时发现和修复. 为了刻画数据集中资源泄露的隐蔽性, 本文对每个资源泄露的持续时间进行了统计分析. 具体而言, 本文定义资源泄露的持续时间 (又称缺陷存活时间) 为从缺陷引入系统到被修复之间的时间间隔. 然而, 准确地获取资源泄露的引入时间是一个具有挑战性的任务. 由于版本控制管理系统中缺乏直接标识资源泄露引入时间的记录, 本文采取了一种近似方法, 将缺陷引入时间视为缺陷文件在修复前的上一次修改时间. 如果该文件之前未被修改过, 则将缺陷引入时间视为项目的创建时间. 虽然这一近似方法并非完全精确, 但在实践中已被证明能够合理反映资源泄露的引入过程和时间趋势. 图 3(d) 展示了数据集中缺陷存活时间分布, 其中大多数缺陷存活时间均在 100 天以内.

3 实验设计

本文的总体测试框架如图 4 所示, 分别测试了多种传统模型和语言模型. 针对传统模型, 本文测试了 3 种流行的代码表征方式. 针对语言模型, 本文考虑了提示词工程、模型微调以及大模型与静态分析结合的调整方式.



(a) 基于深度学习的缺陷检测方法 (b) 基于语言模型的缺陷检测方法

图 4 测试框架设计图

在测试基于传统模型的缺陷检测方法时, 本文同时考虑了 3 种代码表征形式: Token、AST 和图. 首先, 本文将 Token 表征的代码输入到 n-gram 模型中, 复现了 Bugram 工具[10]. 其次, 将 Token 表征的代码输入到前馈神经网络中, 复现了 DeepBugs 工具[11]. 接着, 本文将代码表征为 AST, 并结合门控循环单元, 复现了 ASTNN[35]. 此外, 通过将代码表征为代码属性图, 并结合图神经网络, 复现了论文 Devign[14].

在针对大模型资源泄露检测能力的测试中, 本文选择了表 2 中列举的 7 种语言模型. 其中, GPT-3.5-Turbo-0125 (下文使用 GPT-3.5-Turbo 表示)、GPT-4-0125-preview (下文使用 GPT-4 表示)、Gemini Pro 和 DeepSeek-Coder 为大模型, Phi3、Gemma 和 CodeT5 为小模型. 为了深入探讨这些模型的资源泄露检测能力, 本文将模型的调整方式分为 3 类. 首先, 在提示词设计方面, 本文对 7 种语言模型均测试了零样本和链式询问两种提示词. 针对 GPT-4, 本文依据文献 [19] 中的思想额外测试了单样本提示词和多样本提示词. 图 5 展示了本文使用的不同的提示词, 其表述方式参考了 INFERROI[5]和 GRACE[19]. 在 One-Shot/Few-Shot 提示词中的 node information 以及 edge information 为 Joern 生成代码属性图. 其次, 在模型微调方面, 本文使用 OpenAI 的 Fine-tuning 接口来微调 GPT-3.5-Turbo, 以提升其在资源泄露检测这一特定任务的表现. 最后, 在静态分析和语言模型结合方面, 本文将大模型 GPT-4 与静态分析相结合, 以复现论文 INFERROI[5]中的相关工作. 需要注意的是, 实验结果中, 模型上标为 1 表示使用了图 5 中第 1 个提示词, 模型上标为 2 表示使用了图 5 中第 2 个提示词, GPT-3.5-Turbo^{FT1} 为微调 GPT-3.5-Turbo 且

使用第 1 个提示词. GRACE-oneshot 为使用与输入最相似的一段代码作为示例, 而 GRACE-twoshots 是使用与输入最相似的两段代码作为示例.

表 2 语言模型详细情况 (输入和输出价格均以美元/百万 Token 为单位, 截至 2024 年 8 月)

分类	模型 API	最多Token 数量 (k)	知识截止时间	输入价格	输出价格	temperature	frequency_penalty	presence_penalty	top_p
大模型	GPT-3.5-Turbo-0125	16	09/2021	0.5	1.5	0	0	0	1
	GPT-4-0125-preview	128	12/2023	10	30	0	0	0	1
	Gemini Pro	32	—	2	6	0	—	—	0.95
	DeepSeek-Coder	16	03/2023	0.14	0.28	0	0	0	1
小模型	Phi3	4	10/2023	—	—	0	—	—	1
	Gemma	8	—	—	—	0	—	—	1
	CodeT5	0.5	09/2021	—	—	0	—	—	1

Raw 提示词

Analyze the information about resource leaks in the provided code snippet bellow, and determine if the code snippet contains resource leaks.
Code Snippet:
{code}
Desired format:
'Contains resource leaks' or 'Does not contain resource leaks'
Response:

Chain 提示词

Please analyze the provided code snippet and determine if it contains any resource leaks by following the steps below:
1. Resource Identification: Identify all the resources being used in the code snippet.
2. Resource Management: Examine how each resource is being managed within the code snippet.
3. Potential Leaks: Identify any parts of the code where resources are not being properly managed or released.
4. Final Determination: Based on your analysis, determine if the code snippet contains any resource leaks.
Code Snippet:
{code}
Desired format:
'Contains resource leaks' or 'Does not contain resource leaks'
Response:

One-Shot/Few-Shot 提示词

Analyze the provided code snippet and determine if it contains any resource leaks.
Code Snippet:
{code}
The node information is as follows:
{node information}
The edge information is as follows:
{edge information}
Here is an example (are tow examples) for you to learn from:
It {contains resource leaks or does not contain resource leaks.}
Desired format:
'Contains resource leaks' or 'Does not contain resource leaks'

图 5 不同提示词样式

3.1 研究问题

为了全面比较基于传统模型的缺陷检测方法与基于语言模型的缺陷检测方法在资源泄露检测中的表现, 本文设定并探究了以下 4 个关键研究问题 (research question, RQ).

RQ1: 不同模型在资源泄露检测任务上的整体能力如何?

第 1 个研究问题旨在评估不同技术在资源泄露检测任务中的总体效果. 为此, 本文在由 JLeaks 和 DroidLeaks 共同构成的基础资源泄露数据集中, 对多种模型进行了对比分析, 并探究了各个模型检测出的缺陷的交叉情况.

RQ2: 不同模型在检测由不同原因导致的资源泄露时, 是否存在差异?

第 2 个研究问题旨在探讨不同模型在检测因不同原因导致的资源泄露时的表现差异. 为此, 本文统计了各个模型针对不同资源泄露原因的检测效果.

RQ3: 不同模型在识别不同资源类型的资源泄露时, 是否存在差异?

第 3 个研究问题探究了不同模型在不同资源类别、资源数据类型以及关键变量作用域引发的资源泄露上的检测能力. 为此, 本文在基础数据集上补充标注了资源类别、资源数据类型和关键变量作用域等, 并统计了各个模型在不同维度下的差异.

RQ4: 不同模型在处理不同复杂度的代码时, 是否存在差异?

第 4 个研究问题探究了不同模型在处理不同复杂度的代码时检测效果的差异. 为此, 本文对多个具有代表性的模型进行了拟合分析, 以研究不同代码指标 (如 NLOC、CCN 和缺陷存活时间) 与检测准确率或精确度之间的关系.

3.2 方法选择及实现

为了评估不同模型的资源泄露检测能力, 本文选择了以 Bugram[10]、DeepBugs[11]、ASTNN[35]和 Devign[14]为

代表的基于传统模型的缺陷检测方法, 以及语言模型 GPT-3.5-Turbo、GPT-4、Gemini Pro、DeepSeek-Coder、Phi3、Gemma 和 CodeT5. 表 2 详细列举了语言模型的相关信息及实验中的参数配置, 其中 Phi3、Gemma 以及 CodeT5 等小模型进行了本地部署. 此外, 本文探究了生成 One-Shot 和 Few-Shot 提示词的 GRACE 方法[19]以及将大模型与静态分析结合的 INFERROI[5].

针对基于传统模型的缺陷检测方法, 本文选择了基于 Token、基于抽象语法树和基于图的 3 种代码表征形式, 根据李韵等人[16]的研究, 这 3 种代码表征方式在软件缺陷检测中具有较好的性能. 针对语言模型, 本文考虑了当前最先进的模型 (如 GPT-3.5-Turbo、GPT-4 等) 以及一些新兴模型 (如 Gemini Pro、Phi3 等), 这些模型在多项自然语言处理任务中展示了较大潜力. 此外, 本文在选择模型和方法时还考虑了各个模型的可获得性和社区支持, 以便于重复实验. 下文将对各个模型和方法进行详细介绍.

• Bugram 模型. Bugrams 使用 n-gram 模型对代码中的 Token 序列进行建模, 以识别潜在的缺陷序列[10]. 具体而言, Bugram 利用 n-gram 模型计算代码中不同 Token 序列的概率, 低概率的序列被标记为可能存在缺陷的代码. 根据原文中的经验, 本文选择了窗口大小为 3 的 n-gram 模型, 并将代码划分为长度从 2 到 10 的序列进行建模. 此外, 为了保证模型的有效性, Token 的最小出现次数被设置为 3. 在模型评估过程中, Bugram 首先使用 Eclipse JDT Core 构建抽象语法树并解析 Token 类型, 其次构建了 9 个不同长度 (从 2 到 10) 的序列模型, 如果某个序列在两个或更多的序列模型中的预测概率位于 Bottom-50 中, 则包含该序列的方法被视为潜在的缺陷方法. 需要注意的是, Bugram 的训练集仅为无缺陷方法, 因此实验中使用原本划分的训练集的 991 个无缺陷方法作为训练集.

• DeepBugs 模型. DeepBugs 是基于命名的缺陷检测器, 专注于通过分析标识符名称的语义表示来预测与命名相关的缺陷[11]. 通常情况下, 资源变量的命名、资源获取和释放 API 的命名均遵循一定规律. 例如, 与流相关的资源类型名称通常包含 Stream, 如 java.io.OutputStream、java.io.PipedOutputStream; 与 SQL 相关的资源类型名称通常包含 SQL, 如 java.sql.Connection. 本文对基础数据集的分析发现资源泄露实例中 stream、connection 等词汇的词频较高. 实验中, 本文使用 Eclipse JDT Core 构建抽象语法树进而获取带有推理标识的代码 Token 序列, 使用 CBOW 编码 Token 序列, 编码过程将窗口大小设置为 20, 输出向量维度设置为 200, 最后使用前馈神经网络训练缺陷预测模型. 使用的训练集、验证集和测试集的情况如表 1 所示.

• ASTNN 模型. ASTNN 旨在深入学习源代码语法和语义信息, 具有较高的通用性, 可用于源代码分类、代码克隆检测等多种程序理解相关任务[35]. 该方法将大型抽象语法树拆分为一系列的小语法树来捕获语法和语义信息. 这一策略不仅有效解决了梯度消失问题, 还克服了将语法树视为完全二叉树时对源代码原始语法结构造成破坏的问题. 进一步, ASTNN 利用递归编码器捕获语句的自然性, 并通过门控循环单元获取整个代码片段的向量表示. 本文实验借鉴了该方法在代码功能进行分类的任务中的应用, 将代码类别设置为有缺陷和无缺陷两类, 其余配置均采用 ASTNN 的默认设置. 实验中, 训练集、验证集和测试集的详细情况如表 1 所示.

• Devign 模型. Devign 基于代码属性图和图神经网络进行代码缺陷检测[14]. 代码属性图是一种综合了所有语法和语义的代码表征形式, 包括抽象语法树、控制流图、数据流图和自然代码序列等. 这种复合图结构能够全面捕获代码中的复杂模式和关系, 为缺陷检测提供了丰富的上下文信息. Devign 设计了门控图递归层用于聚合和传递图中相邻节点的信息来学习节点的特征, 并使用卷积模块提取关键节点特征用于表示图级别的预测. 在本文的实验中, 使用 Joern 工具生成了基础数据集中所有实例的代码属性图. 训练过程中采用了 Devign 模型的默认设置. 实验的训练集、验证集和测试集的具体配置详见如表 1.

• 语言模型. 表 2 展示了本文选用的语言模型的详细情况. 作为大型预训练语言模型的先驱之一, OpenAI 相继发布了 GPT-3.5-Turbo 和 GPT-4-preview 两款模型. GPT 系列模型基于 Transformer 架构, 该架构使用高效的自注意力机制捕获文本中的长距离依赖关系. 这些模型通过在互联网上收集的大规模文本数据集进行预训练, 学习到了丰富的语言表示能力. 同时, OpenAI 提供了 GPT-3.5-Turbo 的微调接口, 使研究人员和开发者能够根据特定的任务进一步优化模型. 经过微调后, GPT-3.5-Turbo 在某些特定任务上的表现甚至可以超越 GPT-4-preview 的基本功能[44]. 相比之下, GPT-4-preview 是一个更加复杂的模型, 拥有显著增加的参数规模, 并且具备处理多模态数据的能力[45]. Gemini Pro 是 Google 公司构建的多模态模型, 具有理解、解释和生成高质量代码的能力, 在 Natural2Code

基准测试集中的准确率高达 82.6%[46]. DeepSeek-Coder 是一个由一系列代码语言模型组成的集合. 在编程方面, DeepSeek-Coder 在多种编程语言和各种基准测试的开源代码模型中实现了最先进的性能. 同时, 本文本地部署了 Phi3、Gemma 和 CodeT5 模型. Phi3 和 Gemma 能够理解和生成自然语言文本, 处理复杂的语言任务. 而 CodeT5 是 Salesforce Research 发布的用于代码理解和生成的模型. 特别地, CodeT5 能够感知代码中的标识符并在掩码处理时恢复这些标识符. 此外, CodeT5 通过引入语言 ID 和双模态生成机制, 提高了自然语言和编程语言之间的一致性. 实验过程中, 本文在资源泄露基础数据集上对这些语言模型进行了评测, 并分别使用了图 5 中零样本和链式两种提示词. 在测试过程中, 部分模型可能出现冗长回答或无关回答. 为了保证评测结果的准确性, 本文在数据处理统计过程中忽视了这些回答. 另一方面, 本文使用 OpenAI 提供的 fine-tuning 接口微调了 GPT-3.5-Turbo. 本文使用训练集中的 1 944 个样例进行微调, 并使用剩余的 834 个样例进行测试.

• GRACE 方法. GRACE[19]是由 Lu 等人提出的漏洞检测工具, 旨在解决大模型在代码缺陷检测中忽略代码的语法和语义信息的问题. 为了增强大模型的检测能力, GRACE 通过在代码中加入图结构信息和语境学习进行优化. 具体而言, GRACE 首先利用 CodeT5 模型获取了输入代码语义最为相似的 5 个代码示例, 之后分别计算这 5 个示例与输入代码之间的词法和语法混合相似度分数, 从而选择出与输入代码词法、语法和语义上均最接近的代码作为 One-Shot 提示词中的示例. 此外, GRACE 还利用 Joern 提取输入代码的图结构信息, 并将这些信息融合到提示词中. 由此构成的提示词, 如图 5(c) 所示. 本文使用 1 944 条训练数据作为寻找相似代码的基准库, 并使用 834 条测试数据进行了验证. 由于该训练数据集中包含 953 条缺陷样本和 991 条非缺陷样本, 因此选取到的与输入样本最为相似的示例既可能为缺陷样本也可能为非缺陷样本. 在此 GRACE 基础上, 本文增加了 Few-Shot 提示词方案, 提示词中包含与输入代码最为相似的两个示例.

• INFERROI 方法. INFERROI 是由 Wang 等人[5]提出的一种专门用于资源泄露检测的方法. 传统的资源泄露缺陷检测技术通常依赖于预定义的资源获取和释放 API 和校验条件来发现未释放的资源, 但这类方法往往存在资源获取和释放 API 不完整以及资源可达性验证分析不健全的问题. INFERROI 利用大模型强大的代码理解能力, 推断代码中资源获取和释放 API 以及校验条件语句. 之后, 该方法结合了两阶段的静态分析技术, 根据大模型推断的意图检查控制流路径, 从而识别存在泄露风险的路径. INFERROI 充分考虑了资源泄露检测中资源识别以及可达性分析的关键特征, 是一种结合大模型和静态分析方法的新探索. 本文在复现过程中, 对搜寻可达路径的数量进行了限制, 以最多 1 000 条为上限. 如果路径搜寻时间超过 1 h, 则将可达路径数量限制为最多 100 条. 然而, 在 2 778 个样本中, 有 6 个样本仍遇到再次超时的情况, 这些样本的结果在数据统计中已被过滤掉.

4 实验分析

4.1 RQ1: 不同模型针对资源泄露检测的效果

● 评估指标

不同模型针对资源泄露检测效果的评价指标有准确率 (*Accuracy*)、精确度 (*Precision*)、召回率 (*Recall*)、*F*1-值 (*F*1-score, *F*1)、检测覆盖率 (Coverage, *C@M*) 以及成对预测 (pair-wise prediction). 针对模型的组合效果, 评价指标有真阳率 (true positive rate, *TPR@K*) 和假阳率 (false positive rate, *FPR@K*).

成对预测由 Ding 等人[40]提出, 通过在缺陷代码及其修复代码对上述评估模型的表现, 可以判断模型是否仅依赖于表面的文本模式进行预测, 还是掌握了深层次的安全含义. 本文采用了 JLeaks 和 DroidLeaks 组合构成的资源泄露数据集, 并将修复版本的代码作为无资源泄露的对照组. 在此基础上, 成对预测结果的定义如下.

成对正确预测 (P-C): 模型正确预测了成对代码, 即缺陷代码和无缺陷代码均被正确分类;

成对缺陷预测 (P-V): 模型正确预测了缺陷代码, 但错误地将无缺陷代码预测为缺陷代码;

成对良性预测 (P-B): 模型将缺陷代码预测为无缺陷代码, 但正确预测了无缺陷代码;

成对反向预测 (P-R): 模型错误地将缺陷代码预测为无缺陷代码, 同时将无缺陷代码预测为缺陷代码.

检测覆盖率是评估两个资源泄露检测模型在识别相同资源泄露问题上的重叠程度的重要指标, 能够帮助确定

不同方法之间的检测效果和互补性. 检测覆盖率定义如下:

$$C@M = \frac{D_M \cap D_{M'}}{D_M} \times 100\%,$$

其中, M 和 M' 是两种检测模型, D_M 表示模型 M 检测到的资源泄露的数量, $D_M \cap D_{M'}$ 表示模型 M 和模型 M' 共同检测成功的资源泄露数量. $C@M$ 指标的值介于 0 和 100% 之间, 值越接近 100% 表示 M' 的检测能力越能覆盖 M 的检测能力.

$TPR@K$ 是 K 个模型同时检测资源泄露的真阳率, $FPR@K$ 是 K 个模型检测资源泄露缺陷时的整体假阳率. 这两个指标对于评估多个模型在协同检测资源泄露中的效果具有重要参考价值. 余下评价指标的计算方法如下所示:

$$Accuracy = \frac{TP+TN}{TP+TN+FN+TN}, Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN}, F1 = \frac{2\times(Precision\times Recall)}{Precision+Recall}.$$

● 结果分析

本文首先测试了基于传统模型、基于小模型和基于大模型的缺陷检测方法在资源泄露数据集上的检测效果, 实验结果如表 3 所示. 模型上标为 1 时表示使用了图 5 中第 1 个提示词, 模型上标为 2 时表示使用了图 5 中第 2 个提示词. GRACE-oneshot 表示提示词中提供了一个与输入代码最为相似的代码示例, GRACE-twoshots 表示提示词中提供了两个代码示例. 需要注意的是, Bugram 中无需验证集, 因此测试样本数量为表 1 中测试集和验证集之和, 共计 834 个样本. 同样地, GRACE 将训练集作为提示词中示例的基准库, 微调 GPT-3.5-Turbo 时同样使用训练集作为训练样本, 因此测试样本数量也为 834 个. 其他大模型均使用 2 778 个样本进行测试. 然而, 由于部分模型回答与资源泄露检测无关, 本文在数据处理中排除了这些无关的回答. 在评估成对预测结果时, 由于训练集、验证集和测试集是随机划分的, 加上部分模型回答与资源泄露无关, 所以并非所有测试样本都能够成对评测. 因此, 本文仅关注测试结果中成对出现的样本用于评估成对预测结果, 并将成对样本数量标注在括号中.

表 3 不同模型在不同设置下的资源泄露检测的效果

模型	样本数量	准确率 (%)	精确度 (%)	召回率(%)	$F1$ (%)	P-C (%)	P-V (%)	P-B (%)	P-R (%)
Bugram	834 (124)	47.60	50.00	14.19	22.10	2.42	13.71	80.65	3.23
DeepBugs	556 (51)	44.06	46.56	59.09	52.08	7.84	72.55	17.65	1.96
ASTNN	556 (51)	68.71	70.00	68.53	69.26	50.98	25.49	17.65	5.88
Devign	556 (51)	49.10	50.33	80.42	61.91	1.96	80.39	9.8	7.84
Phi3[1]	2 759 (1 374)	51.21	63.57	5.94	10.86	4.73	1.24	91.85	2.18
Phi3[2]	2 308 (1 028)	55.94	53.42	56.88	55.10	18.48	37.45	36.96	7.1
CodeT5[1]	1 666 (626)	46.28	49.12	50.63	49.86	13.9	34.35	32.11	19.65
CodeT5[2]	684 (146)	52.34	56.12	69.27	62.00	19.18	46.58	15.07	19.18
DeepSeek-Coder[1]	2 778 (1 387)	67.67	67.65	67.84	67.74	39.65	28.26	27.83	4.25
DeepSeek-Coder[2]	2 777 (1 386)	69.00	66.46	76.83	71.27	43.72	33.12	17.39	5.77
Gemini Pro[1]	2 776 (1 385)	64.45	61.49	77.55	68.60	35.02	42.6	16.32	6.06
Gemini Pro[2]	2 763 (1 372)	58.89	56.57	76.63	65.09	28.21	48.4	12.9	10.5
Gemma[1]	2 773 (1 384)	49.95	49.97	63.59	55.96	7.23	56.29	29.05	7.44
Gemma[2]	2 773 (1 384)	48.50	42.44	8.29	13.87	2.6	5.64	86.13	5.64
GPT-3.5-Turbo[1]	2 778 (1 387)	53.46	61.05	19.28	29.31	11.1	8.22	76.57	4.11
GPT-3.5-Turbo[2]	2 778 (1 387)	53.60	61.45	19.50	29.60	11.75	7.79	75.99	4.47
GPT-3.5-Turbo^{FT1}	834 (124)	86.45	88.21	85.58	86.88	70.97	13.71	15.32	0
GPT-3.5-Turbo^{FT2}	834 (124)	86.21	87.27	86.27	86.77	70.97	16.13	12.1	0.81
GPT-4-preview[1]	2 778 (1 387)	68.97	75.24	56.62	64.61	40.23	16.44	41.1	2.24
GPT-4-preview[2]	2 773 (1 382)	64.37	79.88	38.38	51.85	30.17	8.25	60.13	1.45
GRACE-oneshot	834 (124)	66.19	64.71	78.03	70.75	37.9	41.94	15.32	4.84
GRACE-twoshots	834 (124)	65.47	64.14	77.35	70.12	33.87	44.35	19.35	2.42
INFERROI	2 772 (1 382)	62.01	61.90	62.75	62.33	30.82	31.98	30.39	6.8

图 6 总结了不同模型在测试集上资源泄露检测结果的交叉情况, 即检测覆盖率, 进一步对比了模型间的检测结果差异. 图 6 中的主对角线表示模型独立检测到的资源泄露的数量. 例如 Bugram 检测到 38 个资源泄露, INFERROI 检测到 179 个资源泄露. 单个模型检测到的资源泄露的数量在 38–249 之间不等. 此外, 图 6 中行和列详细展示了不同模型之间的检测覆盖率. 例如 ASTNN 模型检测到了 196 个资源泄露, 其中 116 个被 GPT-4-preview[1] 同时检测到, 占比 59%. 然而, GPT-4-preview[1] 检测到的 161 个资源泄露中, 只有 116 个与 ASTNN 检测结果重叠, 占比为 72%. 因此图 6 并不对称. 图 7 展示了真阳率和假阳率随着判别模型个数 K 增加的变化趋势, 横坐标为 K 值, 表示模型数量.

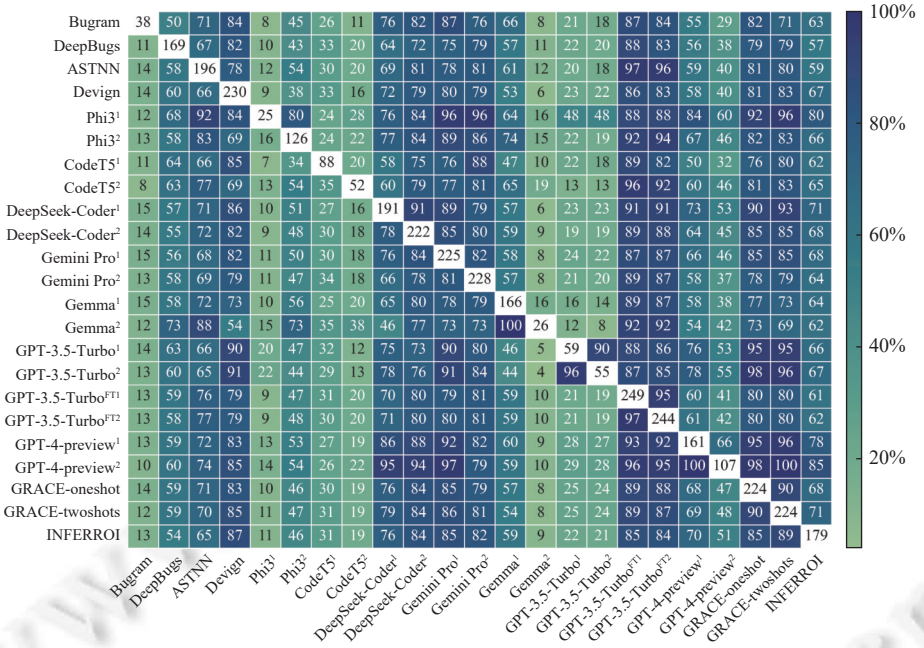


图 6 模型间缺陷检测覆盖率热力图

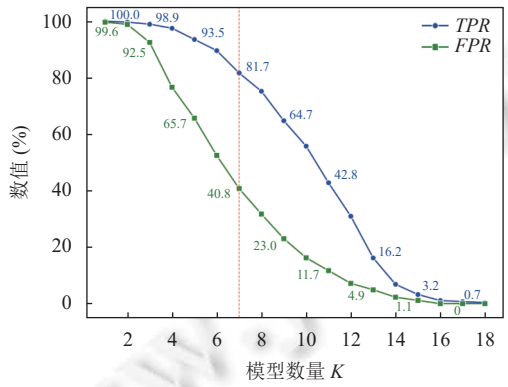


图 7 真阳率和假阳率随判别模型个数的变化趋势

Finding-RQ1.1: 微调后的 GPT-3.5-Turbo 在资源泄露任务上表现出显著优势. 初始测试中, 未经过微调的 GPT-3.5-Turbo 在资源泄露检测任务中表现不佳, 其精确率在零样本提示词和链式提示词两种情景下均未超过 54%, 召回率则低于 20%, $F1$ 值甚至不足 30%. 然而, 在采用 1 944 个样本进行微调后, GPT-3.5-Turbo 的效果实现了显著的提升, 精确度达到了近 86.5%, 较未经微调的版本提升了超 32 个百分点. 这一结果突显了微调对提升生成式模

型在资源泄露检测能力方面的关键作用. 相比之下, 尽管 GPT-4-preview 作为新一代的语言模型在诸多任务中表现出色, 其在资源泄露任务中的表现仍然被微调后的 GPT-3.5-Turbo 超越. 同时, 如图 6 所示, 微调后的 GPT-3.5-Turbo 几乎能够覆盖 85% 以上的其他工具或模型的检测结果. 未来的研究可以进一步探讨微调过程中的关键因素, 以期进一步提高模型资源泄露检测效果.

Finding-RQ1.2: 部分模型在区分资源泄露代码及其相似代码时能力较弱. 成对评估能够检验模型是否真正学会了识别安全漏洞, 而不是简单的识别某种模式, 不仅为模型评估提供了一个更为细致的视角, 还达到了对实际应用可行性的压力测试作用[40]. Phi3 和 CodeT5 模型在分析资源泄露这一复杂任务上表现出明显的局限性, 不仅更易输出与缺陷检测无关的回答, 而且检测准确率均未突破 56%. 其中, Phi3 在零样本提示词的情形下, P-C 值仅为 4.73%, 而 P-B 值则高达 91.85%, 这表明 Phi3 倾向于将样例视为无缺陷代码, 而缺乏对代码的实质理解能力. 类似的, Gemma 模型在链式提示词下 P-B 值也远高于 P-V 值. 然而, Gemma 模型在零样本提示下显示出较高的 P-V 值, 这一现象可能意味着 Gemma 模型在特定条件下具备更高的漏洞识别敏感性, 但是也可能导致较高的误报率. DeepSeek-Coder、Gemini Pro 以及 GPT-4-preview 在资源泄露任务中表现相似, 准确率介于 59%–69% 之间. 然而, GPT-4-preview 模型的 P-B 值显著高于 P-V 值, 表明其更倾向将资源泄露代码错误地判定为无缺陷代码. 相反地, DeepSeek-Coder 和 Gemini Pro 的 P-V 值则显著高于 P-B 值, 表明这些模型更倾向于将无缺陷的代码判定为资源泄露代码.

Finding-RQ1.3: 小样本学习不完全依赖于示例标签. 在 GRACE 模型的设计中, 通过在提示词中加入与输入代码相似的示例及其标签, 以提升 GPT-4 在缺陷检测任务中的效果. 然而, 本文使用的基准数据集将修复后的代码视为无缺陷代码, 这导致在提示词中插入的代码示例可能会与输入代码的标签不一致. 根据统计, 在单样本学习的情况下, 在 834 个测试样本中, 410 个输入样本给出了标签相反的示例, 占比为 49.16%. 标签不一致的情况很难避免, 因为在数据集中查询最相似的代码时, 很难保证不会查询到标签相反的示例. 为了深入探究示例标签不一致对模型检测资源泄露的影响, 本文从“上帝视角”出发, 为输入代码提供标签与输入代码一致的相似示例. 实验结果显示, 此时 GRACE-oneshot 准确率为 66.31%, 相较于原始情况下的 66.19% 仅有微小提升, 而 GRACE-twoshots 的准确率则未改变. 这表明, 在资源泄露检测任务中, 小样本学习的效果不完全依赖于示例标签的一致性.

Finding-RQ1.4: 静态分析能提高模型的召回率. INFERROI 使用 GPT-4 识别代码中资源获取和释放等操作, 并结合静态分析, 判别代码中是否存在潜在的资源泄露. 表 3 显示, INFERROI 检测资源泄露时相较于 GPT-4 在召回率上有所提升. 具体而言, INFERROI 在 1 388 个资源泄露中成功检测出 871 个, 缺陷预测召回率为 62.75%; 相比之下, GPT-4-preview[1] 仅能在 1 390 个资源泄露中检测出 787 个资源泄露, 缺陷预测召回率为 56.62%. 对于缺陷检测任务而言, 其主要目标是尽可能多地捕获潜在的漏洞, 以确保软件系统的安全性和可靠性[40], 因此, 从实用性的角度来说, 静态分析结合生成式模型的方式具有一定前景.

Finding-RQ1.5: 多种工具结合能够提高真阳率, 但同时具有较高的假阳率. 图 6 显示, 除了微调后的 GPT-3.5 模型外, 其余部分模型在缺陷覆盖率上表现有限. 为了进一步提升其他模型的资源泄露检测精确度, 本文对模型进行了组合分析, 并统计了这些模型在预测资源泄露时的综合真阳率和综合假阳率实验过程中, 本文排除了易给出无关回答的 Phi[2]、CodeT5[1]、CodeT5[2] 模型和效果最好的微调后的 GPT-3.5 模型, 选用了 278 个资源泄露样例和 265 个非资源泄露样例为基准. 图 7 展示了综合真阳率和综合假阳率随着判别模型个数增加的变化趋势, 其中横坐标为 K 值. 如图 7 所示随着模型数量的增加, 真阳率逐步下降, 但是假阳率也随之下降. 当有 7 个模型共同检测时, 达到了真阳率和假阳率的相对平衡, 此时真阳率为 81.7%, 假阳率为 40.8%. 当有 5 个模型共同检测时, 真阳率为 93.5%, 但假阳率为 65.7%, 相对于 7 个模型的判别, 真阳率和假阳率均大幅度增加.

4.2 RQ2: 模型针对不同原因引发的资源泄露的检测效果

- 评估指标

本文使用召回率评估模型在不同原因引发的资源泄露中的有效性. 实验中, 本文重点分析了模型在检测 3 种典型资源泄露原因时的表现: 常规路径下未关闭资源 (noCloseRPath)、异常路径下未关闭资源 (noCloseEPath) 和分支路径未关闭资源 (noCloseCPath). 表 1 中详细列出了 3 种缺陷原因在基础数据集中的分布情况. 召回率能够反

映缺陷被正确预测的比例, 在缺陷检测任务中具有较高的意义.

● 结果分析

图 8 展示了不同模型在不同缺陷原因下的召回率. 其中 Bugram、GRACE 以及微调后的 GPT-3.5-Turbo 模型基于表 1 所示的验证集和测试集, 而 DeepBugs、ASTNN 以及 Devign 的结果则仅基于测试集. 其余模型的结果基于全数据集. 然而, 由于部分模型回答与资源泄露无关, 本文排除了这些模型的无关回答, 以确保数据的准确性.

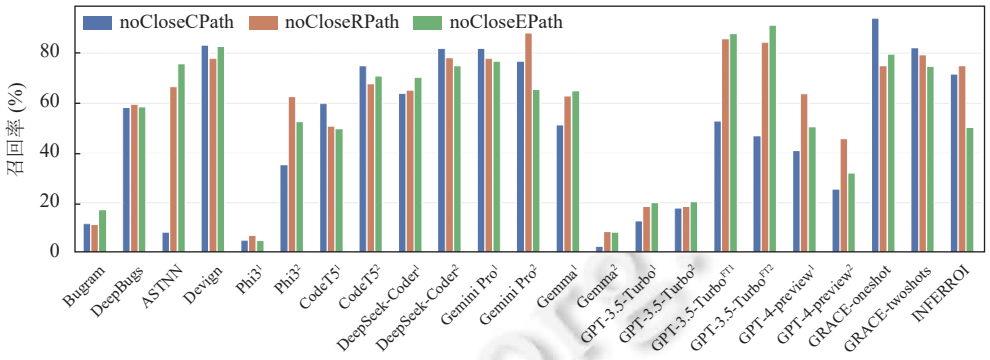


图 8 不同模型在不同缺陷原因下的召回率

Finding-RQ2.1: 对于不同缺陷原因, 大部分模型的召回率相差不大. 预想中, 一方面语言模型可能对常规路径下没有关闭资源 (noCloseRPath) 的情况表现良好, 因为 noCloseRPath 通常具有更为简单和明确的模式; 另一方面, 需进一步训练的模型可能对在异常情况下没有关闭资源 (noCloseEPath) 的情况表现良好, 因为 noCloseEPath 在训练集中占比最高. 然而, 大部分模型针对不同缺陷原因下的缺陷检测召回率并无明显规律. 值得注意的是传统模型 Devign 以及语言模型 CodeT5[2]、DeepSeek-Coder[2]、Gemini Pro[1] 和 Gemini Pro[2] 均表现较好, 结合表 3, 我们可以明显观察到这些模型的 P-V 值明显高于 P-B 值, 表示这些模型更倾向于将代码判别为有缺陷.

Finding-RQ2.2: 静态分析和大模型结合需进一步加强对 noCloseEPath 引发的资源泄露的检测. 相较于 GPT-4 而言, 静态分析和大模型结合后资源泄露检测的召回率有所提高, 这表明静态分析和大模型结合检测资源泄露具有前景. 但同时, 我们观察到尽管静态分析结合大模型的方式在因 noCloseRPath 引发的资源泄露检测中表现较好, 但在处理由 noCloseEPath 引发的资源泄露时, 其召回率仍不足 60%. 图 9(a) 展示了来自 apache/hbase 项目的一段代码示例, 揭示了异常处理不当导致资源无法正常关闭的问题. 在该示例中, is 和 zos 作为参数传入 copyToZipStream 方法, 该方法完成读写操作后调用 close 方法试图关闭 InputStream is 以及 ZipOutputStream zos. 由于资源获取到关闭路径是可达的, 因此静态分析结合大模型的方式可能会错误地将其判定为无缺陷代码. 然而, 在实际执行过程中, 若发生异常, close 方法将无法被调用, 进而导致资源泄露. 若程序中使用显式的异常捕获方式, 那么该静态分析和大模型结合的方式仍具有一定优势. 例如图 9(b) 展示了来自 alibaba/nacos 项目中由于 noCloseEPath 引发的资源泄露. 静态分析和大模型结合的方式能够准确识别 catch(Exception e) 分支中未关闭资源的情况, 从而成功检测到识别图 9(b) 中的资源泄露. 为了提高静态分析结合大模型的方式在 noCloseEPath 引发的资源泄露中的准确性, 建议进一步加强对异常处理路径的分析, 从而提升其在复杂代码环境下的整体检测效果.

Finding-RQ2.3: 检测因 noCloseCPath 原因导致的资源泄露具有挑战性. noCloseCPath 通常是由于代码中的逻辑错误导致的资源泄露, 检测该类型的资源泄露通常需要对代码逻辑有更深刻的理解. 图 8 显示 Bugram、ASTNN、Phi3[1]、Phi3[2]、Gemma[2]、GPT-3.5-Turbo[1]、GPT-3.5-Turbo[2]、GPT-4-preview[1] 以及 GPT-4-preview[2] 在检测 noCloseCPath 类型资源泄露时的表现相对较差, 其中 Gemma[2] 几乎未能检出任何此类型资源泄露. 图 10 展示了 osmadapp/OsmAnd 项目中由于 noCloseCPath 引发的资源泄露示例, 左图为存在缺陷的代码, 右图为修复后的代码. 通过分析开发者的修复方法可以确定, 无论 cursor.moveToFirst() 的结果如何, cursor 都应被关闭. 然而, 众多模型未能识别出该代码中的资源泄露问题. 这一结果表明, 现有模型在识别 noCloseCPath 引发的资源泄露时存在不足, 未来的研究应进一步增强模型对此类缺陷的识别能力.

```
private static void copyToZipStream(InputStream is, ZipEntry entry,
ZipOutputStream zos) throws IOException {
   zos.putNextEntry(entry);
   byte[] arr = new byte[4096];
   int read = is.read(arr);
   while (read > -1) {
      zos.write(arr, 0, read);
      read = is.read(arr);
   }
   is.close();
   zos.closeEntry(); }
```

throws

throws

Exception

throws

(a) apache/hbase 项目中 noCloseEPath 引发的资源泄露示例

```
public static byte[] tryCompress(String str, String encoding){
   if (str == null || str.length() == 0) {
      return null; }
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   GZIPOutputStream gzip;
   try {
      gzip = new GZIPOutputStream(out);
      gzip.write(str.getBytes(encoding));
      gzip.close();} catch (Exception e) {
         e.printStackTrace(); }
      return out.toByteArray();   }
```

(b) alibaba/nacos 项目中 noCloseEPath 引发的资源泄露示例

图 9　noCloseEPath 引发的资源泄露示例

```
private void indexCachedResources(Map<String, Long> fileModified,
Map<String, SQLiteTileSource> rs) {
   Cursor cursor = sqliteDb.rawQuery("SELECT filename, date_modified,
left, right, top, bottom FROM TILE_SOURCES",new String[0]);
   if(cursor.moveToFirst()) {
      do{
         …} while(cursor.moveToNext());
      cursor.close();
   }
}
```

```
private void indexCachedResources(Map<String, Long> fileModified,
Map<String, SQLiteTileSource> rs) {
   Cursor cursor = sqliteDb.rawQuery("SELECT filename, date_modified,
left, right, top, bottom FROM TILE_SOURCES",new String[0]);
   if(cursor.moveToFirst()) {
      do{
         …} while(cursor.moveToNext());
   }
   cursor.close();
}
```

图 10　osmadapp/OsmAnd 项目中 noCloseCPath 引发的资源泄露示例

4.3　RQ3: 模型针对不同资源类别、资源数据类型和变量作用域的资源泄露检测效果

● 评估指标

本文使用准确率评估不同模型检测不同资源类别、不同资源数据类型以及不同作用域下的资源泄露的检测效果的差异. 实验中, 本文将资源泄露涉及的资源类别分为文件、套接字和线程这 3 大类, 将资源变量 (即关键变量) 按作用域不同分为全局变量和局部变量 (其中匿名变量和参数因其特殊性被单独划分出来), 并将资源数据类型分为标准库 (包括 Java 标准库和 Android 标准库) 和第三方库. 通过这种细粒度的分析, 本文旨在全面评估模型的资源泄露检测能力, 从而为之后的资源泄露检测提供有价值的参考.

● 结果分析

图 11–图 13 分别展示了不同模型在不同资源类别下、不同资源数据类型下以及不同资源作用域下的准确率.

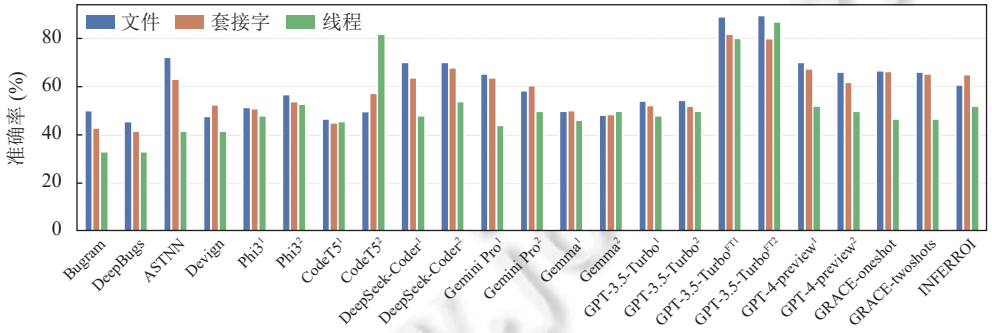


图 11　不同模型在不同资源类别下的准确率

Finding-RQ3.1: 文件类别的准确率总体最高. 如图 11 所示, 大多数模型针对文件类别的缺陷检测准确率均高于 50%. 其中, ASTNN、GPT-3.5-Turbo^{FT1} 以及 GPT-3.5-Turbo^{FT2} 拥有最高的文件类别下的准确率. 对于使用基本数据集训练的传统模型 Bugram、DeepBugs 以及 ASTNN 来说, 文件类别的准确率均明显高于其他类别的准确率. 类似地, GPT-3.5-Turbo^{FT1} 以及 GPT-3.5-Turbo^{FT2} 也表现出在文件类别上的准确率高于其他类别的情况. 这一现象可以归因于基础数据集中文件类别的样本数量为 914, 占总数的 65.75%. 这使得模型在训练过程中, 文件类别资源

泄露的检测能力得到了更多的关注和提升. 值得注意的是, CodeT5[2] 在线程资源类别上的准确率异常高. 这一结果可能源于 CodeT5[2] 在资源泄露检测中更倾向将样本判定为缺陷样本. 如表 3 所示 CodeT5[2] 的 P-V 值高达 46.58%.

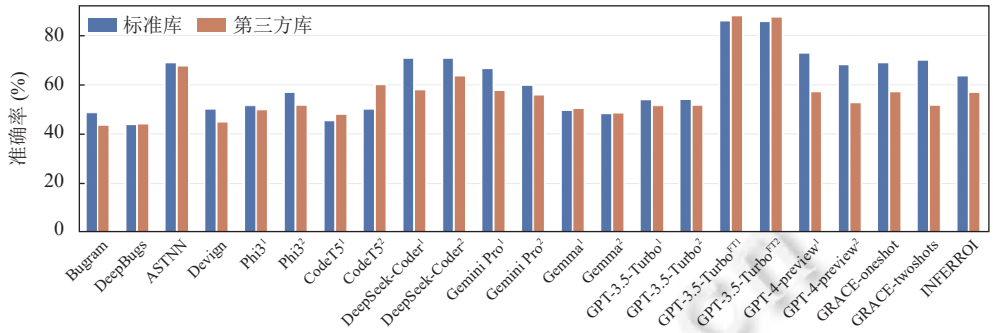


图 12 不同模型在不同资源数据类型下的准确率

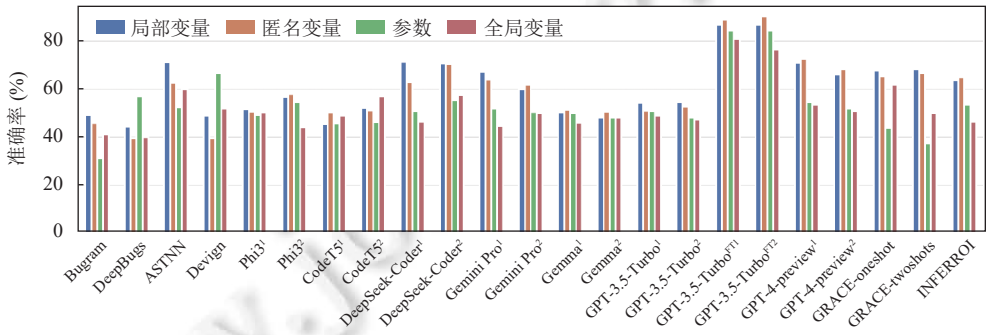


图 13 不同模型在不同资源变量作用域下的准确率

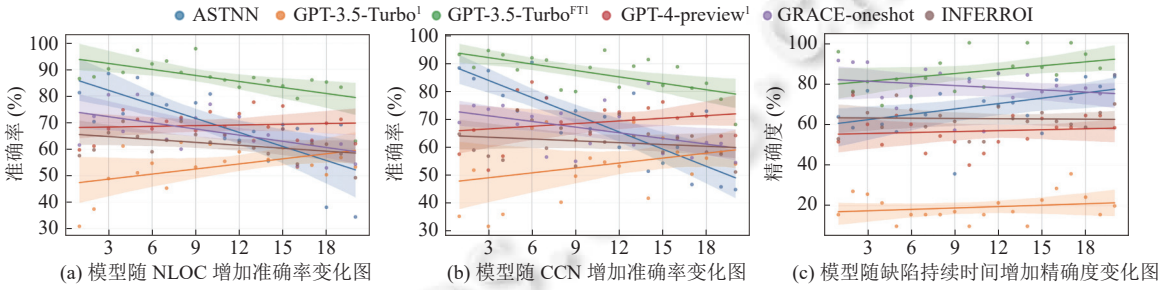
Finding-RQ3.2: 标准库类型准确率总体高于第三方库类型准确率. 如图 12 所示, 大多数模型对使用标准库 API 创建的资源具有更强的检测能力, 且部分模型在检测标准库和第三方库引起的资源泄露时显示出相对明显的准确率差异, 如 GPT-4-preview、INFERROI 等. 其中 INFERROI 依赖于 GPT-4-preview 对资源获取和释放操作的识别能力进行进一步的静态分析, 因此 GPT-4-preview 对资源的识别能力直接影响了 INFERROI 的整体检测效果. 模型在标准库中的准确率总体高于针对第三方库的准确率, 主要原因在于标准库提供的 API 在资源泄露场景中更为常见. 第三方库提供获取和释放资源的 API 通常是在项目中将标准库提供的 API 进一步封装或添加新功能后供开发者使用. 在实际应用中, 使用第三方库的 API 获取资源的情况也相当普遍. 在基础数据集中, 27.48% 的样例使用了第三方库提供的 API. 由此可见, 识别第三方库 API 是资源泄露检测中一个亟待解决的问题.

Finding-RQ3.3: 匿名变量准确率和局部变量准确率明显高于全局变量准确率和参数变量准确率. 从定义上来说, 匿名变量和参数均属于局部变量. 参数变量是在方法签名中定义的, 通常由外部方法传入, 其作用范围受限且具有输入特性, 匿名变量是未显式命名的变量, 通常出现在简化代码的场景中, 由于两种变量的特殊性, 在探究不同模型在资源泄露检测中的差异时, 本文将匿名变量和参数从局部变量中独立出来进行分析. 图 13 显示了大多数模型在匿名变量占和局部变量上均呈现出更高的准确度. 然而, 在 DeepBugs 模型中, 匿名变量占比率最低, 这是由于 DeepBugs 是基于命名的缺陷检测器, 专注于通过分析标识符名称的语义表示来预测与命名相关的缺陷. 由于匿名变量在源码级别上不具备标识符名称, 因此 DeepBugs 模型对匿名变量的检测能力较弱. 在方法粒度的缺陷检测中, 方法内部可能并不涉及参数或全局变量的资源获取操作, 缺乏上下文信息, 这可能是导致全局变量和参数变量准确度较低的原因之一. 为了验证这一假设, 本文随机选取了 5 个涉及全局变量的缺陷样本, 并在代码中增加变量的定义语句, 之后使用 GPT-4-preview[1] 再次进行检测. 结果显示, 原本未能检测出资源泄露的 5 个样本中, 有 3 个在加入上下文信息后被成功检测出资源泄露. 这表明在跨过程的资源泄露检测中, 获取资源的上下文信息能够提升模型的检测效果.

4.4 RQ4: 不同代码属性对模型资源泄露检测效果的影响

本研究问题主要关注不同行数、不同复杂度以及不同隐蔽程度对模型资源泄露检测效果的影响. 其中, 隐蔽程度由缺陷持续时间来衡量. 此次探究中, 本文仅选用了几个代表性的模型. 为了分析这些因素的影响, 本文首先对不同的代码属性进行了排序和分组, 分别计算了在各个分组区间内, 不同模型的准确率或精确度. 此次实验仅选取了在基于传统模型的方法中表现突出的 ASTNN 模型以及部分具有代表性的大模型.

图 14 展示了不同代码属性和模型资源泄露检测效果之间的关系. 为了深入分析这些关系, 本文将每个代码属性排序后分为了 20 组, 并以组索引为横坐标. 在图 14 中, 针对代码行数和圈复杂度, 计算了不同模型的准确率. 代码行数和圈复杂度分别代表了代码规模和逻辑复杂度, 通过这些属性可以评估模型在处理简单和复杂代码时的检测效果差异. 此外, 为了探讨缺陷的隐蔽性对模型检测效果的影响, 本文计算了不同模型在不同缺陷持续时间下的精确度.



(a) 模型随 NLOC 增加准确率变化图 (b) 模型随 CCN 增加准确率变化图 (c) 模型随缺陷持续时间增加精确度变化图

图 14 行数 (NLOC)、圈复杂度 (CCN) 以及缺陷持续时间对模型检测效果的影响

Finding-RQ4.1: 代码行数及复杂度对传统模型的资源泄露检测准确率影响较大. 正如图 14(a) 和 (b) 所示, 随着 NLOC 和 CCN 的增加, ASTNN 模型的准确率显著下降. ASTNN 使用抽象语法树作为代码的表征形式, 将其转化为多个小语法树进行编码. 尽管这种方式在一定程度上缓解了模型的梯度消失问题, 但并未彻底解决. 因此, 当代码行数和复杂度增加时, ASTNN 模型在缺陷检测任务中的表现会受到限制, 导致准确率降低.

Finding-RQ4.2: 缺陷持续时间对模型检测效果影响较小. 图 14(c) 展示了不同缺陷持续时间下模型精确度的变化, 可以看到, 无论缺陷持续时间长短, 各模型的准确率变化幅度都较小. 这一现象表明, 模型在检测缺陷时可能对缺陷的隐蔽性并不敏感. 这可能是因为缺陷的隐蔽性主要是从开发者的角度来判断的, 而模型在检测过程中更多地依赖于代码特征和模式, 而不是对缺陷隐蔽性的主观理解. 因此, 缺陷持续时间对传统模型和大模型的缺陷检测能力影响较小. 此外, 由于本文使用缺陷文件上一次修改的时间来粗略表示缺陷引入的时间, 这种方法可能无法完全准确地反映缺陷的实际引入时间, 因此在一定程度上影响了缺陷持续时间对模型检测效果影响的评估结果. 这一限制需要在未来研究中进一步探讨, 以确保评估的准确性和全面性.

5 有效性威胁

本文探讨的内部有效性威胁主要来自于模型复现和技术实现. 为了减低这些威胁, 实验使用的传统模型以及语言模型的调整方式大部分采用官方或开源实现, 并且模型参数等均采用默认配置. 此外, 本文作者对实验代码进行了细致审查, 以确保逻辑严谨和实验设计的完整性.

本文的外部有效性威胁主要体现在基准测试集的规模限制. 为此, 本文结合了 DroidLeaks 和 JLeaks 两个 Java 资源泄露数据集, 共计获取了 2 778 个代码片段, 其中包含 1 390 个资源泄露实例. 尽管 JLeaks 被认为是目前已知的最大的 Java 资源泄露数据库, 但是其规模依然有限, 这在一定程度上制约了本文的研究. 针对这一问题, 未来研究将致力于扩展实验场景, 使用更复杂的代码示例, 以进一步深入探讨语言模型在 Java 资源泄露检测中的有效性.

另一个外部有效性威胁则与数据泄露有关. 当测试代码来自于开源项目, 存在模型预训练阶段已经使用了这些程序代码作为训练数据的可能性. 尽管 DroidLeaks 和 JLeaks 中的代码均来自开源项目, 但是其缺陷代码和非缺陷代码跨越了同一项目的不同版本, 这在一定程度上缓解了数据泄露的风险. 在之后的研究工作中, 我们将进一步

探究数据泄露对评估大模型在特定缺陷类型检测中的影响.

此外, 小模型和大模型输出的不确定性也是本文面临的一项有效性威胁. 为了减少这种不确定性, 我们在模型评估时将 temperature 设置为 0. 虽然这一设置可能会限制模型的创造性, 但能够显著减低结果的随机性, 从而减少了输出结果不一致的可能性. 未来的研究将进一步探究在不同参数设置下, 语言模型在 Java 资源泄露检测中的有效性及其表现差异.

6 对未来工作的可能影响

本文的实证研究对未来的工作可能有以下两方面启发.

(1) 大模型微调: 实验中, 使用 OpenAI 提供的 Fine-tuning 接口对 GPT-3.5-Turbo 进行微调, 可显著提高其在资源泄露检测任务中的检测效果, 准确率由 53.46% 提升至 86.45%. 未来的缺陷检测工作可以利用高质量数据集微调大模型, 提升大模型在特定类型缺陷上的检测效果.

(2) 大模型与静态分析技术结合: 当前已有许多研究尝试将大模型与静态分析技术结合, 借助大模型的代码理解能力降低静态分析的误报率. 实验中, 本文测试了结合大模型和静态分析结合的工具, 该工具提高了大模型在资源泄露检测任务上的召回率, 但仍有改进空间. 未来工作可以加强对异常处理路径的分析, 从而提升其在复杂代码环境下的整体检测效果.

7 总 结

本文深入研究了基于传统模型、小模型和大语言模型的多种资源泄露检测方法, 系统对比了不同模型在检测 Java 资源泄露方面的表现, 并探讨了基于小样本学习、模型微调及模型与静态分析结合的改进方法. 在实验中, 采用了 JLeaks 和 DroidLeaks 数据集, 从资源泄露根本原因、资源类型、代码复杂度等多个维度进行了分析, 揭示了不同模型的优势与局限性, 并提出了未来研究的方向. 未来的研究应继续关注微调技术在特定缺陷检测中的应用, 以及通过结合静态分析与大语言模型来提升整体检测效果.

References:

[1] Ghanavati M, Costa D, Seboek J, Lo D, Andrzejak A. Memory and resource leak defects and their repairs in Java projects. Empirical Software Engineering, 2020, 25(1): 678–718. [doi: 10.1007/s10664-019-09731-8]

[2] Liu TY, Ji WX, Dong XH, Yao WH, Wang YZ, Liu H, Peng HY, Wang YX. JLeaks: A featured resource leak repository collected from hundreds of open-source Java projects. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: IEEE, 2024. 1–13.

[3] Kellogg M, Shadab N, Sridharan M, Ernst MD. Lightweight and modular resource leak verification. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 181–192. [doi: 10.1145/3468264.3468576]

[4] Utture A, Palsberg J. From leaks to fixes: Automated repairs for resource leak warnings. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023: 159–171. [doi: 10.1145/3611643.3616267]

[5] Wang C, Liu JN, Peng X, Liu Y, Lou YL. Boosting static resource leak detection via LLM-based resource-oriented intention inference. arXiv:2311.04448, 2023.

[6] Lo D, Nagappan N, Zimmermann T. How practitioners perceive the relevance of software engineering research. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo: ACM, 2015. 415–425. [doi: 10.1145/2786805.2786809]

[7] Wang C, Lou YL, Peng X, Liu JN, Zou BH. Mining resource-operation knowledge to support resource leak detection. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 986–998. [doi: 10.1145/3611643.361631]

[8] Li W, Cai HP, Sui YL, Manz D. PCA: Memory leak detection using partial call-path analysis. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2020. 1621–1625. [doi: 10.1145/3368089.3417923]

[9] PMD source code analyzer. 2002. https://pmd.github.io

[10] Wang S, Chollak D, Movshovitz-Attias D, Tan L. Bugram: Bug detection with n-gram language models. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: IEEE, 2016. 708–719.

[11] Pradel M, Sen K. DeepBugs: A learning approach to name-based bug detection. Proc. of the ACM on Programming Languages, 2018, 2: 147. [doi: 10.1145/3276517]

[12] Zhang J, Wang X, Zhang HY, Sun HL, Liu XD, Hu CM, Liu Y. Detecting condition-related bugs with control flow graph neural network. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 1370–1382. [doi: 10.1145/3597926.3598142]

[13] Zou DQ, Wang SJ, Xu SH, Li Z, Jin H. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. on Dependable and Secure Computing, 2021, 18(5): 2224–2236. [doi: 10.1109/TDSC.2019.2942930]

[14] Zhou YQ, Liu SQ, Siow J, Du XN, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 915.

[15] Nguyen VA, Nguyen DQ, Nguyen V, Le T, Tran QH, Phung D. ReGVD: Revisiting graph neural networks for vulnerability detection. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering: Companion Proc. Pittsburgh: IEEE, 2022. 178–182. [doi: 10.1145/3510454.3516865]

[16] Li Y, Huang CL, Wang ZF, Yuan L, Wang XC. Survey of software vulnerability mining methods based on machine learning. Ruan Jian Xue Bao/Journal of Software, 2020, 31(7): 2040–2061 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]

[17] Fu M, Tantithamthavorn C. LineVul: A Transformer-based line-level vulnerability prediction. In: Proc. of the 19th IEEE/ACM Int'l Conf. on Mining Software Repositories. Pittsburgh: IEEE, 2022. 608–620. [doi: 10.1145/3524842.3528452]

[18] Liu JY, Ai J, Lu MY, Wang J, Shi HX. Semantic feature learning for software defect prediction from source code and external knowledge. Journal of Systems and Software, 2023, 204: 111753. [doi: 10.1016/j.jss.2023.111753]

[19] Lu GL, Ju XL, Chen X, Pei WL, Cai ZL. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. Journal of Systems and Software, 2024, 212: 112031. [doi: 10.1016/j.jss.2024.112031]

[20] Chen TY, Li L, Zhu LC, Li ZY, Liu XQ, Liang GT, Wang QX, Xie T. VulLibGen: Generating names of vulnerability-affected packages via a large language model. In: Proc. of the 62nd Annual Meeting of the Association for Computational Linguistics. Bangkok: ACL, 2024. 9767–6780. [doi: 10.18653/v1/2024.acl-long.527]

[21] Li HN, Hao Y, Zhai YZ, Qian ZY. Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proc. of the ACM on Programming Languages, 2024, 8: 111. [doi: 10.1145/3649828]

[22] Sun YQ, Wu DY, Xue Y, Liu H, Wang HJ, Xu ZZ, Xie XF, Liu Y. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 166. [doi: 10.1145/3597503.3639117]

[23] Yu JX, Liang P, Fu YJ, Tahir A, Shahin M, Wang C, Cai YX. An insight into security code review with LLMs: Capabilities, obstacles and influential factors. arXiv:2401.16310, 2024.

[24] Sun YQ, Wu DY, Xue Y, Liu H, Ma W, Zhang LY, Liu Y, Li YJ. LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. arXiv:2401.16185, 2024.

[25] Liu YP, Wang J, Wei LL, Xu C, Cheung SC, Wu TY, Yan J, Zhang J. DroidLeaks: A comprehensive database of resource leaks in Android APPs. Empirical Software Engineering, 2019, 24(6): 3435–3483. [doi: 10.1007/s10664-019-09715-8]

[26] TableInputFormat/TableRecordReaderImpl leaks HTable. 2014. https://github.com/apache/hbase/commit/e04009c9894b1ace20759c5f97f30126f3129aa3

[27] HBase. TableInputFormat/TableRecordReaderImpl leaks HTable. 2014. https://issues.apache.org/jira/browse/HBASE-10330

[28] Chen X, Gu Q, Liu WS, Liu SL, Ni C. Survey of static software defect prediction. Ruan Jian Xue Bao/Journal of Software, 2016, 27(1): 1–25 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]

[29] Younis A, Malaiya Y, Anderson C, Ray I. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In: Proc. of the 6th ACM Conf. on Data and Application Security and Privacy. New Orleans: ACM, 2016. 97–104. [doi: 10.1145/2857705.2857750]

[30] Shin Y, Williams L. Is complexity really the enemy of software security? In: Proc. of the 4th ACM Workshop on Quality of Protection. Alexandria: ACM, 2008. 47–50. [doi: 10.1145/1456362.1456372]

[31] Li RH, Feng C, Zhang X, Tang CJ. A lightweight assisted vulnerability discovery method using deep neural networks. IEEE Access, 2019, 7: 80079–80092. [doi: 10.1109/ACCESS.2019.2923227]

[32] JavaParser. https://javaparser.org/
[33] ANTLR. https://www.antlr.org/
[34] Anbiya DR, Purwarianti A, Asnar Y. Vulnerability detection in PHP Web application using lexical analysis approach with machine learning. In: Proc. of the 5th Int'l Conf. on Data and Software Engineering. Mataram: IEEE, 2018. 1–6. [doi: 10.1109/ICODSE.2018.8705809]
[35] Zhang J, Wang X, Zhang HY, Sun HL, Wang KX, Liu XD. A novel neural source code representation based on abstract syntax tree. In: Proc. of the 41st IEEE/ACM Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 783–794. [doi: 10.1109/ICSE.2019.00086]
[36] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 590–604. [doi: 10.1109/SP.2014.44]
[37] Wang HT, Ye GX, Tang ZY, Tan SH, Huang SF, Fang DY, Feng YS, Bian LZ, Wang Z. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. on Information Forensics and Security, 2021, 16: 1943–1958. [doi: 10.1109/TIFS.2020.3044773]
[38] Cheng X, Wang HY, Hua JY, Xu GA, Sui YL. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. on Software Engineering and Methodology (TOSEM), 2021, 30(3): 38. [doi: 10.1145/3436877]
[39] Han K, Xiao A, Wu EH, Guo JY, Xu CJ, Wang YH. Transformer in Transformer. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Curran Associates Inc., 2021. 1217.
[40] Ding YRB, Fu YJ, Ibrahim O, Sitawarin C, Chen XY, Alomair B, Wagner D, Ray B, Chen YZ. Vulnerability detection with code language models: How far are we? arXiv:2403.18624, 2024.
[41] Chen XP, Hu X, Huang Y, Jiang H, Ji WX, Jiang YJ, Jiang YY, Liu B, Liu H, Li XC, Lian XL, Meng GZ, Peng GZ, Peng X, Sun HL, Shi L, Wang B, Wang C, Wang JY, Wang TT, Xuan JF, Xia X, Yang YB, Yang YX, Zhang L, Zhou YM, Zhang L. Deep learning-based software engineering: Progress, challenges, and opportunities. SCIENCE CHINA Information Sciences, 2025, 68(1): 111102. [doi: 10.1007/s11432-023-4127-5]
[42] LangChain. Announcing LangSmith, a unified platform for debugging, testing, evaluating, and monitoring your LLM applications. 2023. https://blog.langchain.dev/announcing-langsmith/
[43] Khare A, Dutta S, Li ZY, Solko-Breslin A, Alur R, Naik M. Understanding the effectiveness of large language models in detecting security vulnerabilities. arXiv:2311.16169, 2023.
[44] GPT-3.5 Turbo fine-tuning and API updates. 2023. https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/
[45] GPT-4 is OpenAI's most advanced system, producing safer and more useful responses. 2024. https://openai.com/index/gpt-4/
[46] Gemini Pro. 2024. https://deepmind.google/technologies/gemini/pro/

附中文参考文献:

[16] 李韵, 黄辰林, 王中锋, 袁露, 王晓川. 基于机器学习的软件漏洞挖掘方法综述. 软件学报, 2020, 31(7): 2040–2061. http://www.jos.org.cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]
[28] 陈翔, 顾庆, 刘望舒, 刘树龙, 倪超. 静态软件缺陷预测方法研究. 软件学报, 2016, 27(1): 1–25. http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]

刘天阳(1995－), 女, 博士生, 主要研究领域为程序分析与优化, 软件缺陷库构建.

计卫星(1980－), 男, 博士, 教授, 博士生导师, 主要研究领域为计算机体系结构, 程序分析与优化, 并行与高性能计算.

叶嘉威(2001－), 男, 硕士生, 主要研究领域为程序分析与优化.

刘辉(1978－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能软件工程, 数据挖掘, 深度学习.