摘要

源代码漏洞检测主要是判断一段软件代码里是否存在安全隐患，在软件测试和维护等软件工程领域中应用广泛，对于保障软件功能及加强应用安全性具有重要作用。随着互联网的快速发展，软件的功能和对应的代码也愈发复杂，软件漏洞的复杂性和检测难度也不断提升。随着深度学习在源代码漏洞检测领域的应用，神经网络模型能够自动化学习漏洞代码的模式，节省了人工检查和开发高性能漏洞检测工具的成本，显著提高了漏洞检测的效率。然而，现有方法仍面临诸多挑战，如依赖低质量漏洞数据集、学习不相关代码特征、不合适的模型设计，以及模型难以定位到漏洞的位置等问题。

为解决高质量漏洞数据集稀缺的问题，本文提出了一种自动化的漏洞代码库构建框架。该框架从CVE数据库和开源代码库中提取漏洞信息和候选补丁信息，然后利用多个静态分析工具、机器学习模型和大语言模型，对漏洞代码和补丁代码进行综合分析，以标注准确的标签。为了能够标注漏洞的具体位置，本文针对主流的漏洞类型，制定了一套用于标记漏洞可能触发位置的PoIs特征。基于PoIs特征，结合代码的修订记录，本文对漏洞触发的位置和路径进行了详细标注。本文基于此框架构建了一个高质量的漏洞数据集ReliVul，与现有数据集相比，ReliVul在数据集基本信息、准确性、唯一性和完整性等方面均有显著提升，为本文后续的研究奠定了基础，填补了漏洞检测领域高质量漏洞数据集的空缺。

为了解决现有漏洞检测模型的不足之处，本文提出了基于多模态注意力机制的源代码漏洞检测与定位模型MaliVD。MaliVD不仅能够识别出具体的漏洞类型，还能准确定位到漏洞触发的具体代码行。MaliVD从源代码中提取出代码序列、抽象语法树和代码属性图三种结构的特征，并使用三个对应结构的先进神经网络进行学习。为了提升漏洞类型识别和漏洞定位的能力，MaliVD引入了PoIs特征，通过在原始特征中标注与PoIs相关的语句，同时把其他位置施加掩码，掩码后的特征也通过相应结构的神经网络模型学习，最后将各个模型的输出连接起来，通过线性连接以及注意力排序的方式进行漏洞类型的识别和漏洞位置的判断。本文将MaliVD同多个先进的漏洞检测模型进行对比，在多个大型漏洞数据集上，MaliVD都取得了最优的漏洞检测和漏洞定位能力，保持了较高的准确率和F1分数，相对于现有方法减少了较多的漏报和误报，验证了MaliVD在现实环境下的应用价值。

关键词：漏洞检测；漏洞定位；多模态注意力；漏洞代码库

Abstract

Source code vulnerability detection is essential for identifying security risks withinsoftware code, playing a critical role in software testing and maintenance.This process is vital for ensuring software functionality and enhancing application security. With therapid development of the internet, software functionalities and their corresponding codes have become increasingly complex, leading to heightened complexity and detection dif-ficulty of software vulnerabilities. The application of deep learning in source code vul-nerability detection enables neural network models to automatically learn patterns fromvulnerable codes, savingcosts associatedwith manual inspection and the development ofhigh-performance vulnerability detection tools, thereby significantly improving detection efficiency. However, existing methods still face numerous challenges, such as reliance on low-quality datasets, learning irrelevant code features, inappropriate model designs,and difficulties in accurately locating vulnerabilities.

To address the scarcity of high-quality vulnerability datasets,this paper proposes anautomated framework for constructing a reliable vulnerability codebase. This framework extracts vulnerability information and candidate patch details from the CVE database and open-source repositories, then utilizes multiple static analysis tools, machine learning models, and large language models to conduct comprehensive analyses of both vulnerableand patched codes, ensuring accurate labeling. To annotate the specific locations of vul- nerabilities, we developed a set of Points of Interest (PoIs) features tailored to mainstreamvulnerability types. Based on PoIs features and combined with code revision records, wemeticulously annotated the locations and paths where vulnerabilities are triggered. Us-ing this framework, we constructed a high-quality vulnerability dataset named ReliVul.Compared with existing datasets, ReliVul shows significant improvements in dataset ba- sics, accuracy, uniqueness, and completeness, laying a solid foundation for subsequentresearch and filling the gap for high-quality vulnerability datasets in the field of vulnera- bility detection.

To tackle the limitations of current vulnerability detection models, this paper intro- duces MaliVD, a multi-modal attention-based source code vulnerability detection andlocalization model. MaliVD not only identifies specific vulnerability types but also ac-curately pinpoints the exact lines of code where vulnerabilities are triggered. MaliVD

extracts three types of structural features from source code: code sequences, abstractsyntax trees, and code property graphs, using advanced neural networks tailored to eachstructure for learning. To enhance vulnerability type recognition and localization capabil- ities, MaliVD incorporates PoIs features by marking statements related to PoIsin originalfeatures while masking other positions. Masked features are also learned through cor- responding neural network models. Finally, outputs from various models are integrated using linear connections and attention-based ranking for vulnerability type identificationand location determination. When compared with several state-of-the-art vulnerabilitydetection models across multiple large datasets, MaliVD demonstrated superior vulner-ability detection and localization capabilities, maintaining high accuracy and F1 scores. Relative to existing methods, MaliVD significantly reduced false negatives and positives, validating its practical application value in real-world environments.

**Keywords:** vulnerability detection;vulnerability localization;multimodal attention;vul- nerability dataset

目录

[摘 要](#bookmark1) [I](#bookmark1)

[Abstract](#bookmark2) [II](#bookmark2)

[目 录](#bookmark3) [IV](#bookmark3)

[插图清单](#bookmark8) [VII](#bookmark8)

[附表清单](#bookmark10) [VIII](#bookmark10)

[符号和缩略语说明](#bookmark12) [IX](#bookmark12)

第 1 章 [引言](#bookmark14) [1](#bookmark14)

1.1 [研究背景](#bookmark16) [1](#bookmark16)

1.2 [研究意义](#bookmark18) [3](#bookmark18)

1.3 [研究内容与贡献](#bookmark20) [4](#bookmark20)

1.4 [论文组织架构](#bookmark22) [5](#bookmark22)

第 2 章 [研究现状与文献综述](#bookmark24) [7](#bookmark24)

2.1 [漏洞代码库的构建](#bookmark26) [7](#bookmark26)

2.1.1 [基于人工或工具生成](#bookmark28) [7](#bookmark28)

2.1.2 [基于开源代码库爬取](#bookmark30) [9](#bookmark30)

2.1.3 [基于漏洞库抽取](#bookmark32) [9](#bookmark32)

2.1.4 [当前漏洞代码库构建方法存在的问题](#bookmark34) [10](#bookmark34)

2.2 [源代码漏洞检测技术](#bookmark36) [11](#bookmark36)

2.2.1 [传统源代码漏洞检测方法](#bookmark38) [11](#bookmark38)

2.2.2 [基于深度学习的源代码漏洞检测方法](#bookmark40) [12](#bookmark40)

2.2.3 [基于深度学习的源代码漏洞定位方法](#bookmark42) [14](#bookmark42)

2.2.4 [当前漏洞检测模型存在的问题](#bookmark44) [15](#bookmark44)

2.3 [本章小结](#bookmark46) [15](#bookmark46)

第 3 章 [代码行级的漏洞代码库构建框架](#bookmark48) [17](#bookmark48)

3.1 [代码行级的漏洞代码库构建](#bookmark50) [17](#bookmark50)

3.1.1 [模型概述](#bookmark52) [17](#bookmark52)

3.1.2 [源代码获取](#bookmark54) [18](#bookmark54)

3.1.3 [漏洞补丁判定](#bookmark56) [22](#bookmark56)

3.1.4 [漏洞行号标注](#bookmark58) [25](#bookmark58)

3.2 [实验设置](#bookmark60) [27](#bookmark60)

3.2.1 [比较基线](#bookmark62) [28](#bookmark62)

3.2.2 [评价指标](#bookmark64) [28](#bookmark64)

3.3 [实验结果](#bookmark66) [29](#bookmark66)

3.3.1 [基本信息对比实验](#bookmark68) [30](#bookmark68)

3.3.2 [准确性对比实验](#bookmark70) [31](#bookmark70)

3.3.3 [唯一性对比实验](#bookmark72) [32](#bookmark72)

3.3.4 [完整性对比实验](#bookmark74) [33](#bookmark74)

3.4 [本章小结](#bookmark76) [34](#bookmark76)

第 4 章 [基于多模态注意力的源代码漏洞检测技术](#bookmark78) [35](#bookmark78)

4.1 [基于多模态注意力的源代码漏洞检测](#bookmark80) [35](#bookmark80)

4.1.1 [模型概述](#bookmark82) [35](#bookmark82)

4.1.2 [数据预处理与特征提取](#bookmark84) [36](#bookmark84)

4.1.3 [多模态注意力模型设计](#bookmark86) [38](#bookmark86)

4.1.4 [分类模型设计](#bookmark88) [42](#bookmark88)

4.1.5 [定位模型设计](#bookmark90) [43](#bookmark90)

4.2 [实验设置](#bookmark92) [43](#bookmark92)

4.2.1 [数据集](#bookmark94) [43](#bookmark94)

4.2.2 [比较基线](#bookmark96) [44](#bookmark96)

4.2.3 [评价指标](#bookmark98) [45](#bookmark98)

4.3 [实验结果](#bookmark100) [47](#bookmark100)

4.3.1 [漏洞检测实验](#bookmark102) [47](#bookmark102)

4.3.2 [漏洞定位实验](#bookmark104) [51](#bookmark104)

4.3.3 [消融实验](#bookmark106) [57](#bookmark106)

4.3.4 [参数调优实验](#bookmark108) [59](#bookmark108)

4.4 [本章小结](#bookmark110) [62](#bookmark110)

第 5 章 [总结与展望](#bookmark112) [63](#bookmark112)

5.1 [论文工作总结](#bookmark114) [63](#bookmark114)

5.2 [未来工作计划](#bookmark116) [64](#bookmark116)

[参考文献](#bookmark118) [66](#bookmark118)

附录 A [与漏洞修复相关的提交可能包含的关键词](#bookmark120) [71](#bookmark120)

[致 谢](#bookmark122) [72](#bookmark122)

[声 明](#bookmark124) [73](#bookmark124)

[个人简历、在学期间完成的相关学术成果](#bookmark126) [74](#bookmark126)

[指导教师评语](#bookmark128) [75](#bookmark128)

[答辩委员会决议书](#bookmark130) [76](#bookmark130)

插图清单

图 1.1 [国家信息安全漏洞库 2015-2024 年漏洞收录情况[1]](#bookmark132) [1](#bookmark132)

图 1.2 [2024 年漏洞威胁等级和产生原因统计](#bookmark134) [2](#bookmark134)

图 1.3 [本文各章节组织结构图](#bookmark136) [6](#bookmark136)

图 3.1 [基于 CVE 数据库和开源代码库抽取的自动化漏洞样本构建模型](#bookmark138) [18](#bookmark138)

图 4.1 [MaliVD：基于多模态注意力的源代码漏洞检测与定位模型](#bookmark140) [35](#bookmark140)

图 4.2 [代码属性图（CPG）生成过程示例](#bookmark142) [36](#bookmark142)

图 4.3 [MaliVD 与对比方法在漏洞定位上的 Top-k 统计结果](#bookmark144) [51](#bookmark144)

图 4.4 [各方法在漏洞定位 IFA 指标上的统计结果](#bookmark146) [53](#bookmark146)

图 4.5 [各方法在六类主流漏洞定位 IFA 指标上的统计结果](#bookmark148) [55](#bookmark148)

图 4.6 [MaliVD 在不同嵌入维度和隐藏层维度下的性能表现](#bookmark150) [60](#bookmark150)

图 4.7 [MaliVD 在不同批量大小和学习率下的性能表现](#bookmark152) [61](#bookmark152)

附表清单

表 2.1 [漏洞代码库构建方法总结](#bookmark154) [8](#bookmark154)

表 2.2 [深度学习技术在源代码漏洞检测中的应用总结](#bookmark156) [13](#bookmark156)

表 3.1 [从 CVE 数据库中提取的有关漏洞的关键信息](#bookmark158) [20](#bookmark158)

表 3.2 [从开源代码平台获取的有关提交的关键信息](#bookmark160) [20](#bookmark160)

表 3.3 [从开源代码平台获取的有关代码的关键信息](#bookmark162) [22](#bookmark162)

表 3.4 [CWE 分类、示例和对应的 PoIs 标准](#bookmark164) [26](#bookmark164)

表 3.5 [不同层次的代码克隆及解释](#bookmark166) [29](#bookmark166)

表 3.6 [ReliVul 与 5 个流行的漏洞数据集的基本信息对比](#bookmark168) [30](#bookmark168)

表 3.7 [各漏洞数据集在准确性、唯一性和完整性上的实验结果](#bookmark170) [32](#bookmark170)

表 4.1 [代码特征的节点信息和边信息](#bookmark172) [38](#bookmark172)

表 4.2 [各方法在 Juliet 和 Devign 数据集上的漏洞检测实验结果](#bookmark174) [48](#bookmark174)

表 4.3 [各方法在 DiverseVul 和 ReliVul 数据集上的漏洞检测实验结果](#bookmark176) [50](#bookmark176)

表 4.4 [消融实验结果](#bookmark178) [57](#bookmark178)

符号和缩略语说明

Sard 软件保障参考数据集（Software Assurance Reference Dataset）

CNVD国家信息安全漏洞共享平台（China National Vulnerability

Database）

NVD 美国国家漏洞数据库（National Vulnerability Database）

CVE 通用漏洞披露（Common Vulnerabilities and Exposures）

CWE 通用缺陷枚举（Common Weakness Enumeration）

NCS 自然代码序列（Natural Code Sequence）

AST抽象语法树（Abstract SyntaxTree）

CFG 控制流图（Control Flow Graphs）

DFG 数据流图（Data Flow Graphs）

CPG 代码属性图（Code Property Graphs）

PDG 程序依赖图（Program Dependence Graphs）

CNN卷积神经网络（Convolutional Neural Networks）

RNN循环神经网络（Recurrent Neural Networks）

GRU 门控循环单元（Gated Recurrent Unit）

LSTM 长短期记忆网络（Long Short-Term Memory）

Tree-LSTM 树形结构的长短期记忆网络（Tree-structural LSTM）

GCN 图卷积神经网络（Graph Convolutional Network）

GGNN 图循环神经网络（Gated Graph Neural Networks）

LLMs 大语言模型（Large Language Models）

GAT 图注意力网络（Graph Attention Network）

PoIs 兴趣点（Points of Interest），指漏洞可能触发的位置

ReliVul本课题提出的代码行级别的真实漏洞数据集（Real-world Line-

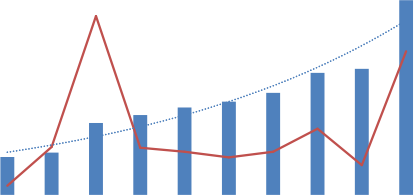
level Vulnerability dataset）

MaliVD本课题提出的基于多模态注意力的漏洞检测模型（Multimodal

Attention-basedLine-levelVulnerabilityDetector）

WLJAN 单词-标签联合注意力网络（Word-Label Joint Attention Network）

IFA 初始误报（Initial False Alarm）



40000

35000

30000

25000

20000

15000

10000

5000

0

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

第1章引言

在当今数字化时代，软件已成为支撑社会运转和经济发展的重要力量。随着软件功能和软件复杂度的不断增加，软件代码中的安全漏洞问题却日益凸显，软件漏洞对个人隐私、企业运营乃至国家安全构成了重要威胁，有效的软件漏洞挖掘技术就显得尤为重要。本章节为源代码漏洞检测技术的引言，首先介绍源代码漏洞检测技术的研究背景和研究意义，然后介绍本文主要的研究内容与主要贡献，最后简要介绍本文的组织架构，概述各章节的主要内容。

1.1研究背景

随着互联网的快速发展和各行各业对软件的依赖，软件和代码已经成为维持社会正常运作的基本组成成分。如今，人们的日常生活越来越离不开各种软件系统，软件质量的好坏直接影响了人们的日常生活，然而，软件的安全性却日益成为当今互联网时代的主要威胁。软件的安全性直接影响到国家的安全、经济的增长以及社会的稳定，人们对软件安全性的要求以及软件漏洞的检测技术的需求越来越高。图[1.1](#bookmark132)展示了从2015年到2024年，国家信息安全漏洞库[[1](#bookmark179)]（China National Vulnerability Database，CNVD）中统计的漏洞数量，可以看出，近十年来CNVD收录的漏洞数量呈现出指数增长的趋势。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2015年 | 2016年 | 2017年 | 2018年 | 2019年 | 2020年 | 2021年 | 2022年 | 2023年 | 2024年 |

新增数量7735 8622 14671 16307 17833 19048 2082724921 25748 39750

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 增长率 | -5.9% | 11.5% | 70.2% | 11.2% | 9.4% | 6.8% | 9.3% | 19.7% | 3.3% | 54.4% |

图1.1国家信息安全漏洞库2015-2024年漏洞收录情况[[1](#bookmark179)]

80.0% 70.0% 60.0% 50.0% 40.0% 30.0% 20.0% 10.0% 0.0%

-10.0%

近年来漏洞数量的增长速度正在加快，特别是2024年，漏洞的收录数量达到了39750个，相比前一年增加了54.4%。软件漏洞数量的加快增长， 表明如今软件

安全仍面临严峻的挑战。一方面，随着当今互联网设备的激增和软件数量的增长，潜在的漏洞数量相应增加，使得漏洞数量不断增多；另一方面，新型软件安全技术也不断涌现，使得更多的漏洞能够被及时发现和披露，这种趋势可能会导致软件漏洞在未来继续快速增长。

图[1.2](#bookmark134)展示了2024年收录漏洞的威胁等级和产生原因的统计情况。首先分析2024年CNVD收录漏洞的威胁等级占比情况，如图[1.2(a)](#bookmark134)所示，2024年收录的漏洞中，低危漏洞占比只有4.20%，而中高危漏洞占比却超过95%，其中高危漏洞占据了约一半的比例。这些漏洞涉及多个国家，关系到通信、医疗、交通、教育等关键基础设施领域，关系到个人生活、企业经营以及国家安全，当今网络安全局势仍旧严峻。

4.20%

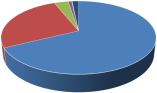


46.94%

48.86%

低危中危高危

(a) 漏洞威胁等级占比

3.50% 0.90% 0.10%0.01%1.59%

26.20%

67.70%

输入验证错误边界条件错误访问验证错误

设计错误

竞争条件

配置错误 其他错误

(b) 漏洞产生原因统计

图1.2 2024年漏洞威胁等级和产生原因统计

[深入分析漏洞的产生原因，如图1.2(b)](#bookmark134)所示，设计错误占比高达67.70%，是导致漏洞产生的主要原因。这表明，如今软件开发过程中，在软件开发初期的设计阶段对安全性的考量仍严重不足。许多漏洞并非是由于程序逻辑的错误或者配置错误，而是由于在软件开发初期就存在设计缺陷。因此，在软件开发的初期阶段加强安全性设计，提高对源代码的漏洞检测能力，对于提升整个软件周期的安全性有重要的帮助。

近年来，随着开源软件的快速发展，基于源代码的静态分析技术也有较大进步。通过对源代码进行静态分析，可以在软件开发初期就扫描出大部分漏洞，从而减少后续软件测试和维护的人力和财力。有研究表明，漏洞检测与修复所耗费的[成本通常占软件开发总成本的 50%-70%[2](#bookmark180)[]。尽管一些静态分析工具如 CodeQL[3](#bookmark181)]、Coverity[[4](#bookmark182)]、Cppcheck[[5](#bookmark183)]和Flawfinder[[6](#bookmark184)]等能够通过扫描源代码发现漏洞，但这些工具主要依赖领域专家根据特定漏洞模式制定的规则，随着软件的更新和新漏洞的出现，这些规则需要不断更新。同时，随着软件复杂度的增加，人工制定漏洞规则的工作量也不断增加，且不同专家在制定漏洞规则时存在主管因素，从而导致现有工具存在大量的漏报、误报。另外，随着新型漏洞模式的不断涌现，现有静态

分析工具的局限性更加明显，同一种漏洞可能有不同的触发条件，这些工具仅仅进行规则匹配，而缺乏对漏洞特定模式的理解，使得这些工具难以辨别代码是否真正存在漏洞。

随着机器学习和深度学习技术的广泛应用，越来越多的方法被提出并应用于源代码漏洞检测。一些方法尝试将代码视为自然语言处理问题，通过对代码中的词汇或符号进行编码，然后使用机器学习和深度学习模型来识别代码中的漏洞。然而，代码不同于自然语言，它是一种复杂的结构数据，代码中蕴含着复杂的控制流和数据流关系，将代码视作自然语言进行处理是不妥的。为了解决这一问题，一些方法从代码中提取出抽象语法树（Abstract Syntax Tree，AST）、控制流图（Control Flow Graph，CFG）和数据流图（Data Flow Graph，DFG）等代码特征，并利用机器学习和深度学习模型从中学习[[7](#bookmark185)]。虽然这种方法提高了漏洞检测的速度，但漏洞检测效果并不令人满意，且缺乏实际应用价值[[8](#bookmark186)]。现有的漏洞检测方法存在的主要问题包括：1）现有漏洞代码库存在重复、不平衡、复杂度低、样本少等缺点，[使得训练有效的模型变得困难[9](#bookmark187)]；2）漏洞检测模型常常学习到不相关的代码特征，影响模型捕捉漏洞本质的能力[[10](#bookmark188)]；3）设计的模型往往缺乏针对性、合理性和可解释性，简单地从其他领域适应模型而未充分考虑其适用性[[10](#bookmark188)]；4）模型无法准确地[定位漏洞发生的具体位置[11](#bookmark189)]。因此，如何设计一个自动化的漏洞代码库构建框架，以及设计一个低漏报、误报的基于深度学习的漏洞检测与定位模型，是当前网络环境下亟待解决的问题。

1.2研究意义

研究源代码漏洞检测的意义可以从如下三个研究内容阐述：自动化构建高质量的漏洞数据集，提高漏洞类型识别的准确性，以及提高模型定位漏洞代码位置的能力，这些研究内容对于漏洞检测领域、互联网的发展以及社会发展具有重要[的意义[8](#bookmark186)]。

首先，通过自动化的数据构建方法收集一个含有足够样本和足够准确的漏洞代码库填补了该研究领域的空缺，有一个统一的、准确的漏洞代码库可以帮助研究人员公平地评估和对比不同的漏洞检测算法和技术的性能，这有助于推动漏洞检测的技术发展和算法改进。另外通过公开和分享漏洞代码样本，人们可以了解常见漏洞的原理、遵守网络规范，从而提高开发者和用户对于网络安全的意识，促进整个社会对于网络安全的重视，从而形成良好的互联网安全文化氛围[[12](#bookmark190)]。

其次，从源代码中识别潜在的漏洞是确保软件安全的重要步骤，传统的基于人工漏洞挖掘和静态分析工具扫描的方法已难以满足如今软件安全的需求。深度

学习具有挖掘深层特征的能力，对专家提取漏洞特征的依赖不强，所以研发出一种基于深度学习的源代码漏洞检测方法至关重要。一方面，在软件开发初期，开发者在软件设计时可以实时扫描自己设计的软件是否存在安全漏洞，并及时修复，避免了后期由于软件功能复杂后难以检测的问题[[12](#bookmark190)]；另一方面，在软件测试与修复阶段，软件安全工作人员可以利用漏洞检测工具进行漏洞排查，参考工具检查的结果，安全员可以更方便的进行漏洞修复。总之，基于深度学习的源代码漏洞检测贯穿整个软件开发的周期，对于节省人力、计算资源和确保软件的安全性具有重要作用。

最后，如果利用深度学习进行漏洞检测的同时可以同时输出漏洞触发的位置，那将大大提高开发者和安全人员进行漏洞修复的效率。如今软件的功能愈发复杂，软件的代码量也越来越大，这使得人工检查漏洞触发的位置变得困难，漏洞修复工[作的难度变大。所以， 在这种大型项目中快速定位漏洞的位置将显得尤为重要[11](#bookmark189)]。如果基于深度学习的方法只能判断一段代码是否含有漏洞，而不指明漏洞的位置，那也不利于进行漏洞原理的理解与修复，其作用可能还不如基于规则的静态分析工具。深度学习模型如果能够输出漏洞的触发位置，还可以证明其真正学习和理解了漏洞的模式，从而提高漏洞检测的准确性。

综上，构建足够高质量的漏洞数据样本和设计高效的源代码漏洞检测与定位模型对于工业和互联网的发展具有巨大的研究价值。对于促进技术进步、工业软[件开发效率、净化互联网环境以及提升用户的安全意识具有重要的意义[13](#bookmark191)]。期望通过本文的研究，能够在源头上解决软件安全问题，为软件安全和社会发展提供强有力的保障。

1.3研究内容与贡献

近年来，研究人员提出了多种基于机器学习和深度学习的漏洞检测方法，但现有方法的检测效果仍难以迎合如今软件安全的需求，现有方法的不足之处包括：依赖低质量漏洞数据集、学习不相关代码特征、不合适的模型设计以及模型难以定位到漏洞的位置等。为了解决现有方法的不足，本文在深入分析现有方法的局限性后，重点从以下两个方面展开研究：（1）研究如何自动化构建高质量的漏洞代码库；（2）通过多模态注意力机制设计高效的漏洞检测与定位模型。具体来说，本文的主要贡献如下：

**第一，提出了一种代码行级的自动化漏洞代码库构建框架，并基于此框架构建了一个高质量漏洞数据集ReliVul。**针对现有漏洞数据集普遍存在的问题，例如样本数量少、标签不准确、代码重复率高、标注粒度较粗等，本文提出了一个代码

行级别的自动化漏洞代码库构建框架。该框架能够从大型通用漏洞披露（CommonVulnerabilities and Exposures，CVE）数据库[[14](#bookmark192)]以及开源代码库中提取漏洞信息及其对应的代码提交记录，然后利用多种静态分析工具、机器学习模型和大语言模型对提取的信息进行综合分析，来标注漏洞和补丁代码。为了进一步提升数据集的标注粒度，我们通过深入分析各类漏洞的模式，定义了一组漏洞兴趣点（Points of Interest，PoIs）特征[[11](#bookmark189)]，即漏洞可能触发的位置信息。将漏洞PoIs特征和代码的修订历史结合，就可以对漏洞触发的具体位置进行标注。基于该框架，我们构建了一个真实世界代码行级的漏洞数据集ReliVul。实验结果表明，与现有的流行数据集相比，ReliVul在数据量、准确性、唯一性和完整性等方面有较大提升。ReliVul的提出为后续研究奠定了基础，也为漏洞检测领域提供了一个更为可靠的数据集基准。

**第二，提出了一种基于多模态注意力机制的源代码漏洞检测模型MaliVD，支持漏洞类型识别和漏洞触发位置的定位。**为解决现有漏洞检测模型在特征提取、模型设计和检测粒度上的问题，本文提出了一种基于多模态注意力机制的源代码漏洞检测和定位模型MaliVD。MaliVD利用代码的语言序列特征、树形语法特性和图形结构特性从代码中提取出序列、树形和图形三种结构的特征，并利用对应结构的三种先进的神经网络模型进行特征学习，在三种模型提取到代码的高层表示后，将它们拼接起来并使用一个线性层连接到待分类的标签上。另外，本文利用漏洞的PoIs特征，在源代码中标注了漏洞可能触发的位置，并使用特征掩码技术，使模型更关注那些可能触发漏洞的语句。MaliVD能够根据其注意力机制输出每行代码可能触发漏洞的概率，从而实现对漏洞位置的输出。本文将MaliVD同8个先进的漏洞检测模型对比，实验结果显示，在四个大型漏洞数据集上，MaliVD相对现有方法能够更为准确地识别漏洞的类型和定位漏洞的位置，减少了较多的漏报和误报，验证了MaliVD的准确性和泛化能力，使其具有较高的实际应用价值。

综上，本文所提出的漏洞代码库ReliVul和漏洞检测模型MaliVD为源代码漏洞检测领域提供了新的技术支持，这些技术不仅拥有较强的实验结果，也具有较高的实际应用价值。通过将这些技术应用于实际项目，能够助力构建更加安全可靠的软件生态系统，为促进互联网发展和保障网络安全具有重要意义。

1.4论文组织架构

本文以基于多模态注意力机制的源代码漏洞检测技术为研究目标，全文总共包括五个章节，结构安排如图[1.3](#bookmark136)所示。

第[1](#bookmark14)章为引言，主要分析源代码漏洞检测的研究背景及意义，并引出了本文的

引言（第1章）



研究现状与文献综述

（第2章）



代码行级的漏洞代码库构建框架

（第3章）

基于多模态注意力的源代码漏洞检 测技术（第4章）



总结与展望（第5章）

图1.3本文各章节组织结构图

主要研究内容和贡献。最后简单介绍了本文的组织架构。

第[2](#bookmark24)章从漏洞代码库的构建与基于深度学习的源代码漏洞检测模型两方面对当下国内外研究现状进行阐述，并分析了当前方法存在的不足之处。

第[3](#bookmark48)章介绍了本文代码行级的漏洞代码库构建框架，并通过实验验证了所构建的漏洞代码库在多项指标上的优越性。

第[4](#bookmark78)章详细介绍了本文提出的基于多模态注意力机制的源代码漏洞检测模型，并通过丰富的对比实验验证了模型在漏洞检测和漏洞定位上突出的性能。

第[5](#bookmark112)章对全文的工作进行了概括总结，并描述了未来工作展望。

第2章 研究现状与文献综述

本文将从源代码漏洞代码库的构建与基于深度学习的漏洞检测模型这两个方面对当下国内外研究现状进行阐述。对于漏洞代码库，根据其数据来源分成了基于人工或工具生成、漏洞库抽取和开源代码库爬取等方向[[13](#bookmark191)]；对于漏洞检测模型，首先对传统方法进行概述，然后根据论文采取的代码表征的结构进行描述，然后描述现有模型的在漏洞定位上的尝试。另外，本章也对目前该领域现有工作存在的问题进行了总结和探讨。

2.1漏洞代码库的构建

从数据来源上划分，当前漏洞代码库的构建来源主要分为三类：基于人工或工具生成、[CVE 漏洞库抽取与开源代码库爬取，表2.1](#bookmark154)梳理了这三类漏洞代码库构建的常见数据集的信息统计。本节将首先介绍这四种方法和相关工作，并总结现有方法的不足之处。

2.1.1基于人工或工具生成

[早期，为了构建漏洞数据集，通常采用人工编写测试样例[15](#bookmark193)-[16](#bookmark194)]或者利用静态[分析工具[17](#bookmark195)-[18](#bookmark196)]生成的方式。虽然利用这些方法构建的数据样本缺乏真实性，但为源代码漏洞检测提供了最初的训练样本，也为后续该领域的发展奠定了基础。

首先，从人工构造的角度来看，在软件开发中，人们常常根据实际需求和特定场景手动设计一些测试样例，例如，SARD（Software Assurance Reference Dataset）[[19](#bookmark197)]由美国国家标准与技术研究院发布，汇集了来自工业界、教育领域以及个人编写的测试样例，涵盖了包括C/C++、Java在内的多种编程语言。截至目前，该数据集包含了大约20万条测试样例，涵盖了许多常见的漏洞类型，对于大部分的漏洞样本（Bad），还提供了补丁代码（Good），而且对漏洞的位置进行了详细的标注，但其样本的复杂度很低，数据依赖关系较简单，难以适用于复杂的场景。此外，OWASP [Benchmark[20](#bookmark198)]是一个专门针对Java项目的综合测试样例集合，旨在为软件漏洞分析工具提供自动评估的支持。尽管人工构造的漏洞数据集能够提供多样化的测试样例，但其主要局限在于所含漏洞流程相对简单，复杂度较低，无法完全反映真实世界中的代码漏洞情况。

另一方面，由于大部分漏洞都有特定的漏洞模式和触发条件，将人为制定的漏洞规则集成到工具中，利用工具就可以快速生成大量漏洞样本。Russell等

人[[17](#bookmark195)]的研究展示了如何结合Juliet [[19](#bookmark197)]、Debian Linux[[21](#bookmark199)[]和 GitHub[22](#bookmark200)]等资源，运用Clang[[23](#bookmark201)]、Cppcheck[[5](#bookmark183)]和Flawfinder[[6](#bookmark184)]等开源静态分析工具进行漏洞检测，并采用投票机制来确定最终的标签。然而，尽管静态分析工具可以快速产生大量的漏洞样本，它们主要依赖于专家预定义的漏洞规则，生成的样本缺乏多样性和真实性，且准确性也难以保证，所以基于静态工具生成的样本已难以作为评估更先进漏洞检测模型的标准。

人工构造的漏洞数据集虽然具有较高的标签准确性，但其复杂性和样本数量有限；基于工具的漏洞数据构建方法虽能提供大量样本，但其准确性和真实性难以保证。虽然这些样本对于漏洞检测模型的设计提供了最初的数据支撑，但当需要构建更加有效的源代码漏洞检测系统时，这些方法已无法作为模型训练和验证的样本来源。

表2.1漏洞代码库构建方法总结

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 来源 | 名称 | 类型a | 粒度 | 总样本 | 漏洞样本b | 占比（%） |
| 人工或工 具生成 | VulSniper [[16](#bookmark194)] | - | 函数 | 28049 | 10561 | 37.7 |
| AppMine [[15](#bookmark193)] | 7 | 文件 | 1698 | 932 | 54.9 |
| Early-DL [[17](#bookmark195)] | 149 | 函数 | 1286262 | 87804 | 6.8 |
| PROMISE [[18](#bookmark196)] | - | 文件 | 8118 | 2887 | 35.6 |
| 漏洞库 抽取 | QuixBugs [[24](#bookmark202)] | 40 | 代码行 | 80 | 40 | 50.0 |
| “VulDeePecker[[25](#bookmark203)] | 40 | 切片 | 181641 | 43119 | 23.7 |
| SySeVR [[26](#bookmark204)] | 126 | 代码段 | 14511 | - | - |
| Big-Vul [[27](#bookmark205)] | 91 | 代码行 | 264919 | 11823 | 4.5 |
| DeepVD [[10](#bookmark188)] | 5 | 函数 | 26260 | 13130 | 50.0 |
| CrossVul [[28](#bookmark206)] | 107 | 文件 | 134126 | 6884 | 5.1 |
| DiverseVul [[9](#bookmark187)] | 150 | 函数 | 330492 | 18945 | 5.7 |
| ReposVul [[29](#bookmark207)] | 236 | 代码行 | 10639 | 5940 | 56.8 |
| 代码库 爬取 | BugDetection [[30](#bookmark208)] | - | 函数 | 4973000 | 1824000 | 36.7 |
| Devign [[7](#bookmark185)] | - | 函数 | 58965 | 27652 | 46.9 |
| DeepWukong [[31](#bookmark209)] | - | 切片 | 140670 | 44521 | 31.6 |
| D2A [[32](#bookmark210)] | - | 函数 | 1295623 | 18657 | 1.4 |
| ReVeal [[33](#bookmark211)] | - | 代码段 | 18169 | 1657 | 9.1 |

a表示数据集内含有的漏洞类型，若为“-”，表示未区分漏洞类型。b漏洞样本列为“-”表示不确定是否是漏洞的样本。

2.1.2基于开源代码库爬取

随着开源软件和开源代码平台的快速发展，开发人员在更新软件版本或修复软件问题时会提交变更信息。通过这些变更信息和软件修复的版本交替使得从开源代码库中爬取漏洞信息成为可能。[GitHub[22](#bookmark200)]是一个广泛使用的开源代码平台，平台内包含大量开发人员在更新代码仓库时留下的版本变更信息。在版本变更的过程中如果某个版本是包含漏洞的，而提交人员通过一次代码变更进行了漏洞修复，那么变更前后的代码就可以标注为有漏洞和无漏洞的。虽然代码没有绝对的安全性，修复后的代码也可能包含新的安全问题，但这些样本仍包含对于漏洞的修复记录，对于训练漏洞检测模型也是具有帮助的。通过识别这些提交信息，判断哪些提交是和漏洞修复相关的，就可以进行漏洞样本和补丁代码的收取。但是，开发者在进行代码修复时，往往不会详细描述修复的具体原因或漏洞的具体类型，这就给识别与漏洞相关的提交带来了困难，所以，如何有效地鉴别这些提交并从中提取漏洞信息成为扩充漏洞样本的重要研究方向。

在研究初期，一般通过提交信息的关键词过滤进行粗略标注，如BugDetec- tion[[30](#bookmark208)]数据集，尽管其数据量庞大，但准确率难以保证。为了提高准确率，De-vign[[7](#bookmark185)]和DeepWukong[[31](#bookmark209)]是两个经过人工标注和审查的数据集，但这种标注方法耗费人力且数据样本数较少。后来，一些工作通过将机器学习模型应用于漏洞代码库构建中，以判断提交信息是否与漏洞修复有关。例如，使用支持向量机（Support Vector Machine, SVM）、K近邻（K-Nearest Neighbor, KNN）和随机森林（Random [Forest, RF）等方法进行识别[34](#bookmark212)-[35](#bookmark213)]，或者训练多个模型进行投票。然而，这种方法的准确率仍然难以令人满意，且难以标注漏洞的具体类型。

2.1.3基于漏洞库抽取

中国国家信息安全漏洞库[[1](#bookmark179)]与美国国家漏洞数据库（NVD）[[36](#bookmark214)]是两个广泛认可的漏洞数据库。CNVD由中国的计算机网络应急技术处理协调中心联合国内的重要信息系统单位、电信运营商、网络安全公司、软件开发商以及互联网企业共同建立，旨在共享信息安全漏洞的知识。NVD则由美国国土安全部支持的联邦计算机应急准备小组维护，它不仅是权威的源代码漏洞数据库之一，还被广泛应用于各个领域。除此之外，Mend[[37](#bookmark215)]也是一个重要的安全数据库，它从多个渠道汇总了丰富的漏洞信息，并提供了详细的修复指南。这些数据库不仅同步CVE发布的最新条目，还在CVE的基础上增加了CWE[[38](#bookmark216)]分类、安全评分、影响评级、修复建议等信息，并通过人工审核将漏洞与原始项目的信息进行关联。

Zou 等人[[25](#bookmark203)]的研究展示了如何从NVD和SARD中提取基于C/C++语言的切

片级漏洞数据集，选取了一些开源项目的相关漏洞，并考虑到了数据依赖和控制依赖。然而，这个数据集中实际来自NVD的数据不足1%，限制了其应用价值。相比之下，Big-Vul [[27](#bookmark205)]和DeepVD[[10](#bookmark188)]从大型CVE数据库中收集数据，每个CVE条目都包含了指向具体漏洞修复提交的链接，允许研究人员收集提交前后版本的代码，进而识别出大型C/C++项目中的漏洞函数。此外，[Lin 等人[24](#bookmark202)]利用Quixey挑战赛平台[[39](#bookmark217)]，翻译了40个Python和Java程序，创建了一个代码行级别的小型漏洞数据集。尽管近年来越来越多的研究工作开始从CVE漏洞数据库中抽取数据[（如 CrossVul [28](#bookmark206)]、DiverseVul [[9](#bookmark187)]和ReposVul [[29](#bookmark207)]），显著提升了样本的数量和准确性，但依然存在标注不准确、缺乏漏洞补丁信息以及未明确标注漏洞触发位置等问题。Bhandari等人的研究[[40](#bookmark218)]表明，在NVD的约20万条漏洞记录中，只有5000多条能与具体的代码提交相匹配，这进一步凸显了高质量漏洞样本稀缺的问题。

2.1.4 当前漏洞代码库构建方法存在的问题

根据表[2.1](#bookmark154)的总结结果，可以看到当前的漏洞代码库在各项技术环节均存在较大的差异，如不同的粒度、样本数量等。现有方法存在的主要问题如下：

• **人工及工具生成的样本缺乏实际应用价值**：通过人工构造和分析工具生成的漏洞样本往往与现实生活中的代码存在较大差距，使得这些样本缺乏实际的应用价值。同时这些样本通常缺乏复杂的代码结构，其中的数据和控制依赖关系较为简单，基于这些数据集训练出的模型在真实项目中的表现难以取得令人满意的效果。

• **CVE数据库中的样本有限且缺少代码行号**：CVE漏洞数据库中的样本虽然可靠性较高，但样本数量较少，而且这些数据库大多只是记录了漏洞代码的链接，缺少详细的漏洞补丁和漏洞触发位置的标注信息。这些问题使得利用这些数据进行有效的漏洞检测和漏洞定位难以进行。

• **开源代码库爬取数据的准确性难以保证**：开源代码库中的代码复杂多样，仅仅根据提交信息和代码变更信息收集的漏洞样本难以保证其准确性。而且这些数据集面临正负样本数量不平衡以及难以标注漏洞类型和触发位置等问题。这些问题影响了数据集的质量，通过此类数据训练的漏洞检测模型难以学习到真正的漏洞知识，也难以训练得到更强大的模型。

• **样本完整性问题**：现有的数据集往往存在样本不完整的情况。本文通过分析相关开源漏洞代码库的内部代码，发现许多漏洞样本不包含完整的漏洞路径。因为数据样本往往是根据漏洞补丁位置（diff中修改的函数）来决定的，而补丁位置并不一定与漏洞在同一函数中，导致样本的漏洞路径不完整。此外，一些函数级的样本缺少相关的调用关系，有些漏洞的触发路径涉及多个

函数，但数据样本仅包含一个函数，导致漏洞信息不全。由于项目的复杂性，一个程序中可能多个函数组合起来是无漏洞的（假设其中某个函数进行了检验或防护），如果在抽取样本时只抽取其中某个函数，则很可能是缺少相关检验防护的，导致有漏洞的样本被标注为无漏洞。

2.2源代码漏洞检测技术

源代码漏洞检测技术经历了从传统的数据流和控制流分析，到基于规则的分析工具，再到机器学习和深度学习方法的应用。本节首先简述传统漏洞检测方法，然后阐述基于深度学习的漏洞检测技术，并对当前模型在漏洞定位上的表现和当前该领域存在的问题进行了探讨。

2.2.1传统源代码漏洞检测方法

早期的源代码漏洞检测主要是利用基于规则的静态分析工具进行，这些工具通过分析源代码的词法结构、语法特性、控制流以及数据流等多方面的信息来识别代码中的漏洞。例如，[CPPCheck[5](#bookmark183)]、[CodeQL[3](#bookmark181)]和Coverity[[4](#bookmark182)]等是典型漏洞检测工具，在工业界被广泛使用。但是这些工具通常由专业机构开发，并依赖于专家手工定义的漏洞模式和制定的漏洞规则。如今软件系统的日益复杂，漏洞的模式和规则也更加复杂和难以提取，大大增加了规则制定的成本。另外，不同专家对于不同的漏洞可能存在不同的理解，导致提取的规则和开发的工具具有主观性，难以作为通用的漏洞检测工具。虽然这些工具在漏洞检测领域被广泛使用，但大量的漏洞、误报问题仍需要人工排查，大大影响了软件漏洞挖掘和修复的效率。

随着机器学习技术的进步，一些研究人员开始将机器学习模型应用在源代码漏洞检测领域。[Hindle 等人[41](#bookmark219)]发现编程语言与自然语言在统计特性上有许多相似之处，且编程语言表现出更高的规整性，验证了代码具有“自然性”的假说。基于这一理论，一些研究尝试从代码中提取“语言类”特征，并采用机器学习模型训练和预测的方法进行漏洞挖掘。例如，VCCFinder[[42](#bookmark220)]通过分析代码的结构来提取代码特征，并使用支持向量机对独热编码后的特征进行训练，从而识别代码中是否包含漏洞。然而现有的大多数机器学习方法仍然依赖于专家设计的特征作为输入，影响了漏洞检测的效率。另外当前的机器学习技术在实际环境中的误报率仍然较高，无法完全满足现实世界的需求。这主要是因为传统的机器学习模型在挖掘深层特征方面的能力有限，降低了其在复杂漏洞检测任务中的有效性[[43](#bookmark221)]。因此，尽管这些早期的努力为后续的研究奠定了基础，但在效率和准确性上仍有很大的改进空间。

2.2.2基于深度学习的源代码漏洞检测方法

深度学习首先应用在图像识别和自然语言处理领域，并取得了突出的成果，深度学习方法不需要专家设计的特征，能够自动挖掘代码深层的特征。近年来，众多研究致力于探索如何将深度学习应用于源代码漏洞检测中，并取得了不少成果。[2.2](#bookmark156)总结了现有的具有代表性的基于深度学习的源代码漏洞检测技术。

早期的工作主要基于自然代码序列（Natural Code Sequence, NCS）进行学习，采用卷积神经网络（Convolutional Neural Networks,CNN）和循环神经网络（Recurrent [Neural Networks, RNN）这类基于序列的模型[17](#bookmark195),[25](#bookmark203)[,45](#bookmark222)-[46](#bookmark223),[51](#bookmark224)]。然而，这些模型忽略了代码中的结构信息，导致模型难以学习和利用各种上下文语义信息。

代码除了具有自然语言的特征，它还包含丰富的结构化信息，例如数据的流动和控制关系，所以后续的研究开始探索新的代码结构化表征，并采用图神经网络（Graph Neural Networks, GNN）进行漏洞识别。早期通用的代码结构化表征包括抽象语法树（Abstract Syntax Tree,AST）[[7](#bookmark185)[]、控制流图（Control Flow Graphs, CFG）[44](#bookmark225)]、数据流图（Data Flow Graphs, DFG）[[44](#bookmark225)]和程序依赖图（Program Dependence Graphs, [PDG）[57](#bookmark226)]，其中PDG是结合了CFG和DFG的一种复合代码表示。更先进的代码表征包括跨文件图表征PDGcross[[62](#bookmark227)]、代码属性图（Code Property Graphs, CPG）[[63](#bookmark228)]和带有原始代码顺序信息的[[7](#bookmark185)]。CPG将AST、CFG和PDG的边信息融合到一张图中，包含更多的信息，被广泛应用于现有方法中。在模型方面，部分工作使用门控循环单元（Gated Recurrent Unit,GRU）[[48](#bookmark229)]、树形结构的长短期记忆网络（Tree-structural Long Short-Term Memory, Tree-LSTM）[[10](#bookmark188),[48](#bookmark229)[,64](#bookmark230)]、图卷积神经网络（GraphConvolutional Network, GCN）[[7](#bookmark185),[33](#bookmark211)]进行学习各类图结构特征。GCN可以将节点信息传播到相邻节点，学习每个节点的嵌入，然后在分类之前对其进行平均或求和。

然而，GCN在每次迭代中无法记忆（即传播）节点的原始特征向量，并且难[以学习输入结构中的远距离关系。为了解决这个问题，后续的研究[47](#bookmark231)-[49](#bookmark232),[58](#bookmark233)]使用图[循环神经网络（Gated Graph Neural Networks, GGNN）[65](#bookmark234)]，GGNN基于GRU，利用循环层来记忆传递给邻居节点的信息。然而，在这些网络中，层数决定了传播迭代的次数，通常只有一到两次，导致模型的学习深度不够[[18](#bookmark196)]。近年来，大部分研究工作在神经网络模型上进行了改进，并取得了较好的结果，例如在GCN中添加残差网络（Residual Network, ResNet）[[52](#bookmark235)[]，添加注意力机制[57](#bookmark226),[59](#bookmark236)[,66](#bookmark237)]，改进GNN中[节点和边的学习过程使其对流信息敏感（Flow-Sensitive GNN, FS-GNN）[53](#bookmark238)]，以及利用消息传递神经网络（Message-Passing Neural Networks, MP-NN）[[11](#bookmark189)]进行代码分析和漏洞识别。然而，这类方法通常是将其他领域的模型套用在源代码漏洞分析上，并未深入分析漏洞模式和考虑模型是否真正适合学习这些代码表征。有些工作

表2.2深度学习技术在源代码漏洞检测中的应用总结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参考文献 | 年份 | 代码表征 | 模型 | 漏洞 定位 |
| NCS  AST  CFG  DFG  CPG  CUSa |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Russell’18 [[17](#bookmark195)] | 2018 | √ |  |  |  |  |  | CNN/RF | X |
| “VulDeePecker[[25](#bookmark203)] | 2019 |  |  |  |  |  | √ | BiLSTM | X |
| Devign [[7](#bookmark185)] | 2019 | √ | √ | √ | √ |  |  | GCN/DNN | X |
| Xia’19 [[44](#bookmark225)] | 2019 |  |  | √ | √ |  |  | BiLSTM | X |
| Li’20 [[45](#bookmark222)] | 2020 | √ |  |  |  |  |  | CNN | X |
| Zagane’20 [[46](#bookmark223)] | 2020 | √ |  |  |  |  |  | DNN | X |
| Funded [[47](#bookmark231)] | 2020 |  |  |  |  | √ |  | GNN/GRU | X |
| SySeVR [[26](#bookmark204)] | 2021 | √ |  |  |  |  | √ | BiRNN | X |
| IVDetect [[48](#bookmark229)] | 2021 | √ | √ | √ |  |  | √ | GRU/Tree-LSTM | √ |
| BGNN4VD [[49](#bookmark232)] | 2021 |  |  |  |  | √ |  | GNN/GRU | X |
| Vuldeelocator [[50](#bookmark239)] | 2021 | √ |  |  |  |  | √ | BiRNN | √ |
| ReVeal [[33](#bookmark211)] | 2021 |  |  |  |  | √ |  | GCN/DNN | X |
| VulCNN [[51](#bookmark224)] | 2022 | √ |  |  |  |  |  | CNN | X |
| ReGVD [[52](#bookmark235)] | 2022 | √ |  |  |  | √ |  | Res-GCN | X |
| MVD [[53](#bookmark238)] | 2022 | √ |  | √ | √ |  |  | FS-GNN | X |
| LineVul [[54](#bookmark240)] | 2022 | √ |  |  |  |  |  | Transformer | √ |
| LineVD [[55](#bookmark241)] | 2022 | √ |  | √ | √ |  |  | Transformer/Graph | √ |
| CPVD [[56](#bookmark242)] | 2023 |  |  |  |  | √ |  | GAT/ConvPool | X |
| VulChecker [[11](#bookmark189)] | 2023 |  |  |  |  |  | √ | MP-GNN | √ |
| DeepVD [[10](#bookmark188)] | 2023 | √ | √ |  |  |  | √ | GRU/GCN | X |
| mVulSniffer [[57](#bookmark226)] | 2023 |  |  | √ | √ |  |  | BiGRU/CNN | X |
| EnGS2F [[58](#bookmark233)] | 2024 |  | √ | √ | √ |  |  | GGNN | X |
| MGVD [[59](#bookmark236)] | 2024 | √ | √ | √ |  |  |  | GAT/CNN | X |
| DetectVul [[60](#bookmark243)] | 2025 | √ |  |  |  |  |  | Self-Attention | √ |
| SecureQwen [[61](#bookmark244)] | 2025 | √ |  |  |  |  |  | CodeQwen | X |
| MaliVD（本文） | 2025 | √ | √ |  |  | √ | √ | Hybrid/ATT | √ |

a CUS (Customized)，表示研究人员自定义的其他代码表征。

考虑了不同代码表征的组合，使用不同的模型学习不同结构的特征（例如组合序列模型和图模型），但没有充分考虑特征的类分离特性（Class-Separation Features, [CSF），缺乏对漏洞模式和特征选取的深入分析[10](#bookmark188)]。

2.2.3基于深度学习的源代码漏洞定位方法

尽管基于深度学习的源代码漏洞检测技术在实验中验证了可行性，超过了传统的基于规则或机器学习模型的方法，但深度学习方法大都只能提供有无漏洞的检测结果，而不能定位到漏洞触发的位置。相比之下，一些静态分析工具基于漏洞的检测规则却能够更好地定位到漏洞的触发位置。此外，模型难以定位漏洞的触发位置，也使得科研人员无法判断基于深度学习的漏洞检测器是否真实学习到了漏洞的模式。

最初，科研人员通过对神经网络进行解释来进行漏洞位置的判断，Zou等人[[67](#bookmark245)]在这方面进行了初步探索，他们从漏洞检测模型的判断中提取出一系列规则，用于识别哪些语句对漏洞检测的结果更重要，并基于这些语句构建了决策树规则，以辅助专家理解和判断。然而，这种方法只是根据神经网络的判断结果进行判断，并不能判断规则中重要的语句就是对漏洞检测有用的。为了解决这一局限，Mao等人[[68](#bookmark246)]在基于RNN的漏洞检测模型中加入了注意力机制，并根据注意力机制提供可解释性的效果，但该方法也没有直接用于漏洞语句的定位。此外，Li等[人[48](#bookmark229)]提出了一个基于图网络的可解释漏洞检测器IVDetect，其通过带有注意力的GCN学习代码的上下文感知表示，从而进行粗粒度漏洞检测，同时利用GNNEx- plainer[[69](#bookmark247)]来提供漏洞相关语句的细粒度解释，并支持对漏洞语句的定位，但仍难以取得准确的漏洞定位。随着图注意力网络（Graph Attention Network, GAT）[[70](#bookmark248)]的广泛应用，[一些工作[56](#bookmark242),[59](#bookmark236)]利用GAT模型进行漏洞检测和定位，但仍未能达到满意的效果。随着自注意力网络[[71](#bookmark249)]的提出，一些工作也基于此进行代码行级的漏洞检测，例如DetectVul [[60](#bookmark243)]。值得注意的是，这种通过解释神经网络的方法并未结合漏洞模式进行分析，对于可解释的结果也缺少定量表示，具有一定的主观因素。

近年来，Vuldeelocator[[50](#bookmark239)]和VulChecker[[11](#bookmark189)]等方法试图通过编译后的中间代码（如LLVM）进行漏洞定位。Vuldeelocator将C语言项目编译成LLVM中间代码，并使用带有注意力机制的双向RNN网络进行学习，然后根据注意力的大小来输出哪些语句是漏洞触发的位置。类似地，VulChecker将C/C++项目转化为自定义的程序依赖图，然后根据漏洞可能触发位置的PoIs特征提取子图，然后通过对子图的分类判断该PoIs是否是真实漏洞触发的位置。虽然这两种方法在漏洞定位上取得了一定的成功，但它们依赖于编译后的代码信息，这使得代码必须通过编译才可行，从而在实际的软件开发中难以利用。随着大语言模型（Large Language

Models, LLMs）在多种任务中的成功应用，一些研究开始探索其在漏洞检测领域的能力。例如，基于CodeBERT[[72](#bookmark250)]的LineVul [[54](#bookmark240)[]和 LineVD[55](#bookmark241)]，基于CodeQwen[[73](#bookmark251)]的[SecureQwen[61](#bookmark244)]等模型，可以进行漏洞类型的识别和漏洞触发位置的判断，但这些方法的准确率仍然不够理想。这是因为代码不仅仅是文本，它包含丰富的结构化信息，直接应用LLMs处理源代码会导致结构化特征的丢失。此外，在庞大的代码库中，漏洞往往表现为细微的特征变化，这对于训练信息复杂且强调全局特征的LLMs来说，难以捕捉到这些微小但关键的特征，从而难以做出准确的判断。

2.2.4 当前漏洞检测模型存在的问题

根据分析现有工作在漏洞检测上的表现，可以总结出如下几点当前基于深度学习的漏洞检测模型存在的问题：

• **依赖低质量的漏洞数据集：**现有数据集来源不一，多数据来源混用导致该领域缺乏统一的衡量标准，使得某些模型检测效果虚高。此外，数据集中存在的重复样本、不平衡的正负样本等问题，且现有数据集大都没有标注漏洞行号，使得难以训练得到细粒度的漏洞检测模型。

• **不合理的特征选取：**现有模型通常组合多类特征进行实验，但未充分考虑特征对于检测漏洞的类分离度。应首先深入分析漏洞模式，确定哪些特征对分类有用，再使用相关特征重要性评价技术进行特征选择。选取重要性高的特征能够显著提升模型性能和解释能力。

• **不合适的模型设计：**现有方法未充分考虑模型是否适合漏洞检测领域，大多是将其他领域的先进模型应用于代码分析上。然而，代码不同于图像和文本，其结构化特点要求使用更合理的模型。强行将各种图神经网络应用于代码特征上也是不合理的，因为代码还包含序列特征和树结构特征，应综合进行考虑。单一使用图模型不仅降低了模型的学习效果，还增加了检测的复杂度。

• **难以定位漏洞的位置：**大多数现有的漏洞检测方法主要是二分类任务（即判断是否存在漏洞），难以识别具体漏洞类型以及漏洞触发的位置。依赖编译后的代码信息进行漏洞定位的方法在实际软件开发过程中存在较大的局限性；尝试通过LLMs来实现漏洞定位的方法，损失了代码的结构信息，在实际应用中准确率较低。

2.3本章小结

本章节全面回顾了当前源代码漏洞检测领域的研究现状，包括漏洞代码库的[构建和源代码漏洞检测方法两部分内容。在2.1](#bookmark26)小节，按照人工和分析工具生成、开

源代码库爬取以及漏洞库抽取的顺序详细介绍了这三类主要的漏洞代码库构建方法。然后在[2.2](#bookmark36)小节，介绍了从传统的漏洞检测方法到基于深度学习的漏洞检测和漏洞定位技术的发展历程。在介绍每部分内容时，本章还探讨了各类方法的优缺点，最后总结了该领域面临的主要问题，如缺乏高质量漏洞数据集、特征选取不合理、模型设计不合适以及模型难以定位漏洞的位置等。

第3章 代码行级的漏洞代码库构建框架

本章节将详细介绍本文提出的代码行级的漏洞代码库构建框架，并通过实验验证所构建的漏洞代码库ReliVul的有效性。[3.1](#bookmark50)小节详细阐述了ReliVul的构建框架，包括源代码获取、漏洞补丁判定和漏洞行号标注三部分内容。[3.2](#bookmark60)节描述了实[验设置，主要介绍了比较基线和所使用的评价指标。第3.3](#bookmark66)小节展示了实验结果，与其他广泛使用的漏洞数据集进行对比，从数据集的基本信息、准确性、唯一性和完整性四个方面验证了ReliVul的优越性。

3.1代码行级的漏洞代码库构建

本文希望通过CVE数据库和开源代码库的提交信息来设计一个自动化的漏洞代码库构建系统。[CVE 数据库，例如 Mend[37](#bookmark215)]和NVD[[36](#bookmark214)]，收录了大量的CVE漏洞资源，在每个CVE中，还含有CWE漏洞信息、语言、漏洞资源地址等。经研[究，这些数据库中包含了大量开源代码库中的代码资源，例如 Github[22](#bookmark200)]平台。这些开源代码库中拥有大量的代码提交和修订记录，开发者一般会在提交信息中说明每次修订代码的原因，如果某次提交涉及漏洞的修复，那么修订前的代码可标记为有漏洞，修订后的代码一般是没有漏洞的（也可能未完全修复或者带来其他漏洞）。本文设计了一个自动化的漏洞代码库构建模型，用于从CVE数据库和开源代码库中抽取漏洞样本，并根据开源项目的提交信息判断是否是涉及漏洞修复的提交。另外，本文通过分析各类CWE漏洞的触发特点，设计了一组特定漏洞的可能触发语句，即PoIs，通过结合PoIs和代码的修订记录，可准确标注漏洞的触发位置和触发路径。

本节首先对本文自动化漏洞代码库构建模型进行概述，然后按照源代码获取、漏洞补丁判定和漏洞行号标注的步骤进行详细描述。

3.1.1模型概述

本文提出了一种基于CVE数据库和开源代码库抽取的自动化漏洞样本构建模型，并依赖此模型收集了一个代码行级别的真实漏洞数据集ReliVul（Real-worldLine-levelVulnerability dataset），ReliVul的构建流程如图[3.1](#bookmark138)所示。

ReliVul的构建流程可分为三个步骤：源代码获取、漏洞补丁判定和漏洞行号标注。本文首先从CVE数据库中提取漏洞的信息，例如CWE的编号、解释和相关的资源地址，并依托开源代码平台提取相关项目的提交历史，并根据提交历史

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **1. 源代码获取** | |  | | --- | | **2. 漏洞补丁判定** | | **SAST**  a11  ⋮  am1  … a1n 、 ⋮ … amn  **ML** →  Yes  No  **DL**  Tree-sitter  结果集成  SVM/RF/KNN …  向量化   |  | | --- | | Prompt |  |  |  | | --- | --- | |  | … | | |
| |  | | --- | | CWE-ID | | CWE-Description | | Language | | URL | | …… |  |  | | --- | | Commit-ID | | Commit-Message | | URL | | Parent Patch | | …… |  |  | | --- | | Project-ID | | Code-Before | | Code-Changes | | …… |   CVE数据库  DATABASE  !  提取漏洞信息   |  |  | | --- | --- | |  |  |   代码平台  获取补丁信息  …  项目地址  爬取代码信息  API, 爬虫脚本 |
| 标注  Joern  ∩  攻击路径及 漏洞触发点  CPG   |  |  | | --- | --- | | **3. 漏洞行号标注** | | | 制定  **-------**    **-----**  CWE信息  标准化   |  | | --- | | **+** ~ ~ ~  -  + ~ ~ ~  ~ ~ ~  Code Changes | | |  | | --- | | PoIs | |  | | 代码修订 | |  | | 漏洞源代码 | | |  | |

图3.1基于CVE数据库和开源代码库抽取的自动化漏洞样本构建模型

和爬虫脚本爬取代码的修订信息，这样就得到了准确的漏洞样本和可能的补丁样本，所以需要对补丁的准确性进行判定，本文使用4个静态工具、多个机器学习模型和大语言模型进行综合判断，并将结果集成来判断补丁的准确性。另外，利用设计好的漏洞PoIs特征，结合代码的修订记录，就可以定位到漏洞触发的位置，通过分析代码的数据依赖和控制依赖关系，还可以标注好漏洞触发的路径。

总之，本文利用CVE数据库和开源代码平台，可快速自动化构建大量高质量漏洞样本。样本中的代码都来自于真实的项目，在真实性、复杂度和准确性方面相对现有数据集都有较大改进，另外对漏洞行号和触发路径的标注也是该领域的第一次尝试，填补了该领域的空白，对行级的漏洞检测研究提供可靠的数据支撑。

3.1.2源代码获取

源代码获取是构建漏洞代码库的第一步，包括漏洞信息的提取、补丁信息的获取和代码信息的爬取。

**1）漏洞信息提取**

根据[2.1.3](#bookmark32)小节的介绍，有一些机构或组织对软件开发中发现的漏洞进行了收集、整理与发布，并构成了CVE漏洞数据库，因此通过这些漏洞库即可获得漏洞条目的入口。这些数据库中的漏洞样本一般经过了广泛的验证，准确性得到了保障。其中，Mend和NVD是两个广受认可的漏洞信息平台，它们提供了大量的漏洞数据和相关的代码链接，是构建漏洞数据集的重要来源。

• **Mend**[[37](#bookmark215)]：是一个广泛使用的开源组件管理和安全平台，旨在帮助企业发现和修复开源软件中的安全漏洞。Mend平台会实时更新最新的漏洞信息，确保数据的时效性和准确性；在每个漏洞界面，其提供了丰富的漏洞信息，包括CVE编号、漏洞描述、影响的软件版本、修复建议等。另外，Mend提供了RESTful应用程序编程接口（Application Programming Interface，API），方便开发者自动化地获取漏洞信息。

• **NVD**[[36](#bookmark214)]：由美国国家标准与技术研究院维护，是全球最权威的漏洞数据库之一。NVD包含了大量且详细的漏洞类型信息，并为每种漏洞类型提供了真实项目下的实例，是构建漏洞数据集的重要来源。NVD中的漏洞信息包括CVE编号、漏洞描述、通用漏洞评分系统（CommonVulnerability Scoring System，CVSS）的打分、受影响的软件版本、修复建议等。NVD的数据是公开可用的，也提供了丰富的API接口。

本文通过分析Mend和NVD平台的漏洞信息，结合其提供的API接口，通过Python爬虫的方式获取特定CVE的详细信息，并将提取的漏洞属性信息保存为键值对的格式，主要的漏洞属性包括：CVE编号、CWE编号、代码语言、资源地址等，这些漏洞的关键属性信息列表如表[3.1](#bookmark158)所示。

**2）补丁信息获取**

在从CVE数据库获取到漏洞信息后，本文首先从与该CVE相关的代码资源中提取出与开源项目有关的信息。对于Mend和NVD数据库中的记录，首先筛选出C/C++项目，然后进一步过滤出包含提交信息网址的资源[①](#bookmark252),以方便收集漏洞代码和补丁信息。对于大多数开源项目，GitHub是一个重要的代码来源。另外，Android、Chrome和Linux作为大型开源项目，拥有庞大的用户基础和活跃的开发者社区。这些项目的代码由他们专门的平台进行管理和发布。为了能够收集漏洞的补丁代码，本文记录了如提交ID、提交信息、提交的平台及地址，以及每个提交的前后版本的ID，以便于后续对补丁的分析和提取。这些提交信息的关键属性信息如表[3.2](#bookmark160)所示。

本文首先通过开源平台和漏洞信息定位到漏洞代码的版本（Commit-ID），那么其子提交可能是与漏洞修复相关的。由于项目代码一般存在多个字提交，为了能较准确地查找到补丁版本（即与漏洞修复相关的提交），本文首先定义了一组关键字，这些关键字可能与漏洞修复相关。本文将常见的漏洞关键词分为六类：溢出类、内存损害类、越权类、常见Web类、漏洞相关类以及安全规范类，这些关键词如附录[A](#bookmark120)中的表[A.1](#bookmark253)所示。其中括号里的词组表示提交信息中要同时包含每一个

① 例如：<https://github.com/inspircd/inspircd/commit/eba7e6655a21c752912995aa04355f0df112f8e5>

表3.1从CVE数据库中提取的有关漏洞的关键信息

|  |  |
| --- | --- |
| 属性 | 描述 |
| CVE-ID | CVE 编号（如：CVE-2012-1836） |
| CWE-ID | CWE 编号（如：CWE-119） |
| Language | 该 CVE 的代码语言（如：C++） |
| Resource | 该 CVE 相关的代码资源 |
| CVE Description | 该 CVE 的描述信息 |
| Publish-Date | 该 CVE 的发布日期 |
| CVSS | 通用漏洞评分系统 |
| CVE-AV | 该 CVE 的攻击向量（Attack Vector） |
| CVE-AC | 该 CVE 的攻击复杂度（Attack Complexity） |
| CVE-PR | 该 CVE 的权限要求（Privileges Required） |
| CVE-UI | 该 CVE 的用户交互（User Interaction） |
| CVE-S | 该 CVE 的评分（Score） |
| CVE-C | 该 CVE 的机密性（Confidentiality） |
| CVE-I | 该 CVE 的完整性（Integrity） |
| CVE-A | 该 CVE 的可靠性（Availability） |
| CWE Description | 该 CWE 的描述信息 |
| CWE Solution | 该 CWE 的可能解决方案 |
| CWE Consequence | 该 CWE 可能的触发原因 |
| CWE Method | 该 CWE 可能的检测方法 |

表3.2从开源代码平台获取的有关提交的关键信息

|  |  |
| --- | --- |
| 属性 | 描述 |
| Commit-ID | 提交的 ID（或者称为：Sha） |
| Commit-Msg | 提交信息 |
| Commit-Date | 提交时间 |
| Platform | 提交平台（Github,Android, ...） |
| URL | 提交地址 |
| Project | 所属项目名称 |
| Parent-ID | 父提交 ID（该版本的前一版本） |
| Child-ID | 子提交 ID（该版本的后以版本） |

关键词，该提交才可能是与漏洞修复相关的，例如文件上传漏洞（表中对应：(file, upload)），如果只包含“文件”或“上传”则可能仅是上传文件功能的优化。

对于开源代码库中的海量代码提交，本文首先通过这些关键词进行初步过滤，如果提交信息中不包含这些关键词，那么该提交很可能不与漏洞修复相关，且对于自动化样本构建模型以及判断具体漏洞类型都十分困难。

**3）代码信息爬取**

在经过关键词初步过滤后，就得到对于某个漏洞的多个可能的修复提交信息，然后使用开源平台提供的API来爬取代码信息，这些API可以方便地提取开源项目中的代码信息。本文从四个平台爬取代码信息，即Github[[22](#bookmark200)]，[Google Git [74](#bookmark254)]，bugs.chromium[[75](#bookmark255)]和Linux Kernel Git [[21](#bookmark199)]，以下是以下是实现代码信息爬取的关键步骤：

• **API接口调用**：首先，针对从CVE数据库提取的与漏洞相关的提交信息，使用与平台对应的API接口来爬取代码信息。例如对于GitHub，可以利用其REST API或GraphQL API来查询特定Commit-ID下的文件列表、文件内容和文件的变更信息。由于各个平台都有API请求频率限制，所以该步骤需要合理处理API请求的频率。

• **数据解析**：从API返回的数据通常为JSON格式或其他结构化的数据。需要对这些数据进行解析，以提取出有用的信息，如文件名、最后修改日期以及代码内容本身。此外，还需要关注代码的差异部分（diffs），这些部分往往直接反映了开发者对漏洞所做的修正。

• **增量更新机制**：考虑到需要爬取的代码量很大，特别是在提取漏洞补丁时，需要提取多个子提交的内容，所以在爬取补丁时仅爬取代码的修订部分，然后结合父提交版本的代码以获取该版本的代码，从而减少不必要的网络传输和存储开销。

• **错误处理与日志记录**：在执行爬虫任务过程中，可能会遇到由于网络不稳定引起的连接失败或API响应超时等问题。对于错误的处理并使用日志进行记录是非常重要的，对于后续的问题排查和系统优化至关重要。

• **合规性考虑**：在执行代码爬取任务之前，本文了解并确保遵守了目标网站的服务条款及隐私政策，确保了所爬取的项目是完全开源可用的，在将构建的数据集公开之前会检查代码中敏感信息并进行匿名化处理。

通过上述的代码爬取工作，就得到了漏洞代码和候选的补丁代码，这些代码的关键属性信息如表[3.3](#bookmark162)所示。通过这些关键信息，不仅可以方便进行代码信息的获取，有利于在数据集收集完毕后保证样本的真实性。

表3.3从开源代码平台获取的有关代码的关键信息

|  |  |
| --- | --- |
| 属性 | 描述 |
| Project-ID | 所属项目名称 |
| File-Name | 文件路径（在该项目中的路径） |
| Commit-ID | 提交的 ID（用于判断文件的版本） |
| Code-Before | 有漏洞代码（对应 CVE 数据库中的版本） |
| Code-Changes | 代码变更（有漏洞代码的多个子提交） |

3.1.3漏洞补丁判定

在爬取了漏洞代码和对应的提交列表后，下一个关键的任务是对漏洞补丁的[判定，如2.1.3](#bookmark32)小节的描述，虽然从漏洞库可以抽取到漏洞样本，但现有数据通常缺乏补丁代码，一些方法直接抓取现有项目中暂时“安全”的代码作为“无漏洞”的代码。虽然这些项目暂时没有发现漏洞，但“安全无绝对”，没有绝对安全的代码，这些项目也是基于现有的工具和漏洞知识来判断为“暂时未发现漏洞”，随着新漏洞的出现和检测工具的升级，这些项目可能会暴露出其他的漏洞。另外，这种直接抓取现有项目的方法并没有准确地对应原来有漏洞代码的补丁代码，一些大型项目可能经历了成千上万次版本更替，代码的样貌可能经历了很大的变动。实际上的补丁代码应该是在漏洞代码的基础上，只进行漏洞的修复，而不应该有其他的改变，这样的补丁代码才有利检测模型的训练，可以让它们更关注于代码的细微变化，从而学习到漏洞的模式。

为了能构建高质量的补丁数据，提高现有数据集的质量与实用性，本文根据漏洞代码的提交记录，爬取了其多个子提交，并根据子提交循环判断哪些是与漏洞修复相关的提交。具体来说，对于每一个子提交，我们在根据代码修订信息获取到源代码后，使用4个静态分析工具、4个机器学习模型和一个大语言模型进行综合判断，以确定这些提交是否涉及了漏洞的修复，并判断补丁是否有效。这样构建的数据集可以保证补丁代码相对于漏洞样本的改动不那么大，增强了其在实际环境中的可应用性。接下来将具体介绍三类方法的判别流程以及结果的集成方法。

**1）静态分析工具**

市场上对C/C++代码有很多静态扫描工具，虽然其准确率有限，但也具有[很强的参考价值，本文使用的 4 个静态分析工具包括 Cppcheck[5](#bookmark183)]、Flawfinder[[6](#bookmark184)]、RATS[[76](#bookmark256)]和Semgrep[[77](#bookmark257)]。静态分析工具能够提供详细的代码结构分析，帮助识别常见的编程错误和安全漏洞。并且具有较低的误报率，从而提高了数据集的质量。此外，通过结合多个工具的结果，可以进一步提高漏洞检测的准确性和全面性。将

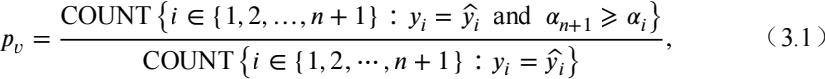
可能修复后的代码让这四个静态分析工具扫描，如果未扫描出漏洞，可在本阶段标记为良性样本。

**2）机器学习模型**

为了使用机器学习模型，需要部分已标注的数据进行模型训练，本文的训练样本由提交信息、代码修订信息和标签组成。模型的主要训练样本来源于现有数据集。现有不少工作通过开源代码库抽取漏洞数据集，根据其数据集的年份、代码语言、数据集信息完整度、论文引用率等进行筛选，最终选择了四个数据集作为候选训练数据集：Devign[[7](#bookmark185)]、[Big-Vul [27](#bookmark205)]、ReVeal [[33](#bookmark211)]、D2A[[32](#bookmark210)]。这些数据集来源于开源代码项目，包含详细的文件路径、提交信息和代码修订信息，这些已标注好的数据可以用作本文的数据构建模型的训练数据集。

模型采用了一种类似于集成学习的策略，通过训练多个模型，并结合每个模型的预测结果及其置信度来确定最终样本的选择。共形预测[[78](#bookmark258)]（Conformal Prediction,CP）是一种用于评估和量化机器学习模型预测不确定性的方法。不同于传统的预测方式，共形预测在给出样本的预测结果时，还会输出该样本的置信水平，即模型对该样本预测的把握程度。这种方法是基于统计学的原理，给定一个置信水平，可以根据预测的结果落在置信区间的概率来判断该样本是否符合要求。对于本文使用的SVM、KNN、RF和GB等算法，可以利用共形预测技术提升这些模型的性能。[PyCP 库[79](#bookmark259)]中提供了用于计算预测结果奇异程度的方法，可利用PyCP库进行机器学习置信度的计算。

为了计算样本的置信度，本文使用10%的数据作为校验集，来确定置信度的阈值。具体来说，假设校验集中有n个数据样本，那么对于输入的数据xi和其标签yi，模型ℎ的预测结果为i，假设使用函数A(xi,yi,ℎ)来计算模型ℎ对预测结果的置信度为αi，其中i = 1,⋯,n。那么对于一个新的样本xn+1，可以计算得到模型对其预测标签i的置信度为αn+1，可以通过式（[3.1](#bookmark260)）计算来判断该样本的置信[度[47](#bookmark231)]。



此式的含义是统计对于那些模型预测结果正确的样本，新样本置信度大于等于原有样本的置信度的比例。这说明，如果计算得到的pv较小，也就是接近其下界1/ (n + 1)，即置信度较低，那么说明该预测结果是非常不可信的；如果pv较大，则说明输入xn+1预测为标签i是较为可信的。所以设置合适大小的阈值（将作为超参数在实验中进行调整和选取），可以有效提高样本的准确程度。

训练好的多个机器学习模型即可进行自动化的漏洞补丁判定。另外还可以使

用低可信度的样本来改进模型的学习结果。一般来说，低可信度的样本一般可能是较为复杂的漏洞或者更难识别的提交信息，这些样本可以通过少量人工的检查获取其正确的标签。通过在训练数据中加入新的低可信度的样本及其标签，可以不断提升模型识别复杂漏洞修复提交的能力。

**3）大语言模型**

近年来，大语言模型在自然语言处理方面的发展迅速，这些LLMs在代码理解与源代码漏洞检测方面也表现了较好的性能。一些LLMs甚至可以理解代码复杂的上下文信息，且根据预训练知识中的漏洞知识，可以给出代码是否含有漏洞的判断和解释。本文将LLMs也纳入补丁判断的方法之一，本文尝试了多个流行的LLMs，最终选取了通义千问大模型，一方面该大模型在本文选取的训练数据集上表现优越且稳定，而且它还提供了API方便自动化调用。为了增强LLMs的漏洞检测能力，本文针对补丁判断任务设定了一个提示模板，包括系统提示、CWE描述、推荐解决方案、提交信息和受影响函数等内容。具体来说，该提示模板包括如下的组成部分：

• **系统提示**：LLMs扮演专家角色，分析代码漏洞及其对应的修复方案。

• **上下文提示**：

**– CWE 描述**：基于每个补丁关联的CWE-ID提供简要的漏洞描述。

**– CWE 解决方案**：提供根据CWE-ID推荐的解决方案，帮助LLMs评估代码变更与漏洞解决方案的一致性。

**– 提交信息**：提交信息帮助LLMs理解修复的目的。

**– 代码变更**：通过Tree-sitter工具提取受代码变更影响的内容，进一步细化上下文信息。

• **代码内容**：具体的代码变更内容，包括漏洞代码和根据Code-Changes生成的补丁代码。

• **答案提示**：LLMs会根据提供的详细漏洞和补丁信息，输出“YES”或“NO”，表示代码变更是否与漏洞修复相关。

不同于静态分析工具和机器学习模型，LLMs在预测代码结果时使用了不同的视角，本文将漏洞的知识、原来有漏洞的代码提供给LLMs用于挖掘现有漏洞的知识，将提交信息和代码的变更信息提供给LLMs，一方面利用其强大的自然语言处理能力判断这些提交信息是否和漏洞修复相关，一方面通过代码的变更信息判断该版本的代码是否真正修复了代码中的漏洞。

**4）结果集成**

虽然静态分析工具、机器学习模型和大语言模型目前的漏洞检测能力均不是

很突出，但他们从多个视角审视代码的修订信息，通过综合这些方法的分析结果，将这些输出结合起来，就可以得到相对准确的结果。本文以提高补丁的准确性为主要目标，只有当三者都得出相同结论时，该版本的代码才会被标记为与漏洞修复相关。如果某个版本的代码的置信度较低，或者大语言模型和静态分析工具的结果存在冲突，则排除该版本，并循环利用子提交进行判断，如果最新版本也与漏洞修复无关，则不再标注该漏洞的补丁。通过模型集成，能够最大化每种方法的优势，从而减少错误的判断。

3.1.4漏洞行号标注

现有方法例如Big-Vul [[27](#bookmark205)]数据集在标注漏洞行号的时候，是仅仅基于提交版本补丁中的代码变更信息来完成的，将两个小版本之间的修改行视为潜在的漏洞行。这种方式可能会将一些与漏洞无关的代码行标记为漏洞行。例如，某些修改可能只是优化或重构代码，而不是修复漏洞，这可能导致大量与实际漏洞无关的代码行被标注，增加了数据集的噪声。有些漏洞可能涉及多个文件或函数之间的交互，而不仅仅是单个文件内的修改。如果这些交互没有被正确识别，可能会导致部分漏洞行未被标注。

为了解决现有标注漏洞行号的不足之处，本文通过联合分析漏洞特征和代码变更信息，大大提高了漏洞行号标注的准确性。接下来，本文将首先分析各类漏洞的模式，并从中提取PoIs特征，然后介绍如何对漏洞代码行进行标注。

**1）PoIs提取**

本文通过在CWE官网[[38](#bookmark216)]和互联网上的统计信息，根据漏洞的流行度和危害程度，重点分析了流行度高和危害性大的漏洞，并将这些漏洞分成缓冲区溢出、值计算漏洞、资源管理漏洞、输入验证漏洞、权限管理漏洞和命令执行漏洞六类，这六类包含大多数的CWE，其中部分CWE示例列在了表[3.4](#bookmark164)中。

为了准确地标注漏洞触发点（PoIs），本文对每类漏洞进行了详细的分析，并制定了相应的PoIs标准，这些标准列在了表[3.4](#bookmark164)中。

对于缓冲区溢出漏洞，例如CWE-119（不恰当的内存缓冲区操作限制）、CWE- 121（栈缓冲区溢出）和CWE-122（堆缓冲区溢出），这些漏洞通常由于程序未能正确处理内存的边界而引起。缓冲区溢出漏洞可能导致数据的读写错误，甚至破坏原有的代码和数据，导致系统崩溃。因此，本文将本地或动态分配内存的任何读写命令作为PoIs的标准，包括strcpy, memcpy, malloc,free等函数调用。

对于值计算漏洞，如CWE-190（整数溢出或环绕）、CWE-369（除零）和CWE- 682（错误的计算，然后结果被用作内存分配和访问上），这些漏洞主要是因为程序在计算过程中出现了违法操作。值计算漏洞可能导致程序崩溃或产生意外的行

表3.4 CWE分类、示例和对应的PoIs标准

|  |  |  |
| --- | --- | --- |
| 漏洞类型 | CWE 编号 | PoIs 标准 |
| 缓冲区溢出 | CWE-119, CWE-121, CWE-122, CWE-125, CWE-119 | 本地或动态分配内存的任何读写命令 |
| 值计算漏洞 | CWE-190, CWE-369, CWE-682 | 如果一个变量经过了运算符计算 或函数调用，那么标记该变量 |
| 资源管理漏洞 | CWE-120, CWE-399, CWE-400, CWE-404, CWE-415, CWE-416, CWE-476, CWE-908, CWE-909 | 指针变量和内存分配、访问指令 |
| 输入验证漏洞 | CWE-20, CWE-89, CWE-134 | 来自用户输入的任何变量和访问语句 |
| 权限管理漏洞 | CWE-255, CWE-264, CWE-269, CWE-284, CWE-287, CWE-863 | 涉及文件、网络端口、数据库、命令 执行等操作前的验证语句 |
| 命令执行漏洞 | CWE-77, CWE-78, CWE-88 | 执行系统命令的语句及相关变量 |

为。为了识别这些漏洞，本文将经过运算符计算或函数调用的所有变量进行了标注，并作为PoIs的标注。例如，算术运算、逻辑运算或函数调用的变量。

资源管理漏洞，例如CWE-400（未控制的资源消耗）、CWE-415（双重释放）和CWE-416（释放后使用），通常与指针的使用和内存管理有关系。资源管理漏洞可能导致内存泄漏或内存损害等问题，导致越权或非法闯入，威胁资源安全。为此，本文将指针变量和内存分配、访问指令作为PoIs的标准，包括malloc, free, new, delete等函数调用，该类型漏洞与缓冲区溢出漏洞较为相似，在结合代码场景和对应的语句条件时，可以进行细粒度划分，这些函数对于这些漏洞的识别都很重要。

针对输入验证类漏洞，如CWE-20（不恰当的输入验证）、CWE-89（SQL注入）和CWE-134（使用外部控制的格式字符串），通常发生在对用户输入的数据未做充分的检查的情况下。输入验证类漏洞可能导致攻击者构造的恶意输入被直接传递给应用程序，导致恶意命令执行或病毒木马的传播，从而造成其他安全威胁。本文将来自用户输入的任何变量和访问语句作为PoIs的标准，包括scanf,fgets, gets等函数调用。

权限管理漏洞，如CWE-255（凭证管理错误）、CWE-264（访问控制不当）和CWE-269（权限管理不当），设计对不同用户和敏感资源的管理，在复杂软件中容易出现。这类漏洞可能导致未经授权的用户访问敏感资源问题。本文将涉及文件、网络端口、数据库、命令执行等操作前的验证语句作为PoIs的标准，包括open, read, write, exec等函数调用。

最后，对于命令执行漏洞，如CWE-77（不安全的命令执行）、CWE-78（系统命令注入）和CWE-88（参数化命令执行），这类漏洞危险极大且较为常见，当一些系统命令的参数来自用户的输入或者这些命令管理不当，就可能导致一些危险

的系统命令执行，甚至引起系统崩溃和重要资源损失。本文将执行系统命令的语句及相关的变量作为PoIs的标准，包括system, exec, popen等函数调用。

通过对这六类漏洞的深入分析，本文为每类漏洞制定了一套PoIs标准。这些标准不仅能帮助本文在收集数据时准确定位到漏洞的代码行，也为后续研究漏洞检测模型提供了漏洞特征。另外，这些PoIs标准也可以帮助更多的研究人员在设计软件或检测漏洞时更加关注这些语句，从而增强软件的安全性。

**2）漏洞行号标注**

在介绍了漏洞PoIs特征后，结合补丁代码相对于漏洞代码的修订信息，就可以标注漏洞触发的位置信息。具体来说，有如下几个步骤：

• **代码清洗与提取：**首先对漏洞代码和补丁代码进行数据清洗，包括删除注释、空值以及常量替换等，然后提取补丁代码相对漏洞代码的变更代码，将这部分代码作为后续提取漏洞位置的代码；

• **PoIs特征匹配：**对于候选的代码，针对漏洞样本的CWE类型，利用预定义的PoIs标注进行匹配，如果没有匹配到任何PoIs语句，则排除该样本，如果匹配到多个结果，则将这些PoIs语句均标注为候选漏洞位置；

• **标注触发位置：**[在初步筛选的多个候选漏洞位置中，结合第3.1.3](#bookmark56)的各个分析工具的检测结果，选取最可能触发漏洞的PoIs语句作为漏洞的触发位置；

• **生成触发路径：**在标注好漏洞触发位置后，利用Joern提取漏洞样本的数据依赖和控制依赖关系，然后将从主函数到该漏洞触发位置的数据流和控制流标注为漏洞触发的路径。

通过这些步骤，本文在尽可能保证漏洞触发位置的准确性的基础上，大大减少了现有方法标注位置不够精细的问题。

总的来说，本小节从漏洞代码的获取、补丁代码的获取到漏洞位置的标注，详细介绍了ReliVul数据集的构建流程。从CVE数据库提取漏洞信息确保了漏洞代码的来源可靠性，通过集成学习的方式判断漏洞的补丁也提高了补丁代码的准确性，对于漏洞位置的标注结合了PoIs漏洞特征，提高了现有方法标注的准确性和精细度。基于本文所调研的结果，对于漏洞触发路径的标注也是该领域的第一次尝试，为漏洞样本提供了更详细的上下文信息，有利于训练出更为强大的漏洞检测模型，为该领域做出更多的贡献。

3.2实验设置

为了更好地评估本文提出的漏洞代码库ReliVul，本文将ReliVul和四个广泛使用的数据集进行对比，并在数据集基本信息、标签准确性、数据重复率、数据完

整性等指标进行评价，以突出本文提出的数据集的优势。

3.2.1比较基线

本文选取五个流行的C/C++数据集进行比较，分别是Juliet [[19](#bookmark197)]、Devign[[7](#bookmark185)]、[Big-Vul [27](#bookmark205)]、D2A[[32](#bookmark210)]和DiverseVul [[9](#bookmark187)]。其中Juliet是人工构造的漏洞数据集，漏洞复杂度相对较低，但提供了较为准确的标签。其他数据集都是来自真实的开源项目，在项目复杂性和漏洞复杂度上具有保障，其中Devign的数据标注方法是研究人员手动筛选的，Big-Vul是从CVE数据库中抽取的漏洞数据集，但对补丁的获取仅仅依赖于现有的项目，对漏洞触发位置的标注也仅仅使用了代码变更信息；D2A数据集的来源是开源代码仓，通过识别提交信息获取漏洞样本和补丁样本，其标签的准确性可能较低；DiverseVul是一个较新的漏洞数据集，通过爬取安全问题网站、提取修复漏洞的提交和源代码等方法进行构建漏洞样本，该方法从多个开源项目中提取漏洞函数和非漏洞函数，并使用三个静态分析器来筛选和标注数据样本。该数据集相对之前的工作扩充了数据样本，但标注粒度只在函数，没有对漏洞的触发位置进行标注。

3.2.2评价指标

漏洞数据集的评估一直以来是一个难题。除了数据集的基本信息，例如样本数量、标注粒度、含有的信息等，这些基本信息很容易通过统计的方法进行度量。然而研究者更关注的是数据集本身的质量问题，例如准确性、唯一性和完整性等[指标[80](#bookmark261)]。为了全面评价本文的提出的ReliVul数据集，本文使用四个评价指标来验证ReliVul的优越性。

**1）基本信息**

漏洞数据集的基本信息包括样本的标注粒度、标签数量、标记方法、样本信息丰富度、正负样本个数等，这些基本信息对于数据集的应用至关重要，通过对比这些基本信息，可以明显看出各个数据集的优劣之处。

**2）准确性**

准确性（Accuracy）定义了组成数据集的样本的正确性。这在很大程度上与语义标签正确性有关，即标记为某种类型的漏洞或非漏洞的样本是否真正一致。由于代码没有绝对的安全性，即使某个程序被标记为无漏洞，那么可能随着软件和互联网的发展以及新漏洞的出现，该程序可能存在新的漏洞，所以本文更关注被标记为有漏洞的代码，对于数据集内有漏洞的代码，定义FP为标记为无漏洞样本的个数，TN为标记为有漏洞样本的个数，那么在衡量漏洞代码库时，准确性可以

表示为式（[3.2](#bookmark262)）的形式。

Acc .′=（3.2）

**3）唯一性**

唯一性（Uniqueness）即数据集中不存在重复的样本，该指标可评估组成数据集样本的重复率。由于数据收集过程中有许多样本来自相同的项目，且现实世界存在大量代码复制的现象，所以数据集中可能包含重复的样本。数据集中含有重复的样本不利于检测模型的评价，训练集内部中如果有重复样本，会导致模型偏向这些样本，而训练集和测试集之间存在重复样本，会不利于模型向真实的数据分布推广。对于代码克隆，实际上存在不同层次的代码克隆现象，表[3.5](#bookmark166)展示了四类代码克隆的解释。

表3.5不同层次的代码克隆及解释

|  |  |
| --- | --- |
| 层次 | 解释 |
| Level-1 | 相同的代码片段，除了空白、布局和注释的不同 |
| Level-2 | Level-1，以及自定义标识符的不同 |
| Level-3 | Level-2，以及语法相似的代码片段，但在语句级别有所不同。 |
| Level-4 | Level-3，以及语法上不同的代码片段实现相同的功能 |

如今互联网上存在不少用于检测代码克隆的方法，本文利用Allamanis[[81](#bookmark263)]发布的代码重复检测器工具来识别代码克隆并将Level-3作为评估代码克隆的标准。

**4）完整性**

完整性（Completeness）既可以指数据集中信息的完整性，也可以指单个样本的完整性。由于前者需要外部参考信息，本文更关注单个样本的完整性，因为它是数据的固有属性，对于漏洞数据集，如果值不包含原始函数的所有代码，则源代码可能会丢失信息。

由于项目代码在抽取和切片的过程中，依赖现有工具和人工操作，导致某些样本存在不合理的截断或遗失现象。如果总样本数为N，不完整的样本个数为D，那么完整性的定义可由式（[3.3](#bookmark264)）表示。

COmpleteness = 1−（3.3）

3.3实验结果

本小节将详细对比ReliVul和现有数据集的指标，包括基本信息比较，准确性、唯一性和完整性的实验和分析。

3.3.1基本信息对比实验

本文将ReliVul和六个广泛使用的数据集进行对比，数据集内的项目个数、CWE的个数、总函数个数、漏洞样本占比、提交个数、是否标注漏洞行号以及是否有额外信息如表[3.6](#bookmark168)所示。

表3.6ReliVul与5个流行的漏洞数据集的基本信息对比

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 数据集 | 项目 | CWEs | 函数 | % 漏洞函数 | 提交 | 漏洞行号 | 触发路径 |
| Juliet | - | 117 | 253,002 | 36.77 | - | √ | X |
| Devign | 2 | - | 26,037 | 45.61 | - | X | X |
| Big-Vul | 348 | 91 | 264,919 | 5.78 | 3,754 | √ | X |
| D2A | 6 | - | 1,295,623 | 1.44 | 14,447 | √ | X |
| DiverseVul | 797 | 150 | 330,492 | 5.73 | 7,514 | X | X |
| ReliVul | 1178 | 237 | 324,216 | 13.66 | 24278 | √ | √ |

根据表格，可以得出如下的结论：

ReliVul 包含1178个项目，覆盖了237个CWEs类型，远多于其他数据集，这表明ReliVul具有更广泛的代表性，从而提供更全面的漏洞分析能力。相比之下，Juliet是人工合成的数据集，没有来自真实的项目，Devign和D2A来源的项目只有个位数，当使用这些数据集进行训练时，漏洞检测模型很难推广到真实的其他项目中，难以提升模型的泛化能力。在函数数量上，ReliVul包含324,216个函数，虽然不及D2A和DiverseVul，但仍然比其他数据集要多，这是因为D2A数据集从开源项目中抽取漏洞数据集，其数据样本大都没有CWE编号，这很难保证数据集的准确性，另外DiverseVul的漏洞函数占比是小于ReliVul的，经过计算可以看出ReliVul要比DiverseVul的漏洞样本多两倍多。ReliVul的所有样本均附加了提交信息，这提供了数据的来源，有利于研究人员核实数据的准确性，也便于根据具体项目进行漏洞分析。另外，ReliVul对漏洞行号进行了标注，这是其显著优势之一。相比之下，Devign和DiverseVul没有标注漏洞行号。尽管Juliet、Big-Vul和D2A也标注了漏洞行号，但ReliVul通过结合PoIs特征和代码变更信息的方法，提供了更精确的标注结果。ReliVul提供了额外的信息，如漏洞触发路径等，这在其他数据集中是缺失的。这种额外信息对于深入研究漏洞的触发条件和影响范围非常有价值。

ReliVul 在多个方面展示了其优越性，具体体现在以下几个方面：

• **数据多样性和代表性**：ReliVul涵盖了1178个项目和237个CWE类型，远超其他数据集。这说明ReliVul的来源具有多样性，能更好地反映现实世界

中的漏洞情况，利用ReliVul数据集训练出的模型可能具有更好的泛化性能。

• **真实性和复杂度**：ReliVul的样本都是来自真实的开源项目，在每个样本中都含有提交的链接和信息。这说明ReliVul的样本更具有真实性，能够更接近实际项目中的代码分布。另外ReliVul的样本大都来自各个大型项目，例如Linux、Qemu等，这些项目功能复杂多变，代码的复杂性较高，能够提供更为有价值的样本。

• **详细的漏洞触发路径**：ReliVul不仅标注了漏洞行号，还生成了漏洞的触发路径，这是在该领域的第一次尝试。这些漏洞触发路径为漏洞样本提供了更详细的上下文信息，有利于研究人员更好地理解漏洞的触发条件，有利于训练出更为强大和实用性的漏洞检测模型。

•**自动化构建流程**：从漏洞信息获取，到漏洞代码和补丁代码的标注，再到触发位置和触发路径的标注，ReliVul都是可以自动完成的，无需人工干预，这说明ReliVul可以更为高效地构建大量高质量漏洞样本，可以支持构建漏洞收集平台，创造实际应用价值。

总之，ReliVul在数据多样性、真实性、复杂度、和标注粒度上都表现出色，填补了现有数据集的空白，为代码行级的漏洞检测领域提供了更为可靠的数据支撑。这一创新方法将有利于推动软件安全领域的发展。

3.3.2准确性对比实验

由于现有的漏洞分析工具，包括机器学习和深度学习方法，都难以准确地判断代码中是否含有漏洞，参考目前大多数工作在评估数据集时的方法[[9](#bookmark187),[29](#bookmark207)[,80](#bookmark261)]，本文通过人工分析每个数据集的标签机制来确定漏洞标签是否正确。本文检查了每个数据集的70个随机样本，首先提取了每个数据集的漏洞代码对应的提交信息。除了Juliet和Devign数据集，其他数据集都提供了提交信息的ID，D2A数据集还提供了静态分析工具分析的结果。然后根据提交信息、源代码和漏洞修复信息，并结合静态分析工具和大语言模型进行判断，最终得出漏洞样本标注是否准确的结论。实验结果如表[3.7](#bookmark170)所示。

可以看出，ReliVul的数据集在准确性方面表现出色，达到了0.871。这一结果显著高于其他自动化标注的数据集，如Big-Vul（0.557）、D2A（0.286）和DiverseVul（0.629）。虽然Juliet的数据集准确性为1.000，这是因为其数据都是人工撰写生成的，而非真实世界中的漏洞。虽然这保证了标签的准确性，但缺乏现实世界的复杂性和多样性。Devign的准确性为0.800，相对较高，但仍低于ReliVul。这是由于Devign通过关键字分析有效地区分了与漏洞相关和无漏洞相关的提交，并在此基础上，研究人员手动筛选与漏洞相关的提交，以确保结果的准确性，手动筛选过程

表3.7各漏洞数据集在准确性、唯一性和完整性上的实验结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据集 | 自动标注 | 准确性 | 唯一性 | 完整性 |
| Juliet | X | **1.000** | 0.263 | **1.000** |
| Devign | X | 0.800 | 0.584 | 0.744 |
| Big-Vul | √ | 0.557 | 0.630 | 0.824 |
| D2A | √ | 0.286 | 0.121 | 0.781 |
| DiverseVul | √ | 0.629 | 0.641 | 0.495 |
| ReliVul | √ | 0.871 | **0.736** | 0.950 |

虽然提高了准确性，但也增加了工作量和主观性，且其漏洞来源仅有两个大型项目，不利于推广到其他项目的漏洞检测任务上。Big-Vul通过从CVE Details数据库的关联补丁中抓取数据，将所有发生变更的函数均标记为有漏洞。其依赖于外部漏洞报告，受到补丁质量和误报或漏报的影响，导致数据集不准确。D2A的准确性仅为0.286，是所有数据集中最低的。D2A通过在六个开源仓库的漏洞修复提交前后运行静态分析工具来收集源代码。易受攻击的类由修复后消失的工具警告组成。这种方法的有效性高度依赖于所使用静态分析工具的准确性和全面性。如果工具产生大量误报或漏报，将直接影响数据集的准确性。DiverseVul通过爬取安全问题网站、提取修复漏洞的提交和源代码等方法构建漏洞样本。虽然该数据集相对之前的工作扩充了数据样本，但由于其样本是函数级别的，样本的复杂度和数据依赖关系相对简单。此外，标注的粒度也在函数级别，没有对漏洞的触发位置、触发条件进行标注，这限制了其在细粒度分析中的应用。

总之，ReliVul结合了PoIs特征和代码变更信息，通过静态分析工具、机器学习模型和大语言模型进行综合判断，显著提高了标签的准确性，另外，ReliVul的数据来源于真实的开源项目，具有更高的真实性和复杂度，能够更好地反映现实世界中的漏洞情况，相对于现有的数据集，具有真实准确的特点。

3.3.3唯一性对比实验

唯一性是评估漏洞数据集时的重要指标，因为它直接影响后续模型的泛化能力和性能。唯一性判断的是数据集中是否存在重复的样本，如果某个数据集中存在的重复性样本太多，且分布在后续模型的训练和验证数据集划分中，那么对于模型的评估将起不到太大价值。本文使用Allamanis[[81](#bookmark263)]中的检测代码克隆的工具对各个数据集进行计算，得出的结果如表[3.7](#bookmark170)所示。

从这些结果可以看出，ReliVul的表现最佳，在唯一性方面的指标达到了0.736。

相比之下，D2A的数据集唯一性最低，仅为0.121，Juliet的数据集唯一性也非常低，为0.263，其他数据集的唯一性指标分布在0.6左右。Juliet数据集均是人为编写的，其样本之间的结构化非常相似，对于有漏洞和无漏洞的样本均存放在“Bad”和“Good”函数中，且大部分样本仅仅只更改了几行代码，这使得其代码的重复性较高。结合上一小节的分析，虽然D2A的样本数量最多，但其样本来源仅只有六个开源的仓库，这说明其在构建样本时存在同一个项目中过去多个版本的代码，这些版本的代码相似性较高，导致代码的重复率最高，这种数据来源较为单一的数据集很难训练出泛化性能高的漏洞检测模型。类似的，Devign的样本只有两个项目（FFmpeg和Qemu），但其样本数量较少，所以唯一性没有D2A那么低，这可能也是因为Devign中的数据样本都是研究人员手动筛选的，所以可能有意避免了一些重复的内容。Big-Vul和DiverseVul的数据来源是一些CVE数据库，这些数据库内的样本来源丰富，涵盖多个大型项目和漏洞类型，一定程度上避免的重复的代码，但其代码重复率仍然在ReliVul之下，这可能是因为这些数据集中的样本不平衡导致的，大量的“无漏洞”标签的代码难免会导致代码的重复率。对于ReliVul，其从两个大型CVE漏洞数据库中爬取数据，涵盖上千个大型项目和237个CWE编号，从数据分布上看大大避免了代码重复率。由于ReliVul针对大部分漏洞代码都提供了补丁代码，实际上补丁代码和漏洞代码就不可避免地带来一定的代码重复率，但这对于数据集的正负样本的平衡性和丰富性又是至关重要的。虽然如此，ReliVul仍然取得了最高的唯一性，使其具有更能代表现实世界的代码情况，成为漏洞检测领域的理想数据集。

3.3.4完整性对比实验

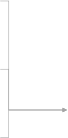
在本实验中，本文通过分析C/C++函数语法自动检查不完整的代码片段，以检[测缺失的信息，根据第3.2.2](#bookmark64)[小节的指标描述，本文将实验结果放在了表3.7](#bookmark170)中。从这些结果可以看出，除了Juliet取得100%的完整性，各数据集均存在不完整性的现象，ReliVul的数据集在完整性方面表现次佳，达到了0.950，相比之下，DiverseVul的数据集完整性最低，仅为0.495。Juliet数据集是基于预定义规则自动生成的程序示例，每个程序都包括一个易受攻击版本和一个非易受攻击版本。由于这些示例是精心设计的，因此不存在代码截断和缺失的问题，但其代码复杂度和真实性却是所有数据集中最差的。Devign和D2A通过在开源仓库的漏洞修复提交前后运行静态分析工具来收集源代码，可能存在分析工具不准确，人工难以排查全部代码，且项目中的提交可能本身存在不完整的现象，这导致数据集中存在代码截断不完整的问题。Big-Vul和DiverseVul的漏洞样本来自漏洞数据库，但它们均进行了不同程度的代码切片，只提取了可能发生漏洞的代码部分，这大大增加了漏洞代码

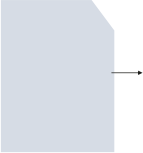
的不准确性。DiverseVul的数据样本全部在函数级别，存在函数嵌套未定义、引用不全等代码不完整现象，而且其样本的复杂度和数据依赖关系也相对简单。

相比之下，ReliVul在代码完整性上远超其他真实漏洞数据集，ReliVul不仅收集了漏洞代码和补丁代码，还利用漏洞特征标注了漏洞代码行和触发路径，这种基于PoIs对数据依赖和控制依赖的分析，大大保证了原始代码的完整性，且在抽取漏洞代码时可避免大量无关代码，在漏洞复杂度、真实性、准确性、唯一性和完整性中取得了最好的综合性能，使得ReliVul成为研究和开发高质量漏洞检测和防御系统的理想数据集。

3.4本章小结

本章节详细介绍了本文提出的自动化漏洞代码库构建框架的设计与实现，在此基础上收集了一个高质量漏洞数据集ReliVul，并通过一系列实验验证了其有效性。ReliVul相对现有的漏洞数据集，包含丰富的漏洞来源、类别和代码行级别的粒度，还通过实验验证了其在准确性、唯一性和完整性上的突出性能，这些结论验证了ReliVul数据集是一个高质量的漏洞数据集，对于漏洞检测领域和网络安全任务具有较大应用价值。





**void Func() {**

**int64\_t \* data; data = NULL; if(staticTrue) {**

**data = new int64\_t[100]; size\_ti;**

**for(i = 0; i < 100; i++) data[i] = 5LL;**

**delete [] data;**

**}**

**if(staticTrue) {**

**printLine(data[0]);**

**} }**

**// 标签：CWE-416**

Joern

特征 提取

|  |
| --- |
| yi |

L1

L2

LN

漏洞数据样本

第4章 基于多模态注意力的源代码漏洞检测技术

本章节首先对提出的基于多模态注意力的源代码漏洞检测模型进行了详细的描述，包括模型概述、数据预处理与特征提取、多模态注意力模型的设计、分类模型以及定位模型的设计。然后通过丰富的实验进行了性能的验证，通过四个数据集和八个对比方法验证了本文模型在漏洞检测和漏洞定位上的优越性。

4.1基于多模态注意力的源代码漏洞检测

本文充分考虑漏洞检测模型的学习效果和漏洞定位能力，采用不同的网络结构学习不同结构的代码表征，并将结果集成用于漏洞代码的检测，另外，通过利用各类漏洞的PoIs特征，让模型重点学习可能发生漏洞的代码，并最终实现漏洞代码行的预测。本节将首先对所提的模型进行概述，然后分别介绍数据预处理、模型的设计与训练过程以及实验验证的方法。

4.1.1模型概述

本文提出了一种基于多模态注意力的漏洞检测与定位模型，叫作MaliVD（Multimodal Attention-based Line-level Vulnerability Detector），MaliVD的结构如图[4.1](#bookmark140)所示。

1. 特征提取模块2.多模态注意力学习模块3.分类模块

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NCS PoIs  AST  Mask  Mask       |  | | --- | | PoIs | | |  |  | ...   |  | | --- | |  | | ResNet |  |  | | --- | | **Tree-LSTM** | |  |  |  | | --- | | **GGNN** | |  |   **WLJAN**  单词 嵌入  节点 嵌入  边 嵌入 |  | +    线性 连接  排序 推荐 |
| CPG | |  | | --- | | PoIs |       Mask |

漏洞类型

触发位置

图4.1 MaliVD：基于多模态注意力的源代码漏洞检测与定位模型

如第[2.2](#bookmark36)小节所述，源代码可以表示成三种结构的代码表征：序列结构表征、树结构表征和图结构表征。对于不同类型的表征本模型将分别使用先进的序列结构、树结构和图结构的带有注意力机制的神经网络进行学习。相比于现有的工作直接将复杂的图神经网络模型应用在多种表征上，本文的方法更为合理且能够大大减

少图神经网络的复杂度和检测时间。在三类网络模型分别学习不同表征后，再使用一个集成模型将不同模型的结果集成起来，体现了对表征的综合考虑。这也符合人类和静态分析工具在进行漏洞检测时，先分别考虑各个表征（例如某个数据流，某个控制条件），然后综合起来看是否符合漏洞的模式从而得出判断。

另外，本文为了能够实现漏洞代码行的定位，重点考虑漏洞的PoIs特征，并采用启发式的方式引导模型重点学习可能发生漏洞的代码，具体来说，在将代码编码成序列、树和图形结构的数据后，分别根据PoIs所在的位置，提取带有PoIs的子序列、子树和子图，并在PoIs所在位置之后的语句和路径以及其他没有PoIs的语句和路径设置掩码，原始特征和掩码后的特征均通过三种模型进行学习。最后将两部分的结果进行拼接，最终连接到每种漏洞类型的分类结果上。利用模型学习过程中的对每个节点的注意力，可以计算得到模型对每行代码可能触发漏洞的预测概率，从而定位到漏洞可能触发的位置。

4.1.2数据预处理与特征提取

**1）代码表征**

对于一段C/C++代码，其本质上是由单词和符号组成的序列，但由于其含有丰富的结构信息，故可以将其转换成树结构表示和图结构表示。树结构表示主要以抽象语法树（AST）为主；图结构表示主要包括：控制流图（CFG）、数据流图（DFG）、程序依赖图（PDG）和代码属性图（CPG）等。其中CPG结合了AST、CFG和PDG的优点，能够较为全面地反映源代码的语句结构、数据流和控制流等关键信息，图[4.2](#bookmark142)给出了代码属性图的生成过程示例。

|  |
| --- |
| 1 void FUNC1(VAR1, VAR2) {  2 int VAR3 = Source();  3 if (VAR1 < VAR2) {  4 int VAR4 = FUNC2(res); 5 }  6 } |

解析



合并



cfalse

false

Dx



Dres

true

ε

ε

ε

ε

ε

ENTRY

DECL

DECL

PRED

DECL

EXIT



Dy



DECL



|  |
| --- |
| VAR1 |

AST edge

CFG edge

PDG edge



|  |
| --- |
| VAR2 |



|  |
| --- |
| = |



CALL

|  |
| --- |
| VAR1 |

|  |
| --- |
| VAR3 |



|  |
| --- |
| Source |



ctrue

|  |
| --- |
| < |

|  |
| --- |
| = |



|  |
| --- |
| VAR2 |

CALL

|  |
| --- |
| VAR4 |



ARG

|  |
| --- |
| FUNC2 |



|  |
| --- |
| VAR3 |

图4.2代码属性图（CPG）生成过程示例

**2）数据预处理**

对于一段代码样本，其中可能包含注释、字符串、用户自定义变量名和函数名等，需要清洗和标准化，然后才能进行特征的提取和输入到模型中。本文数据预处理的方式如下：

• **获取源代码**：分析漏洞数据集的结构和特点，有些数据集提供了所有样本的源代码，有些则仅提供了源代码的地址，需要根据代码地址和爬虫脚本爬取源代码。

• **去除注释和空行**：利用一组正则表达式去除代码中的注释（包括单行注释和多行注释）、空行和多余的空格。

• **常量替换**：将代码中的常量（如数字、字符串等）统一替换成特定单词。例如，将所有的整数替换成“NUM”加上编号的形式，将所有的字符串替换为“STR”加上编号的形式。

• **标识符匿名化**：将代码中用户自定义的函数名和变量名进行匿名化处理，按照顺序重命名为标准化的形式，例如将函数名依次重命名为“FUNC0”和“FUNC1”等，将变量名重命名为“VAR0”和“VAR1”等。

通过上述步骤，本文将代码预处理成标准化代码的形式，大大减少了原始代码中的噪声，为下一步特征提取和模型训练做准备。

**3）特征提取**

在完成代码的清洗和标准化后，需要从预处理后的代码中提取有价值的特征。目前市场上有许多工具可用于解析源代码并生成代码表示，利用这些工具可以有效生成AST，CPG等代码表示并进行静态程序分析。本文使用Joern工具进行特征提取，Joern是专门针对C/C++的分析工具，可直接生成AST、CFG、CPG等，并通过一个数据库进行代码属性查询，还支持自定义特征提取。本文使用代码顺序序、抽象语法树和代码属性图作为三类特征的基本结构，并在此基础上，添加节点信息和边信息。

对于预处理后的代码，为了统计在经过数据处理后这些单词有多少个，本文分别统计了关键字、符号、系统函数、标准化后的函数名和变量名、节点本身包含的属性特点以及PoIs特征，共得到1289个单词或符号，另外本文对于树和图的边进行的信息提取，包括AST（抽象语法树）、CFG（控制流图）和DFG（数据流图）三类边类型以及[14 种节点之间的调用关系。表4.1](#bookmark172)列举了本模型使用的节点信息和边信息。

将这些节点信息和边信息分别加入到NCS、AST和CPG中就得到了本文三种结构的代码特征。值得一提的是，对于特征中的PoIs节点，本文通过将其他部

表4.1代码特征的节点信息和边信息

类型

名称

个数

|  |  |  |
| --- | --- | --- |
| 节点 | 关键字（include, return, ...） 符号（+ ，{ ，; ...）  系统函数（malloc, read, ...）  标准化变量（VAR1, VAR2, ...）  标准化函数（FUNC1,FUNC2, ...） 属性特征（dtype,expression, ...）  PoIs 节点 合计 | 98  31  928  100  100  26  6  1289 |
| 边 | 类型（AST, CFG, DFG）  调用关系（函数, 返回值类型, ...） 合计 | 3  14  17 |

分设置掩码的形式重新得到了三种带掩码的特征。具体来说，对于代码序列特征，如果某行代码的某个节点被标记为PoIs节点，那么保留该行PoIs节点及其之前的节点，对于没有PoIs的语句以及每行PoIs节点之后的节点，均被填充为0；对于AST，如果树中某个节点被标记为PoIs节点，那么从根节点到该节点的所有路径都可能是漏洞触发的漏洞，需要保留，其他不在这些路径上节点都被填充为0；对于CPG，和AST类似，本文保留了从根节点到PoIs节点的所有路径，而其他位置被做了掩码。对于没有做掩码的部分，这么操作实际上得到了一个子序列、子树和子图，但为了保证神经网络的输入大小相同，本文通过做掩码的形式进行处理。该步骤的目的是让模型能够重点学习可能触发漏洞的代码，从而更利于漏洞检测和漏洞定位。

通过在生成的NCS、AST和CPG特征中标注这些节点信息和边信息，能够为检测模型提供更为丰富的代码信息，这些信息对于代码的理解以及漏洞模式的提取至关重要。虽然AST节点和CPG节点中也包含部分特征信息，例如变量类型、函数返回值类型等，但是显式地指出这些特征信息能够让模型更好地理解代码的实际逻辑，真正学习到每一种漏洞代码的模式，以及区分和无漏洞代码的区别。

4.1.3多模态注意力模型设计

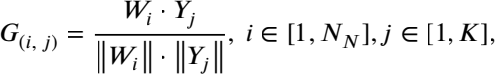
对于一个代码样本，经过特征处理和掩码，本文得到了6个特征空间，分别是NCS、AST、CPG和对应的掩码特征，本文使用三类神经网络模型进行学习，分别

是：单词-标签联合注意力网络（Word-Label Joint Attention Network，[WLJAN）[82](#bookmark265)]、[树结构长短期记忆网络（Tree-LSTM）[64](#bookmark230)[]和门控图神经网络（GGNN）[65](#bookmark234)]。接下来本小节将详细介绍这三类模型及学习过程。

**1）序列结构模型**

WLJAN是对RBLJAN[[82](#bookmark265)]模型的改进，原模型是对网络流量数据进行识别，本质上序列化的代码数据也像网络流量数据一样，具有结构化的序列特征，就像RBLJAN论文中提到的，传统的神经网络模型大都应用在图像和文本数据上，直接应用在流量数据上不够合理，这是因为流量数据既没有像图像这么大的空间特征也没有像文本这么大的词汇空间，流量数据仅有256个字节。这一点和代码很相似。实际上经过处理后的代码节点也只有1289个特征，且大部分节点不常用到。所以本文改进了原论文中的模型，并得到本文处理序列代码特征的重要模型。

具体来说，本文将代码中的节点类比做网络流量中的字节，首先将每个单词和标签编码成向量的形式，假设代码序列的长度为NN，标签的数量有K个，那么经过嵌入后就得到NN个单词向量和K个标签向量Y = {yi其中wi和yi分别表示第i位置的单词的嵌入向量和第i个标签向量，Qi表示位置i的位置编码，那么接下来根据式（[4.1](#bookmark266)）计算单词和标签的相似度矩阵。

（4.1）

其中G = [G(i, j)]是就是得到的相似度矩阵，将输入到神经网络中学习单词注意力和标签注意力。

与原论文不同的是，为了更好地处理代码的序列信息，本文的核心网络使用

门控循环单元GRU[[83](#bookmark268)]来进行学习，GRU的计算方式如式（[4.2](#bookmark267)）至（[4.5](#bookmark269)）所示。zt=σ(wz⋅[ℎt−1,xt] + bz) ,（4.2）rt =σ(wr⋅[ℎt−1,xt] + br) ,（4.3）

ℎt= tanℎ(wℎ⋅[rt⊙ℎt−1,xt] + bℎ) ,（4.4）

̃

ℎt=(1−zt)⊙ℎt−1 + z t⊙ℎt,（4.5）

̃

其中z、r和ℎ分别表示更新门、重置门和候选隐藏层的参数，w(∗)和b(∗)表示对应公式中各个门的变换矩阵和偏移量向量。在t时刻，xt表示当前时刻的输入信

̃

息，ℎt−1为上一时刻的隐藏状态，ℎt代表候选的隐藏状态，rt和zt分别表示重置门和更新门，σ和tanℎ为sigmoid和tanℎ激活函数，⊙表示点积计算。通过GRU对代码序列的学习，使得整个序列的节点逐个往下传递，同时会保留相关的信息

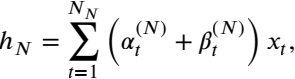
传递到下一个单元，适合长序列的学习。

假设GRU每一步的输出层的维度为fN，将每一步的输出ℎt∈ℝfN进行加权连接和池化操作，就得到了每个单词的注意力分数，可以通过式（[4.6](#bookmark270)）计算得到：

= tanh（4.6）

其中MP和FC分别指最大池化（Max Pooling）和全连接（Fully-Connected）层，ATTt(N)代表单词t的注意力分数。

如前所述，序列特征包含原始序列特征和带掩码的序列特征，实际上两个代码序列都经过了上述的计算，并分别得到了对应的注意力分数，假设不带掩码和带掩码的注意力分数分别是αt(N)和βt(N)，然后将两个注意力分数相加，并和单词向量加权求和，就得到了单词注意力向量。对于标签注意力和生成整个代码的表征向量的方法，采用和原论文相同的方式。最终本文通过WLJAN模型，得到式（[4.7](#bookmark271)）的代码表示。

（4.7）

其中ℎN∈ℝE表示WLJAN模型的输出，即E维的序列代码的高层表示向量。

**2）树结构模型**

对于AST这类树结构表征，使用树结构的神经网络学习较为合适，这里介绍Tree-LSTM的一种方式Child-sumTree-LSTM，其相对于经典LSTM来说，其父节点的信息是来源于所有子节点的信息的传递，该模式将子节点的隐藏层都求和然后再去更新父节点的隐藏层。其计算方式如式（[4.8](#bookmark272)）至式（[4.14](#bookmark273)）所示。

（4.8）

ij =σ(w(i)xj+ U(i)j+ b(i)) ,（4.9）

fjk=σ(w(f)xj + U(f)ℎk+ b(f)) ,（4.10）

oj =σ（4.11）

uj= tanh（4.12）

cj= ij⊙uj+ kfjk⊙ck,（4.13）

ℎj= oj⊙tanh (cj ) ,（4.14）

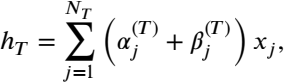
其中W(∗)、U(∗)、b(∗)分别是各个门输入特征的变换矩阵、隐藏层的变换矩阵和偏移量，i、f、O作为上标分别表示输入门、遗忘门和输出门对应的参数，可以从式（[4.8](#bookmark272)）看出，父节点的隐藏层是所有子节点求和而来，所以对子节点的个数没有限制。传统的LSTM，可以看作所有子节点个数都为1的Tree-LSTM模型。

通过自底向上的计算后，本文得到每个节点的输出向量，为了能更好地进行漏洞类型检测和漏洞代码行定位，本文引入一层注意力机制计算每个节点的权值，计算方式如式（[4.15](#bookmark274)）所示：

⊙tanhσ激活函数将输入进行归一化，挑选出对神经网络贡献较大的节点，tanh激活函数用于确定该节点是正向作用还是负向作用，同时加快收敛的速度和防止梯度消失。

所以ATT就是第j个节点的注意力分数。与序列特征一样，树形特征也有原始

特征和带掩码的特征，这两部分分别使用Tree-LSTM进行学习，假设不带掩码和带掩码的注意力分数分别是αt(T)和βt(T)，然后将两个注意力分数相加，并节点向量加权求和，得到最终的树形高层表示如式（[4.16](#bookmark275)）所示。

（4.16）

其中ℎT∈ℝE表示Tree-LSTM模型的输出，也是E维的序列代码的高层表示向量。

**3）图结构模型**

对于数据流、控制流等程序依赖关系可以表示成图的形式，可以表示成G =(V, E)的形式，其中v∈V中存储D维的特征向量值，e∈E中存储D×D维边的邻接矩阵。这里以GGNN为例介绍图模型的传播和输出方式，其传播模型如式（[4.17](#bookmark276)）至式（[4.22](#bookmark277)）所示。

ℎ1)= [X,0]T,（4.17）

at) = A[ℎt−1)T⋯ℎ|1)T]T+ b,（4.18）

（4.19）

（4.20）

= tanh（4.21）

ℎt) =(1−z)⊙ℎt−1) + z⊙t),（4.22）

其中ht)表示t时刻节点v的状态，A是E和E的转置拼接起来的D×2D的矩阵，

Av表示从矩阵A中选出对应节点v的两列，式（[4.19](#bookmark278)）至式（[4.22](#bookmark277)）类似GRU的计算过程，也是通过邻居节点来更新本节点的状态。

在经过GGNN计算得到每个节点的输出后，类似于式（[4.15](#bookmark274)），本文同样引入一层图注意力机制来筛选哪些节点和整个图的输出最相关，最终得到第v个节点

的注意力分数ATTG)，如式（[4.23](#bookmark279)）所示：

ATTG)=σ(FC ([h,xv]))Θtanh (FC ([h,xv])), v∈V,（4.23）

另外，本文希望图神经网络的输出为对应的特征向量（而不是特征图），可将每个节点加权求和得到。由于原始特征和经过掩码后的特征均进行了计算，假设两部分得到的注意力分数分别是αt(G)和βt(G)，那么最终得到的图表示向量如式（[4.24](#bookmark280)）所示。

（4.24）

其中hG∈RE表示GGNN模型的输出，也是E维的序列代码的高层表示向量。

4.1.4分类模型设计

为了将模型对代码的学习结果连接到待分类的漏洞标签上，本文首先将上述得到的三个高层表示向量hN、hT和hG进行连接：

h =[hN llhT llhG] ,（4.25）

其中ll代表向量的拼接，h∈R3E代表最终的代码高层表示，由序列高层表示、树形高层表示和图形高层表示组成。然后本文使用一个线性层将代码表示连接到待分类的k个漏洞标签上，如式（[4.26](#bookmark281)）所示。

= softmax (Wh + q) ,（4.26）

其中W∈Rk×3E and q∈Rk代表线性连接层的权重和偏移量，代表分类的结

果，softmax是归一化函数。本文使用*Focal Loss*[[84](#bookmark282)]作为本文分类模型的损失函数，因为它在不平衡的数据集中表现优越，可以更有利于学习分类困难的样本。*FocalLoss*的计算方式如式（[4.27](#bookmark283)）所示。





（4.27）

其中y代表真实的标签（Ground Truth），y代表在真实标签上的预测概率，α和Y

是*Focal Loss*的超参数。本文将α设置为1减去该标签的样本数量占总样本数的

比例，将Y设置为2[[82](#bookmark265)]。

4.1.5定位模型设计

模型对漏洞触发位置的预测可以通过神经网络每个节点的注意力计算得到，序列、树形和图形神经网络的注意力如式（[4.6](#bookmark270)）式（[4.15](#bookmark274)）和式（[4.23](#bookmark279)）所示。由于MaliVD最终预测的是原始代码触发的行号，一行代码可能对应多个AST或CPG的节点，本文首先将某行代码对应的序列注意力、该行对应的所有节点的树形注意力和图形注意力进行相加，作为该行代码的注意力。定义MASK(⋅)函数为掩码函数，即针对序列数据、树形数据和图形数据根据没有PoIs的位置进行掩码（置零）。这样处理的是为了保留了PoIs所在的代码行，因为只有PoIs的位置才可能是触发漏洞的位置。假设序列、树形和图形注意力相加后的注意力为ATT∈ℝM，其中M代表代码样本的总行号。对ATT进行掩码和归一化操作就得到了模型对每一行可能触发漏洞的概率Li，如式（[4.28](#bookmark284)）所示。

L = [L1, ...LM]



（4.28）

漏洞检测模块将一段代码序列分类为指定的漏洞类别，使用*Focal Loss*作为损失函数。同样，在漏洞定位模块，模型预测的是每行代码可能是漏洞触发位置的概率，本文同样使用*Focal Loss*作为定位模块的损失函数，类似于式（[4.27](#bookmark283)），其中的y和y就代表真实漏洞触发的行号以及模型在真实漏洞触发行号上的预测概率。通过分别对漏洞类型和漏洞行号进行梯度回传，模型不仅能够识别出代码中存在的漏洞，还能准确地定位到具体的漏洞触发位置。

4.2实验设置

为了全面评估MaliVD的性能，本文选取了5个大型漏洞数据集和8个流行的对比方法进行实验。本小节将详细介绍实验过程中使用的数据集、比较基线和评价指标。

4.2.1数据集

由于本文提出的漏洞检测模型不仅能识别代码是否含有漏洞，还能识别漏洞的类型以及定位到漏洞的位置。本文在选取数据集上选择了不同来源、和不同粒度的数据集，分别是：Juliet、Devign+Reveal、Big-Vul、DiverseVul和本文提出的漏洞数据集ReliVul。这些数据集在第[2](#bookmark24)章和第[3](#bookmark48)章都有所描述。其中Juliet是人工撰写和工具生成的数据集，其余数据集来自真实的开源项目，由于Devign和Reveal

数据集都来自几个大型的开源项目，且两者都没有标注具体漏洞类型，仅标注了有漏洞和无漏洞两类标签，所以本文将其合并并用于二分类的实验。Juliet、Big-Vul和ReliVul都标注了漏洞类型和漏洞行号，这三个数据集将用于漏洞定位的实验，以评估模型寻找漏洞触发位置的能力，也可以评估模型的可解释性。

4.2.2比较基线

由于基于静态扫描工具和基于机器学习的漏洞检测方法的准确率较低，且本文主要研究基于深度学习的源代码漏洞检测，所以在选择对比方法时也仅选择了基于深度学习的模型。本文一共选择了8个对比方法：Code2Vec[[85](#bookmark285)]，[Devign[7](#bookmark185)]，[ReVeal [33](#bookmark211)]，IVDetect [[48](#bookmark229)]，Funded[[47](#bookmark231)]，MGVD[[59](#bookmark236)]，[LineVul [54](#bookmark240)]和VulChecker[[11](#bookmark189)]，其中LineVul和VulChecker的检测粒度做到了代码行级别，可用于漏洞定位任务的对比实验。具体来说，这些方法的介绍如下：

• Code2Vec[[85](#bookmark285)]：使用神经网络模型将代码片段映射到一个固定长度的向量空间，然后通过这些向量进行分类，Code2Vec提出主要是用于代码理解和功能预测任务，但也适用于漏洞代码检测。

• Devign[[7](#bookmark185)]：以AST为中心，将不同级别的数据依赖和控制依赖编码为联合图，然后通过GNN和CNN模型学习代码表征并进行漏洞检测。

• ReVeal [[33](#bookmark211)]：使用图神经网络(GNN)检测漏洞，将控制流、数据流、语法树和依赖图整合到代码属性图(CPG)中进行综合代码分析。

• IVDetect [[48](#bookmark229)]：通过PDG表示源代码，并从中提取信息作为代码的向量表征，通过带有注意力的GCN学习代码的上下文感知表示，从而进行粗粒度漏洞检测，同时利用GNNExplainer来提供漏洞相关语句的细粒度解释。

• Funded[[47](#bookmark231)]：使用多关系GGNN进行源代码的漏洞检测，通过捕捉程序的语法、语义和流信息，来生成用于下游任务的更好代码标识。其还利用迁移学习来移植漏洞检测模型到不同的编程语言上。

• LineVul [[54](#bookmark240)]：是基于CodeBERT的序列模型，将代码块作为输入，并使用Code- BERT模型对源代码进行分类，能够精确定位代码中的漏洞行。

• MGVD[[59](#bookmark236)]：使用两种方式将AST和PDG编码为两张图，结合原始代码序列，编码为特征矩阵，然后使用GAT和多层CNN进行学习并识别代码中的漏洞。

• Cppcheck[[5](#bookmark183)]：是一个广泛使用的C/C++代码安全分析工具，其主要依赖于一套规则来匹配代码中的问题，包括内存泄漏、逻辑错误、未使用的变量等。上述对比方法使用了不同的代码特征和检测方法，通过将这些方法同MaliVD

进行对比，可以更全面地评估MaliVD在漏洞检测和漏洞定位任务中的优势和不

足，为进一步优化和改进提供参考。

4.2.3评价指标

**1）分类指标**

根据不同的数据集和研究目标，在对漏洞类型进行识别时，可以看作二分类或多分类问题。对于二分类问题，漏洞检测模型用于判断代码是否包含漏洞；对于多分类问题，模型则可以识别代码中存在的具体漏洞类型（例如，缓冲区溢出）或者将代码识别为无漏洞的类型。本文将使用常见的查准率（Precision，P.），召回率（Recall，R.），F1分数（F1-score，F1.）和准确率（Accuracy，Acc.）来作为漏洞检测模型的评价指标。

对于二分类问题，即数据集中只将样本分成“有漏洞”和“无漏洞”两个类别。在这种情况下，本文先定义以下四个基本概念：

• 漏报（False Negative，FN）：实际有漏洞但被模型错误地预测为无漏洞的样本数。高漏报率意味着模型容易遗漏实际存在的漏洞，导致漏洞没有被及时发现和修复，从而引起安全风险。

• 误报（False Positive，FP）：实际无漏洞但被模型错误地预测为有漏洞的样本数。高误报率意味着模型容易将正常的代码误判为有漏洞，这增加了安全人员排查漏洞的工作量。

• 真正例（True Positive，TP）：实际有漏洞且被模型正确预测为有漏洞的样本数。高真正例率意味着模型能够有效地检测出实际存在的漏洞，确保漏洞被及时发现，从而保证代码的安全性。

• 真负例（True Negative，TN）：实际无漏洞且被模型正确预测为无漏洞的样本数。高真负例率意味着模型能够有效地排除正常的代码，避免对正常代码进行不必要的分析。

基于以上定义，我们可以计算以下的评价指标：

查准率代表模型预测为有漏洞的样本中，实际有漏洞的样本所占的比例。查准率反映了模型在预测有漏洞时的准确性，即模型的“保守性”。高查准率意味着模型很少将无漏洞的样本误判为有漏洞。在漏洞检测中，高查准率可以减少误报，降低安全分析师的工作负担。其计算公式可以表示成式（[4.29](#bookmark286)）的形式。

Precision=.（4.29）

召回率代表实际有漏洞的样本中，被模型正确预测为有漏洞的样本所占的比例。召回率反映了模型在检测实际有漏洞样本时的能力，即模型的“敏感性”。高召回率意味着模型很少遗漏有漏洞的样本。在漏洞检测中，高召回率体现了模型具

有较低的漏报，即大部分的漏洞能够被模型检测出来。召回率可以通过式（[4.30](#bookmark287)）来计算。

Recall =（4.30）

F1分数通过综合考虑查准率和召回率，以它们的调和平均值来平衡两者之间的关系，是一个更全面公平的评价指标。高F1分数表明模型在查准率和召回率之间达到了良好的平衡，这意味着模型既不过于保守（减少误报），也不过于宽松（减少漏报）。因此在漏洞检测中，高F1分数意味着模型在减少误报和漏报方面都表现良好。其具体的计算公式如式（[4.31](#bookmark288)）所示。

F1 = 2×（4.31）

准确率代表模型正确分类的样本数占总样本数的比例。准确率反映了模型整体的分类性能，但当数据不平衡时（即某一类别的样本数远多于另一类），准确率可能不是一个理想的指标。在漏洞检测中，如果无漏洞样本远多于有漏洞样本，高准确率可能掩盖了模型在检测有漏洞样本时的不足。其计算公式为式（[4.32](#bookmark289)）。

Accuracy =（4.32）

然而，有些漏洞数据集不仅标注了代码是否包含漏洞，还标注了具体的CWE类型。为了更全面地评估漏洞检测模型在各类漏洞上的检测结果，我们将二分类的评价指标扩展到多分类的情况。在多分类问题中，模型的任务是将代码划分到多个类别中的一个，例如缓冲区溢出、命令执行以及无漏洞等，而每一种类别都可以参照上述定义计算查准率、召回率和F1分数。

为了综合评估模型在所有类别上的性能，本文采用宏平均（Macro-average）的方法。在对每个类别分别计算查准率、召回率和F1分数之后，取它们的平均值作为模型在该数据集上的结果。之所以没有采用依据类别数量进行加权平均的方式，这是因为现有数据集中的类别存在不平衡的现象，例如大部分数据集的“无漏洞”样本的数量占比较多，如果模型单纯将样本均分类到无漏洞的类别，那么也可以取得不错的结果。宏平均就可以避免这种情况。

**2）定位指标**

本文提出的MaliVD模型不仅可以识别代码的漏洞类型，还可以定位到漏洞触发的位置。由于MaliVD最终给每行代码赋予了不同的权值，权值越大代表漏洞代码越可能是漏洞触发的位置。这本质上是一个推荐问题，类似于一般的推荐系统，本文使用Top-k准确率和初始误报（Initial False Alarm, IFA）两个指标进行衡量模型漏洞定位的能力。

对于一段代码，模型能够预测出每行代码触发漏洞的概率，按照每行代码可

能触发漏洞的概率进行从大到小排序，如果在前k行代码中包含真实的漏洞触发位置，那么该模型在预测该样本的Top-k指标是准确的。所以对于一个数据集来说，Top-k准确率反映了模型能够在前k行推荐中包含漏洞触发位置的样本比例。其计算公式可以表示成式（[4.33](#bookmark290)）的形式：

Accuracy@k =包含实际易受攻击代码的前k行推荐数.（4.33）

总的易受攻击函数

初始误报（Initial False Alarm, IFA）是另一个衡量漏洞定位模型的指标。假设一种场景，安全分析师利用某个漏洞检测模型，得到了一系列推荐的可能触发漏洞的代码行号，那么他们需要从概率最大的代码行出发，依次排查模型预测错误的行号，直到发现真正的漏洞代码行。那么初始误报就是安全分析师在找到实际漏洞触发位置前需要检查的非漏洞代码行的数量，也就是模型误报的数量。低IFA值表示安全分析师在检查误报上花费的时间较少，体现了模型在定位方面的性能较好。

Top-k 准确率指标反映了模型对各类漏洞定位的准确程度，特别是如果模型能够在较低的k时仍能保持较高的准确率，那么该模型的定位能力是更好的。IFA指标则反映了模型在减少误报方面的能力，体现了模型在实际应用中的价值。通过综合使用这两个指标，可以全面评测漏洞定位模型的性能。

4.3实验结果

为了验证本方案提出的MaliVD的效果，本文设计了多个实验，包括验证模型在漏洞分类准确率、漏洞代码行定位方面的性能、通过消融实验验证各个模块的重要性以及针对模型在参数、收敛性以及检测速度方面的优化分析。

4.3.1漏洞检测实验

MaliVD同各对比方法在四个数据集上的漏洞检测实验结果如表[4.2](#bookmark174)和表[4.3](#bookmark176)所示。其中Devign数据集是进行的二分类实验（有漏洞和无漏洞），其余三个数据集因为有具体的CWE编号，本文将其分为六类进行实验（缓冲区溢出漏洞、值计算漏洞、资源管理漏洞、输入验证漏洞、权限管理漏洞和命令执行漏洞），表格中的结果为6种类别指标的平均值。接下来本小节针对每个数据集进行详细的实验结果分析。

**1）Juliet数据集**

Juliet 数据集是人工撰写或工具生成的漏洞样本示例，是一个正负样本相对平衡的数据集，且包含具体的CWE类型，各方法在Juliet数据集上各指标的实验结

表4.2各方法在Juliet和Devign数据集上的漏洞检测实验结果

模型

**Juliet Devign**

Acc . P. R. F1 . Acc . P. R. F1 .

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Code2Vec | 0.6129 | 0.5167 | 0.6287 | 0.5672 | 0.5932 | 0.5587 | 0.5823 | 0.5703 |
| Devign | 0.6310 | 0.4668 | 0.6524 | 0.5442 | 0.5966 | 0.5324 | 0.5889 | 0.5592 |
| ReVeal | 0.6754 | 0.5141 | 0.7055 | 0.5948 | 0.6023 | 0.5401 | 0.5694 | 0.5544 |
| IVDetect | 0.7436 | 0.5877 | 0.6543 | 0.6192 | 0.6177 | 0.5523 | 0.6288 | 0.5881 |
| Funded | 0.7109 | 0.5311 | 0.6437 | 0.5820 | 0.6098 | 0.5876 | 0.6138 | 0.6004 |
| LineVul | 0.6731 | 0.5121 | 0.6697 | 0.5804 | 0.6159 | 0.4903 | 0.6390 | 0.5549 |
| MGVD | 0.7441 | 0.6155 | 0.7992 | 0.6954 | 0.6569 | 0.5904 | 0.6291 | 0.6091 |
| MaliVD | **0.8167** | **0.6854** | **0.8112** | **0.7430** | **0.7533** | **0.6837** | **0.7283** | **0.7053** |
| 提升（%） | 9.76 | 11.36 | 1.50 | 6.84 | 14.67 | 15.80 | 13.97 | 15.79 |

果如表[4.2](#bookmark174)所示。可以看出，MaliVD在准确率、查准率、召回率和F1四个指标上都取得了最优的水平。MaliVD的准确率达到了0.8167，比第二高的MGVD（0.7441）提升了近10%。另外MaliVD在差准率和F1值上的表现也显著优于其他模型。在召回率上，MGVD也取得了将近80%的表现，但其查准率只有61.55%。总的来看，其他方法在F1分数指标上大约分布在0.5到0.6左右，且各类方法的召回率都显著高于查准率，这表明这些方法在检测漏洞时更注重漏报的减少，倾向于将样本分类为某些类别，而没有专注于区分各个类别的差异。MaliVD其实也存在召回率比查准率要高的问题，但相比其他方法，MaliVD的查准率也优于其他方法超过10%。

值得注意的是，MGVD相对其他方法表现较好，究其原因，可能是因为MGVD在特征选取上也选择了代码序列、AST和PDG三种表征，使其能够更好地捕捉代码的上下文和结构信息，这其实也是MaliVD的优势所在。该数据集的样本和漏洞的复杂度都比较低，但Devign、ReVeal、Funded和LineVul等方法的F1分数不足60%，IVDetect方法虽在F1分数上处于第三的水平，但其值仅有61.92%。这是因为Juliet的样本较为简单，但其正负样本之间的差异非常细微，大部分的漏洞样本和对应的补丁样本仅有一行代码之差，这些仅仅使用一种结构的代码特征或者使用没有精心设计的模型的方法难以捕捉这种细微的差距。相反，MaliVD利用多种代码特征，从不同的视角审视代码，特别是使用了PoIs特征定位漏洞可能触发的位置，使得模型可以捕捉到漏洞代码相对于补丁代码的区别，也能够学习到各类漏洞的模式，从而取得更好的性能。

**2）Devign数据集**

在Devign这个数据集上，MaliVD同样表现出色，F1分数达到了0.7053，比第二高的MGVD（0.6091）提升了约15%。Devign数据集虽然样本数量较少，但[其样本是经过人工筛选的，结合3.3.2](#bookmark70)的实验结论，其样本的准确性还是较高的。与人工和工具生成的Juliet数据集相比，Devign来自FFmpeg和Qemu两个真实的开源项目，其代码的复杂度更高，数据依赖关系更为复杂。但是该数据集只标注了正负两个类别，并没有标注具体的漏洞类型。但即便执行二分类任务，从表[4.2](#bookmark174)中可以看出，大部分方法在各个指标上的性能不如Juliet数据集。

根据实验结果，可以看出，类似的，使用单一模型的方法，例如基于线性模型的Code2Vec和LineVul，以及仅基于图神经网络的Devign和ReVeal模型，F1分数都在60%以下。像IVDetect、Funded和MGVD模型，在方法设计上使用了多特征或多模型，能够更好地捕捉代码的结构和语义信息，在该数据集上的表现也更好。值得注意的是，MaliVD在四个指标上的结果都超过次优方法10%以上，这说明，在更真实和更复杂的Devign数据集上，MaliVD具有更多的优势。MaliVD的优势在于其对多种代码表征的综合考虑以及其多模态注意力模型的设计，即使真实环境下的代码含有较多的噪声，MaliVD仍能取得较优的效果。另外，虽然PoIs特征是重点用来进行漏洞定位的，但其对漏洞检测也具有重要作用。这是因为PoIs正是指明了那些可能发生漏洞的位置，让模型能够重点去学习漏洞代码和良性代码的区别，从而在像Devign这种二分类的任务上取得最优的效果。

**3）DiverseVul数据集**

DiverseVul 数据集的来源是安全问题网站和开源代码平台，相比于前两个数据集，其扩充了数据样本，但该数据也在标签准确性和样本完整性上有明显的劣势。从表[4.3](#bookmark176)中可以看出，各方法在该数据集上的表现都有了明显的下降，除了上述分析了模型本身的劣势，[一个主要原因是该数据集的标签的准确性不够，如第3.3.2](#bookmark70)所述，其标签准确率仅有62.9%左右，而且其抓取的“无漏洞”的样本是来自“暂时安全”的开源代码，这也会导致标签不准确，从而不利于模型的学习。

另外，Funded方法的准确率达到了82.15%，超过了MaliVD的81.78%，在该数据集上，除了Code2Vec，其他方法的准确率都超过了0.7，这主要也是因为该数据集的正负样本不平衡的问题，DiverseVul漏洞数据集中的漏洞函数占比仅有5.73%，如果漏洞检测模型简单地将一些不确定的样本归为无漏洞，那么也有较大的概率分类正确，从而能取得较高的准确率，但这会引起大量的漏报现象，所以一些方法（Code2Vec、Devign和ReVeal）的查准率甚至不足50%。相比之下，MaliVD在这种不平衡的数据集上，依然能够平等的对待每一个类别，通过结合代码多种结

构的特征和漏洞的PoIs特征，在F1分数上达到了68.92%，相对于第二的MGVD提升了5.48%。这表明MaliVD在类别不平衡的数据集上表现更优，值得注意的是，现实世界的漏洞代码往往也是占据一小部分，而大部分的代码都是“暂时无漏洞的”，所以这也证明了MaliVD更具有实际应用价值。

表4.3各方法在DiverseVul和ReliVul数据集上的漏洞检测实验结果

模型

**DiverseVul ReliVul**

Acc . P. R. F1 . Acc . P. R. F1 .

Code2Vec0.6578 0.4078 0.4810 0.44140.5571 0.4452 0.5318 0.4847

Devign 0.7212 0.4156 0.5137 0.45950.5643 0.3957 0.5759 0.4691

ReVeal 0.7281 0.4375 0.5383 0.48270.6050 0.4786 0.6693 0.5581

IVDetect 0.7828 0.5638 0.6119 0.58690.63390.5713 0.6113 0.5906

Funded **0.8215**0.5189**0.7178**0.60240.6022 0.4613 0.5854 0.5160

LineVul 0.7548 0.5725 0.6254 0.59780.6250 0.4968 0.5665 0.5294

MGVD0.7344 0.6256 0.6837 0.65340.6189 0.5980 0.5964 0.5972

MaliVD 0.8178**0.6671** 0.7128**0.6892****0.7143 0.6865 0.7737 0.7275**

提升（%）-0.45 6.63 -0.70 5.4812.68 14.8015.60 21.82

**4）ReliVul数据集**

ReliVul 是由本文提出的数据集，具有高度的真实性、复杂性和准确性，详细标注了漏洞样本的CWE类型、触发位置和触发路径。相比其他数据集，该数据集的漏洞检测任务更具有挑战性。但有趣的是，各方法在ReliVul数据集上的表现比DiverseVul要高，这也侧面验证了DiverseVul数据集存在标签不准确和样本不平衡的问题。针对实验结果，可以看出，MaliVD的表现是最好，在四个指标上都取得了最优的表现。在准确率上提升了12.68%，在F1分数上提升了21.82%。对于其他所有方法，其F1指标都在60%以下。

除了多模态机制和PoIs注意力机制，MaliVD还有一个重要的改进是使用了Focal Loss损失函数，同大部分方法使用的均方误差损失（Mean SquareError Loss，MSE Loss）不同，Focal Loss可以重点学习那些样本数量少、难以识别的漏洞类别，而其他方法可能更偏向于那些样本数量多的类别，只是单纯追求整体损失的最小化。所以MaliVD利用Focal Loss专注于那些样本数量少的类别，而不是只追求整体的损失最小，所以MaliVD在宏平均的指标上都取得了最好的表现，如前所述，MaliVD这种能力使得其在面对现实环境这种类别不平衡的代码时更具有优势，另外这也验证了MaliVD在面对复杂和真实的代码时能够发挥出更大的优势。

**5）总结**

MaliVD 在四个大型漏洞数据集上的漏洞检测任务都表现出色，特别是在真实性和复杂度较高的ReliVul数据集上。MaliVD通过学习多种结构的代码表征、利用多模态注意力模型以及重点关注漏洞PoIs特征等机制，提升了其挖掘漏洞特征的能力。MaliVD在准确率和F1分数等指标上的优秀结果验证了其较高的漏洞识别能力和泛化性能，体现出其在真实环境下的应用价值。

4.3.2漏洞定位实验

为了评估MaliVD在漏洞定位方面的性能，本文利用提出的漏洞数据集ReliVul进行实验，这是因为只有ReliVul内的样本是标注了CWE类型和漏洞触发位置的真实项目代码。本文将漏洞类型分为六个进行实验（缓冲区溢出漏洞、值计算漏洞、资源管理漏洞、输入验证漏洞、权限管理漏洞和命令执行漏洞），由于大部分方法不支持漏洞定位任务，我们选择LineVul和IVDetect两个深度学习方法，以及CppCheck这个静态分析工具作为对比方法，并使用Top-k和IFA指标衡量MaliVD和它们的表现，另外本节还针对每种漏洞类型的IFA情况进行了深入的探讨。

**1）Top-k指标分析**

LineVul、IVDetect、CppCheck和MaliVD在ReliVul数据集上漏洞定位的Top-k（k = 5, 10, 20, 50）实验结果如图[4.3](#bookmark144)所示。

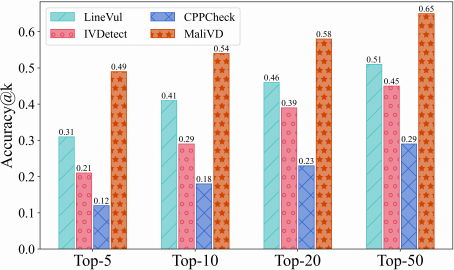


图4.3 MaliVD与对比方法在漏洞定位上的Top-k统计结果

实验结果显示，MaliVD在Top-5、Top-10、Top-20和Top-50的指标上取得了最优的表现，LineVul次优，CppCheck。总的来看，各方法随着k的增加准确率几乎呈现对数增长趋势（统计图中近似线性，但因为k的取值不是线性的，所以总体增长趋势应是次线性）。在Top-5指标上，MaliVD的准确率达到了0.49，而第二高

的LineVul仅取得0.31的准确率，这说明MaliVD具有较低的误报率。LineVul的增长趋势不如IVDetect，可以看出在k=50时，IVDetect已经接近于LineVul的准确率。CPPCheck的性能最差，即使k到了50，准确率仍不足30%。实验结果表明MaliVD不仅在粗粒度的漏洞检测上表现出色，而且在细粒度的漏洞定位上也具有明显的优势。

具体而言，结合各个模型的原理，可以得出如下的结论：LineVul模型的基于CodeBERT模型，本质上是和Transformer类似的序列结构的模型，将代码作为自然语言进行处理，并根据预训练的知识库进行漏洞检测与漏洞定位。虽然LineVul能够进行简单的漏洞检测与定位任务，并且给出可解释的结果，给安全工作者提供一定的建议，但其模型忽略了代码结构化的信息，难以学习到各个漏洞的特别之处，大模型也更难捕捉到代码的细微信息，从而在漏洞定位任务上仍难以取得令人满意的效果。对于IVDetect，它将代码表示为程序依赖图的形式，并使用基于注意力机制的图卷积网络进行学习，然后使用GNNExplainer来提供漏洞相关语句的细粒度解释，并依据注意力的大小来进行漏洞的定位。虽然这种方法可行，但其在模型设计上有两个明显的不足之处，一是其本身并未使用漏洞位置来进行训练模型，并没有通过GNNExplainer的解释结果来进行神经网络的梯度回传，所以模型仍是按照漏洞检测任务来进行的训练和优化；而是在学习过程中并未学习漏洞的知识，不像MaliVD具有PoIs特征、LineVul具有预训练的知识，仅仅利用神经网络根据源代码自动学习各类漏洞的特征是不够的。对于CPPCheck，其基于专家预定义的规则来对源代码进行规则匹配，以识别潜在的漏洞。然而各个样本在代码风格和漏洞触发上具有多样复杂的关系，难以简单利用规则进行匹配，所以CPPCheck容易产生大量的误报，在所有的Top-k指标上都显著低于其他基于深度学习的方法。

相比之下，本文提出的MaliVD方法利用简单易得的代码特征和漏洞特征，以及先进的神经网络模型就可以自动地定位到漏洞的位置。MaliVD不仅考虑了代码的序列结构特征、树形结构特征以及图形结构特征，而且针对各类漏洞提取了漏洞可能触发位置的PoIs特征，将这些特征进行融合，引导模型全面深入理解代码，重点学习漏洞可能触发位置的代码行，所以在漏洞定位上具有显著的优势。在实际应用中，对于Top-5指标，MaliVD取得了接近50%的准确率，说明安全工作人员在进行漏洞排查时，利用MaliVD的帮助，平均只需要检查5行代码，就有一半的概率发现真实漏洞的位置。而随着k的增加，MaliVD可以发现更多的漏洞，因此MaliVD同时改进了现有方法的漏报和误报问题，可以更准确、更高效的定位漏洞的位置，为漏洞挖掘和提升软件安全提供有力的支持。

**2）IFA指标分析**

IFA指标反映了安全分析师在找到实际漏洞位置之前需要检查的误报代码行数。四个方法在ReliVul数据集上各漏洞的平均IFA分布情况如图[4.4](#bookmark146)所示。

80

60

IFA

40

20

0

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |  | | | | |
|  | | | | | | | |  | | | | |
|  | | | | |  | |  |  |  | | | |
|  | |
|  | |  |  |  |  |  | **M = 27** | |  | |  | |
| **M = 20** | |
|  |  | |  | |
|  | **M = 15** | |  |  | |  |  | |  |  |  |  |
| **M = 9** | |
|  | |
|  | |  |
|  |  |  | |
|  | |  |
|  | | | | | | | | | | | | |

LineVul IVDetect CPPCheck MaliVD

图4.4各方法在漏洞定位IFA指标上的统计结果

图[4.4](#bookmark146)是一个箱形图，表示了各方法IFA指标的离散统计情况，中间的箱体的上下界表示离散数据的第一四分位数和第三四分位数，中间的线表示中位数，向外延伸的线条代表数据的变异性，代表大部分数据的分布范围。对于IFA的分布，显然上述各个指标越小越好，代表模型产生的误报最少。根据实验结果，可以看出MaliVD在IFA指标上的表现最优，不仅取得最低的中值（M=9），而且箱体的上下界也是最低，整体的箱体也相对最窄，说明MaliVD取得的误报平均来看是最少的。LineVul和IVDetect的中值分别是15和20，说明安全工作人员在进行漏洞排查时平均要检查15行代码，且由于数据的不稳定，有些样本的检查次数可能更多。CPPCheck方法的IFA分布最差，中值是27，且CPPCheck对于很多样本分类错误，使得其漏报和误报较高。相比之下，MaliVD较低的误报能力使其具有在复杂软件环境中提供可靠漏洞定位服务的能力。

实际上，MaliVD之所以可以取得较低的误报，关键在于其引入的PoIs掩码机制。MaliVD通过预定义的漏洞特征，排除了大部分与漏洞触发位置无关的代码，根据本文的统计，大部分的样本的PoIs个数都不足50个（通过规则判断可能发生漏洞位置的代码行），所以即使仅仅通过PoIs也可以平均在25行内定位到漏洞位置。MaliVD将PoIs特征融合到多模态模型中，使模型能够在理解代码的基础上重点关注那些可能发生漏洞位置的代码，从而实现对漏洞位置的精确判断。对于LineVul，其基于CodeBERT的序列模型，虽然基于大量预训练的漏洞知识和优秀

的序列信息处理能力，但由于其难以捕捉复杂的数据依赖和控制依赖关系，且大模型可能容易忽略那些代码的细节部分，所以在漏洞定位方面容易产生误报现象。对于IVDetect，其在模型训练时并未加入漏洞信息以及漏洞代码行信息的训练，仅仅依赖模型的可解释性进行定位是不够的，所以虽然IVDetect利用了代码的数据依赖和控制依赖关系，并采用先进的图神经网络和注意力机制进行学习，其表现仍不如LineVul。而CPPCheck，由于基于预定义的规则和单一的漏洞模式，在面对现实环境下的复杂代码和漏洞场景时，漏洞的检测能力和定位能力都受限，取得了最多的误报。

通过对IFA指标和各个模型的分析，可以看出，MaliVD凭借其多模态模型和PoIs注意力机制在漏洞定位方面取得了优秀的表现，在真实的有漏洞代码上取得了最低的误报，能够成为安全工作人员在进行漏洞排查时的有力助手，为提升软件安全提供有力支持。

**3）六类漏洞IFA指标的深入分析**

为了深入理解各模型针对特定漏洞的分类能力，本文将MaliVD、LineVul、[IVDetect 和 CPPCheck 这四个模型在六类漏洞上的 IFA 值进行了可视化，如图4.5](#bookmark148)所示。图中的黑点代表IFA值离散点的分布情况，箱形图代表了这些离散数据的统计特性。根据实验结果，对于这些漏洞类型，我们可以得出如下的结论：

总体来看，除了值计算漏洞外，MaliVD在其他五类漏洞中的表现都是最优的，特别是在缓冲区溢出和命令执行漏洞上，MaliVD的数据分布更为集中，且数据整体中心在靠近1的位置（推荐的第一行代码就准确定位了漏洞的位置），与其他方法的对比更为明显。这说明MaliVD能够在大部分的漏洞场景下取得较低的误报能力，相对于其他的方法，具有更高的应用价值。这六类漏洞各有特点，各方法在上面的表现也各不相同，接下来通过分别对这六类漏洞进行详细分析，来捕捉这些模型在每类漏洞上的分类行为。

针对缓冲区溢出漏洞，MaliVD的表现最好，CPPCheck的表现最差，LineVul和IVDetect的IFA值分布处在中等水平。相对于IVDetect，LineVul的分布较为集中，但其均值较高。缓冲区溢出主要发生在内存的分配和读写上，这种一般通过系统函数来实现，例如strcpy、gets函数等。虽然CPPCheck和LineVul可以扫描到这些系统函数，但容易导致误报，IVDetect方法并没有将这些系统函数考虑在内，虽然通过图神经网络的学习，能够识别到缓冲区溢出漏洞可能与这些函数有关，但准确率也不够高。MaliVD就巧妙地将这些系统函数标注为PoIs特征的形式，然后结合对代码的理解和分析重点学习哪些PoIs可能发生漏洞的概率更高，所以通过PoIs可以更好的减少现有模型的误报问题。

100

80

60

IFA

40

20

0

100

80

60

IFA IFA

40

20

0

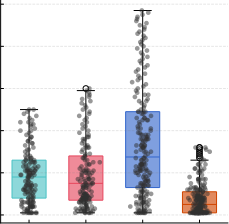
80

60

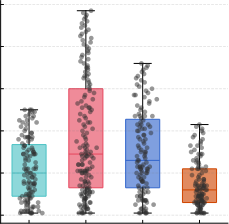
40

20

0

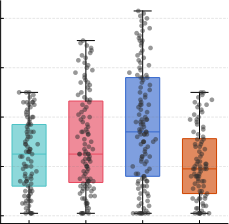


LineVul IVDetect CPPCheck MaliVD (a)缓冲区溢出漏洞



LineVul IVDetect CPPCheck MaliVD

(c)资源管理漏洞



LineVul IVDetect CPPCheck MaliVD (e)权限管理漏洞

60

50

40

IFA IFA

30

20

10

0

80

60

40

20

0

100

80

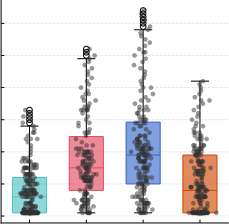
60

IFA

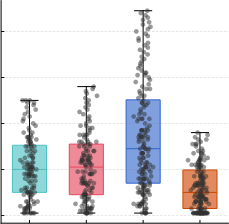
40

20

0

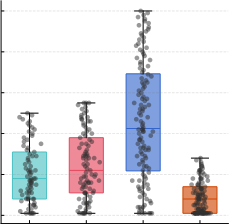


LineVul IVDetect CPPCheck MaliVD (b)值计算漏洞



LineVul IVDetect CPPCheck MaliVD

(d) 输入验证漏洞



LineVul IVDetect CPPCheck MaliVD (f)命令执行漏洞

图4.5各方法在六类主流漏洞定位IFA指标上的统计结果

对于值计算漏洞，LineVul的表现最优，超过了MaliVD。值计算漏洞一般发生在数值运算上，例如除零、浮点数的精度丢失或者整数溢出。值计算漏洞涉及对数值范围的精确处理，这种特殊之处对神经网络模型提出了挑战。像IVDetect和MaliVD，在数据预处理上为了减少嵌入的数量，将大部分数字进行了替换，这本身损失了一定的漏洞判断能力。LineVul基于CodeBERT的大模型可以直接将带有数值的源代码作为输入，保留了可能发生漏洞的条件，所以取得了较低的误报。虽然MaliVD难以精确地判断值计算漏洞，但也根据其挖掘到的代码特征，取得了不错的性能，在平均IFA上低于10，四分位数也低于20，在减少误报上也远低于基于静态分析工具的CPPCheck。

资源管理漏洞主要是因为程序对资源的不正确管理而导致的，例如双重释放或者资源泄露。同缓冲区溢出漏洞不同，缓冲区溢出主要是由于数组或者缓冲区越界导致的。资源管理类漏洞涉及更复杂的代码逻辑和程序依赖关系，检测难度相对较大。所以各类方法在该漏洞上的检测效果都较差，特别是IVDetect，其准确性甚至低于CPPCheck工具。这类漏洞主要发生的位置可能是不正确的调用free和delete这类内存释放函数或在释放后使用了相应的内存。MaliVD将这些系统函数和对资源调用的变量或函数进行了标注，使得模型更能学习到这类漏洞的模式，从而取得了最优的表现。

对于输入验证漏洞，常存在于程序未能正确检查和过滤外部输入的数据上。在这种漏洞上，基于模式匹配的CPPCheck工具由于难以判断用户的输入，对于程序的过滤和检查规则也难以正确识别，所以取得了最差的表现。MaliVD将来自外部输入的所有变量及衍生变量作为PoIs特征，同时又对涉及危险操作的代码（例如对数据库的操作、内存的使用等）部分重点关注，取得了低于10的平均误报，且四分位数在20以下，其他方法的IFA中值都在20以上，这体现了MaliVD对输入验证漏洞的优秀检测能力。

权限管理漏洞则更为复杂，包括对用户权限和敏感操作的不正当管理，通常涉及复杂的数据依赖和控制依赖关系，检测难度更高，所以在这类漏洞上，所有模型的IFA值分布都较高。但MaliVD仍保持最低的IFA中值和更为集中的分布，显示出处理这类复杂漏洞的准确性和稳健性。MaliVD保留了代码的序列特征、树形语法特征和各数据流和控制流图，再通过多模态注意力模型的学习，能够深入挖掘这类漏洞的触发特点，从而克服了其他方法的不足之处。

命令执行漏洞的危害较大，属于高危漏洞，可能造成信息泄露、系统崩溃、权限被非法获取等严重后果。实际上命令执行漏洞一般涉及具体命令的构造，涉及字符串拼接或用户命令注入，这些漏洞对于将常量字符串进行替换的模型而言检

测难度较大。但MaliVD在IFA的整体分布上要显著低于其他方法，说明其在命令执行漏洞检测上具有较强的鲁棒性。MaliVD的多模态注意力机制有效关注到代码中的关键部分，如PoIs，提高了对依赖于复杂条件和动态输入场景的命令执行漏洞的检测能力。

通过对这六类漏洞的深入分析，可以看出，对于现有的流行的这些漏洞，对于基于规则的静态分析工具而言，难以对漏洞的位置做出判断；序列模型虽同时学习了漏洞知识和代码知识，但面临复杂的数据关系和控制关系时也难以取得满意的效果；IVDetect虽使用多个模型和多种特征，但其在模型中没有学习漏洞的知识，仅仅靠神经网络的权重参数来定位漏洞是不够的，将带来很多误报。MaliVD通过集成多种特征，并使用PoIs特征来引导模型学习漏洞的模式，挖掘代码中潜在的漏洞触发点，取得了较高的漏洞定位准确性，为漏洞定位领域带来了新的突破。

4.3.3消融实验

为了评估MaliVD中各个模块对漏洞检测和漏洞定位的贡献，本文还做了一系列消融实验。MaliVD的核心机制在于其多模态的表征学习和PoIs注意力机制，所以我们分别在模型中移除这些模块，然后使用查准率（P.）、召回率（R.）和F1分数（F1.）评估模型识别漏洞类型的能力，并使用Top-10准确率（Acc@10）评估模型定位漏洞的能力。具体来说，消融实验主要包含以下几个模块：①原始代码序列特征（NCS）和序列神经网络（WLJAN）；②基于AST树结构的特征和Tree-LSTM模型；③基于CPG图的特征和GGNN模型；④基于PoIs的注意力机制（掩码特征）和两部分的特征拼接。此外，我们还评估了⑤仅使用基于PoIs的掩码特征，而不使用原始特征和特征拼接时的性能。这些实验结果如表[4.4](#bookmark178)所示。

表4.4消融实验结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编号 | 模型 | P. | R. | F1 . | Acc@10 |
| ① ② ③ ④ ⑤ | w/o NCS & WLJAN | 0.63 | 0.74 | 0.68 | 0.44 |
| w/o AST & Tree-LSTM | 0.62 | 0.72 | 0.67 | 0.41 |
| w/o CPG & GGNN | 0.54 | 0.59 | 0.57 | 0.26 |
| w/o Masked features & PoIs | 0.58 | 0.65 | 0.61 | 0.31 |
| w/o Original features | 0.52 | 0.56 | 0.54 | 0.33 |
| - | MaliVD | 0.69 | 0.77 | 0.73 | 0.54 |

**1）多模态表征学习的作用**

多模态表征学习是MaliVD的核心创新点之一，模型从源代码中提取出原始代码序列、抽象语法树和代码属性图三个特征，并使用三种对应结构的先进神经

网络进行学习。为了评估这三种代码表示形式分别对模型做出的贡献，我们分别移除了NCS、AST和CPG，当然也分别移除了对应的WLJAN、Tree-LSTM以及GGNN模型，实验结果在表[4.4](#bookmark178)的①、②和③中。

实验结果表明，移除任何一种代码特征和模型都会导致所有分类指标和定位指标的下降，这说明三类特征对于漏洞检测和定位都是不可或缺的，其中移除图类表征学习的影响最大，F1分数下降0.16，Acc@10下降0.28，这也说明图类表征对于漏洞检测能够提供更多的信息，实际上，代码属性图能够有效地保留代码中的复杂依赖关系，例如数据流、控制流以及函数调用关系等。这些依赖关系往往是漏洞产生的关键因素。通过GGNN对CPG的学习，MaliVD能够更好地捕捉可能由于这些依赖关系导致的漏洞，从而在漏洞检测和定位上取得更好的效果。

除了图类表征学习，移除基于AST树的表征学习也会导致各指标大幅下降。虽然CPG中本身就包含了AST树结构的一些信息，但Tree-LSTM模型作为专门处理树形结构的数据，更适合对AST特征进行学习，同图神经网络的学习传播方式不同，Tree-LSTM通过递归的方式学习子节点的内容，并传播给根节点，更能保留关键的语义信息。另外，在移除NCS表征和WLJAN网络后，模型的F1分数下降了7%，Acc@10下降了约20%，这表明序列特征仍是源代码漏洞检测的重要特征，它提供了代码的“语言特性”，对应代码的执行顺序和关键语句做了保留，对于漏洞检测和定位也是至关重要的。

**2）PoIs特征与原始代码特征的共同作用**

PoIs 特征的引入，让MaliVD有了更强的漏洞类型判断和漏洞定位的能力。为了验证PoIs特征的作用，本文移除PoIs特征，只使用原始的三类代码特征，当然原模型中使用PoIs掩码的部分以及与原始特征拼接的部分都进行了移除，来查看模型的性能变化。如表[4.4](#bookmark178)④中的实验结果所示，在移除PoIs、掩码特征和特征拼接后，模型的Acc@10大幅下降，仅为0.31，模型的查准率、召回率和F1分数也[有了较大程度的下降，和表4.3](#bookmark176)中的其他模型相比，MaliVD在失去这些关键部分后没有了竞争力。

为了进一步验证PoIs特征和原始代码特征的共同作用，本文只使用PoIs掩码[的部分，而不使用代码的原始三类特征，又得出了如表4.4](#bookmark178)⑤的实验结论。有趣的是，移除掩码特征和PoIs（0.31）与移除原始特征（0.33）会导致大致相似的Acc@10下降，甚至在查准率、召回率和F1分数上下降更多，F1分数只有0.54。这实际上可以从以下两个方面进行解释：一方面，PoIs不仅提供了漏洞的类型信息，还帮助模型排除了大部分的与漏洞无关的代码，使得模型更专注于学习那些可能发生漏洞的位置，对于提升漏洞的类型识别和位置判定起到很大的作用；另一方面，虽

然PoIs特征至关重要，但它们仅仅是原始代码中的一小部分（如前所述，大部分的样本的PoIs个数都不足50个），所以如果只使用PoIs特征，而移除原始完整的代码，模型则学不到原始代码中的重要上下文信息，只关注局部信息而损失全局信息会使模型丧失整体的代码理解以及漏洞模式学习。例如对于一个可能触发漏洞的语句，被标注成了PoIs特征，但程序又在其他代码部分对这条语句做了充分的条件限制和危险操作过滤，那么这条语句就大概率不存在漏洞，如果只保留关键代码部分，那么模型很大可能将这条语句判定为有漏洞，从而引起误报。

经过实验结果的验证，原始代码特征和PoIs掩码特征对于模型的贡献都是非常重要的，因此本文将这两部分特征都送入模型中训练，并将这两部分的高层表示进行连接，然后进行漏洞检测和定位。通过这种方式，MaliVD能够更好地结合代码的全局信息和局部信息，实现更准确的漏洞检测和漏洞定位。

**3）总结**

消融实验充分证明了MaliVD模型中各个组成部分的重要性。序列、树形和图形这三种代码表示形式相互补充，结合PoIs特征和原始代码特征的拼接，共同提升了MaliVD在漏洞检测和漏洞定位上的性能。MaliVD模型的设计是模拟人类在进行漏洞检测时的思维方式，当人类在进行漏洞检测和定位时，也是先根据原始代码（对应NCS和WLJAN模型）结合代码的语法（对应AST和Tree-LSTM模型）特点理解代码内容，然后针对可能触发漏洞的位置（对应PoIs特征），提取关键数据流和控制流（对应CPG和GGNN模型）深入分析，从而判断漏洞的类型和漏洞的触发位置。

4.3.4参数调优实验

为了进一步提升MaliVD的性能，本文使用ReliVul数据集进行了一系列的参数调优实验，包括各个特征的嵌入（Embedding）维度、各个模型的隐藏层（Hidden Layer）维度、学习率（Learning Rate）和批量大小（BatchSize）。特征的嵌入维度决定了对代码信息的表示能力，也直接影响着模型的学习效果，各个模型的隐藏层维度决定了模型的参数大小，影响着模型挖掘隐藏信息的深入和泛化能力，学习率和批量大小则影响着模型在训练中参数更新的幅度和最终模型的稳定性。下面将详细介绍各个参数的调优过程和实验结果。

**1）特征嵌入维度**

在将各类特征送入神经网络模型之前，需要将特征进行嵌入，例如对于代码序列，需要在分词后对每个单词和符号进行编码，然后嵌入成多维向量的形式，对于树形和图形特征，则需要对不同的节点和边做不同的嵌入，以便神经网络进行数值化计算和寻找各不同单词或节点的关系。过小的嵌入维度可能无法让模型充

分捕捉代码的语义信息，而过大的嵌入维度则增大了计算复杂度，也容易引起模型的过拟合，为了探寻单词、节点和边的最佳嵌入维度，本文将嵌入维度的取值范围设置为16, 32, 64, 128, 256，分别查看各个嵌入维度的训练效果，由于CPG中[边的类型较少（如表4.1](#bookmark172)所示，只有17种），还额外测试了边嵌入维度为8的情况，[实验结果如图4.6](#bookmark150)（a）所示。

0.75

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | | |  |
|  |  |  |  | | |  |
|  |  |  |  |  | |  |
|  |  |  | | 单词 |
|  |  |  |  |  | | 边  节点 |
|  |
|  |  |  |  |

0.70



0.65

F1

0.60

0.55



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  | WL | JAN |
|  |
|  |  |  |  | Tree | -LSTM |
|  | |  |  |



0.70

0.75



F1 农

0.65



0.60



0.55

8 16 32 64 128 256

(a)嵌入维度



















GGNN



8 16 32 64 128 256

(b)隐藏层维度

图4.6 MaliVD在不同嵌入维度和隐藏层维度下的性能表现

总体来看，随着单词、节点和边的嵌入维度的升高，MaliVD的F1分数都呈现上升的趋势，这确实验证了更大的嵌入维度包含更多的参数，更有利于模型进行特征学习和漏洞检测。随着嵌入维度的降低，节点的影响大于单词，边的影响更小。根据上一节消融实验的结论，树形表征和图形表征对MaliVD的贡献最大，而节点正是这些代码表征的重要组成部分，当节点嵌入维度只有16时，模型的F1分数已不足60%。相比之下，边的嵌入维度对漏洞检测效果的影响很小，这是因为边本身的数量较少，在较小的嵌入维度下就已经可以取得较好的检测结果，但高维的边嵌入维度确实对模型更有力，也说明了边在漏洞检测方面也提供了有价值的信息。值得注意的是，当嵌入维度大于64之后，F1分数的增长趋势都趋于平缓。对于节点来说，当维度达到128时，F1分数已经接近0.72，而将维度提升到256后，F1分数并没有明显增加，对于单词和边来说，对应的F1分数几乎没有变化。由于当嵌入维度翻倍之后，模型的训练时间和预测时间也会成倍增加，降低了模型的效率，因此，本文最终选择将单词嵌入维度、节点嵌入维度和边嵌入维度都设置为128。

**2）模型隐藏层维度**

MaliVD 包含三个模型，隐藏层维度主要影响WLJAN中的GRU单元、Tree- LSTM中的LSTM单元以及GGNN中的门控神经单元。类似于嵌入维度的调参方式，通过将各个模型的隐藏层维度的设置为16, 32, 64, 128, 256，然后分别查看模型在ReliVul的平均F1分数，由于WLJAN模型更适合隐藏层维度更低的结果，本

文还测试了其隐藏层维度在8的情况。实验结果如图[4.6](#bookmark150)（b）所示。

不同于F1分数随嵌入维度的规律，随着隐藏层维度的增加，并不是所有模型的性能都会提升。对于序列模型WLJAN，在隐藏层维度为16时模型的效果最优，当维度下降或减少时，F1分数都会下降。这是因为WLJAN模型中含有对隐藏层的池化操作，过大的隐藏层维度会损失模型学习到的一些信息，导致检测效果下降，所以最终将WLJAN的隐藏层维度设置为16。对于GGNN和Tree-LSTM，随着隐藏层维度的增大，F1分数逐步提高，且在维度低于64时，增长的趋势更为明显，在维度大于64后，F1分数没有太大的变化，可见隐藏层维度在64-128时，模型就可以学习到代码和漏洞的特点，在漏洞检测上较为准确。在权衡模型的准确性和参数大小后，将GGNN的隐藏层维度设置为64，Tree-LSTM的隐藏层维度设置为128。

**3）学习率和批量大小**

本文还研究了学习率和批量大小两个重要的参数，学习率决定了模型每次更新权重时的步长大小，批量大小则是指模型每次进行正向传播和梯度回传时同时学习了多少个样本，这两个参数对于深度学习模型的训练至关重要。本文测试了学习率在{10−6 , 10−5 , 10−4 ,10−3}内，批量大小在{16, 32, 64, 128}内的模型性能，并使用F1 [分数作为评价指标。为了节省训练的时间， 本文引入了早停机制[82](#bookmark265)]，即模型最多训练100轮次，如果在连续的五轮训练中，模型的所有指标（例如F1分数或Top-10准确率）都没有提升，则停止训练。根据本文的实验结论，MaliVD模[型基本上在40轮左右就能收敛，且稳定性较好。实验结果如图4.7](#bookmark152)所示。

0.7

0.6



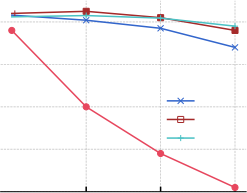
0.5

F1

0.4

0.3





学习率:

—●— 1e-06

1e-05

 1e-04 1e-03

16 32 64 128

批量大小

图4.7 MaliVD在不同批量大小和学习率下的性能表现

实验结果表明，MaliVD在学习率为10−4、批量大小为32时的F1分数最高。实际上当学习率在{10−5 , 10−4 , 10−3}时，F1分数都超过了70%，而当学习率为10−6时，由于模型的更新步长过小，模型在有限的训练周期内未能收敛。当学习率为

10−3时，虽然模型收敛较快，但随着批量大小的增加，模型的准确性有所下降，整体性能不如当学习率设置为10−4时。且过大的学习率可能导致模型在训练过程中发生震荡，不够稳定，从而难以找到最优解。另外考虑批量大小对结果的影响，较大的批量大小可以加速模型的训练，使得模型在每次更新参数都能考虑到多样化的样本，但也可能让模型收敛到一个较宽泛但非最优解的区域，也会带来更大的内存开销；较小的批量大小可以增强训练过程的随机性，有助于模型跳出局部最优解，但可能使模型不够稳定，同时也使训练次数增加、降低了训练效率。结合实验结果、模型的内存消耗和训练时长，本文最终将MaliVD在训练过程中的批量大小设置为32，学习率设置为10−4。

4.4本章小结

本章详细介绍了本文提出的基于多模态注意力机制的源代码漏洞检测与定位模型MaliVD[的原理和实验结果。第4.1](#bookmark80)小节概述了MaliVD模型的基本框架，并按照数据预处理与特征提取、多模态注意力模型设计、分类模型设计以及定位模型设计详细介绍了其原理。在实验结果部分，第[4.2](#bookmark92)节描述了实验设置，包括所使用的数据集、比较基线和评价指标。第[4.3](#bookmark100)节列出了详细的实验结果和分析，同多个先进的方法进行对比，MaliVD在漏洞检测和漏洞定位上都展现出了最优的表现，本章还通过消融实验验证了多模态表征学习和PoIs特征的作用，并通过详细的参数调优实验优化了MaliVD的性能。

第5章总结与展望

5.1论文工作总结

在当下的互联网时代，软件与人们的生活息息相关，人们的隐私、财产甚至人身安全都与各类软件进行了绑定，所以软件的安全关系到整个社会的安全。研究基于源代码的漏洞检测技术对于提升软件的安全性至关重要。然而随着软件代码量和漏洞复杂度的不断上升，传统的基于规则的静态分析方法由于需要专家提取规则且难以赶上现有漏洞增长的趋势，已难以满足现代软件安全的需求。尽管深度学习方法在漏洞检测领域取得了一定的成果，但也面临着缺乏高质量漏洞数据集、学习与漏洞无关的代码特征、应用不合适的神经网络模型以及现有方法难以定位漏洞的位置等问题。本文为解决这些问题，从漏洞代码库的构建和漏洞检测模型两个方向出发，以做出创新性的成果。

为了收集足够多的高质量漏洞数据样本，本文提出了一种自动化的代码行级的漏洞代码库构建框架。该框架从CVE数据库和开源代码库中提取漏洞信息、代码信息和代码的修订记录。然后利用多种静态分析工具、机器学习模型和大语言模型对提交信息和代码修订信息进行判断，以获取准确的补丁代码。为了能标注漏洞的触发位置，本文针对各类漏洞，制定了一套漏洞PoIs特征，PoIs特征是根据每类漏洞的特点提取的那些可能发生漏洞的位置。通过结合PoIs特征以及代码的修订信息，对漏洞触发位置和触发路径进行了详细标注。基于此框架，本文构建了一个代码行级的高质量漏洞数据集ReliVul。ReliVul含有足够的样本，且数据均来自于真实的项目，标注粒度也达到了代码行级，同一些流行的数据集相比，ReliVul在数据集的基本信息、准确性、唯一性和完整性等方面具有显著优势，不仅为后续漏洞检测模型的设计提供了更准确的数据支持，也为该行业提供了一个可靠的数据基准。

为了解决现有漏洞检测模型在特征选取、模型设计以及定位能力方面的不足之处，本文提出了基于多模态注意力机制的源代码漏洞检测与定位模型MaliVD。MaliVD从源代码中提取出代码序列、抽象语法树和代码属性图三种结构的特征，并使用对应结构的三个先进的神经网络模型WLJAN、Tree-LSTM和GGNN进行学习。为了能更好地进行漏洞类型的识别和漏洞位置的判断，MaliVD引入了PoIs特征，根据PoIs的位置对原始的特征施加掩码，并将掩码后的特征也送入三个模型中学习，最后将这些结果连接起来，共同为漏洞检测和定位做贡献。MaliVD能根据不同结构的代码表征从多个视角挖掘代码的隐藏信息，通过PoIs特征重点关

注那些可能发生漏洞的语句，从而大大提升了漏洞检测与定位的能力。实验结果显示，同多个先进的漏洞检测模型对比，MaliVD在漏洞类型识别和漏洞定位上都取得了最优的性能。对于目前主流的六类漏洞类型，MaliVD能够保持较低的漏报和误报率，验证了其在现实环境下的应用价值。

综上所述，本文通过深入研究源代码漏洞检测领域的各项挑战，提出了两个创新性的解决方案：ReliVul漏洞数据集的提出为后续的研究提供了数据基础，也为漏洞检测领域提供了一个代码行级的高质量漏洞数据集；MaliVD模型通过利用多模态注意力机制，在提升漏洞类型识别能力的基础上也提高了漏洞定位的能力。实验结果显示，本文的这两项贡献相对于现有方法均有较大提升，且具有较高的实际应用价值。期望通过本文的研究，可以推动源代码漏洞检测技术的发展，为软件安全提供更有力的支持。

5.2未来工作计划

本文针对现有源代码漏洞检测方法中存在的问题进行了探索与研究，提出了一个代码行级的漏洞数据集ReliVul和高效的漏洞检测与定位模型MaliVD，但仍有许多方面值得进一步思考和改进，如在其他漏洞和编程语言上的尝试以及在现实环境中的应用。因此，未来的工作将集中优化以下三个方面：（1）基于本文提出的漏洞代码库构建方法，构建可持续学习的漏洞知识库；（2）探索更广泛的漏洞模式，特别是新兴的复杂漏洞类型；（3）将MaliVD应用于真实项目中，以验证其在实际环境中的效果。

**1）构建可持续学习的漏洞知识库：**本文提出的漏洞代码库构建方法能够从多个CVE数据库和开源代码平台自动提取漏洞代码和补丁代码。但对补丁的判断是基于现有的漏洞样本、现有的工具和模型上的，未能考虑新出现的漏洞。为了能够紧跟安全领域的前沿，将新出的漏洞加入到现有模型中是一个重要的研究方向，未来本文希望构建一个可持续学习的漏洞知识库。该知识库不仅能够存储已知的漏洞信息，还能通过主动学习或增量学习等技术，不断吸收新的漏洞数据和补丁代码，从而提升对新型漏洞的识别能力和泛化能力。一种可行的方式是将新收集到的漏洞代码和补丁信息作为新的训练数据，对预训练的模型或工具进行微调，使其能够更好地适应新的漏洞模式。另外也可以探索利用生成对抗网络等技术，生成更多样化的漏洞样本，以增强模型的泛化能力。通过构建可持续学习的漏洞知识库，本文希望可以有效地提升漏洞数据集的质量，并为漏洞检测领域提供源源不断的数据支持。

**2）探讨更广泛的漏洞模式：**尽管本文对六类常见漏洞进行了深入分析，涵盖

了大部分CWE类型，但随着技术的不断进步，新漏洞类型层出不穷，如零日漏洞、供应链攻击等。这些新兴漏洞往往更加复杂多变，难以检测。未来本文将进一步研究这些新兴漏洞，探索其特征和模式，提升MaliVD的检测能力。由于本文的研究主要集中在C/C++语言上，只是广泛漏洞库中的一部分，而其他编程语言（如Java、Python）以及跨语言的漏洞（如Web应用程序中的漏洞）也值得进一步研究。希望能扩展本文的方法到更多的编程语言和平台上，这样可以构建一个更加全面和通用的漏洞检测系统，为不同类型的软件提供安全保障，从而提升整个软件生态的安全。

**3）在真实项目中的应用：**为了验证MaliVD模型在现实环境和真实代码中的有效性，本文计划将其应用于开源项目中，通过现有漏洞数据集训练一个通用模型，在开源项目中开展大规模扫描，以识别潜在的安全漏洞问题。通过在真实项目中的应用，可以针对出现的问题做进一步改进，从而不断提升其应用价值；另外可以将MaliVD集成到开源平台上，帮助开发者及时发现和修复代码中存在的问题，进一步提升开发者的安全意识和整个开源系统的安全性。期望将MaliVD作为安全编程助手，为个人、企业、代码平台乃至整个互联网的安全做出更大的贡献。

参考文献

[1] CNVD:国家信息安全漏洞共享平台[EB/OL]. 2025. <https://www.cnvd.org.cn>.

[2] LaToza T D, Venolia G, DeLine R. Maintaining mental models: a study of developer workhabits[C]//Proceedings of the 28th international conference on Software engineering. 2006: 492-501.

[3] CodeQL[EB/OL]. 2025. <https://codeql.github.com>.

[4] Coverity: Coverity scan static analysis[EB/OL]. 2025. <https://scan.coverity.com>.

[5] Cppcheck[EB/OL]. 2025. <http://cppcheck.net>.

[6] Flawfinder[EB/OL]. 2025. <https://dwheeler.com/flawfinder>.

[7] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning com-prehensive program semantics via graph neural networks[J]. Advances in neural information processing systems, 2019, 32.

[8] 刘嘉勇,韩家璇,黄诚.源代码漏洞静态分析技术[J].信息安全学报, 2022, 7(4): 100-113.

[9] Chen Y, Ding Z, Alowain L, et al. Diversevul: A new vulnerable source code dataset for deeplearning based vulnerability detection[C]//Proceedingsof the 26th International Symposium onResearch in Attacks, Intrusions and Defenses. 2023: 654-668.

[10] Wang W, Nguyen TN, Wang S, et al. Deepvd: Toward class-separation features for neural net-work vulnerability detection[C]//2023 IEEE/ACM 45th International Conference on SoftwareEngineering (ICSE). IEEE, 2023: 2249-2261.

[11] Mirsky Y, Macon G, Brown M, et al. {VulChecker}: Graph-based vulnerability localization in source code[C]//32nd USENIX Security Symposium (USENIX Security 23). 2023: 6557-6574.

[12] 顾绵雪,孙鸿宇,韩丹,等.基于深度学习的软件安全漏洞挖掘[J].计算机研究与发展,2021, 58(10): 2140-2162.

[13] 邓枭,叶蔚,谢睿,等.基于深度学习的源代码缺陷检测研究综述[J].软件学报, 2023, 34

(2): 625-654.

[14] CVE 平台[EB/OL]. 2025. <https://cve.mitre.org>.

[15] Ghaffarian S M, Shahriari H R. Neural software vulnerability analysis using rich intermediategraph representations of programs[J]. Information Sciences, 2021, 553: 189-207.

[16] Duan X, Wu J, Ji S, et al. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. [C]//IJCAI. 2019: 4665-4671.

[17] Russell R, Kim L, Hamilton L, et al.Automated vulnerability detection in source code usingdeep representationlearning[C]//2018 17th IEEE international conference on machine learningand applications (ICMLA). IEEE, 2018: 757-762.

[18] Dam H K, Pham T, Ng S W, et al. Lessons learned from using a deep tree-based model forsoftware defect prediction inpractice[C]//2019 IEEE/ACM 16th International Conference onMining Software Repositories (MSR). IEEE,2019: 46-57.

[19] National Institute of Standards and Technology. Sard: Software assurance reference dataset [EB/OL]. 2025. <https://samate.nist.gov/SARD>.

[20] Owasp benchmark[EB/OL]. 2025. <https://owasp.org/www-project-benchmark>.

[21] Linux kernel git[EB/OL]. 2025. <https://git.kernel.org>.

[22] Owasp benchmark[EB/OL]. 2025. <https://owasp.org/www-project-benchmark>.

[23] Clang[EB/OL]. 2025. <https://clang.llvm.org>.

[24] Lin D, Koppel J, Chen A, et al. Quixbugs:A multi-lingual program repair benchmark setbased on the quixey challenge[C]//Proceedings Companion of the 2017 ACM SIGPLAN in- ternational conference on systems, programming, languages, and applications: software for humanity. 2017: 55-56.

[25] Zou D, Wang S, Xu S, et al.“vuldeepecker: A deep learning-based system for multiclassvulnerability detection[J]. IEEE Transactions on Dependable and Secure Computing, 2019, 18

(5): 2224-2236.

[26] LiZ,ZouD,Xu S,et al. Sysevr:A framework for using deep learning to detect software vulner- abilities[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 19(4): 2244-2258.

[27] Fan J, Li Y, Wang S, et al. Ac/c++ code vulnerability dataset with code changes and cve sum-maries[C]//Proceedings of the 17th International Conference on Mining Software Repositories.2020: 508-512.

[28] Nikitopoulos G, Dritsa K, Louridas P, et al. Crossvul: a cross-language vulnerability datasetwith commit data[C]//Proceedings of the 29th ACM Joint Meeting on European Software En- gineering Conference and Symposium on the Foundations of Software Engineering. 2021: 1565-1569.

[29] Wang X, Hu R, Gao C, et al. Reposvul:A repository-level high-quality vulnerability dataset[C]//Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineer-ing: Companion Proceedings. 2024: 472-483.

[30] Li Y, Wang S, Nguyen TN, et al. Improving bug detection via context-based code representationlearning and attention-based neural networks[J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-30.

[31] Cheng X, Wang H,Hua J, et al. Deepwukong: Statically detecting software vulnerabilities usingdeep graph neural network[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2021, 30(3): 1-33.

[32] Zheng Y, PujarS, Lewis B, et al. D2a: A dataset built for ai-based vulnerability detection meth-odsusing differential analysis[C]//2021 IEEE/ACM 43rd International Conference on SoftwareEngineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021: 111-120.

[33] Chakraborty S,Krishna R, Ding Y, et al. Deep learning based vulnerability detection: Are wethere yet?[J]. IEEE Transactions on Software Engineering,2021, 48(9): 3280-3296.

[34] Zhou Y, Sharma A. Automated identification of security issues from commit messages andbugreports[C]//Proceedings of the 2017 11th joint meeting on foundations of software engineering.2017: 914-919.

[35] Sabetta A, Bezzi M. A practical approach to the automatic classification of security-relevantcommits[C]//2018 IEEE International conference on software maintenance and evolution (IC- SME). IEEE, 2018: 579-582.

[36] NVD: 美国国家漏洞数据库[EB/OL]. 2025. <https://nvd.nist.gov>.

[37] Mend[EB/OL]. 2025. <https://www.mend.io>.

[38] CWE 平台[EB/OL]. 2025. <https://cwe.mitre.org/>.

[39] Quixey[EB/OL]. 2025. <https://quixy.com>.

[40] Bhandari G, Naseer A, Moonen L. Cvefixes: automated collection of vulnerabilities andtheirfixes from open-source software[C]//Proceedings of the 17th International Conference on Pre-dictive Models and Data Analytics in Software Engineering. 2021: 30-39.

[41] Hindle A, Barr E T, Gabel M, et al. On the naturalness of software[J]. Communications of the ACM, 2016, 59(5): 122-131.

[42] Perl H, Dechand S, Smith M, et al. Vccfinder: Finding potential vulnerabilities in open-sourceprojects to assist code audits[C]//Proceedings of the 22nd ACM SIGSAC conference on com-puter and communications security. 2015: 426-437.

[43] 李韵,黄辰林,王中锋,等.基于机器学习的软件漏洞挖掘方法综述[J].软件学报, 2020, 31

(7): 2040-2061.

[44] 夏之阳,易平,杨涛.基于神经网络与代码相似性的静态漏洞检测[J/OL].计算机工程, 2019, 45(12): 141-146. DOI:[10.19678/j.issn.1000-3428.0053136](https://doi.org/10.19678/j.issn.1000-3428.0053136).

[45] Li X, Wang L, Xin Y, et al. Automated vulnerability detection in source code using minimum intermediate representation learning[J]. Applied Sciences, 2020, 10(5): 1692.

[46] Zagane M, Abdi M K, Alenezi M. Deep learning for software vulnerabilities detection usingcode metrics[J]. IEEE Access, 2020, 8: 74562-74570.

[47] Wang H, Ye G, Tang Z, et al. Combining graph-based learning with automated data collectionfor code vulnerability detection[J]. IEEE Transactions on Information Forensics and Security, 2020, 16: 1943-1958.

[48] Li Y, Wang S, Nguyen T N. Vulnerability detection with fine-grained interpretations[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conferenceand Symposium on the Foundations of Software Engineering. 2021: 292-303.

[49] Cao S, Sun X, Bo L, et al. Bgnn4vd: Constructing bidirectional graph neural-network forvulnerability detection[J]. Information and Software Technology, 2021, 136: 106576.

[50] Li Z, Zou D, Xu S, et al. Vuldeelocator: a deep learning-based fine-grained vulnerability de-tector[J]. IEEE Transactions on Dependableand Secure Computing, 2021, 19(4): 2821-2837.

[51] Wu Y, Zou D, Dou S, et al. Vulcnn: An image-inspired scalable vulnerability detection system[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 2365- 2376.

[52] Nguyen V A, Nguyen D Q, Nguyen V, et al. Regvd: Revisiting graph neural networks for vulnerability detection[C]//Proceedings of the ACM/IEEE 44th International Conference onSoftware Engineering: CompanionProceedings. 2022: 178-182.

[53] Cao S, Sun X,BoL, et al. Mvd: memory-related vulnerability detection based on flow-sensitivegraph neural networks[C]//Proceedings of the 44th international conference on software engi-neering. 2022: 1456-1468.

[54] Fu M, Tantithamthavorn C. Linevul: A transformer-based line-level vulnerability prediction[C]//Proceedings of the 19th International Conference on Mining Software Repositories. 2022: 608-620.

[55] HinD,KanA,Chen H,et al.Linevd: statement-level vulnerability detection using graph neural networks[C]//Proceedings of the 19th international conference on mining software repositories.2022: 596-607.

[56] Zhang C, Liu B, Xin Y, et al. Cpvd: Cross project vulnerability detection based on graph attention network and domain adaptation[J]. IEEE Transactionson Software Engineering, 2023,49(8): 4152-4168.

[57] 张学军,张奉鹤,盖继扬,等. mVulSniffer:一种多类型源代码漏洞检测方法[J].通信学报, 2023, 44(9): 149-160.

[58] Xiao P, Xiao Q, Zhang X, et al. Vulnerability detection based on enhanced graph representationlearning[J]. IEEE Transactions on Information Forensicsand Security, 2024.

[59] Qiu F, Liu Z, Hu X, et al. Vulnerability detection via multiple-graph-based code representation[J]. IEEE Transactionson Software Engineering, 2024.

[60] Tran H C, Tran A D, Le K H. Detectvul: A statement-level code vulnerability detection for python[J]. Future Generation Computer Systems, 2025, 163: 107504.

[61] Mechri A, Ferrag M A, Debbah M. Secureqwen: Leveraging llms for vulnerability detection in python codebases[J]. Computers & Security, 2025, 148: 104151.

[62] 熊可欣,李涛,余琴,等. PDGcross:基于跨文件图表征的源代码漏洞检测[J].计算机技术与发展, 2023, 33(8): 102-107.

[63] 杨宏宇,杨海云,张良,等.基于特征依赖图的源代码漏洞检测方法[J].通信学报, 2023, 44

(1): 15.

[64] Tai K S, SocherR, Manning C D. Improved semanticrepresentations from tree-structured longshort-term memory networks[A]. 2015.

[65] Li Y, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks[A]. 2015.

[66] 潘禺涵舒远仲洪晟罗斌聂云峰.基于多关系结构图神经网络的代码漏洞检测[J].网络安全与数据治理, 2022, 41(11): 36-42.

[67] Zou D, Zhu Y, Xu S, et al. Interpreting deep learning-based vulnerability detector predictionsbased on heuristic searching[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2021, 30(2): 1-31.

[68] Mao Y, Li Y, Sun J, et al. Explainable software vulnerability detection based on attention-basedbidirectional recurrent neural networks[C]//2020 IEEE International Conference on Big Data(Big Data). IEEE, 2020: 4651-4656.

[69] Ying Z, Bourgeois D, You J, et al. Gnnexplainer: Generating explanations for graph neuralnetworks[J]. Advances in neural information processing systems, 2019, 32.

[70] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[A]. 2017.

[71] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[72] Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages[A]. 2020.

[73] Hui B, Yang J,Cui Z, et al. Qwen2.5-coder technical report[A]. 2024.

[74] Google git[EB/OL]. 2025. <https://code.googlesource.com/git>.

[75] Chromium issues[EB/OL]. 2025. <https://issues.chromium.org>.

[76] andrew d. rough-auditing-tool-for-security[EB/OL]. 2025.[https://github.com/andrew-d/rough](https://github.com/andrew-d/rough-auditing-tool-for-security)[-auditing-tool-for-security](https://github.com/andrew-d/rough-auditing-tool-for-security).

[77] Semgrep[EB/OL]. 2025. <https://semgrep.dev>.

[78] Shafer G, Vovk V. A tutorial on conformal prediction.[J]. Journal of Machine Learning Re- search, 2008, 9(3).

[79] Balasubramanian V N, Baker A, Yanez M, et al. Pycp: an open-source conformal predic-tions toolkit[C]//Artificial Intelligence Applications and Innovations: 9th IFIP WG 12.5 Inter-national Conference,AIAI 2013, Paphos, Cyprus, September 30–October 2, 2013, Proceedings

9. Springer, 2013: 361-370.

[80] Croft R, Babar M A, Kholoosi M M. Data quality for software vulnerability datasets[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023: 121- 133.

[81] Allamanis M. The adverse effects of code duplication in machine learning models of code[C]//Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, NewParadigms, and Reflections on Programmingand Software. 2019: 143-153.

[82] Xiao X, Wang S, Hu G, et al. Rbljan: Robust byte-label joint attention network for networktraffic classification[J]. IEEE Transactions on Dependable and Secure Computing, 2024.

[83] Tang D, Qin B, Liu T. Document modeling with gated recurrent neural network for sentimentclassification[C]//Proceedingsof the 2015 conference on empirical methods in natural language processing. 2015: 1422-1432.

[84] Ross T Y, Dollár G. Focal loss for dense object detection[C]//proceedings of the IEEE confer-ence on computer vision and pattern recognition. 2017: 2980-2988.

[85] Alon U, Zilberstein M, Levy O, et al. code2vec: Learning distributed representations of code[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1-29.

附录A与漏洞修复相关的提交可能包含的关键词

表A.1与漏洞修复相关的提交可能包含的关键词

|  |  |
| --- | --- |
| 类别 | 关键词 |
| 缓冲区溢出 | 缓冲区, 溢出, 上溢, 下溢, (格式化, 字符串), (format, string), (buffer, access), overflow, underflow, buffer, format string |
| 内存损害类 | 泄漏, 泄露, 双重释放, 重复释放, 空指针, 野指针, 解引用, 悬空指针,  悬垂指针, 消耗, 耗尽, 越界, uaf, memory leak, double free,  uncontrolled resource consumption, resource leak, (use, after, free), (dangl, pointer), (wild, pointer), (derefer, pointer), (out, of, bound) |
| 越权类 | 权限, 授权, 鉴权, 越权, 提权, 特权, 限制, 控制, 访问, 仿冒, 伪装, 伪造, 假冒, 绕过, 认证, 管理员, 管理面, 管理者, 后台, 后门, valid, verif, right, permi, competence, authori, authenticat, restrict, access, control, falsific,  forgery, bypass, certifi, counterfeit, impersonat, masquerad, root, manager, sudo, admin, shell, (back, door), (privilege, escalation), (ultra, vires) |
| 漏洞相关类 | 漏洞, 漏洞, cve, cwe, 病毒, 木马, 蠕虫, 僵尸, 脚本, 恶意, 非法, 异常,  风险, 安全, 满足, 符合, 故障, 攻击, 危害, 补丁, 修复, exception, vulnerab, exploit, hole, bug, virus, trojan, zombie, botnet, script,  malicious, malware, illegal, risk, safe, secur, meet, satisf, conform, hack, attack, hazard, threat, patch, fix, (defect, fix), (defect, repair), (fault, fix), (fault, repair) |
| 常见 Web 类 | 注入, inject, 任意, 未公开接口, undisclosed interface, 逃逸, escape, 拒绝服务, dos, 文件上传, 文件下载, (file, upload), (file, download), xss, csrf, 跨站, ssrf, crlf, 劫持, hijack, 重定向, retarget, redirect,  序列化, serializ, (命令, 执行), (代码, 执行), (command, exec), (code, exec), (command, inject), (denial, of, service), (cross, site) |
| 安全规范类 | 滥用, 缺失, 缺少, 欠缺, 严重, 损失, 防御, 防范, 防护, 防止, 抵御, 校验,  监视, 排查, 排除, 过滤, 红线, 规范, 明文, 敏感, 匿名, 口令, 身份证, 账号, 加密, 解密, 密钥, 私钥, 基线, 未声明, 违反, 违背, defens, prevent, preserv, protect, monit, watch, check, exclus, filter, abuse, miss, lack, integr,  severe, severity, loss, redline, plaintext, ciphertext, sensitiv, anonymi,  passwd, password, token, crypt, key, cipher, baseline, declar, violat, against, topn, top-n, top\_n, top n, (red, line), (red, lining), (id, card), (bank, account) |

致谢

在时光的长河中漫步，我如一片随风起舞的叶，有幸落在了清华这棵参天大树上。回首往昔，心中满是幸运与感激。感谢命运让我在这片土地上遇到如此多美好的灵魂，他们如同夜空中的繁星，编织了我梦想的轨迹。

亦师亦友，师恩难忘。首先我要衷心感谢我的导师肖喜老师，以“天道酬勤”的精神，引领我在学术的道路上不断探索与进步。肖老师治学严谨，诲人不倦，不仅在学业上给予我悉心指导，更在生活上给予我无私的关怀。同时，我也要感谢所有计算机学科的老师们，给予我课程、科研以及实践上最有价值的建议与帮助；感谢实习期间的崔鹍老师和蒋振超老师，为我打开了工程世界的大门；感谢李清、胡光武等老师在科研论文发表过程中给予的宝贵指导；感谢在学工任职期间的郝治翰老师，帮助我在个人能力和综合素质上全面发展；感谢我的班导师卢姝和班助王锦鹏，是你们的支持让我的研究生生活更加顺遂且丰富多彩。

同窗岁月，挚友情长。感谢肖文韬、董宇韬、毛科龙师兄带领我走进科研的大门；感谢林铸天、孙文举、沈晓腾、朱皓博、陈志成、岳浩东等所有同门战友的一路支持与陪伴；感谢在马协遇到的石明聪、覃忠甜、寇周斌、李亮、高丽、彭路、崔姗姗、查耀华、季慧珍、孔德强、吴旭、管柯琴、李峻航、李梦涵、魏昶旭、覃宝钻等所有可亲可爱的小伙伴们，让南国清华成为了我心中的第二个家；感谢领客九班为我的研究生生涯带来了梦幻般的开端；感谢生活部不仅赋予我成长的机会，也让我收获了真挚的友谊；特别感谢那些多年来的挚友，周金发、李家宝、吴裕辰、周博博、王成龙、李龙甄、张高峰、户辽方、李通、唐亿、孟响、陈文曦、李朝恒，无论天涯海角，你们的支持始终如一。

初见乍惊欢，久处仍怦然。感谢我的女孩童话， 给予我无尽的鼓励、支持与信任，你常说我是你的太阳，可在这漫天星辰中，你才是那独一无二、最美丽、最温柔、照亮我每一个夜晚前行道路上的月亮呀。

谁言寸草心，报得三春晖。最后， 我要深深感谢我的父母和奶奶。是你们用无尽的爱和牺牲，为我搭建了一个可以追逐梦想的平台，无论何时何地，你们的支持都是我最坚强的后盾。愿岁月温柔以待，让我在未来慢慢报答你们的养育之恩。

雁过留声，风过疏竹。感谢生命中的每一次遇见，愿我们都能珍惜这份因缘际会带来的美好，在未来的岁月里继续书写属于自己的精彩篇章。

声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签名：日期：

个人简历、在学期间完成的相关学术成果

个人简历

2000 年5月10日出生于安徽省宿州市砀山县。

2019 年9月考入北京邮电大学网络空间安全学院网络空间安全专业，2022年6月本科毕业并获得工学学士学位。

2022 年9月免试进入清华大学深圳国际研究生院攻读计算机技术专业硕士至今。

在学期间完成的相关学术成果

学术论文：

[1] Xiao X, **Wang S**, Hu G, et al. Rbljan: Robust byte-label joint attention networkfor network traffic classification[J]. IEEE Transactions on Dependable and Secure Computing, 2024.（CCF-A类期刊、学生一作）

指导教师评语

论文提出了……

答辩委员会决议书

论文提出了……

论文取得的主要创新性成果包括：

1.……

2.……

3.……

论文工作表明作者在×××××具有×××××知识，具有××××能力，论文××××,答辩××××。

答辩委员会表决，(×票/一致）同意通过论文答辩，并建议授予×××（姓名）×××（门类）学博士/硕士学位。