软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007375] [CSTR: 32375.14.jos.007375]

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

基于结构感知图神经网络的多类别漏洞检测*

曹思聪[1], 孙小兵[1], 薄莉莉[1], 吴潇雪[1], 李 斌[1], 陈 厅[2], 罗夏朴[3], 张 涛[4], 刘 维[1]

[1](扬州大学 信息工程学院, 江苏 扬州 225127)

[2](电子科技大学 计算机科学与工程学院, 四川 成都 611731)

[3](香港理工大学 计算机系, 香港 999077)

[4](澳门科技大学 计算机科学与工程学院, 澳门 999078)

通信作者: 孙小兵, E-mail: xbsun@yzu.edu.cn

摘 要: 软件漏洞威胁着现实世界系统的安全. 近年来, 基于学习的漏洞检测方法 (尤其是基于深度学习的方法) 由于其从大量漏洞样本中挖掘隐式漏洞特征的显著优势, 得到了广泛的研究. 然而, 由于不同类型漏洞之间的特征差异和数据分布不平衡问题, 现有基于深度学习的漏洞检测方法难以准确识别具体的漏洞类型. 因此, 提出一种基于深度学习的多类型漏洞检测方法 MulVD. MulVD 构建了一种新型的结构感知图神经网络 (SA-GNN), 它可以自适应地为不同类型的漏洞提取局部典型的漏洞模式, 并在不引入噪声的情况下重新平衡数据分布. 检验所提方法在二分类和多分类漏洞检测任务中的有效性. 实验结果表明, MulVD 显著提高了现有基于深度学习的漏洞检测技术的性能.

关键词: 漏洞检测; 注意力机制; 图神经网络; 多类别分类

中图法分类号: TP311

中文引用格式: 曹思聪, 孙小兵, 薄莉莉, 吴潇雪, 李斌, 陈厅, 罗夏朴, 张涛, 刘维. 基于结构感知图神经网络的多类别漏洞检测. 软件学报. http://www.jos.org.cn/1000-9825/7375.htm

英文引用格式: Cao SC, Sun XB, Bo LL, Wu XX, Li B, Chen T, Luo XP, Zhang T, Liu W. Multi-class Vulnerability Detection with Structure-aware Graph Neural Network. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7375.htm

Multi-class Vulnerability Detection with Structure-aware Graph Neural Network

CAO Si-Cong[1], SUN Xiao-Bing[1], BO Li-Li[1], WU Xiao-Xue[1], LI Bin[1], CHEN Ting[2], LUO Xia-Pu[3], ZHANG Tao[4], LIU Wei[1]

[1](College of Information Engineering, Yangzhou University, Yangzhou 225127, China)

[2](School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China)

[3](Department of Computing, Hong Kong Polytechnic University, Hong Kong 999077, China)

[4](Faculty of Information Technology, Macau University of Science and Technology, Macao 999078, China)

Abstract: Software vulnerabilities pose significant threats to real-world systems. In recent years, learning-based vulnerability detection methods, especially deep learning-based approaches, have gained widespread attention due to their ability to extract implicit vulnerability features from large-scale vulnerability samples. However, due to differences in features among different types of vulnerabilities and the problem of imbalanced data distribution, existing deep learning-based vulnerability detection methods struggle to accurately identify specific vulnerability types. To address this issue, this study proposes MulVD, a deep learning-based multi-class vulnerability detection method. MulVD constructs a structure-aware graph neural network (SA-GNN) that can adaptively extract local and representative vulnerability patterns while rebalancing the data distribution without introducing noise. The effectiveness of the proposed approach in both

* 基金项目: 国家自然科学基金 (62202414); 江苏省“六大人才高峰”高层次人才项目 (RJFW-053); 江苏省“333”工程中青年科学技术带头人项目; 云南省软件工程重点实验室开放基金 (2023SE201)
收稿时间: 2023-07-03; 修改时间: 2023-11-03, 2024-05-17; 采用时间: 2024-12-20; jos 在线出版时间: 2025-04-23

binary and multi-class vulnerability detection tasks is evaluated. Experimental results demonstrate that MulVD significantly improves the performance of existing deep learning-based vulnerability detection techniques.

Key words: vulnerability detection; attention mechanism; graph neural network (GNN); multi-class classification

漏洞 (vulnerability) 是软件在设计、实现、配置等过程中引入的安全缺陷[1], 一旦被攻击者利用, 可导致敏感信息泄露、恶意勒索、系统崩溃等后果, 轻则危害计算机系统安全, 重则造成重大财产损失甚至危及生命安全. 随着越来越多的行业使用软件作为其业务载体, 由漏洞导致的软件安全问题也正在成为当今社会的基础性问题之一. 传统的漏洞检测技术通常以静态分析为基础, 利用预定义的漏洞规则或形式化验证等手段对程序源代码进行审计, 具有覆盖率高、分析速度快等优势[2]. 然而, 随着软件与漏洞复杂性的增加, 此类方法需要较强的领域知识与复杂的计算规则的劣势也逐渐显露, 导致误报率和漏报率较高.

近年来, 随着以深度学习 (deep learning, DL) 为代表的人工智能技术的兴起, 其展现出的强大的数据挖掘与分析能力也为漏洞检测技术带来了新的发展契机[3,4]: 一方面, 开源软件的蓬勃发展使得漏洞数据库与开源社区中积累了海量的漏洞代码, 为基于学习的漏洞检测技术研究提供了坚实的数据基础; 另一方面, 源代码中包含了丰富的语法语义信息, 使得神经网络模型从复杂数据中提取漏洞特征成为可能. 这类方法主要通过将训练数据解析为语法树、程序图等包含丰富代码特征信息的抽象表示, 并设计合适的深度神经网络模型以充分挖掘漏洞的隐式模式, 实现软件漏洞检测的自动化与智能化.

尽管围绕基于深度学习的漏洞检测已涌现出一批高质量的研究成果[5−10], 但几乎所有方法仅停留在判别给定源代码是否包含漏洞, 无法提供细粒度的漏洞类型信息. 最近的研究表明, 带有详细漏洞类型的检测结果对于开发人员而言更加实用, 一定程度上有助于减轻人工漏洞分析的工作量[11,12]. 为了满足特定场景下的检测需求, 现有工作大多通过对目标类型的漏洞代码特征进行针对性的表示与抽取, 以训练一个高精度的专用检测模型[7,8]. 尽管此类方法在特定类型的漏洞检测任务中取得了良好的效果, 但仍具有较高的局限性: 一方面, 由于漏洞类型众多, 仅针对几种常见的漏洞类型构建检测模型难以满足不同场景下的检测需求; 另一方面, 不同类型的漏洞在真实场景中往往呈现出项目内不平衡、项目间差异大的分布态势[13], 使得为每种漏洞类型分别训练一个检测模型所需的样本数量严重不足, 导致方法性能大幅下降.

针对上述问题, 本文提出了一种基于结构感知图神经网络的多类别漏洞检测方法 MulVD (multi-class vulnerability detection). 首先, 本方法对源代码进行解析, 构建代码属性图 (code property graph, CPG)[14]以对不同类型的漏洞特征进行统一建模. 接着, 为了在不同级别上学习准确的代码语义, 我们分别使用 Word2Vec[15]和 Node2Vec[16]对 CPG 中的代码令牌节点和语句节点进行特征编码. 然后, 我们构建了一种新颖的结构感知图神经网络 (structure-aware graph neural network, SA-GNN). 该模型能够自适应地挖掘不同类型漏洞的局部典型特征, 并在不引入噪声的情况下动态平衡漏洞分布差异, 实现高质量的特征嵌入表示. 最后, 通过多类别分类器实现对漏洞类型的识别.

为了验证所提方法的有效性, 本文采用了 3 个广泛使用的真实漏洞数据集来分别评估本方法在漏洞函数预测 (二分类) 和漏洞类型识别 (多分类) 两个任务上的有效性. 本文选取了 3 个基于规则的漏洞检测工具 (Flawfinder[17]、RATS[18]、Cppcheck[19]) 和 5 个基于深度学习的漏洞检测方法 (VulDeePecker[6]、SySeVR[20]、Devign[21]、ReVeal[22]和 LineVul[23]) 作为对比基线. 实验结果表明, 本文方法在漏洞函数预测和漏洞类型识别任务上 $F1$ 分数达到了 66.3% 和 45.5%, 相较于最优基线性能 ReVeal 分别提升了 3.27% 和 13.47%. 此外, 对于所提出的结构感知图神经网络 SA-GNN 的消融实验, 证明其可以自适应地捕获不同类型漏洞的局部典型特征, 证明了本方法的有效性.

本文的主要贡献可总结如下.

(1) 提出了一种面向源代码多类别漏洞检测方法 MulVD, 实现对多种漏洞类型的统一建模与识别, 提升了现有基于深度学习的漏洞检测方法在现实场景中的通用性.

(2) 设计了一种新颖的结构感知图神经网络模型, 可以自适应地挖掘不同漏洞类型的局部典型特征, 并动态平衡分布差异, 增强了多类别漏洞特征学习的准确性.

(3) 通过将本文所提出的方法与现有漏洞检测方法在两个真实的漏洞数据集上进行实验对比, 结果表明, 本文所提出的方法在漏洞函数预测和漏洞类型识别任务上均有显著提升.

本文第 1 节介绍基于深度学习的漏洞检测的相关工作. 第 2 节介绍本文所需的基础知识, 包括图神经网络和小样本学习的相关概念. 第 3 节介绍本文构建的基于结构感知图神经网络的漏洞类型识别方法以及实现细节. 第 4 节提出研究问题并设计实验. 第 5 节对实验结果进行分析. 第 6 节总结全文并展望未来工作.

1 相关工作

近年来, 开源社区中代码资源的不断积累以及深度学习在图像处理、语音识别、自然语言处理等领域的突破性进展, 为以代码为中心的软件漏洞检测提供了新的解决思路[24]. 基于深度学习的漏洞检测方法通常将程序转换为低维稠密的向量表示, 利用神经网络模型提取高质量特征和拟合复杂函数的能力, 挖掘漏洞代码的隐式模式. 相比于传统依赖人工定义的启发式规则挖掘程序的特征信息, 基于深度学习的漏洞检测具有特征抽取准确、端到端构建等优点. 依照不同的关注点, 现有工作可从漏洞特征建模和检测模型构建两个角度进行分类.

具体而言, 依照漏洞特征建模方式的不同, 基于深度学习的漏洞检测方法可进一步分为基于代码令牌和基于代码结构两类.

基于代码令牌的漏洞特征表示方法将代码视为与自然语言一样具有统计特性的文本, 利用长短期记忆网络 (long short term memory, LSTM)、Transformer[25]等模型提取漏洞代码中常见的词法或句法特征以执行检测. 例如: Dam 等人[26]利用 LSTM 分别从代码令牌和语句序列中抽取漏洞程序的局部和全局特征, 利用随机森林等 4 种分类器进行漏洞检测.

相较于自然语言, 程序中包含了丰富、明确且复杂的结构信息. 因此, 获取代码的结构语义显得尤为重要. 漏洞代码常见的特征建模形式大多是经典程序抽象表示的复合或优化, 包括抽象语法树 (abstract syntax tree, AST)、控制流图 (control flow graph, CFG)、程序依赖图 (program dependence graph, PDG) 等. 抽象语法树 AST 在细粒度的代码令牌级别反映了代码的语法信息, 有助于揭示如不安全的参数使用等漏洞特征. 相比之下, 控制流图 CFG 和程序依赖图 PDG 在较粗粒度的语句级别刻画了代码的执行逻辑与数据流转等语义信息, 有利于捕获如缓冲区溢出、代码注入等更为复杂的漏洞特征. 此外, 一些工作也尝试通过程序切片、图简化等启发式方法进一步精化特征表示, 以更加关注漏洞相关的局部代码片段.

根据特征学习方式的不同, 基于深度学习的漏洞检测方法可分为基于卷积神经网络、基于循环神经网络、基于图神经网络和基于 Transformer 这 4 种模型.

基于卷积神经网络 (convolutional neural network, CNN) 的漏洞检测模型将代码片段视为图像, 借助卷积操作在二维网格数据上强大的特征提取能力来学习漏洞特征. 例如: Russell 等人[27]利用卷积核在由连续的代码令牌及其嵌入表示构成的滑动窗口上进行特征提取, 并输入到随机森林中进行分类.

基于循环神经网络 (recurrent neural network, RNN) 的漏洞检测模型将程序视为一段连续的代码序列, 利用长短期记忆网络 LSTM、门控递归单元 GRU 等模型来学习漏洞的上下文特征. 例如: Li 等人[6]提出了 VulDeePecker, 通过将源代码表示为切片级别的序列, 并使用双向 LSTM 等 RNN 模型来训练其检测模型.

随着图神经网络 (graph neural network, GNN) 的流行, 一些研究者尝试利用 GNN 的节点信息传播机制, 从抽象语法树、控制流图等包含丰富程序结构化语义的抽象表示中挖掘漏洞代码的结构语义[28]. 代表性模型包括图卷积神经网络 GCN[29]、门控图神经网络 GGNN[30]、图注意力网络 GAT[31]等. 例如: Zhou 等人[21]通过在基于 GGNN 的特征表示模块后添加一层卷积层以从代码复合图表示中挖掘全面丰富的语法语义特征.

基于 Transformer 的漏洞检测模型同样将程序代码视为连续的语句序列, 但其利用 Transformer 架构捕获语句间长依赖语义关系的能力可以更好地挖掘漏洞语义. 例如: Fu 等人[23]利用 CodeBERT 模型在漏洞数据集上进行微调, 并利用自注意力机制以定位漏洞语句.

目前, 基于深度学习的漏洞检测方法主要侧重于判别目标程序是否包含漏洞, 鲜有研究关注漏洞类型的识别问题. Zou 等人[12]首次引入了代码注意力 (code attention) 这一概念, 通过在程序切片时考虑控制和数据依赖关系

以包含更加全面的漏洞语法和语义信息以支持多类型的漏洞检测. 然而, 程序切片高度依赖专家经验, 且难以覆盖所有漏洞类型. 考虑到图表示在建模漏洞程序结构化的语法语义特征方面的有效性, 本文构建了一种新颖的结构感知图神经网络 SA-GNN, 基于漏洞函数的代码属性图 CPG 自适应地挖掘不同类型漏洞的局部典型特征. 此外, 尽管一些研究者[22,32]通过 SMOTE[33]等数据采样技术有效地缓解了模型训练过程中存在的数据不平衡问题, 但当迁移到多类别漏洞检测场景下时, 随机生成少数类样本可能会导致不同漏洞类型的特征空间存在重叠与错误泛化的风险 (即属于某种少数类漏洞的合成样本出现在其他类型漏洞的特征空间中), 阻碍了深度学习模型学习到精确的漏洞特征. 因此, 本文在结构感知图神经网络 SA-GNN 之后添加过采样层, 通过动态评估合成样本可能带来的潜在负面影响, 在不引入噪声的情况下生成最优样本以平衡漏洞分布差异.

2 基础知识

本节将对本文涉及的相关概念和基本知识予以介绍, 主要包括图神经网络和不平衡样本学习.

2.1 图神经网络

鉴于在学习程序语义方面的卓越能力, 图神经网络 GNN 被广泛应用于代码检索、缺陷定位、克隆检测等面向源代码的软件工程任务, 并取得了显著的性能提升. 目前主流的 GNN 模型大多采用邻域聚合方案, 通过迭代聚合周围 k 跳邻居的特征嵌入来更新中心节点的向量表示:

$$h_v^{(t)} = \sigma\left(h_v^{(t-1)}, AGG^{(t)}\left(\left\{h_u^{(t-1)} :\ u \in \mathcal{N}(v)\right\}\right)\right) \tag{1}$$

其中, $h_v^{(t)}$ 表示节点 v 在时间步长 t 下的特征嵌入, $u \in \mathcal{N}(v)$ 表示节点 v 的邻居节点, $AGG^{(t)}$ 和 $\sigma(\cdot)$ 分别表示更新节点嵌入表示所采用的图聚合函数和激活函数.

通过多轮迭代, 模型可以捕获节点间的依赖关系, 进而学习到图的结构信息. 根据下游任务的属性不同, T 轮后的节点嵌入表示 $h_v^{(T)}$ 可用于图分类、节点分类和链接预测任务. 在本文中, 多类别漏洞检测任务被形式化为多类图分类任务, 即: 给定一个漏洞函数 i 所对应的图表示 G_i, 检测模型学习一个映射函数 f 以预测其所属的类型 $l \in \{0,1,2,\ldots,n\}$. 其中, 0 表示不存在漏洞, 1–n 表示漏洞类型标签.

2.2 不平衡样本学习

由于软件工程领域中标签数据的稀缺性, 面向漏洞检测[32]、缺陷预测[34]等分类任务的深度学习方法普遍面临严重的数据不平衡问题. 目前已经提出了一些措施来缓解不平衡的数据分布对模型性能的负面影响. 根据所采用的方法不同, 大致可分为基于数据选择和基于数据采样两种.

基于数据选择的方法通过主动学习、原型学习等策略选择少量但重要性较高的典型样本用于模型训练. 例如: Wu 等人[35]提出了一种基于主动学习的缺陷报告检测方法, 利用不确定性采样策略选择模型置信度较低的样本用于训练以提高分类性能.

基于数据采样的方法通过过采样、欠采样等技术来平衡不同类型的样本比例. 最近的研究发现, 过采样技术可以一定程度上缓解基于深度学习的漏洞检测方法的数据不平衡问题并提高模型的检测能力[36]. 最常用的数据过采样技术是合成少数类过采样 (synthetic minority oversampling technique, SMOTE)[33], 其核心思想是在少数类集合中随机选择任意样本及其 k 近邻, 在两者特征空间的连线上生成合成样本, 从而扩充少数类样本数据.

尽管这些方法在二分类场景下均取得了不错的性能提升, 但在数据分布高度不平衡的多类别漏洞检测场景下, 过度泛化与类重叠风险可能损害模型学习少数类漏洞实例特征的能力. 因此, 本方法采用了一种基于选择权重的面向多分类任务的合成少数类过采样技术 SMOM[37], 通过在特征空间中动态评估合成样本可能带来的潜在负面影响, 在不引入噪声的情况下生成最优样本以平衡漏洞分布差异.

3 基于结构感知图神经网络的多类别漏洞检测方法

为了更准确地检测软件中的潜在漏洞并提供支撑性的类型信息, 本文提出了一种基于结构感知图神经网络的

多类别漏洞检测方法. 该方法的整体框架如图 1 所示, 主要由以下 3 个模块构成: 漏洞特征抽取模块、检测模型构建模块以及漏洞检测模块. 具体而言, 首先对训练数据 (包括漏洞代码和非漏洞代码) 进行静态分析, 构建代码属性图, 并使用词嵌入技术初始化图节点特征; 然后, 将代码属性图及其结点特征输入到结构感知的图神经网络中, 学习不同类型漏洞间差异化的局部模式, 训练多类别漏洞检测模型; 最后, 将源代码输入到训练好的漏洞检测模型中, 输出被测程序是否存在漏洞, 若存在, 输出具体的漏洞类型.

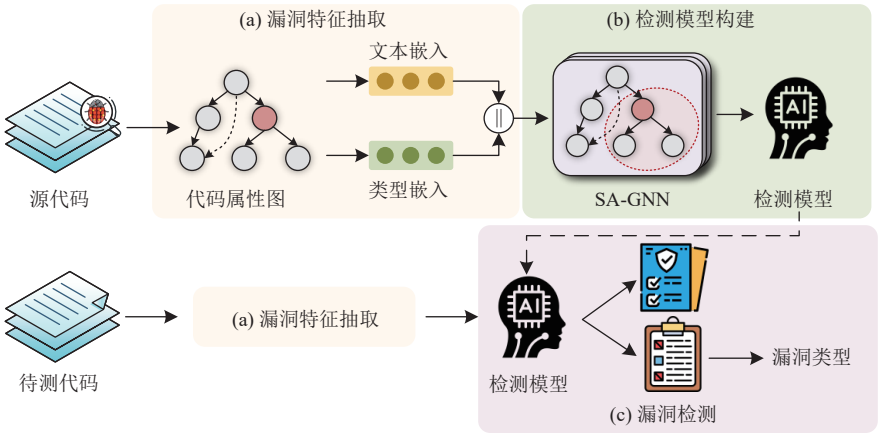


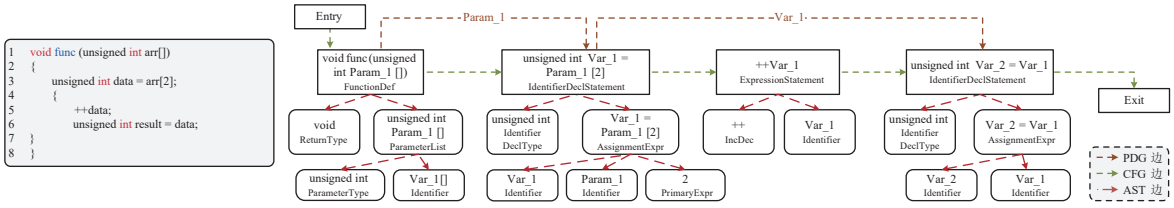
图 1 基于结构感知图神经网络的多类别漏洞检测方法整体框架

3.1 漏洞特征抽取

3.1.1 代码属性图构建与标准化

为了容纳不同类型漏洞的特征信息, 本文借鉴现有基于深度学习的通用漏洞检测方法[21,22], 采用代码属性图 CPG[14]作为程序的表征方式. CPG 是一种综合了抽象语法树 AST、控制流图 CFG 和程序依赖图 PDG 的联合数据结构, 其本质是在程序 AST (通常以函数为构建粒度) 的基础上添加语句间的控制流和程序依赖 (包括控制依赖和数据依赖) 以反映漏洞代码的语法和语义信息. 相较于单一的程序表示形式, 代码属性图 CPG 将程序的语法、控制流和数据依赖性整合到统一的图结构中, 在全面地捕获漏洞代码的语法、语义信息的同时, 能够适配图神经网络模型的特征学习方式, 有助于提升漏洞检测的效果.

图 2(a) 给出了一个存在整数溢出 (CWE-190) 漏洞的代码实例, 其相应的代码属性图如图 2(b) 所示. 图中每个节点包含两种属性: 代码及其语法类型. 为了避免模型学习到频繁出现的变量名等无关属性与漏洞标签间的虚假映射关系, 本方法使用统一的抽象符号 Var_i 和 Param_j 对用户自定义的局部变量名和参数名进行标准化. 例如: 图 2(a) 代码示例第 3 行的局部变量 data 被替换为 Var_1, 其相应的语法类型为 Identifier.



(a) 漏洞代码实例 (b) 代码属性图CPG

图 2 代码属性图构建与标准化样例

3.1.2 图节点特征嵌入

在获得代码结构化的抽象表示 CPG 后, 需要将图中的所有节点转换为向量表示, 以输入到图神经网络模型中进行特征学习. 具体地, 本文首先使用 Word2Vec[15]对抽象语法树 AST 中的叶子节点进行编码. 考虑到使用 Word2Vec

将图转换为低维向量时可能会丢失一些重要信息, 我们使用 Node2Vec[16]对 CPG 中的语句节点 (即控制流图 CFG 和程序依赖图 PDG 中的节点) 进行编码. Node2Vec 是一种被广泛使用的网络嵌入算法. 该算法通过随机游走学习图节点的嵌入表示, 可以在保留图结构信息的同时有效地将复杂大型的图结构转化成低维稠密的向量表示. 在基于图的漏洞检测场景下, 由于 Node2Vec 通过双向传输两个节点的信息, 并使用周围结构的信息对一个节点进行编码 [38], 可以准确捕获语句之间的数据依赖和控制依赖信息. 此外, 与先前的工作类似, 本方法也考虑每个节点的抽象类型 (标识符、表达式等), 因为其反映了每个节点所代表的代码属性, 使得漏洞模式更加通用. 本文通过 One-Hot 编码对每个节点的抽象类型进行特征嵌入. 最后, 将每个节点的特征嵌入 (Word2Vec 或 Node2Vec 编码的向量表示) 与类型嵌入 (One-Hot 编码的向量表示) 进行拼接, 作为后续特征表示学习的输入.

3.2 检测模型构建

为了从分布高度不平衡的数据样本中学习到不同类型漏洞的局部典型特征, 本文提出了一种结构感知的图神经网络 (structure-aware graph neural network, SA-GNN), 如图 3 所示. SA-GNN 由特征挖掘模块、类别平衡模块和分类模块这 3 个模块组成, 背后的关键思想在于使用基于图的注意力机制来捕获与漏洞代码类型相关的局部程序子图, 并自适应地选择少数类合成样本的生成区域以避免类重叠和错误泛化的风险.

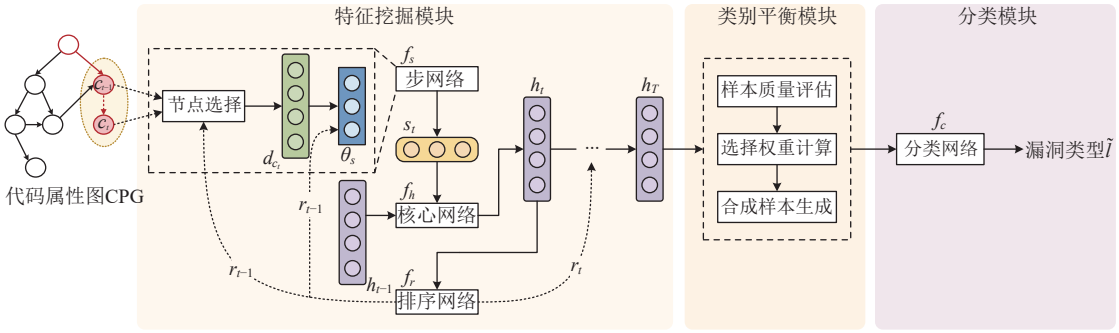


图 3 结构感知图神经网络 SA-GNN 整体架构

3.2.1 特征挖掘模块

漏洞的产生往往只涉及少数几条程序语句, 简单地从函数粒度进行特征挖掘会引入较多与漏洞无关的噪声信息, 而细粒度的代码切片由于高度依赖专家经验, 难以覆盖所有的漏洞类型. 因此, 本文采用了一种注意力引导的随机游走算法 GAM[39], 通过从 CPG 中自适应地选择一组存在语法语义关联 (即节点间存在 AST、CFG 或 PDG 边) 的连续节点, 关注不同类型漏洞的局部典型特征 (即判别子图), 从而提高漏洞特征挖掘的准确性. GAM 将局部判别子图挖掘问题建模为部分可观察的马尔可夫决策过程 (partially observable Markov decision process, POMDP), 即在局部判别子图的挖掘过程中, 仅利用当前时间步 $t \in T$ 下沿着代码属性图 CPG 遍历过的所有节点的特征信息来预测输入程序的类别标签, 并在下一个时间步 $t+1$ 优先考虑对正确预测类别标签贡献更大的相邻节点进行探索. 因此, 在图类别标签信号的监督下, 模型通过迭代训练学会了对不同节点的重要性进行优先级排序, 以便在漏洞分类的过程中专注于不同类型漏洞的局部典型特征, 而忽略其余不相关的噪声信息.

具体而言, 特征挖掘模块主要由步网络 (step network) 和基于 LSTM 的核心网络 (core network) 组成 (如图 3 所示). 给定漏洞样本的代码属性图 $\mathcal{G}$ 及其相应的类型标签 l, 步网络 f_s 基于当前节点 c_{t-1} 的排序向量 r_{t-1} 选择特征重要性最高的邻居节点 c_t, 计算在当前时间步 t 下的步嵌入 s_t:

$$s_t = f_s(d_{c_t}, r_{t-1}; \theta_s) \tag{2}$$

其中, d_{c_t} 表示邻居节点 c_t 的特征嵌入, θ_s 表示步网络 f_s 中用于将特征嵌入映射到低维隐空间的参数化线性层.

基于当前时间步下的步嵌入 s_t 和上一时间步下的历史向量 h_{t-1}, 核心网络 f_h 计算当前时间步下的历史向量 h_t 以更新局部子图中所有节点在时间步 t 下的特征信息总量:

$$h_t = f_h(s_t, h_{t-1}; \theta_h) \tag{3}$$

其中, θ_h 表示核心网络 f_h 中用于更新时间步 t 下的特征信息总量的参数化线性层.

为了挖掘能够准确区分不同类型漏洞特征的局部判别 CPG 子图, 本文通过最大化局部子图在时间步 T 下的总奖励 $J(\theta)$ 以训练最优特征挖掘模型:

$$J(\theta) = \mathbb{E}_{P(s_{1:T};\theta)}[R] \tag{4}$$

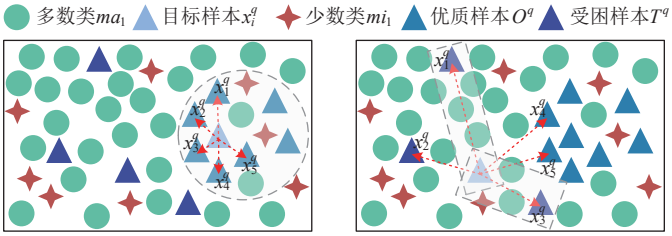
其中, $s_{1:T}$ 表示步网络 f_s 和核心网络 f_h 在时间步 T 内挖掘局部判别 CPG 子图的动作序列, $\theta = \{\theta_h, \theta_s\}$ 表示步网络和核心网络的参数. $R = \sum_{t=1}^{T} r_t$ 表示时间步 T 下特征挖掘模型收到的总奖励. 其中, 如果模型能够基于所挖掘的局部子图正确识别漏洞类型, 则 $r_T = 1$, 否则 $r_T = -1$.

3.2.2 类别平衡模块

为了缓解现有不平衡样本学习方法在多分类场景下的泛化能力较差的问题, 本文采用了一种基于 k 近邻 (k-nearest neighbors, k-NN) 的合成少数类过采样算法 SMOM[37], 通过自适应地为每个漏洞少数类样本邻居方向分配一个选择权重以生成高质量的合成样本 (即此类合成样本不会导致少数类与多数类特征区域的重叠, 降低模型的分类精度). 具体而言, 主要包含以下 3 个步骤.

(1) 样本质量评估

根据所属类型在训练集中的占比, 将漏洞样本 (即通过特征挖掘模块得到的局部判别 CPG 子图) 划分为少数类或多数类. 其中, 样本规模小于类平均样本数 M/L 的漏洞类型被归类为少数类, 否则归为多数类. M 和 L 分别表示漏洞样本数和漏洞类型数. 对于任意的少数漏洞类型 q, 其样本集 X^q 在特征空间中将进一步被划分为优质样本 O^q (即基于此类样本生成的合成实例不会对其他类产生负面影响) 和受困样本 T^q (即基于此类样本生成的合成实例可能会损害模型对其他类的学习能力). 特别地, 本方法采用基于密度的无监督聚类算法 HDBSCAN[40]对特征空间中的所有样本进行聚类. 若漏洞样本 $x^q \in X^q$ 分布于少数类 q 占主导的聚类簇, 则被标记为优质样本 O^q, 否则被标记为受困样本 T^q. 相较于其他流行的聚类算法, HDBSCAN 可以发现任意形状和大小的聚类簇, 适合处理具有不确定分布的漏洞样本. 如图 4(a) 所示, 灰色区域为 HDBSCAN 生成的某一聚类簇, 该聚类簇内绝大多数 (8/11) 漏洞样本的类型为少数类 q. 因此, 该区域内的漏洞样本 $x^q \in X^q$ 为优质样本 O^q, 其余漏洞样本 $\bar{x}^q \in X^q$ 为受困样本 T^q.



(a) 面向优质样本的合成样本生成 (b) 面向受困样本的合成样本生成

图 4 少数类合成样本生成实例

(2) 选择权重计算

针对优质样本 O^q, 本文采取与常用的过采样算法 SMOTE[33]相同的生成策略, 通过沿少数类样本的 k 近邻方向 (如图 4(a) 红色箭头所示, $k = 5$) 随机生成合成样本. 针对受困样本 T^q, 本文计算在其 k 近邻方向生成合成样本时的选择权重:

$$1/\mathrm{e}^{\eta_1 \frac{\gamma^{mi}}{\gamma^q} + \eta_2 E_{mi} + w_2(\eta_1 \frac{\gamma^{ma}}{\gamma^q} + \eta_2 E_{ma})} + w_1 \mathrm{e}^{-\frac{\gamma^q}{\gamma^{ma}+\gamma^{mi}+\gamma^q}} \tag{5}$$

其中, $\mathrm{e}^{\eta_1 \frac{\gamma^{mi}}{\gamma^q} + \eta_2 E_{mi} + w_2(\eta_1 \frac{\gamma^{ma}}{\gamma^q} + \eta_2 E_{ma})}$ 表示过度泛化因子, 用于量化因合成样本落入其他类的特征区域导致错误泛化的风险. $\mathrm{e}^{-\frac{\gamma^q}{\gamma^{ma}+\gamma^{mi}+\gamma^q}}$ 表示难度因子, 其衡量了 k 近邻方向学习少数类特征的难易程度. γ^q、γ^{ma} 和 γ^{mi} 分别表示特征邻域中目标少数类 q、多数类 ma 和其余少数类 mi (不包括目标少数类 q) 的样本个数. 显然, 目标少数类 q 样本在特征邻

域中的占比 $\frac{\gamma^q}{\gamma^{ma}+\gamma^{mi}+\gamma^q}$ 越低, 特征学习难度则越大. w_1 是用于平衡过度泛化因子和难度因子的权重参数. w_2 表示用于调整多数类对过度泛化因子影响的权重参数. η_1 和 η_2 为重缩放参数. 对于这些超参数, 我们采用经过多个大规模数据集验证的最优组合 (即 $w_1=0.2$、$w_2=1/2$、$\eta_1=1/3$、$\eta_2=0.2$). E_{mi} ($/E_{ma}$) 表示特征邻域内所有少数类 (/多数类) 样本的熵:

$$E_{mi}=\sum\nolimits_{q\in S_{mi}}\left(\frac{\gamma^q}{\gamma^{mi}}\log\frac{\gamma^q}{\gamma^{mi}}\right)\Big/\log|S_{mi}| \tag{6}$$

$$E_{ma}=\sum\nolimits_{q\in S_{ma}}\left(\frac{\gamma^q}{\gamma^{ma}}\log\frac{\gamma^q}{\gamma^{ma}}\right)\Big/\log|S_{ma}| \tag{7}$$

其中, S_{mi} ($/S_{ma}$) 表示少数类 (/多数类) 集合. 如图 4(b) 所示, 由于在特征邻域 $x_i^q\to x_3^q$ 方向上包含其他少数类样本 mi_1, 在该方向上生成合成样本比在 $x_i^q\to x_1^q$ 方向上更可能存在过度泛化的风险. 因此, 在特征邻域 $x_i^q\to x_3^q$ 方向生成合成样本时的权重小于 $x_i^q\to x_1^q$ 方向.

(3) 合成样本生成

对于任意少数类漏洞样本 $x_i^q\in X^q$, 本文根据其属于优质样本还是受困样本, 选择相应的权重并在其 k 近邻方向生成合成样本. 这些合成样本将与真实漏洞样本一起作为输入, 用于训练多类别漏洞检测模型.

3.2.3 分类模块

为了识别具体的漏洞类型, 本文采用多层感知机 (multi-layer perceptron, MLP) 作为分类网络 $f_c(\cdot;\theta_c)$, 并使用 *Softmax* 交叉熵损失对其进行训练. 基于局部判别子图的特征嵌入, 分类网络 f_c 输出其相应的预测标签 $\tilde{l}_T$. 该分类网络可以通过最大化 $\log\pi(l_T|s_{1:T};\theta_c)$ 进行优化. 其中, l_T 是训练漏洞样本的真实标签, θ_c 表示分类网络的参数.

3.3 漏洞检测

在漏洞检测阶段, 本文以函数为粒度对输入程序进行拆分, 通过特征挖掘模块生成相应的代码属性图 CPG 以及初始的节点特征嵌入. 然后, 将代码属性图与节点特征嵌入作为输入, 馈送到经过充分训练的漏洞检测模型中, 预测其相应的漏洞标签. 最后, 预测标签为 1–n 的函数将作为潜在的漏洞代码输出.

4 实验设计

验证所提方法的有效性, 本文所设计的实验旨在回答以下 3 个研究问题.

RQ1: MulVD 在漏洞函数预测 (二分类) 任务上的性能是否优于基线方法?

RQ2: MulVD 在漏洞类型识别 (多分类) 任务上的性能是否优于基线方法?

RQ3: 本文提出的结构感知图神经网络 SA-GNN 能否提高漏洞类型识别的性能?

本节中将详细地介绍实验设计的细节, 包括实验数据集、实验环境、基线方法以及评估指标.

4.1 实验数据集

为了验证 RQ1, 本文在两个广泛使用的真实漏洞数据集 Devign[21]和 ReVeal[22]上进行了实验. Devign 数据集完全基于人工标注, 通过对关键词筛选后的安全相关提交进行人工标注与交差验证, 提取与漏洞修复相关的代码片段. 该数据集目前仅开源了 FFmpeg 和 QEMU 两个项目, 共包含 26 037 个函数样本, 漏洞函数占比为 45.7% (11 888 个). ReVeal 数据集基于漏洞跟踪库 Bugzilla 和 Debian security tracker 爬取 Chromium 和 Debian 中安全相关的修复提交. 其中, 变更代码行所涉及的补丁前和补丁后函数分别被标记为漏洞函数和非漏洞函数, 没有任何修改的函数被标记为非漏洞函数. 最终, 该数据集中共包含 1 664 个漏洞函数和 16 505 个非漏洞函数, 漏洞占比为 9.9%.

为了验证 RQ2, 本文采用另一个真实漏洞数据集 Big-Vul[41]进行实验评估. 基于 CVE 漏洞数据库, Big-Vul 爬取了 2002–2019 年间的所有漏洞条目并定位了 348 个相关的 GitHub 源代码仓库. 通过比对漏洞修复提交前后的代码变更, Big-Vul 数据集中共包含 11 823 个漏洞函数和 253 096 个非漏洞函数, 漏洞函数占比约为 4.5%, 涵盖了 91 种漏洞类型. 表 1 给出了每个漏洞数据集的详细信息.

表 1 实验数据集

数据集	项目数	漏洞类型数	总样本数	漏洞样本数	漏洞率 (%)
Devign	2	—	26 037	11 888	45.7
ReVeal	2	—	18 169	1 664	9.9
Big-Vul	348	91	264 919	11 823	4.5

此外, 由于 Big-Vul 数据集采用 CWE 多级抽象的树形漏洞分类标准, 属于同一抽象分支 (例如: CWE-120 缓冲区溢出与 CWE-125 越界读取均属于 CWE-119 缓冲区操作不当的子类型) 的漏洞类型可能存在部分特征重叠的风险, 限制了检测模型学习漏洞特征分类边界的能力. 因此, 本文采用最高级别抽象 (类级别) 的 CWE-ID 作为分类标准, 对数据集中的漏洞进行合并. 类级别的 CWE-ID 通常独立于特定的编程语言且不存在特征重叠. 本文所采用的合并后的漏洞候选类型如表 2 所示. 较低级别的特定子类型 (如 CWE-120 缓冲区溢出与 CWE-125 越界读取) 被包含在它们相应的类级 CWE-ID (即 CWE-119 缓冲区操作不当) 中, 总计 36 种漏洞类型. 不同类型漏洞的数据分布如后文图 5 所示. 其中, 占比前 10 的漏洞类型为 CWE-{119, 672, 754, 362, 327, 404, 20, 682, 834, 400}, 这些漏洞类型囊括了 Big-Vul 数据集中约 95.9% 的漏洞样本. 对于不足平均样本数 (即 $M/L = 11823/36 = 328$) 的少数类漏洞, 本文采用类别平衡模块共生成合成漏洞样本 6 194 个 (平均为每个少数类漏洞生成了 295 个合成样本), 占总漏洞样本数的 34.38%.

表 2 本文所使用的漏洞类型及其对应的 CWE-ID

漏洞类型	CWE-ID	漏洞类型	CWE-ID
Improper input validation	CWE-20	Insecure storage of sensitive information	CWE-922
Race condition	CWE-362	Interpretation conflict	CWE-436
Uncontrolled resource consumption	CWE-400	Improper synchronization	CWE-662
Improper restriction of operations within the bounds of a memory buffer	CWE-119	Externally controlled reference to a resource in another sphere	CWE-610
Improper resource shutdown or release	CWE-404	Improper initialization	CWE-665
Improper check for unusual or exceptional conditions	CWE-754	Use of incorrectly-resolved name or reference	CWE-706
Operation on a resource after expiration or release	CWE-672	Always-incorrect control flow implementation	CWE-670
Excessive iteration	CWE-834	Incorrect resource transfer between spheres	CWE-669
Incorrect calculation	CWE-682	Uncontrolled recursion	CWE-674
Use of a broken or risky cryptographic algorithm	CWE-327	Incorrect permission assignment for critical resource	CWE-732
Improper handling of exceptional conditions	CWE-755	Incorrect comparison	CWE-697
Improper encoding or escaping of output	CWE-116	Incorrect type conversion or cast	CWE-704
Exposure of sensitive information to an unauthorized actor	CWE-200	Improper control of dynamically-managed code resources	CWE-913
Improper privilege management	CWE-269	Exposure of resource to wrong sphere	CWE-668
Improper authentication	CWE-287	Injection	CWE-74
Missing encryption of sensitive data	CWE-311	Missing authorization	CWE-862
Inadequate encryption strength	CWE-326	Incorrect authorization	CWE-863
Use of insufficiently random values	CWE-330	Insufficient verification of data authenticity	CWE-345

4.2 实验环境

本文研究中涉及的实验部分均基于 PyTorch[42]框架实现, 通过使用 tree-sitter[43]生成构建代码属性图 CPG 骨干所需的抽象语法树 AST, 并基于其提供的程序间依赖信息生成相应的控制流图 CFG 和程序依赖图 PDG. 本文采用 DGL[44]深度学习库来实现漏洞检测模型的构建与训练.表 3 给出了实验涉及的超参数和具体的取值, 这些取值基于已有文献的推荐取值和采用网格搜索调优后的实际性能. 本文根据不同类型漏洞在数据集中的占比, 按照 8:1:1 将数据集随即划分为训练集、验证集和测试集. 本文实验设备的配置信息是: Intel(R) Core(TM) i9-12900K CPU、16 GB 显存的英伟达 Tesla T4 显卡、128 GB 内存、Ubuntu 18.04 操作系统、CUDA 10.1.

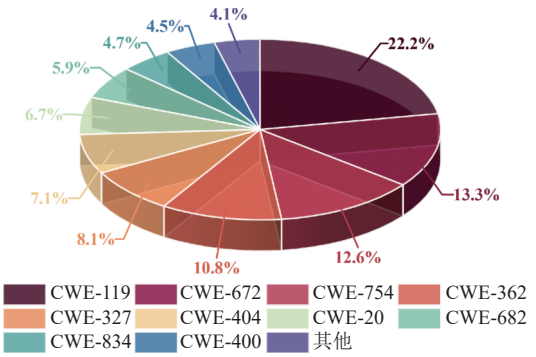


图 5 类型归并后 Big-Vul 数据集中漏洞样本分布情况

表 3 本文模型训练涉及的超参数设置

超参数名称	参数值
Word2Vec维度	100
Node2Vec维度	100
One-Hot维度	69
损失函数	Categorical_CrossEntropy
优化算法	Adam
学习率	0.000 1
权重衰退	0.001
批量大小	32
训练轮数	100

4.3 基线方法

本文针对多类别漏洞检测相关研究展开调研, 发现只有 Zou 等人[12]提出的 μVulDeePecker 模型最为相近. 然而, 该方法目前尚未开源, 且复现效果与论文中报告差距较大. 因此, 为了有效地评估提出的 MulVD 方法的性能, 本文首先选择了 3 种传统的基于规则的漏洞检测工具进行比较, 包括 Flawfinder[17]、RATS[18]、Cppcheck[19]. 此外, 为了更好地评估本文方法是否比基于深度学习的同类型方法更具优势, 本文也选择了 5 种主流的基于深度学习的漏洞检测方法, 包括 VulDeePecker[6]、SySeVR[20]、Devign[21]、ReVeal[22]和 LineVul[23], 作为基线进行比较. 接下来依次简要介绍本文考虑的 8 种基线方法.

Flawfinder: 根据其内置的漏洞模式库扫描源代码, 通过基于语法的文本匹配, 生成潜在漏洞列表. 该工具支持缓冲区溢出、竞态条件、路径遍历等常见类型的漏洞.

RATS: 对源代码进行粗略分析并标记诸如缓冲区溢出、竞态条件等与安全相关的常见编程错误.

Cppcheck: 利用轻量级的数据流分析检测程序中的死指针、整数溢出、空指针解引用等未定义行为.

VulDeePecker: 基于目标代码中敏感 API 的数据流向进行程序切片, 利用双向 LSTM 等 RNN 模型来检测缓冲区错误 (CWE-119) 和资源管理错误 (CWE-399) 漏洞.

SySeVR: 在 VulDeePecker 的基础之上, 添加了数组使用、指针使用和整数使用 3 种切片准则丰富漏洞样本的多样性, 基于程序依赖图 PDG 执行前向和后向切片提取控制和数据流信息.

Devign: 首次提出将漏洞检测视为图分类任务. 该方法通过在抽象语法树 AST 的基础上添加控制流图 CFG、数据流图 DDG 和代码自然序列 NCS 显式刻画漏洞代码的控制流和数据流信息, 然后输入到带有卷积模块的门控图神经网络 GGNN 中学习漏洞特征.

ReVeal: 将代码表示学习与漏洞检测任务解耦, 利用 GGNN 从代码属性图 CPG 中学习良好的代码表示, 通过多层感知机 MLP 和三元损失优化分类器以提高下游漏洞检测任务的性能.

LineVul: 将经过预训练的 CodeBERT 模型编码的代码嵌入输入到标准 BERT 架构中训练漏洞检测模型, 并利用自注意力机制以定位漏洞语句.

4.4 评估指标

为了评估所提方法的性能, 本文分别选择了 4 种二分类指标, 包括准确率 ($Accuracy$)、召回率 ($Recall$)、精确度 ($Precision$) 以及 $F1$ 值 ($F1\ score$), 和 3 种多分类指标 ($Recall_{\text{macro}}$、$Precision_{\text{macro}}$ 和 $F1\ score_{\text{macro}}$). 这些指标被广泛用于基于深度学习的漏洞检测相关研究中[12,20–22].

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{8}$$

$$Recall = \frac{TP}{TP+FN} \tag{9}$$

$$Precision = \frac{TP}{TP+FP} \tag{10}$$

$$F1\ score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{11}$$

$$Recall_{\text{macro}}(M\_R) = \frac{1}{n}\sum_{i=0}^{n} Recall_i \tag{12}$$

$$Precision_{\text{macro}}(M\_P) = \frac{1}{n}\sum_{i=0}^{n} Precison_i \tag{13}$$

$$F1\ score_{\text{macro}}(M\_F1) = \frac{2 \times M\_R \times M\_P}{M\_R + M\_P} \tag{14}$$

其中, $i \in \{0, 1, \ldots, n\}$ 表示待测代码的候选标签集, 0 表示无漏洞, n 是漏洞类型的总数. $Recall_i$ 和 $Precision_i$ 分别代表检测模型在检测 i 类漏洞时的召回率 $Recall$ 和精确度 $Precision$. $Recall_{\text{macro}}$ 和 $Precision_{\text{macro}}$ 表示检测模型在所有漏洞类型上召回率和精确度的算术平均值. $F1\ score_{\text{macro}}$ 是 $Recall_{\text{macro}}$ 和 $Precision_{\text{macro}}$ 的调和平均值, 反映了检测模型在多种漏洞类型上的全局性能. $F1\ score_{\text{macro}}$ 越高, 说明模型在多分类场景下的检测效果越好.

5 实验结果与分析

5.1 RQ1: MulVD 在漏洞函数预测 (二分类) 任务上的性能是否优于基线方法?

为回答该研究问题, 本文分别将所提方法与 8 种基线方法在 Devign 和 ReVeal 两个基准数据集上进行比较. 对比基线包括 3 种传统的基于规则的漏洞检测工具 (Flawfinder[17]、RATS[18]和 Cppcheck[19]) 和 5 种主流的基于深度学习的漏洞检测方法 (VulDeePecker[6]、SySeVR[20]、Devign[21]、ReVeal[22]和 LineVul[23]). 针对基于深度学习的漏洞检测方法 (包括 MulVD 和 5 个基线方法), 将每个数据集按 8:1:1 的比例随机分割以进行模型训练、验证和评估, 并对 10 次独立的实验结果取平均值以避免偶然性.

表 4 给出了 MulVD 方法和选择的 8 种基线方法在 Devign 和 ReVeal 数据集上的实验结果, 对每种指标下的最优结果进行了加粗. 可以发现, 无论是在数据分布相对平衡的 Devign 数据集, 还是在与真实场景类似的分布不平衡的 ReVeal 数据集上, MulVD 的 4 项指标 (准确率、召回率、精确度和 $F1$ 值) 均优于基线方法, 这表明本文提出的 MulVD 方法可以有效地检测软件中的潜在漏洞.

表 4　本文所提方法和基线方法在漏洞函数预测任务上的性能比较 (%)

数据集	类型	方法	准确率	召回率	精确度	$F1$值
ReVeal	人工规则	Flawfinder	55.9	13.7	8.1	10.2
		RATS	58.2	8.3	6.9	7.5
		Cppcheck	60.4	11.5	14.8	12.9
	深度学习	VulDeePecker	87.2	16.8	20.1	18.3
		SySeVR	85.4	38.3	22.6	28.4
		Devign	87.8	31.9	27.8	29.7
		ReVeal	84.3	56.7	31.3	40.3
		LineVul	**90.6**	44.1	30.5	36.1
		MulVD	85.7	**63.6**	**34.0**	**44.3**
Devign	人工规则	Flawfinder	52.1	27.3	21.4	24.0
		RATS	53.5	14.4	12.8	13.5
		Cppcheck	52.4	23.3	28.6	25.7
	深度学习	VulDeePecker	51.8	34.6	45.9	39.5
		SySeVR	54.2	66.1	44.7	53.3
		Devign	57.6	64.3	51.8	57.4
		ReVeal	59.6	71.1	58.5	64.2
		LineVul	57.3	66.8	57.9	62.1
		MulVD	**63.8**	**75.9**	**58.8**	**66.3**

具体来说, 针对不同数据集的性能表现, 几乎所有方法在 Devign 数据集上的表现都更好. 特别是基于深度学习的方法, 可以观察到显著的性能提升. 例如: 与 ReVeal 数据集相比, VulDeePecker 在 Devign 数据集上的召回率、精确度和 *F*1 值分别提高了 2.06 倍、2.28 倍和 2.16 倍. 这种性能差异的主要原因是数据集中漏洞代码和非漏洞代码的比例存在差异. 在 ReVeal 数据集上, 这一比例为 1:9.9, 而在 Devign 数据集上, 这一比例为 1:1.2. 因此, 那些没有为数据不平衡进行额外处理的基于深度学习的方法 (例如我们的基线中的 VulDeePecker、SySeVR、Devign 和 LineVul) 的性能很大程度上取决于数据集的质量. 相反, ReVeal 和 MulVD 采用了过采样技术来平衡数据分布. 因此, 它们在不同数据集上的性能相对稳定. 此外, 由于 Devign 数据集中漏洞函数的数量 (10 067) 远超 ReVeal 数据集中漏洞函数的数量 (1 664), 我们还可以发现 MulVD 在 Devign 数据集上的性能更好.

此外, 我们观察到所有 3 个传统漏洞检测工具都具有较低的召回率或准确度. 以 Flawfinder 为例, 在 ReVeal 和 Devign 数据集上只能达到 10.2% 和 24.0% 的 *F*1 值. 这是合理的, 因为手动创建的漏洞模式或规则无法覆盖所有漏洞. 在实际项目中, 新出现的漏洞比这些简单和有限的规则要复杂得多. 因此, 许多由已知漏洞模式变体引起的漏洞被忽略了. 相比之下, 基于深度学习的方法在每个数据集上的表现都优于这些传统的漏洞检测工具. 其中, MulVD 在召回率 (63.6%/75.9%)、精确度 (34.0%/58.8%) 和 *F*1 值 (44.3%/66.3%) 等方面均取得了最佳表现. 具体来说, 与效果最好的基准方法 ReVeal 相比, 在 Devign 数据集上, MulVD 的召回率高出 6.8%. 这意味着 MulVD 能够更好地关注分类中的重要部分, 从而检测到更多的漏洞. 这种性能提升归功于我们方法中使用的注意力机制. 虽然 ReVeal 和 MulVD 都将函数的图表示作为输入, 并使用 GNN 来训练检测模型, 但 MulVD 通过使用结构化注意力过滤掉对分类没有帮助的图节点, 避免了其余函数图中的噪声信息. 另外, 尽管微调 CodeBERT 等 Transformer 架构在许多以代码为中心的软件工程任务上也展现出良好的性能, 我们发现当数据集规模较小时, 基于 Transformer 的漏洞检测方法与基于 GNN 的方法差异并不大. 如表 4 所示, 在 Devign 和 ReVeal 两个数据集上, 表现最佳的基于 GNN 的基线方法 ReVeal 在精确度上略优于流行的基于 Transformer 的漏洞检测方法 LineVul, 而在召回率方面, ReVeal 相较于 LineVul 则分别提升了 28.57% 和 6.44%.

综上, 在漏洞函数预测 (二分类) 场景下, 本文所提出的 MulVD 可以有效地挖掘漏洞代码和非漏洞代码间的特征差异, 比表现最优的基线方法 ReVeal 仍能在召回率方面提高 6.8%. 此外, 无论是在样本分布较为平衡的 Devign 数据集, 还是在分布较为不平衡的 ReVeal 数据集上, MulVD 的各项指标均优于基线方法, 证明了本方法在真实场景下的有效性.

5.2 RQ2: MulVD 在漏洞类型识别 (多分类) 任务上的性能是否优于基线方法?

为回答该研究问题, 本文将所提方法与 8 种基线方法在 Big-Vul 数据集上进行比较. 由于基于深度学习的基线方法均面向二分类 (有无漏洞) 检测场景, 本文采用为每一种漏洞类型训练一个针对性的检测模型的方式来评估基线方法在漏洞类型识别任务上的性能. 特别的, 考虑到部分漏洞类型的样本过少 (例如: Big-Vul 数据集中仅包含 6 个类型为 CWE-326 的漏洞样本), 不足以训练一个有效的深度学习模型, 本文基于数据集中经过漏洞类型归并后得到的 Top-10 漏洞类型 (如图 4 所示, 包括 CWE-{119, 672, 754, 362, 327, 404, 20, 682, 834, 400}), 为每个基线方法构建相应的面向特定漏洞类型的检测模型, 并按 8:1:1 的比例随机分割数据集以进行模型训练、验证和评估. 其中, 包含目标类型的漏洞函数作为正样本 (标记为“1”), 其余函数 (包括非漏洞函数和非目标类型的漏洞函数) 作为负样本 (标记为“0”). 对于 MulVD, 本文则直接在整个 Big-Vul 数据集上进行训练 (80%)、验证 (10%) 和评估 (10%). 每组实验进行 10 次取平均值以避免偶然性.

表 5 展示了 MulVD 与基线方法在 Big-Vul 数据集上针对 Top-10 漏洞类型的实验结果. 总体而言, 在各类评估指标上, 本文所提 MulVD 在不同类型漏洞检测的平均性能仍优于所有基线. 平均而言, *M_R* 为 51.6%, *M_P* 为 40.7%, *M_F*1 值达到了 45.5%. 具体来说, 与效果最好的基准方法 LineVul 相比, MulVD 的 *M_R* 提高了 11.69%, *M_P* 提高了 2.01%, *M_F*1 提高了 6.31%, 说明了本文方法在漏洞类型识别上的有效性.

为了更加深入地分析每个方法在具体漏洞类型上的有效性, 图 6 报告了每个方法在 Big-Vul 数据集中占比前 10 的漏洞类型检测上的 *F*1 值. 总体而言, 在检测不同类型的漏洞时, 性能差距巨大. 对于传统的漏洞检测工具, 虽

然它们在某些特定类型的漏洞 (如 CWE-682 等) 上取得了与基于深度学习的方法类似的结果, 但它们的性能仍然不尽人意. 对于基于深度学习的方法, 我们观察到 MulVD 在大多数漏洞类型上仍然是表现最佳的方法. 以 CWE-119 漏洞为例, MulVD 和 5 个基于深度学习的基线方法均获得了最佳的 *F*1 分数. 这一结果的主要原因是 CWE-119 是一种常见的漏洞类型. 与其他类型的漏洞相比, 其漏洞模式更容易被深度学习模型学习到. 此外, 我们观察到所有基线方法在检测 CWE-834 漏洞上表现不佳. 对于传统的漏洞检测工具, 它们几乎无法检测到这种漏洞, 因为这些工具是基于静态分析的. 然而, 为了识别源代码中的不可到达退出条件, 需要对目标代码进行动态分析. 相比之下, 基于深度学习的方法可以通过从漏洞样本中学习隐式漏洞模式来部分检测 CWE-834 漏洞. 特别地. 我们观察到 MulVD 相对于其他基于深度学习的基线方法有显著的性能提升. 这种差异的原因可能是由于模型训练数据不足造成的过度拟合问题. 由于在多类场景中使用了少数样本合成技术, MulVD 可以产生高质量的合成实例, 但不会导致严重的过度泛化.

表 5　本文所提方法和基线方法在 Top-10 漏洞类型识别任务上的性能比较 (%)

类型	方法	*M_R*	*M_P*	*M_F*1
人工规则	Flawfinder	12.5	9.1	10.5
	RATS	8.2	4.7	6.0
	Cppcheck	21.8	15.6	18.2
深度学习	VulDeePecker	30.6	33.5	31.9
	SySeVR	38.4	35.9	37.1
	Devign	41.3	36.7	38.9
	ReVeal	43.6	37.2	40.1
	LineVul	46.2	39.9	42.8

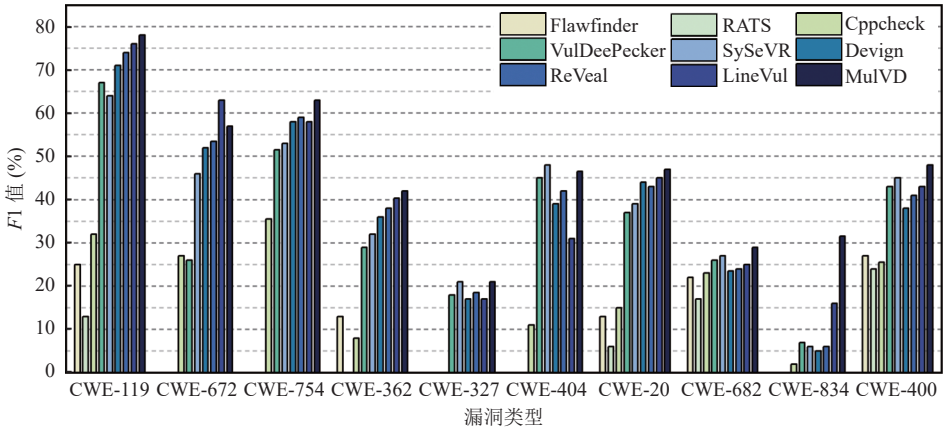


图 6　不同漏洞检测方法对 Top-10 漏洞类型的 *F*1 值

为了进一步说明 MulVD 在漏洞类型识别方面的优势, 本文结合 Linux 项目中一个仅能由本文方法检测出的真实漏洞实例 (CVE-2019-19079[45]) 进行详细阐述. 如图 7 所示, 已分配的指针变量 kbuf 在 If 语句执行完后 (第 8 行和第 12 行) 未被及时释放, 导致内存泄露漏洞 (CWE-401, missing release of memory after effective lifetime). 攻击者可以通过此漏洞导致拒绝服务. 对于基于 RNN/LSTM 的方法 (即 VulDeePecker 和 SySeVR), 它们将此漏洞函数及其相应的补丁函数都错误地识别为了非漏洞函数. 导致该误报的主要原因是 LSTM 以及其他序列模型并不适合对程序的结构化控制流和数据流进行建模. 此外, 尽管基于图神经网络 GNN 的基线方法 (即 Devign 和 ReVeal) 利用 GNN 模型从多个抽象级别的代码图表示中学习到了漏洞程序丰富的结构语义, 并成功识别出了该漏洞函数. 但与此同时, 其相应的补丁函数也被错误地识别为了漏洞函数. 这主要是由于粗粒度 (函数级或切片级) 的漏洞特征表示在涵盖丰富的程序信息的同时, 也不可避免地引入了大量的噪声信息, 导致深度神经网络模型无法从中准

确地推理细粒度的特征差异. 相比之下, 本文使用结构引导的图神经网络来挖掘与漏洞最相关的局部典型子图以更好地区分不同代码段的语义信息. 因此, 在实际场景中, 本文所提方法更能够捕获有益于漏洞检测的语义信息.

```
1     static ssize_t qrtr_tun_write_iter (struct kiocb *iocb,
                                          struct iov_iter *from)
2     {
3         kbuf = kzalloc (len, GFP_KERN EL);
4         if    (!kbuf)
5                 return -ENO MEM;
6   −     if    (!copy_from_iter_full (kbuf, len, from))
7   +     if    (!copy_from_iter_full (kbuf, len, from)){
8   +             kfree (kbuf);
9                 return-EFAULT;
10  +     }
11        ret = qrtr_endpoint_post (&tun->ep, kbuf, len);
12        kfree (kbuf);
13        return ret < 0 ? ret : len;
14    }
```

图 7　仅由 MulVD 检测出的真实漏洞实例分析

综上, 在漏洞类型识别 (多分类) 场景下, 本文所提出的 MulVD 可以有效地挖掘不同类型漏洞代码间的特征差异, 比表现最优的基线方法 LineVul 在 *M_R*、*M_P* 和 *M_F*1 值方面分别提高了 11.69%、6.01% 和 6.31%. 在不同类型漏洞样本分布高度不平衡的 Big-Vul 数据集上的实验表明, MulVD 的各项指标均优于基线方法, 证明了本方法在真实场景下的有效性.

5.3　RQ3: 本文提出的结构感知图神经网络 SA-GNN 能否提高漏洞类型识别的性能?

为回答该研究问题, 本文为 MulVD 构建了两个基于不同组件 (图注意力机制和基于过采样的样本平衡) 的变体, 并在包含 36 种漏洞类型的整个 Big-Vul 数据集上进行训练 (80%)、验证 (10%) 和评估 (10%). 每组实验进行 10 次取平均值以避免偶然性.

实验结果如表 6 所示. 总体上看, 本文方法在 *M_R* (51.6%)、*M_P* (40.7%) 和 *M_F*1 (45.5%) 方面优于所有不同组件的 MulVD 变体. 特别的, 通过添加基于 SMOM 的类别平衡模块后, 本文方法的 *M_R*、*M_P* 和 *M_F*1 分别提高了 60.25%、38.44% 和 48.21%, 性能提升显著.

表 6　SA-GNN 不同组件对 MulVD 性能的影响 (%)

模型	*M_R*	*M_P*	*M_F*1
SA-GNN w/o Oversampling	32.2	29.4	30.7
SA-GNN w/o Attention	38.6	31.8	34.9
SA-GNN	**51.6**	**40.7**	**45.5**

具体而言, 为了研究 SA-GNN 中类别平衡模块在漏洞类型识别中的影响, 本文通过在分类之前不重新平衡漏洞分布以构建 MulVD 变体. 如表 6 所示, 我们可以看到过采样层对提高 MulVD 的召回率和精确度都有很大帮助. 这意味着过采样技术的使用可以有效缓解多类别不平衡问题, 因为原来被错误标记的部分少数类样本现在可以被正确识别.

为了研究基于图的注意力机制的效果, 我们还构建了一个 MulVD 变体, 利用广泛使用的 GGNN 模型来提取漏洞模式. 如表 6 所示, 我们发现添加注意力机制可以有效地提高召回率. 原因是使用注意力机制使检测模型更加关注“感兴趣”的部分, 即帮助检测模型区分程序中不同漏洞类型的特征信息.

综上, 本文所构建的结构感知图神经网络可以自适应地从不同类型漏洞的局部判别子图中捕获差异化的典型特征, 并在不引入噪声的情况下生成最优样本以平衡漏洞分布差异, 提高多类别漏洞检测的性能.

6　总　结

本文提出了一种基于图的多类别漏洞检测方法 MulVD, 在检测代码是否包含漏洞的同时, 识别具体的漏洞类型. 本方法利用代码属性图来覆盖不同类型的漏洞特征, 并提出了一种结构感知的图神经网络以自适应地挖掘不

同类型漏洞的局部判别子图, 并在不引入噪声的情况下动态平衡数据分布以增强检测模型的泛化能力. 实验结果表明, MulVD 在二分类和多分类漏洞检测方面优于现有的最先进方法.

但是, 目前的工作仍具有一定的局限性.

一方面, 本文方法的实施与验证均基于 Big-Vul 漏洞数据集. 但在数据集统计分析的过程中, 我们也发现 Top-10 漏洞类型涵盖了绝大部分 (95.9%) 的漏洞数据. 这种极端的长尾分布态势使得我们通过为少数类漏洞类型合成样本的方式尽管有效, 但对于检测 CWE-326 等样本数极为稀缺的漏洞类型仍不友好. 因此, 未来我们会构建一个规模更大、更加多样的漏洞数据集以评估本文方法的有效性.

另一方面, 本文使用基于图的注意力机制从漏洞代码的属性图 CPG 中捕获与漏洞类型相关的局部判别子图, 但在真实漏洞场景中, 构建的 CPG 可能规模庞大 (包含数百个节点)、结构复杂 (漏洞相关节点距离较远), 导致检测和分类性能不佳. 因此, 受图简化[46]在提升现有基于图的漏洞检测方法性能方面的启发, 未来我们拟设计一种更为有效的代码表示形式以进一步提升方法性能.

在未来工作中, 我们将继续提升漏洞检测模型的检测效果和分类效果, 更好地辅助安全专家进行漏洞挖掘工作.

References:

[1] Liu J, Su PR, Yang M, He L, Zhang Y, Zhu XY, Lin HM. Software and cyber security—A survey. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 42–68 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[2] Li GW, Yuan T, Li L. Study of state-of-the-art open-source C/C++ static analysis tools. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2061–2081 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6569.htm [doi: 10.13328/j.cnki.jos.006569]

[3] Deng X, Ye W, Xie R, Zhang SK. Survey of source code bug detection based on deep learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 625–654 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/j.cnki.jos.006696]

[4] Gu MX, Sun HY, Han D, Yang S, Cao WY, Guo Z, Cao CJ, Wang WJ, Zhang YQ. Software security vulnerability mining based on deep learning. Journal of Computer Research and Development, 2021, 58(10): 2140–2162 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2021.20210620]

[5] Duan X, Wu JZ, Luo TY, Yang MT, Wu YJ. Vulnerability mining method based on code property graph and attention BiLSTM. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3404–3420 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

[6] Li Z, Zou DQ, Xu SH, Ou XY, Jin H, Wang SJ, Deng ZJ, Zhong YY. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2018. [doi: 10.14722/ndss.2018.23158]

[7] Cao SC, Sun XB, Bo LL, Wei Y, Li B. BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. Information and Software Technology, 2021, 136: 106576. [doi: 10.1016/j.infsof.2021.106576]

[8] Cheng X, Wang HY, Hua JY, Xu GA, Sui YL. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. on Software Engineering and Methodology, 2021, 30(3): 38. [doi: 10.1145/3436877]

[9] Wang HT, Ye GX, Tang ZY, Tan SH, Huang SF, Fang DY, Feng YS, Bian LZ, Wang Z. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. on Information Forensics and Security, 2021, 16: 1943–1958. [doi: 10.1109/TIFS.2020.3044773]

[10] Cao SC, Sun XB, Bo LL, Wu RX, Li B, Tao CQ. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 1456–1468. [doi: 10.1145/3510003.3510219]

[11] Zheng W, Gao JL, Wu XX, Liu FY, Xun YX, Liu GL, Chen X. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. Journal of Systems and Software, 2020, 168: 110659. [doi: 10.1016/j.jss.2020.110659]

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

[13] Liu BC, Meng GZ, Zou W, Gong Q, Li F, Lin M, Sun DD, Huo W, Zhang C. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In: Proc. of the 42nd Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1547–1559. [doi: 10.1145/3377811.3380923]

[14] Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of the 35th IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2014. 590–604. [doi: 10.1109/SP.2014.44]

[15] Mikolov T, Sutskever I, Chen K, Corrado G, Dean J. Distributed representations of words and phrases and their compositionality. In:

Proc. of the 27th Int'l Conf. on Neural Information Processing Systems. Lake Tahoe: ACM, 2013. 3111–3119.
[16] Grover A, Leskovec J. Node2vec: Scalable feature learning for networks. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Francisco: ACM, 2016. 855–864. [doi: 10.1145/2939672.2939754]
[17] Flawfinder. 2023. http://www.dwheeler.com/flawfinder/
[18] Rough-auditing-tool-for-security. 2023. https://code.google.com/archive/p/rough-auditing-tool-for-security/
[19] Cppcheck. 2023. http://cppcheck.net/
[20] Li Z, Zou DQ, Xu SH, Jin H, Zhu YW, Chen ZX. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. on Dependable and Secure Computing, 2022, 19(4): 2244–2258. [doi: 10.1109/TDSC.2021.3051525]
[21] Zhou YQ, Liu SQ, Siow JK, Du XN, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: NeurIPS, 2019. 915.
[22] Chakraborty S, Krishna R, Ding YRB, Ray B. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. on Software Engineering, 2022, 48(9): 3280–3296. [doi: 10.1109/TSE.2021.3087402]
[23] Fu M, Tantithamthavorn C. LineVul: A Transformer-based line-level vulnerability prediction. In: Proc. of the 19th Int'l Conf. on Mining Software Repositories. Pittsburgh: ACM, 2022. 608–620. [doi: 10.1145/3524842.3528452]
[24] Cao SC, Sun XB, Wu XX, Lo D, Bo LL, Li B, Liu W. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 155. [doi: 10.1145/3597503.3639168]
[25] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaisaer Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: NeurIPS, 2017. 6000–6010.
[26] Dam HK, Tran T, Pham T, Ng SW, Grundy J, Ghose A. Automatic feature learning for predicting vulnerable software components. IEEE Trans. on Software Engineering, 2021, 47(1): 67–85. [doi: 10.1109/TSE.2018.2881961]
[27] Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications. Orlando: IEEE, 2018. 757–762. [doi: 10.1109/ICMLA.2018.00120]
[28] Cai J, Li B, Zhang T, Zhang JL, Sun XB. Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction. Journal of Systems and Software, 2024, 209: 111919. [doi: 10.1016/j.jss.2023.111919]
[29] Kipf TN, Welling M. Semi-supervised classification with graph convolutional networks. In: Proc. of the 5th Int'l Conf. on Learning Representations. Toulon: OpenReview.net, 2017.
[30] Li YJ, Tarlow D, Brockschmidt M, Zemel RS. Gated graph sequence neural networks. In: Proc. of the 4th Int'l Conf. on Learning Representations. San Juan: OpenReview.net, 2016.
[31] Velickovic P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: OpenReview.net, 2018.
[32] Liu SG, Lin GJ, Han QL, Wen S, Zhang J, Xiang Y. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans. on Fuzzy Systems, 2020, 28(7): 1329–1343. [doi: 10.1109/TFUZZ.2019.2958558]
[33] Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP. SMOTE: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 2002, 16: 321–357. [doi: 10.1613/jair.953]
[34] Tantithamthavorn C, Hassan AE, Matsumoto K. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. on Software Engineering, 2020, 46(11): 1200–1219. [doi: 10.1109/TSE.2018.2876537]
[35] Wu XX, Zheng W, Chen X, Zhao Y, Yu TT, Mu DJ. Improving high-impact bug report prediction with combination of interactive machine learning and active learning. Information and Software Technology, 2021, 133: 106530. [doi: 10.1016/j.infsof.2021.106530]
[36] Yang X, Wang SW, Li Y, Wang SH. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays! In: Proc. of the 45th Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2287–2298. [doi: 10.1109/ICSE48619.2023.00192]
[37] Zhu TF, Lin YP, Liu YH. Synthetic minority oversampling technique for multiclass imbalance problems. Pattern Recognition, 2017, 72: 327–340. [doi: 10.1016/j.patcog.2017.07.024]
[38] Zhang JL, Sui H, Sun XB, Ge CP, Zhou L, Susilo W. GrabPhisher: Phishing scams detection in Ethereum via temporally evolving GNNs. IEEE Trans. on Services Computing, 2024, 17(6): 3727–3741. [doi: 10.1109/TSC.2024.3411449]
[39] Lee JB, Rossi R, Kong XN. Graph classification using structural attention. In: Proc. of the 24th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining. London: ACM, 2018. 1666–1674. [doi: 10.1145/3219819.3219980]
[40] Campello RJGB, Moulavi D, Sander J. Density-based clustering based on hierarchical density estimates. In: Proc. of the 17th Pacific-Asia Conf. on Knowledge Discovery and Data Mining. Gold Coast: Springer, 2013. 160–172. [doi: 10.1007/978-3-642-37456-2_14]
[41] Fan JH, Li Y, Wang SH, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul: ACM, 2020. 508–512. [doi: 10.1145/3379597.3387501]
[42] PyTorch. https://pytorch.org/
[43] Tree-sitter. 2023. https://github.com/tree-sitter/

[44] Deep graph library (DGL). 2023. https://github.com/dmlc/dgl/
[45] CVE-2019-19079. 2023. https://www.cve.org/CVERecord?id=CVE-2019-19079
[46] Wen XC, Chen YP, Gao CY, Zhang HY, Zhang JM, Liao Q. Vulnerability detection with graph simplification and enhanced graph representation learning. In: Proc. of the 45th Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2275–2286. [doi: 10.1109/ICSE48619.2023.00191]

附中文参考文献:

[1] 刘剑, 苏璞睿, 杨珉, 和亮, 张源, 朱雪阳, 林惠民. 软件与网络安全研究综述. 软件学报, 2018, 29(1): 42–68. http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]
[2] 李广威, 袁挺, 李炼. 开源 C/C++静态软件缺陷检测工具实证研究. 软件学报, 2022, 33(6): 2061–2081. http://www.jos.org.cn/1000-9825/6569.htm [doi: 10.13328/j.cnki.jos.006569]
[3] 邓枭, 叶蔚, 谢睿, 张世琨. 基于深度学习的源代码缺陷检测研究综述. 软件学报, 2023, 34(2): 625–654. http://www.jos.org.cn/1000-9825/6696.htm [doi: 10.13328/j.cnki.jos.006696]
[4] 顾绵雪, 孙鸿宇, 韩丹, 杨粟, 曹婉莹, 郭祯, 曹春杰, 王文杰, 张玉清. 基于深度学习的软件安全漏洞挖掘. 计算机研究与发展, 2021, 58(10): 2140–2162. [doi: 10.7544/issn1000-1239.2021.20210620]
[5] 段旭, 吴敬征, 罗天悦, 杨牧天, 武延军. 基于代码属性图及注意力双向 LSTM 的漏洞挖掘方法. 软件学报, 2020, 31(11): 3404–3420. http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

曹思聪(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为智能化软件工程, 软件安全.

陈厅(1987－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件安全, 联邦学习与聚合安全, 区块链, 网络安全.

孙小兵(1985－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为智能化软件数据分析, 软件安全.

罗夏朴(1977－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为移动安全和隐私, 区块链/智能合约, 网络安全和隐私, 软件工程.

薄莉莉(1989－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件安全, 软件测试.

张涛(1982－), 男, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为开源软件数据智能化分析, 软件安全.

吴潇雪(1983－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为软件漏洞分析与检测, 软件测试.

刘维(1982－), 女, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为复杂网络, 机器学习, 数据挖掘.

李斌(1965－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为知识图谱, 软件数据挖掘.