计算机研究与发展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(9): 2348−2361, 2025

LHG-VD: 一种可学习的层次化图表示漏洞检测方法

胡 菘[1] 罗嘉驰[2] 万文凯[2] 闫 阳[1] 郭 帆[2] 曲彦文[2]
[1](江西师范大学数字产业学院 江西上饶 334006)
[2](江西师范大学计算机信息工程学院 南昌 330022)
(songhu@jxnu.edu.cn)

LHG-VD: A Learnable Hierarchical Graph Representation Vulnerability Detection Method

Hu Song[1], Luo Jiachi[2], Wan Wenkai[2], Yan Yang[1], Guo Fan[2], and Qu Yanwen[2]
[1](*School of Digital Industry*, *Jiangxi Normal University*, *Shangrao*, *Jiangxi* 334006)
[2](*School of Computer Information Engineering*, *Jiangxi Normal University*, *Nanchang* 330022)

Abstract Software vulnerabilities pose a serious threat to the safe and stable operation of computer systems and software, so the research related to their automatic detection has been receiving extensive attention. Unlike traditional static vulnerability detection tools that use predefined rules provided by human experts to analyze the code, graph neural network (GNN)-based vulnerability detection methods have surpassed the traditional methods in some datasets by automatically learning the vulnerable code patterns. However, in the current GNN-based vulnerability detection methods, the design of GNN model is not combined with the characteristics of the code itself, which leads to poor detection effect on the real vulnerability code dataset. In this paper, we propose a learnable hierarchical graph representation vulnerability detection method LHG-VD, which is characterized by proposing a learnable readout function for the limitation of the traditional readout function, and designing a cross-granularity loss function based on the idea of comparative learning for the problem of maintaining the local structural information of the code in the process of graph pooling. Experimental results on real vulnerability datasets show that the *F*1 value of LHG-VD is 71.5%, which is improved by 4.9% compared with DeepWukong, a slice-level detection method, and 8.9% compared with AMPLE, a function-level detection method.

Key words vulnerability detection; deep learning; graph neural network; comparative learning; graph pooling

摘 要 软件漏洞严重威胁了计算机系统和软件的安全稳定运行,因此针对其自动检测的相关研究一直受到广泛关注.与传统静态漏洞检测工具采用人类专家提供的预定义规则进行代码分析不同,基于图神经网络(GNN)的漏洞检测方法通过自动学习易受攻击的代码模式,在一些数据集上的检测效果已经超越传统方法.然而,目前基于GNN的漏洞检测方法中,由于未结合代码自身特点对GNN模型进行设计,导致在真实漏洞代码数据集上检测效果较差.提出一种可学习的层次化图表示的漏洞检测方法LHG-VD,其特点是针对传统读出函数的局限提出一种可学习的读出函数,针对图池化过程中的代码局部结构信息保持问题设计了一种基于对比学习思想的跨粒度损失函数.在真实漏洞数据集的实验结果表明,LHG-VD的*F*1值为71.5%,与切片级检测方法DeepWukong相比提升4.9个百分点,与函数级检测方法AMPLE相比提升8.9个百分点.

收稿日期: 2024-03-12; **修回日期**: 2025-05-21
基金项目: 国家自然科学基金项目(61967011)
This work was supported by the National Natural Science Foundation of China (61967011).
通信作者: 郭帆 (fguo@jxnu.edu.cn)

关键词 漏洞检测；深度学习；图神经网络；对比学习；图池化

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440171 **CSTR:** 32373.14.issn1000-1239.202440171

软件源代码安全是软件安全的重要研究领域，其核心在于对软件源代码进行缺陷检测，漏洞作为一种可为攻击者利用的代码缺陷是检测的重点. 在软件开发的过程中，由于人工测试和传统静态分析工具存在局限，很难完全排除软件中的漏洞. 传统漏洞检测方法[1-4]主要采用人类专家提供的预定义规则进行代码分析，劳动强度大且不准确. 最近，各种基于深度学习（deep learning, DL）的漏洞检测方法被提出[5-22]，这些方法通过自动学习易受攻击的代码模式，在一些人工漏洞代码和真实代码数据集上取得了不错的检测效果. 基于深度学习中的自然语言处理（natural language processing, NLP）和图深度学习技术，将程序代码表示成文本序列，利用序列模型进行特征学习[5-10]，或是通过静态分析提取代码中的图结构信息，从而利用图神经网络（graph neural network, GNN）进行特征学习[11-21]. 基于图的表征方式能够对代码的语法信息、语义信息、控制流和数据流等信息进行深层次的表示，相比基于文本表征的方法，在检测效果上更好.

现有基于 GNN 的漏洞检测方法在设计 GNN 模型时，往往直接使用图学习领域的通用方法，并未针对代码自身特点进行设计，具体有 2 方面的问题：

1）真实漏洞代码往往调用链长且复杂，导致代码表征图规模更大、结构更加复杂. 现有基于 GNN 的切片级漏洞检测方法虽在人工漏洞数据集上表现不错，但在真实漏洞数据集上的表现仍不理想.

2）程序代码可以分为多个代码块（如 if-else 块和循环块），每个代码块可以单独视为一个逻辑单元. 在 GNN 模型的处理过程中，代码块被映射为代码表征图的局部结构信息. 图池化过程涉及节点和边的整合和删减，引起图拓扑结构的变动，导致代码表征图的局部结构信息会丢失.

针对上述问题，本文提出了一种可学习的层次化图表示漏洞检测方法 LHG-VD. LHG-VD 属于切片级漏洞检测方法，采用层次化设计的 GNN 模型，能够有效捕捉不同粒度的代码表征图信息. LHG-VD 设计了一个可学习的读出函数来处理不同粒度表征图，得到全图向量表示，其对不同规模代码表征图有自适应性. LHG-VD 设计了一个基于对比学习思想的跨粒度损失函数，能够在图池化过程中保持池化前的细粒度代码表征图中的局部结构信息.

本文将 LHG-VD 与 6 种先进的漏洞检测方法进行了实验对比，包括基于规则的检测方法 Flawfinder[23]、2 种基于文本序列的方法（LineVul[9] 和 CodeBERT[24]）和 3 种基于 GNN 的漏洞检测方法（ReGVD[16], AMPLE[22], DeepWukong[13]）. 在真实漏洞代码数据集上的实验结果表明，LHG-VD 在 $F1$ 得分、召回率、精确率和准确率方面超过了所有的漏洞检测方法.

本文的主要贡献有 3 个方面：

1）从真实漏洞代码的特点出发，针对传统图读出方法受限于固定的计算规则，无法很好适应不同粒度代码表征图中节点层次差异的问题，引入 NLP 中 CLS 的设计思想并结合全局注意力机制提出了一种可学习的图读出（learnable graph readout, LGR）函数.

2）针对在图池化过程中由于图规模的缩减导致细粒度图中局部结构信息可能丢失的问题，设计了一个跨粒度的对比学习损失（cross-granularity contrastive loss, CGCL）函数，能够在池化后的粗粒度图中保持原始细粒度图中的局部结构信息.

3）提出一种可学习的层次化图表示漏洞检测方法 LHG-VD，在真实漏洞代码数据集上的实验结果表明，LHG-VD 的性能优于现有的 6 种检测方法.

1 相关工作

1.1 基于深度学习的漏洞检测方法

基于深度学习的源码漏洞检测方法的核心思想是借助深度神经网络强大的特征学习能力来学习隐含漏洞模式，从而实现自动检测的目的. 源代码的表征形式主要分为文本序列表示和图表示.

将代码表示为文本序列时，主要采用 NLP 中的处理方式，如 VulDeePecker[7] 使用 Word2Vec[25] 技术将输入的代码片段转化为向量表示后，利用双向长短期记忆网络（Bi-LSTM）进行训练. LineVul[9] 引入了预训练编程语言（programming language, PL）模型 CodeBERT[24] 来实现语义嵌入提取的步骤，没有使用传统的 RNN 模型来生成代码表示，而是利用带有自注意力层的 BERT 架构[26]，这些自注意力层能够使用

点积运算捕获长序列中的长期依赖关系. 基于文本序列的表征方式虽然表达直观、处理相对简单，但是无法利用程序代码中丰富的结构信息.

基于代码图表征的检测方法能够充分表示从源代码中抽取的结构信息和语义信息，首先通过数据预处理将代码转化为图表征形式，然后利用 GNN 来学习代码中隐含的漏洞模式，最后对代码表征图进行分类操作. 例如 Devign[12] 使用图注意力神经网络[27](gated graph neural network, GGNN)对节点信息进行汇聚后执行图级别的分类任务，将待检测代码分为有缺陷和无缺陷. AMPLE[22] 使用 Devign 代码表征图，在 GNN 中添加用于捕获远处图节点之间关系的核缩放表示模块，涉及 2 个不同尺度的卷积核，分别关注远距离节点和邻近节点之间的关系. ReGVD[16] 从代码文本的 token 序列中，根据 Unique token-focused 和 Index-focused 自定义策略来构建代码表征图，并使用 CoderBERT 的 token 嵌入层来初始化节点特征向量. ReGVD 以图卷积网络[28](graph convolutional networks, GCN)和 GGNN 为基础模型，通过残差连接(residual connection)来学习代码表征图特征，读出函数通过拼接 sum pooling 和 max pooling 的结果实现. DeepWukong[13] 在 GNN 设计中引入层次化的 *K*-GNN(*K*-dimensional GNN)，该结构由多个相似的模块构成，每个模块都包括图卷积层、池化层和读出层. *K*-GNN 旨在通过整合不同粒度的图信息，从而获得更为全面的图表征服务于分类任务.

1.2 基于 GNN 的图分类

基于 GNN 的图分类方法，首先对节点特征和图结构进行卷积操作得到特征映射后的节点表示，然后进行多次池化操作，缩小图的规模，最终得到全图表示用于后续分类[29].

Ying 等人[30] 提出可微分池化分类模型 DiffPool，GNN 在每次卷积操作后会通过学习一个软分配矩阵将节点聚合到不同的簇中，逐步缩小图的规模. 该聚类过程，计算开销大且面临细节信息丢失问题. Xu 等人[31] 引入跳跃知识(jumping knowledge)机制并提出 JK-Nets(jumping knowledge networks)，可与不同类型的 GNN 层(如 GCN、GraphSAGE 等)结合使用，能够聚合来自不同 GNN 层的节点表示，允许模型在学习时同时考虑局部和全局的图结构信息. Cangea 等人[32] 延续 JK-Nets 的设计，在读出每个 GNN 层的信息时，拼接多个全局池化操作的结果. Morris 等人[33] 受到 *K*-WL(Weisfeiler-Lehman)算法的启发，提出了 *K*-GNN 模型，通过考虑多个尺度来改进对高阶图结构的处理，但在高阶图结构的读出上不够灵活. Wu 等人[34] 的图分类模型设计了全图自注意力来学习图中的长距离依赖关系. 它将额外的可学习节点嵌入〈CLS〉添加到节点嵌入序列中，并从 Transformer 的输出中获取第 1 个节点嵌入作为图表示. 本文的图分类方法，从 *K*-GNN 出发，引入 CLS 设计思路，改善读出函数的自适应性，同时保留了 JK-Nets 风格的设计，结合不同尺度的图信息，获得信息更丰富的全图表示.

2 预备知识

GNN 是一个关于图的特征的学习过程[35]. 对于节点分类任务，GNN 模型旨在学习每个节点的代表性特征，这些特征有助于该类任务的后续处理. 对于图分类任务，GNN 模型的目标是学习整个图的代表性特征，学习节点特征通常是一个中间步骤，通过图卷积层实现.

GCN 是一种常用的图卷积操作，针对目标节点的邻居节点特征以及节点之间的边的权重进行加权融合，将邻居节点的信息传播到目标节点，式(1)说明了逐层传播的规则.

$$\boldsymbol{H}^{(l+1)}=\sigma\left(\boldsymbol{D}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \tag{1}$$

其中 $\tilde{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}_N$ 是增加了自连接的无向图 G 的邻接矩阵，$\boldsymbol{I}_N$ 是一个单位矩阵，$\tilde{\boldsymbol{D}}_{ii}=\sum_j \tilde{\boldsymbol{A}}_{i,j}$ 表示度矩阵，$\boldsymbol{W}^{(l)}$ 是一个特定层可训练的权重矩阵，$\sigma(\cdot)$表示激活函数如 ReLU. $\boldsymbol{H}^{(l)}$表示第 l层的激活矩阵，$\boldsymbol{H}^{(0)}=\boldsymbol{X}$，即为输入节点的特征矩阵.

在基于 GNN 的图分类模型中，图池化层[33] 通常插入在连续的卷积层之间，作用是降低特征维度、增强计算效率并扩大感受野. 引入池化层的目的是降低节点数量，减小图的规模，同时尽量保留最重要的信息. 在图池化策略中，Top-*k* 池化选择最具代表性的 k 个节点组成下一层的图. Top-*k* 池化的核心是评分机制，它为每个节点 v 计算一个分数 s_v，s_v 通过可学习的参数化函数 g 来计算. 以单层神经网络为例，函数 g 可以表示为

$$s_v=g(\boldsymbol{x}_v)=\boldsymbol{w}^{\mathrm{T}}\cdot\boldsymbol{x}_v+b, \tag{2}$$

其中 $\boldsymbol{x}_v$ 是节点 v 的特征向量，$\boldsymbol{w}$ 是权重向量，b 是偏置项. 通过调整 $\boldsymbol{w}$ 和 b，网络可以学习最能代表节点重要性的节点特征组合.

在计算出所有节点的分数后，Top-*k* 池化选择分数最高的 k 个节点或分数高于某个阈值 τ 的节点，构

成新的图层.

图读出层也称为图汇聚层或图全局汇聚层, 是GNN 的重要组成部分, 主要作用是将图的节点特征聚合成全图级别的表示. 图读出层会从图中提取一个固定维度的特征向量, 以便进行图级别的预测任务, 例如图分类和图回归等.

层次化的图分类模型交替地执行卷积操作和池化操作, 将图逐步粗化. 每次池化操作会生成 1 张粗粒度的图, 通过图读出模块提取模型中若干层的图特征, 将这些特征融合后得到全图表示进行图分类. 图分类的目标是学习图和类别标签的映射关系, 并预测未知图的类别标签.

本文研究基于图分类的源代码漏洞检测模型, 首先将非结构化的文本转换为有向图结构, 然后采用层次化的图分类模型对有向图进行分类, 实现漏洞检测.

3 研究动机

DeepWukong[13] 实验结果表明 *K*-GNN 在漏洞检测任务上的分类性能优于 GAT 和 GCN 等方法, 引起了我们的关注. *K*-GNN 使用的信息读出策略是一种 JK-Nets[31] 的 GNN, 核心思想是在网络的不同层之间直接连接, 从而允许信息在多个层次上“跳跃”. 这种“跳跃”策略允许模型在不同深度上可以捕获图的多尺度特性, 从而得到更丰富和更具代表性的图嵌入.

图 1 给出 CWE119 数据集的一个代码切片示例“invalid-memory-access. c”来展示 DeepWukong 的图分类过程. 每一个块由图卷积层、图池化层和图读出函数构成. 代码表征图经过不同块的处理, 可以得到不同粒度的图嵌入(graph embedding, GE), 融合这些 GE 可以得到全图的 GE. 图卷积操作通过传递目标节点的邻域节点信息到目标节点来更新目标节点特征信息, 图 1 中节点的大小表示信息的聚集程度. 图池化方法会利用评分函数来删除显著性分数(significance scores)相对较低的节点(图 1 中虚线部分节点对应被丢弃的节点), 逐步缩小图的规模, 从而产生不同粒度的图. 粒度不同是指图拓扑结构不同和节点的抽象级别不同, 相同节点在不同粒度的图中存在层级差异.

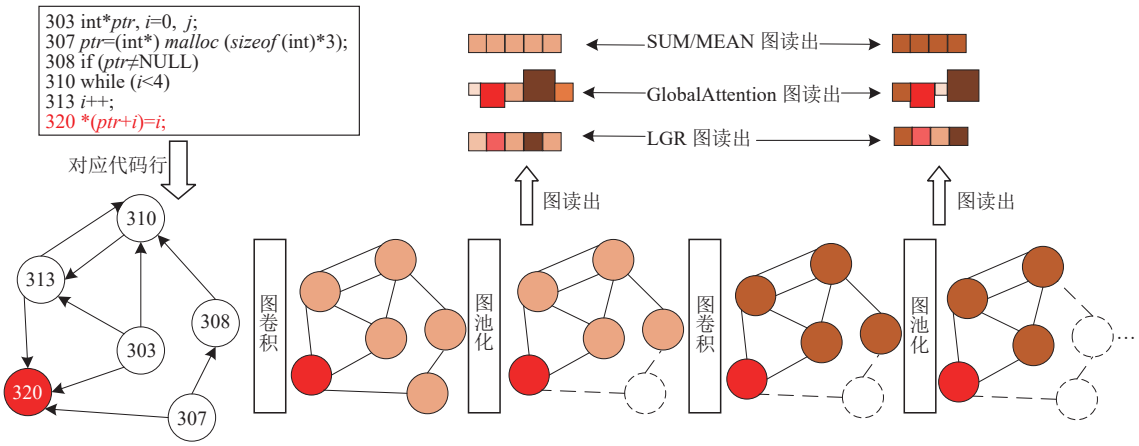


Fig. 1 Inspiring example

图 1 启发性例子

1)传统的读出策略如 SUM 和 MEAN, 尽管操作简洁, 但是均等地处理所有节点, 从而可能遗漏关键节点的重要信息(如图 1 中第 320 行漏洞语句对应的节点信息). 全局注意力方法(GlobalAttention)作为一种权重分配策略, 虽然为每个节点赋予了权重并进行加权平均, 但其固定的计算规则可能导致权重分布不均, 使得某些节点的信息被过度放大, 而其他节点的信息被边缘化. 因此, 为了充分捕获不同粒度图中节点的层次差异, 我们需要一个更为灵活和自适应的读出策略. LGR 图读出方式在结合全局注意力的基础上, 增加了自适应性, 因此可以有效应对不同粒度图中节点的层次差异. 图 1 中图读出部分, 展示了 3 种读出方式的差异, 方块的大小表示节点信息被过度放大或缩小, 颜色深浅表示节点信息汇聚时的重要程度.

2)信息丢失的固有问题也需要被关注. 程序代码可以分为多个代码块(例如, if-else 块、循环块), 每个代码块可以被单独视为一个逻辑单元, 在 GNN 模型的处理过程中, 代码块被映射为代码表征图的局部结构信息. 代码表征图经过不同层的池化后, 图规

模缩减, 原始细粒度图中重要的局部结构信息存在丢失风险. 如图 1 中漏洞行对应 320 号节点邻域的 307 号节点, 在切片代码中可以看出漏洞为数组越界访问造成, 307 号节点对应代码行为内存空间申请操作, 与漏洞产生有关联. 局部结构信息对于理解代码的语义至关重要. 因此我们需要一种策略在池化后的图中保持原图中局部结构信息. 保持局部结构信息, 可以通过提高池化后图中节点与原图中对应节点的相似度实现. 对比学习即通过正负样本对比, 最小化正样本对的损失, 这引导我们考虑对比学习.

4 LHG-VD 漏洞检测方法

图 2 展示了 LHG-VD 方法的工作流程. LHG-VD 的漏洞检测模型采用层次化图表示架构, 通过 LGR 自适应地读出不同层次的图嵌入, 设计 CGCL 来保持原图中的重要局部结构信息. 数据处理模块将源代码处理为代码表征图, 其先采用静态分析工具 Joern[36] 提取代码的程序依赖图(program dependence graph, PDG), 然后进行 PDG 的切片操作, 针对性地识别和分析潜在的敏感点, 即那些可能出现漏洞的代码行. 代码切片是从敏感点开始进行前向和后向节点遍历, 得到程序依赖图的子图. 敏感点包括系统 API 调用和涉及算术运算符(如“+, −, *, /”)的语句, 以及指针成员访问(PtrMemberAccess)和数组索引操作(ArrayIndex).

为了将代码切片转化为模型可利用的图表征, 需要进一步细化描述节点特征. LHG-VD 采用了一种通用的抽象名称替换策略来处理词汇爆炸问题, 即代码中可能出现无限多的标识符(如自定义变量名和函数名)[7]. 抽象名称替换策略则是指将自定义的变量名和函数名替换为通用符号名 *VAR_number* 和 *FUN_number*. 例如, 使用 *VAR*_1, *VAR*_2 等来表示变量名, 以及 *FUN*_1, *FUN*_2 等来表示函数名. *number* 是基于变量名和函数名在代码中的出现顺序进行计数. 使用到的语料为训练集中的源代码. 每一行代码经过字符分割后, 被用作 Word2Vec 模型的输入. Word2Vec 模型训练时使用 Skip-Gram 预测模型, 用于学习词的向量表示. 最终参与表示学习的代码图如图 3 所示, 图 3 中的节点特征为高维向量.

4.1 可学习的读出函数

在基于 GNN 的漏洞检测模型中, 图分类任务需要描述全图的单个嵌入向量, 将节点特征和图结构特征嵌入到固定维度向量的模块称为图读出模块. SUM 和 MEAN 等读出方法将所有节点视为同等重要, 这显然是不合理的. 在代码表征图中, 漏洞触发语句通常更为关键, 应该被赋予更高的权重. 采用 GlobalAttention, 可以使模型将注意力集中在漏洞触

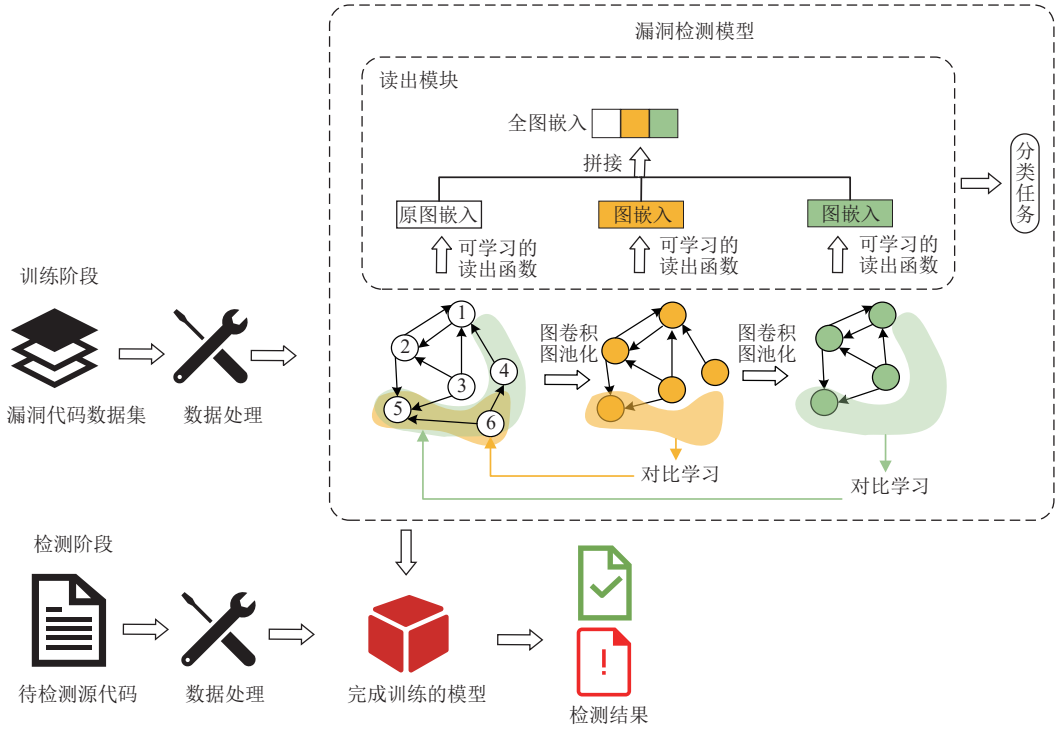


Fig. 2 Flowchart of vulnerability detection

图 2 漏洞检测流程图

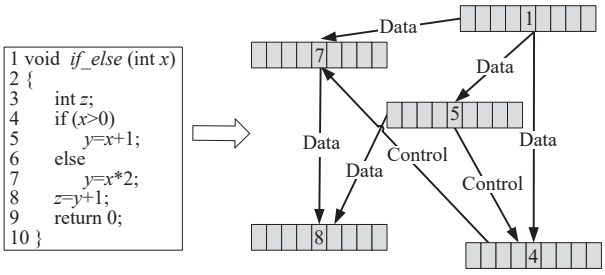


Fig. 3 Example of a code map for participating in representation learning

图 3 参与表示学习的代码图示例

发语句上，而不是均匀处理所有输入，显示出良好的灵活性和通用性. 但这种基于固定计算规则的权重分配策略在实际应用中可能导致某些节点信息被过度放大，而其他节点信息被边缘化. 因此，我们需要一种能够动态调整的自适应策略，以更好地应对不同粒度图中的节点层级差异.

Wu 等人[34]提出可以使用全局注意力表示 GNN 的长距离上下文，他们将特殊 CLS 符号添加到节点嵌入序列，利用 Transformer 层的自注意力机制来捕获全局信息，如图 4 中虚线箭头部分所示. 本文实现思路与之不同，LGR 抛弃了原有拓扑结构，将可训练的 CLS 节点与所有其他节点建立连接，利用 GAT 层的图注意力机制将所有节点对交互聚合到单个分类向量中. 本文 LGR 在使用 GAT 计算 CLS 时和 Transformer 的 attention 在聚合信息的核心思想上具有相似性. 但是 LGR 未使用 position embedding，并且 LGR 仅关注于 CLS 节点的特征，而 Transformer 需要计算所有节点聚集后的特征.

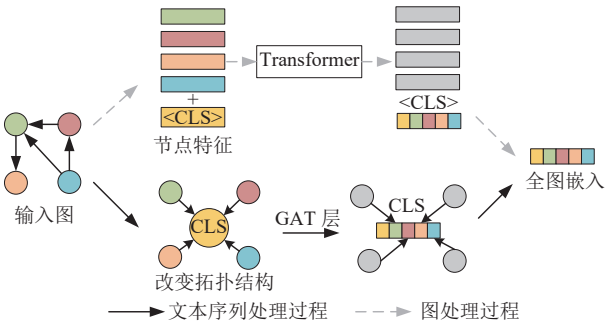


Fig. 4 Readout function design

图 4 读出函数设计

图 2 的读出模块由 3 个可学习的读出函数构成，读出函数的设计如图 4 中实线箭头部分所示，首先向节点特征序列添加 CLS 节点，同时修改输入图的拓扑结构，使图的所有节点均指向 CLS 节点，然后通过 GAT 层来学习原输入图中所有节点和 CLS 节点间的关系. 该函数将 CLS 节点与注意力机制相结合，采用动态权重分配策略来弥补传统方法的缺陷. 同时，CLS 节点参与了训练过程，可以在训练过程中自适应地调整. 从而根据程序代码表征图中数据的特性动态调整权重，使得模型能够更好地生成与任务相关的全图表示，更好地处理不同粒度图的信息. 读出函数的具体实现过程如下：

1）节点初始化

$V=\{v_1,v_2,\cdots,v_n\}$是图中的原始节点集合. v_{cls}是一个可学习的节点，在开始训练之前，CLS 节点的嵌入被随机初始化:

$$e_{\text{cls}}^{(0)} \sim \mathcal{D}(\theta), \tag{3}$$

其中$e_{\text{cls}}^{(0)}$表示 CLS 节点的初始嵌入，$\mathcal{D}(\theta)$表示参数化的分布，比如正态分布或均匀分布，θ表示分布的参数.

2）结构重构

移除原图中所有的边，令$E'=\varnothing$. 连接所有节点到 CLS 节点，对于每个$v_i \in V$，添加一条边(v_i,v_{cls})到E'.

3）应用注意力机制

对于新图$G'=(V\cup v_{\text{cls}},E')$，使用 GAT[37] 网络计算节点间的注意力系数：

$$\alpha_{ij}=\frac{\exp\left(LeakyReLU\left(\boldsymbol{a}^{\mathrm{T}}\left[\boldsymbol{W}\boldsymbol{h}_i\|\boldsymbol{W}\boldsymbol{h}_j\right]\right)\right)}{\sum\limits_{k\in N(i)\cup v_{\text{cls}}}\exp\left(LeakyReLU\left(\boldsymbol{a}^{\mathrm{T}}\left[\boldsymbol{W}\boldsymbol{h}_i\|\boldsymbol{W}\boldsymbol{h}_k\right]\right)\right)}, \tag{4}$$

其中$\boldsymbol{h}_i$是节点i的特征，$\boldsymbol{W}$是权重矩阵，$\boldsymbol{a}$是注意力系数的参数向量，$\|$表示拼接操作，$N(i)$是i的邻居节点集合.

4）信息更新

更新每个节点的特征：

$$\boldsymbol{h}'_{\text{cls}}=LeakyReLU\left(\sum_{i\in V\cup v_{\text{cls}}}\alpha_{i,\text{cls}}\boldsymbol{W}\boldsymbol{h}_i\right)+e_{\text{cls}}^{(0)}. \tag{5}$$

5）全图嵌入

最终，获取 CLS 节点的嵌入$\boldsymbol{h}'_{\text{cls}}$作为整个图的表示. 读出函数不仅重构了图结构，而且通过学习到的全局注意力权重，将所有节点信息有效地集中到了 CLS 节点上，从而得到了一个全图的嵌入表示. 读出函数$R(G)$表示为

$$R(G)=\boldsymbol{h}_{\text{cls}}. \tag{6}$$

4.2 跨粒度对比损失函数

LHG-VD 的图池化策略为 Top-k，因此在层次化图表示学习中，图池化过程涉及节点和边的整合和删减，引起图拓扑结构的变动[38]，导致细粒度的局部结构信息在池化后的粗粒度图中丢失. 随着层次的

增加,图的表示越来越抽象,也会导致局部结构信息丢失.代码表征图中具有丰富的局部结构信息,这些信息对于漏洞发现非常有价值[13].

LHG-VD 设计 CGCL 的核心思想是通过优化模型使得相似(正)样本对的特征表示更接近,而不相似(负)样本对的特征表示更远离.在训练过程中添加 CGCL,能够在学习如何有效地压缩和表征图的同时,保留那些对分类任务至关重要的细粒度局部块结构信息.即使在经过池化后的粗粒度图中,这些关键的局部结构特征也得以保留,从而提高了模型的表征能力和最终的分类性能.CGCL 并不是为了直接修改池化算法本身,而是为了在图卷积时提取和保留那些对分类任务至关重要的细粒度局部块结构信息,即使在经过池化后的粗粒度图中,这些关键的局部结构特征也得以保留.

1)从正样本对角度出发,正样本对由经过图池化层和图卷积层的节点对组成(如图 2 橙色区域内图池化前后的 5 号节点).图卷积层能够捕捉节点的局部邻域信息,而池化层则旨在减少图的复杂性,提炼出更高级的特征.通过最大化正样本对之间的相似度,鼓励模型保持原始节点的关键信息和局部结构特征,通过强化这些节点在池化前后的相似度使得模型能够保留这些关键的局部信息.图 2 中池化后的 5 号节点能够保持原图中 5 号节点的关键信息和局部结构特征(图 2 橙色区域).

2)从负样本对角度出发,负样本对由不共享相同局部块结构的节点组成,通过最小化负样本对之间的相似度,训练模型以区分不同的局部结构,这强化了模型对图中不同区域的敏感性和辨识能力.通过对负样本对施加约束,模型被迫学习更具辨识力的、区分不同结构的嵌入.

3)从图卷积层参数训练过程出发,图卷积层通过聚合邻居节点的信息来更新每个节点的表示,能够捕捉复杂的局部结构模式.在对比学习框架下,引导 GNN 更加关注那些对区分正负样本对有帮助的特征,从而保持和强化局部块结构特征.

LHG-VD 在训练过程中添加 CGCL,能够在学习如何有效地压缩和表征图的同时,保留那些对分类任务至关重要的细粒度局部块结构信息.即使在经过池化后的粗粒度图中,这些关键的局部结构特征也得以保留,从而提高了模型的表征能力和最终的分类性能.

在对比学习中,涉及正样本和负样本部分的对比,对比的是节点之间的相似程度.LHG-VD 使用余弦相似度来计算节点之间的相似程度,如式(7)所示.其中,$\boldsymbol{x}\cdot\boldsymbol{y}$ 是向量$\boldsymbol{x}$ 和向量$\boldsymbol{y}$ 的点积,$\|\boldsymbol{x}\|$ 和$\|\boldsymbol{y}\|$ 分别是向量$\boldsymbol{x}$和向量$\boldsymbol{y}$的欧几里得范数,计算方式如式(8)(9)所示.为了保证数值稳定性,添加了一个小的常数ε($\varepsilon=1\mathrm{E}-8$).

$$\cos(\boldsymbol{x},\boldsymbol{y})=\frac{\boldsymbol{x}\cdot\boldsymbol{y}}{\|\boldsymbol{x}\|\times\|\boldsymbol{y}\|},\tag{7}$$

$$\boldsymbol{x}\cdot\boldsymbol{y}=\sum_{i=1}^{n}x_iy_i,\tag{8}$$

$$\|\boldsymbol{x}\|=\sqrt{\sum_{i=1}^{n}x_i^2+\varepsilon}.\tag{9}$$

正样本对是指经过图池化和图卷积后,粗粒度图中的节点与原始图中对应节点组成的节点对.式(10)计算正样本对的损失,其中x_{pooled} 和x_{unpooled} 分别表示经过池化和未经池化的节点特征.负样本对是指经过图卷积和图池化后,粗粒度图中的节点与原始图中对应节点的非邻域节点组成的节点对.式(11)计算负样本对的损失,$x_{\text{non_neighbor}}$ 表示非邻域节点的特征.

$$L_{\text{pos}}=\left(1-\cos\left(x_{\text{pooled}},x_{\text{unpooled}}\right)\right)^2,\tag{10}$$

$$L_{\text{neg}}=\cos\left(x_{\text{pooled}},x_{\text{non_neighbor}}\right)^2,\tag{11}$$

$$Mean_loss=\frac{1}{M\times K}\sum_{j=1}^{M}\sum_{k=1}^{K}\left(\frac{1}{N_k}\sum_{i=1}^{N_k}L^{(i,j,k)}\right).\tag{12}$$

式(7)~(11)是指单个样本对在单个块(包含池化层和若干卷积层)中的计算方法,在实际使用中,模型里包含若干个相同的块,L_{pos} 和L_{neg} 需要计算平均损失 *Mean_loss*(式(12)).为了保持池化前原图中的重要局部结构信息,在对不同块进行损失计算时,x_{unpooled} 始终为首次池化前的代码表征图中的节点,x_{pooled}为对应经过第 j个块池化后代码表征图中的节点.N_k 为第k 张图中的样本对数量,K 表示每批训练中的图数量,即 *batch_size*. M 表示总块数,完整的 GNN 模型通常包含多个块.$L^{(i,j,k)}$ 表示第k 张图中第i个样本对在第 j 个块中的损失,若为正样本对损失,使用式(10)进行计算,若为负样本对损失则使用式(11)进行计算.

将正样本和负样本对的平均损失结合,得到最终的平均总损失$L_{\text{contrastive}}$. $L_{\text{contrastive}}$ 是对每个批次中所有图、所有块和所有样本对损失的综合平均,计算方式为:

$$L_{\text{contrastive}}=\alpha Mean_loss_{\text{pos}}+\beta Mean_loss_{\text{neg}},\tag{13}$$

其中α 和β 是权重参数,用于控制正样本和负样本损失在总损失中的相对重要性,在实验过程中,α 和β

被设为 1∶1, 以平衡 2 种损失.

4.3 GNN 模型设计

图 5 展示了 LHG-VD 的 GNN 模型设计. 经过多次卷积和池化操作后, 会生成不同粒度的图, LHG-VD 为这些不同粒度的图分别配置了独立且可学习的读出模块. 图 5 包含了 3 次池化过程, 配备 3 个独立的 LGR 来自适应地学习 3 种不同粒度图的特征. 经过 3 层池化, LHG-VD 得到的 3 个读出结果 $R(G^1)$, $R(G^2)$, $R(G^3)$分别对应 3 个不同粒度的代码表征图嵌入, 最终的图嵌入通过拼接这 3 个读出结果得到:

$$\boldsymbol{H}_{\text{final}}(G)=R(G^1)\|R(G^2)\|(G^3). \tag{14}$$

$\boldsymbol{H}_{\text{final}}(G)$代表整个图的最终嵌入向量, ‖表示向量拼接操作. 在 LHG-VD 中, 每层的读出函数都能够捕捉到不同层次的图结构信息, 通过拼接这些信息, 最终的图嵌入能够综合反映整个图的全局结构和局部结构信息, 使得图嵌入更加丰富、表达能力更强.

LHG-VD 是一个端到端方式的图分类任务, 端到端是指从输入到输出通过一个统一的模型完成, 而不是分步骤处理. LHG-VD 通过交叉熵损失函数来主导训练过程, 能够最小化模型预测的输出与实际标签之间的差异. 另外, 加入 CGCL 能够比较不同图之间的相似性或差异性, 增强模型在处理图数据时的判别能力. 为了简化优化过程, 使得模型训练过程中的梯度下降等优化算法更加高效和直接, LHG-VD 设置一个综合的损失函数来平衡交叉熵损失和对比损失, 如式(15)所示.

$$L_{\text{final}}=\gamma\times L_{\text{cross_entropy}}+(1-\gamma)\times L_{\text{contrastive}}, \tag{15}$$

其中, γ是一个介于 0 和 1 之间的权重参数, 用于调整对比函数在训练过程中的影响力度. 在最终模型中, γ值设置为 0.5, 将交叉熵损失和对比学习损失的影响设置为同等重要.

5 实　　验

为了验证 LHG-VD 模型的检测能力, 实验选择了 6 个目前最先进的工具进行对比, 包括 1 个基于检测规则的工具 Flawfinder 和 5 种基于深度学习的工具. 其中, 3 种基于 GNN 的漏洞检测工具分别是 ReGVD, AMPLE, DeepWukong, 2 种基于文本序列的方法分别是 LineVul 和 CodeBERT. CodeBERT 的主体与 LineVul 相同, 差别在于 CodeBERT 使用默认的 BPE 分词进行语义嵌入, LineVul 使用重新训练过的 BPE 分词进行语义嵌入.

为了验证 LGR 和 CGCL 能否有效提升模型性能, 实验从 LHG-VD 模型分离出 LHG-VD-CLS 模型, 表示只添加了 LGR 的 GNN 模型, 该模型没有添加 CGCL. 通过实验希望回答以下 3 个研究问题.

1)在人工合成的数据集上, LHG-VD-CLS 和 LHG-VD 的表现是否能够超过最先进的切片级漏洞级方法 DeepWukong.

2)在真实漏洞数据集上, LHG-VD-CLS 和 LHG-VD 的表现是否优于 DeepWukong.

3)LHG-VD 的漏洞检测能力是否能够超过其他最先进的漏洞检测方法.

5.1 实验数据集

本文在 DeepWukong 项目中开源的 CWE119 数据集和广泛被使用的 Big-Vul 数据集子集上进行所有的实验.

CWE119 是人工合成的漏洞数据集, 大部分漏洞是非法访问内存漏洞. 样本数据来自软件保障参考数据集[39](software assurance reference dataset, SARD), 每个样本(即测试用例)对应 1 个或多个 CWE ID, 即

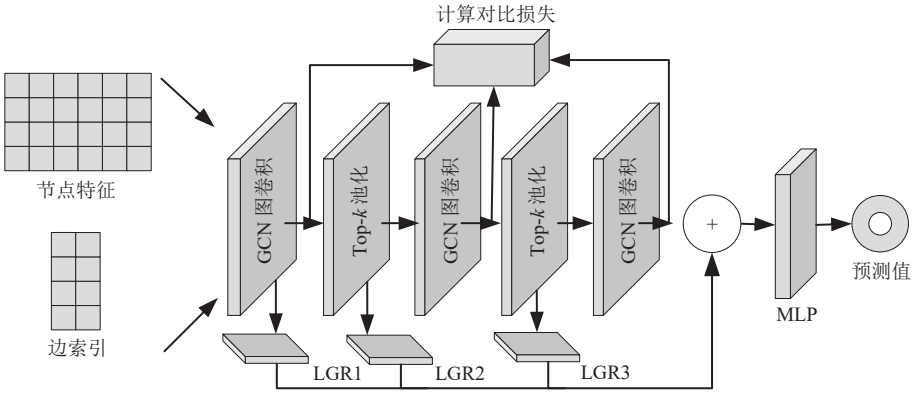


Fig. 5 GNN model of LHG-VD

图 5 LHG-VD 的 GNN 模型

存在多种不同类型的漏洞. 样本以测试用例的形式给出, 不存在函数漏洞标记, 无法直接将 CWE119 数据集应用于函数级漏洞检测方法. 因此, CWE119 数据集上的实验对比在切片级漏洞检测方法 LHG-VD 和 DeepWuKong 之间进行. 该数据集包含漏洞切片样本 14 262 个、无漏洞切片样本 74 935 个.

Big-Vul 数据集[40]是一个高质量的真实漏洞数据集, 涵盖了从 2002—2019 年间 348 个真实开源项目, 共有 11 834 个漏洞函数和 253 096 个无漏洞函数. LHG-VD 是切片级漏洞检测方法, 为了能够与函数级漏洞检测方法进行对比, 要求样本不仅包含函数的漏洞标记还要包含函数中具体漏洞行号的标记. Big-Vul 提供了漏洞修复前后的源码, 我们使用 Big-Vul 数据集中的所有易受攻击的函数共 11 834 个及其对应的非易受攻击的函数, 参考 Hu 等人[21]的处理方法, 通过 diff 工具比对, 确定漏洞代码行信息. 实验将函数修复前的源码设置为包含漏洞的样本, 将修复后的源码设置为无漏洞样本.

AMPLE 是最先进的函数级漏洞检测方法, 但是其数据处理模块无法正常合成 Big-Vul 的部分样本. 为了保证公平性, 从 Big-Vul 数据集中剔除了无对应非易受攻击样本、出错样本和 AMPLE 无法合成的漏洞样本, 保留了 5 145 个漏洞函数样本和 5 464 个无漏洞函数样本. 根据这些函数样本, 最终生成了 16 555 个无漏洞切片样本和 17 000 有漏洞切片样本.

5.2 实施方法

实验使用系统环境为 Ubuntu20.04LTS, CPU 为 Intel Corei7-11700 @2.5 GHz, GPU 为 NVIDIA GeForce RTX3070, 物理内存 32 GB, 硬盘大小为 1 TB. 深度学习环境基于 PyTorch1.9 和 PyTochLightning1.3.7 框架, GNN 使用 PyTorchGeometric2.1.0 搭建.

实验使用 CWE119 数据集和裁剪后的 Big-Vul 数据集, 随机选择 80% 的样本用于训练, 10% 作为验证集数据, 10% 作为测试集数据.

实验使用 PyTorchLightning 框架训练 LHG-VD 模型, ModelCheckpoint 是训练过程中可以保存的检查点. 实验为每轮训练过程设置一个 ModelCheckpoint, 参数 *monitor* 设置为“val_loss”, *mode* 设置为“min”, *save_top_k* 设置为 5. 设置 *save_top_k* 值为 5 主要有 3 个原因: 1)保存性能最佳的前 5 个模型能够平衡模型性能监控与存储资源的需求, 既保证有足够的数据来选择最优模型, 又不至于占用太多存储空间. 2)我们在实验中观察到, 在达到前 5 个最佳模型后, 模型性能的提升变得相对平缓, 表示后续的模型版本不会带来显著的性能改进. 3)在基准模型 DeepWukong 中, *save_top_k* 值也设置为 5, 为保证对比实验的公平性, LHG-VD 同样将 *save_top _k* 设置为 5.

实验把每轮模型训练生成的模型参数应用于验证集数据, 如果验证集上的损失值最小, 就在检查点保存该次训练的模型参数.

5.3 评估指标

真正例(true positive, *TP*)为正确预测为有漏洞的样本数量, 真负例(true negative, *TN*)为正确预测为无漏洞的样本数量, 假正例(false positive, *FP*)为错误预测为有漏洞的样本数量, 假负例(false negative, *FN*)为错误预测为无漏洞的样本数量, 漏洞样本数量为 *TP* 与 *FN* 数量之和, 无漏洞样本数量为 *TN* 与 *FP* 数量之和.

准确率 (accuracy, *A*)是指模型正确预测的样本数量占总样本数量的比例, 一定程度反映了模型对有漏洞和无漏洞样本的分类能力. 准确率显示了模型在所有预测中表现的正确程度. 但在类别不平衡的情况下, 特别是漏洞样本很少时, 即使模型全部预测为无漏洞, 准确率仍然会很高. 因此, 仅凭准确率判断模型性能可能具有误导性.

$$A=\frac{TP+TN}{TP+TN+FP+FN}. \tag{16}$$

精确率(precision, *P*)表示 *TP* 占所有预测为漏洞样本(*TP*+*FP*)的比例, 精确率显示了模型识别漏洞时的准确性, 高精确率也意味着较少的误报.

$$P=\frac{TP}{TP+FP}. \tag{17}$$

召回率(recall, *R*)表示 *TP* 占所有漏洞样本的比例. 漏报(*FN*)可能导致严重后果, 更高的召回率意味着更低的漏报.

$$R=\frac{TP}{TP+FN}. \tag{18}$$

*F*1 分数是精确率和召回率的调和平均数, 提供了更全面的性能度量. 在漏洞检测中, *F*1 分数特别重要, 模型需要在减少误报(更高的准确率)和减少漏报(更高的召回率)之间寻求平衡. 如果模型在测试集上的 *F*1 分数越高, 表示在测试集上的性能越好, 具有更好的泛化能力.

$$F1=2\times\frac{P\times R}{P+R}. \tag{19}$$

5.4 实验分析

5.4.1 针对问题 1 的实验

为了回答问题 1, 本实验在 CWE119 数据集上比较 LHG-VD-CLS, LHG-VD, DeepWukong 模型的漏洞

检测能力，主要验证 LGR 和 CGCL 是否能够提高模型的性能，特别是验证读出函数是否提升了长距离图的特征学习能力.

表 1 给出了实验结果. 在 LHG-VD-CLS 使用 LGR 替换 DeepWukong 使用的传统 GlobalAttention 方式后，*FN* 数量从 160 下降到 90，下降约 43.8%，召回率提高了 5%，表明使用 LGR 能够实现更少的漏报. LHG-VD-CLS 模型的准确率提升了 1 个百分点，精确率提升了 1.4 个百分点，表明模型具有更少的误报. 总之，LHG-VD-CLS 在各项指标上都优于 DeepWukong，*F*1 分数提高了 3.3 个百分点，表明 LGR 能够有效提高模型性能.

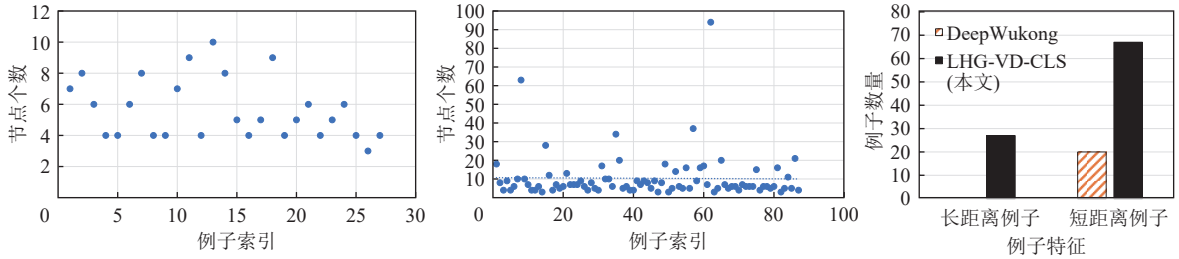
LGR 的设计初衷是为了帮助 GNN 更好地学习图的长距离上下文. 因为图中节点数量可以在一定程度上反映图的复杂度和节点与节点间的距离. 实验分别统计了 DeepWukong 正确预测但是 LHG-VD-CLS 没有正确预测的 *TP* 样本的节点个数，如图 6(a)的散点图所示，以及 LHG-VD-CLS 正确预测但是 DeepWukong 没有正确预测的 *TP* 样本的节点个数，如图 6(b)散点图所示. 图 6(a)(b)散点图的横坐标例子索引无特殊含义，仅表示样本的编号. 图 6(c)柱状图中对节点数大于 10(长距离例子)和小于等于 10(短距离例子)的代码表征图数量进行统计，从中可以看出 LHG-VD-CLS 对长距离图的预测能力显著优于 DeepWukong，DeepWukong 很难预测节点数量较多并且结构复杂的图，LHG-VD-CLS 对长距离图和短距离图均有更好的适应性.

Table 1 Ablation Experiments in Artificial Vulnerability Dataset

表 1 在人工漏洞数据集的消融实验

数据集	模型	*FN*	*FP*	*TN*	*TP*	*F*1/%	*A*/%	*R*/%	*P*/%
CWE119	DeepWukong	160	89	7 409	1 262	91.0	97.2	88.7	93.4
	LHG-VD-CLS	**90**	72	7 426	**1 332**	**94.3**	**98.2**	**93.7**	94.8
	LHG-VD(本文)	93	**71**	**7 427**	1 329	94.1	98.2	93.5	**94.9**

柱：黑体数值表示最优结果.



(a) DeepWukong 正确预测例子的特征 (b) LHG-VD-CLS (本文) 正确预测例子的特征 (c) 正确预测例子对比

Fig. 6 *TP* example features

图 6 *TP* 例子特征

从表 1 中可以看出，添加了 CGCL 后，LHG-VD 相比 LHG-VD-CLS，*F*1 分数略有下降. 因为 CWE119 是人工合成的数据集，测试用例不具备真实代码的复杂性，然而，LHG-VD 提出的 CGCL 的作用是区分细微的结构差异，在处理复杂数据时更有效. CWE119 的测试用例的图结构相对简单，在添加 LGR 后，LHG-VD-CLS 已经能够实现非常有效的分类，LHG-VD 添加对比损失函数未能继续增加模型性能.

5.4.2 针对问题 2 的实验

5.4.1 节验证了可学习读出函数能够有效提升模型性能，但是在人工合成的 CWE119 数据集上没有体现 CGCL 的作用，本节使用 Big-Vul 真实漏洞数据集来验证 CGCL 的作用.

实验结果如表 2 所示，LHG-VD-CLS 的精确率提升了 2.1 个百分点，但是召回率下降了 1 个百分点，*F*1 分数提升了 0.9 个百分点，提升幅度低于在 CWE119 的实验结果，因为真实漏洞代码数据集包含了更复杂的图结构、更多样的代码模式和更微妙的漏洞特征.

Table 2 Ablation Experiments in Real Vulnerability Dataset

表 2 在真实漏洞数据集的消融实验 %

模型	*F*1	*A*	*R*	*P*
DeepWukong	67.6	64.4	75.9	61.0
LHG-VD-CLS	68.5	66.3	74.9	63.1
LHG-VD（本文）	**71.5**	**69.0**	**79.5**	**65.0**

注：黑体数值表示最优结果.

LGR 使用 CLS 节点结合全局注意力机制更适合捕捉全局特征而不是局部细节，在处理复杂的真实数据集时，可能忽视某些局部重要特征，从而影响召回率.

增加 CGCL 后，LHG-VD 的召回率提升了 3.6 个百分点，因为 CGCL 可以强化正样本对之间的相似性和增加负样本对之间的差异性，使模型在特征空

间中形成更加鲜明的分类边界，能够更准确地识别出正样本，从而减少漏报. LHG-VD 的精确率提升了 4 个百分点，因为 CGCL 能够有效地区分正负样本对，提高了模型识别负样本的能力，从而降低误报率，*F*1 分数提升了 2.9 个百分点. 最终，LHG-VD 在所有指标上都超过了 DeepWukong.

在处理复杂的真实代码时，CGCL 能够通过对正负样本对的分析，捕捉到更细微的特征差异，弥补了可学习读出函数可能忽视局部细节的不足.

5.4.3 针对问题 3 的实验

为了回答问题 3，本实验将 LHG-VD 与 6 种最先进的工具在 Big-Vul 数据集上进行检测能力对比，结果如表 3 所示.

Table 3 Comparative Experiment

表 3 对比实验 %

类别	模型	*F*1	*A*	*R*	*P*
基于规则方法	Flawfinder[23]	22.1	48.7	14.1	50.9
基于函数的 token 图方法	ReGVD[16]	61.7	52.0	79.0	50.6
基于函数的图方法	AMPLE[22]	65.1	48.3	**98.8**	48.5
基于函数的文本方法	LineVul[9]	66.2	52.9	94.5	51.0
基于切片级的图方法	DeepWukong[13]	67.6	64.4	75.9	61.0
基于函数的文本方法	CodeBERT	69.3	58.5	95.8	54.3
基于切片级的图方法	LHG-VD（本文）	**71.5**	**69.0**	79.5	**65.0**

注：黑体数值表示最优结果.

LHG-VD 的 *F*1 分数、准确率和精确率都优于其他 6 种方法，*F*1 分数为 71.5%，与 DeepWukong 相比，提升 4.9 个百分点，与 AMPLE 相比，提升 8.9 个百分点.

AMPLE, LineVul, CodeBERT 的召回率都超过了 90 个百分点，高于 LHG-VD，但 3 个模型的精确率和 *F*1 分数都低于 LHG-VD，因为它们倾向于将样本预测为漏洞样本以减少漏报，但是增加了误报的风险.

AMPLE 通过自定义的图简化策略和边缘感知图卷积网络模块来学习代码的结构特征，能够更好地捕捉代码中的复杂关系，尤其是远距离的依赖. 但是，AMPLE 在捕捉代码结构特征时过于敏感，倾向于将包含特定结构或模式的代码错误地识别为漏洞，容易产生误报，导致精确率只有 48.5%. 我们在检查 AMPLE 的实验中发现，AMPLE 设置的损失函数策略倾向于将样本预测为有漏洞，以达到更少漏报，导致在 Big-Vul 数据集的召回率非常高. 另外，AMPLE 受限于自身的构图机制，其数据处理模块无法正常合成 Big-Vul 数据集的部分样本. 最终，实验数据集的正样本和负样本的数量相近，对 AMPLE 模型的训练和泛化能力也造成了一定的挑战.

LineVul 和 CodeBERT 都采用 BERT 架构生成代码的向量表示，专注于捕捉代码中的长期依赖关系和语义信息. BERT 架构能够有效地捕捉长期依赖关系，但是无法理解代码的精确结构和上下文. 它们在尝试捕捉尽可能多的潜在漏洞时，容易将没有漏洞的代码片段错误地分类为漏洞样本，精确率分别只有 51% 和 54.3%. GNN 和大模型在漏洞检测方面各有优势和局限，GNN 擅长捕捉代码中的结构特征，而大模型则在代码理解和推理方面表现出色.

从表 3 可以看出，切片级方法 DeepWukong 和 LHG-VD 的精确率分别为 61% 和 65%，优于函数级方法 AMPLE(48.5%)和 ReGVD(50.6%). 切片级方法会对代码进行更细致的分析，准确地定位和分析与特定行为或状态相关的代码段，更好地理解变量如何在代码中操作和传播的过程，从而更精确地捕捉潜在漏洞. Flawfinder[23] 采用基于规则的方法，缺乏对代码上下文和结构的深入理解，依赖于静态预定义的规则，不能很好地适应复杂或新型的漏洞模式，其各项性能指标垫底.

ReGVD 是基于 GNN 的函数级漏洞检测工具，从代码文本的 token 序列中，根据自定义策略构建代码表征图. 自定义策略限制了模型对代码结构和语义的理解能力，导致 GNN 的结构相对简单，无法捕捉复杂的代码模式和漏洞特征，其各项性能指标都低于 LHG-VD.

图 7 是 Big-Vul 数据集中标记为“1_CVE-2018-1000039_ghostscript_CWE-416”的漏洞样本片段，该样本总计 152 行代码，图 7 截取了部分与漏洞代码(第 58 行)存在数据流和控制流依赖关系的代码行. 该样本对应的代码切片表征图的完整调用链长度为 24，

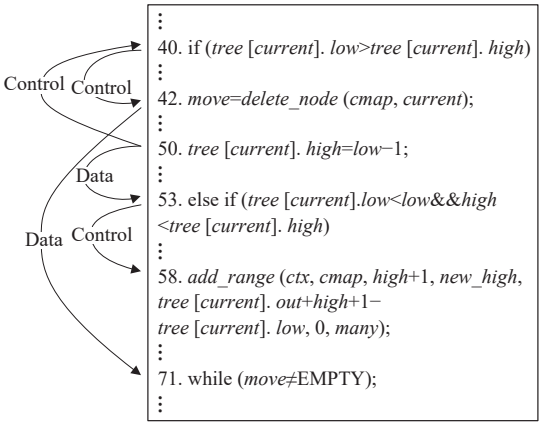


Fig. 7 Vulnerability sample snippet

图 7 漏洞样本片段

LHG-VD 成功检测了该漏洞，而 DeepWuKong 没有正确检测该漏洞，说明 LHG-VD 在长调用链的代码表征图中能获得更好的检测表现. 在该样本中，与漏洞代码相关的数据流和控制流依赖关系较为复杂，导致 3 种函数级方法都没有正确理解样本的结构和上下文，因此没有成功检测.

6 总结与展望

本文根据程序代码特点，在基于 GNN 的漏洞检测问题上，针对现有图读出方法无法很好适应节点的层次差异问题，提出一种可学习的图读出函数 LGR，针对现有图池化过程存在细粒度图的局部结构信息丢失问题，提出一个跨粒度的对比损失函数 CGCL. 最终，实现了一种可学习的层次化图表示模型 LHG-VD. 在真实漏洞代码数据集上的对比实验结果表明，LHG-VD 的检测性能优于目前最先进的 6 种漏洞检测工具.

漏洞数据库的编程语言为 C/C++，如果要在其他语言上实现 LHG-VD，主要难点在于漏洞数据收集、敏感函数收集和语法规则适配等方面. 我们尝试过结合 CodeBERT 等预训练编程语言模型实现对多种编程语言兼容，但是训练时间开销过大，实验效果不佳.

现有真实漏洞数据集的漏洞样本数量和无漏洞样本数量的差距太大，严重影响训练效果. 未来计划引入数据增强，尝试使用源码变换和图增强方式增加漏洞样本的数量. 另外，未来计划融合基于文本序列的方法与图分类方法[41-42]，寻找更优的融合策略，提升漏洞模型检测能力.

作者贡献声明：胡菘提出想法思路，完成系统实现、实验和论文撰写；罗嘉驰参与创新点实现；万文凯和闫阳参与实验部分工作；郭帆提出指导意见并修改论文；曲彦文参与了研究思路的讨论并提供了修改建议.

参 考 文 献

[1] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis[C]//Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 480−491

[2] Ji Xiujuan, Yang Jufeng, Xu Jing, et al. Interprocedural path-sensitive resource leaks detection for C programs[C]//Proc of the 4th Asia-Pacific Symp on Internetware. New York: ACM, 2012: 1−9

[3] Kroening D, Tautschnig M. CBMC–C bounded model checker[C]//Proc of the 20th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2014: 389−391

[4] Fan Gang, Wu Rongxin, Shi Qingkai, et al. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code[C]//Proc of the 41st IEEE/ACM Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2019: 72−82

[5] Chernis B, Verma R. Machine learning methods for software vulnerability detection[C]//Proc of the 4th ACM Int Workshop on Security and Privacy Analytics. New York: ACM, 2018: 31−39

[6] Russell R, Kim L, Hamilton L, et al. Automated vulnerability detection in source code using deep representation learning[C]//Proc of the 17th IEEE Int Conf on Machine Learning and Applications (ICMLA 2018). Piscataway, NJ: IEEE, 2018: 757−762

[7] Li Zhen, Zou Deqing, Xu Shouhuai, et al. VulDeePecker: A deep learning-based system for vulnerability detection[C]//Proc of the 25th Network and Distributed System Security Symp (NDSS 2018). Piscataway, NJ: IEEE, 2018: 1−15

[8] Aumpansub A, Huang Zhen. Detecting software vulnerabilities using neural networks[C]//Proc of the 13th Int Conf on Machine Learning and Computing. New York: ACM, 2021: 166−171

[9] Fu M, Tantithamthavorn C. LineVul: A transformer-based line-level vulnerability prediction[C]//Proc of the 19th Int Conf on Mining Software Repositories. New York: ACM, 2022: 608−620

[10] Feng Hantao, Fu Xiaotong, Sun Hongyu, et al. Efficient vulnerability detection based on abstract syntax tree and deep learning[C]//Proc of the IEEE Conf on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway, NJ: IEEE, 2020: 722−727

[11] Li Zhen, Zou Deqing, Xu Shouhuai, et al. SySeVR: A framework for using deep learning to detect software vulnerabilities[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2244−2258

[12] Zhou Yaqin, Liu Shangqing, Siow Jingkai, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[C]//Proc of the 33rd Int Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2019: 10197−10207

[13] Cheng Xiao, Wang Haoyu, Hua Jiayi, et al. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network[J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(3): 1−33

[14] Chakraborty S, Krishna R, Ding Yangruibo, et al. Deep learning based vulnerability detection: Are we there yet[J]. IEEE Transactions on Software Engineering, 2021, 48(9): 3280−3296

[15] Hu Yutao, Wang Suyuan, Wu Yueming, et al. Slice-level vulnerability detection and interpretation method based on graph neural network [J]. Journal of Software, 2023, 34(6): 2543−2561 (in Chinese)
(胡雨涛，王溯远，吴月明，等. 基于图神经网络的切片级漏洞检测及解释方法[J]. 软件学报，2023，34(6)：2543−2561)

[16] Nguyen V A, Nguyen D Q, Nguyen V, et al. ReGVD: Revisiting

graph neural networks for vulnerability detection[C]//Proc of the 44th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2022: 178−182

[17] Wu Yuelong, Lu Jintian, Zhang Yueyi, et al. Vulnerability detection in C/C++ source code with graph representation learning[C]//Proc of the 11th IEEE Annual Computing and Communication Workshop and Conf (CCWC). Piscataway, NJ: IEEE, 2021: 1519−1524

[18] Cao Sicong, Sun Xiaobing, Bo Lili, et al. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks[C]//Proc of the 44th Int Conf on Software Engineering. New York: ACM, 2022: 1456−1468

[19] Li Yi, Wang Shaohua, Nguyen T N. Vulnerability detection with fine-grained interpretations[C]//Proc of the 29th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering. New York: ACM, 2021: 292−303

[20] Zheng Weining, Jiang Yuan, Su Xiaohong. Vu1SPG: Vulnerability detection based on slice property graph representation learning[C]// Proc of the 32nd IEEE Int Symp on Software Reliability Engineering (ISSRE). Piscataway, NJ: IEEE, 2021: 457−467

[21] Hu Yutao, Wang Suyuan, Li Wenke, et al. Interpreters for GNN-Based vulnerability detection: Are we there yet?[C]//Proc of the 32nd ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2023: 1407−1419

[22] Wen Xincheng, Chen Yupan, Gao Cuiyun, et al. Vulnerability detection with graph simplification and enhanced graph representation learning[C]//Proc of the 45th Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2023: 2275−2286

[23] Ferschke O, Gurevych I, Rittberger M. FlawFinder: A modular system for predicting quality flaws in Wikipedia [C] //Proc of the 3rd Conf and Labs of the Evaluation Forum. Berlin: Springer, 2012:1−10

[24] Feng Zhangyin, Guo Daya, Tang Duyu, et al. CodeBERT: A Pre-Trained model for programming and natural languages[C]//Findings of the Association for Computational Linguistics (EMNLP 2020). Stroudsburg, PA: ACL, 2020: 1536–1547

[25] Church K W. Word2Vec[J]. Natural Language Engineering, 2017, 23(1): 155−162

[26] Devlin J, Chang Mingwei, Lee K, et al. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding[C]//Proc of the 2019 Conf of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Stroudsburg, PA: ACL, 2019: 4171–4186

[27] Li Yujia, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks[J]. arXiv preprint, arXiv: 1511.05493, 2015

[28] Kipf T, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint, arXiv: 1609.02907, 2016

[29] Wang Zhaohui, Shen Huawei, Cao Qi, et al. Survey on graph classification[J]. Journal of Software, 2022, 33(1): 171−192 (in Chinese)
(王兆慧, 沈华伟, 曹婧, 等. 图分类研究综述[J]. 软件学报, 2022, 33(1): 171−192)

[30] Ying Zhitao, You Jiaxuan, Morris C, et al. Hierarchical graph representation learning with differentiable pooling[C]//Proc of the 32nd Int Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2018: 4805−4815

[31] Xu Keyulu, Li Chengtao, Tian Yonglong, et al. Representation learning on graphs with jumping knowledge networks[C]//Proc of the 35th Int Conf on Machine Learning. New York: PMLR, 2018: 5453−5462

[32] Cangea C, Veličković P, Jovanović N, et al. Towards sparse hierarchical graph classifiers[J]. arXiv preprint, arXiv: 1811.01287, 2018

[33] Morris C, Ritzert M, Fey M, et al. Weisfeiler and leman go neural: Higher-order graph neural networks[C]//Proc of the 33rd Conf on Artificial Intelligence. Palo Alto, CA : AAAI, 2019: 4602−4609

[34] Wu Zhanghao, Jain P, Wright M, et al. Representing long-range context for graph neural networks with global attention[J]. Advances in Neural Information Processing Systems, 2021, 34: 13266−13279

[35] Wu Zonghan, Pan Shirui, Chen Fengwen, et al. A comprehensive survey on graph neural networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2021, 32(1): 4−24

[36] Joern Development Team. Joern: Open-source code querying engine for C/C++[CP/OL]. 2020[2025-01-19]. https://joern.io/

[37] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[J]. arXiv preprint, arXiv: 1710.10903, 2017

[38] Liu Chuang, Zhan Yibing, Wu Jia, et al. Graph pooling for graph neural networks: Progress, challenges, and opportunities[J]. arXiv preprint, arXiv: 2204.07321, 2022

[39] National Institute of Standards and Technology. Software assurance reference dataset[DB/OL]. [2025-01-19]. https://samate.nist.gov/SARD

[40] Fan Jiahao, Li Yi, Wang Shaohua, et al. A C/C++ code vulnerability dataset with code changes and CVE summaries[C]//Proc of the 17th Int Conf on Mining Software Repositories. New York: ACM, 2020: 508−512

[41] Hin D, Kan A, Chen Huaming, et al. LineVD: Statement-level vulnerability detection using graph neural networks[C]//Proc of the 19th Int Conf on Mining Software Repositories. New York: ACM, 2022: 596−607

[42] Ding Yangruibo, Suneja S, Zheng Yunhui, et al. VELVET: A novel ensemble learning approach to automatically locate Vulnerable statements[C]//Proc of the 29th Int Conf on Software Analysis, Evolution and Reengineering (SANER). Piscataway, NJ: IEEE, 2022: 959−970



Hu Song, born in 1999. Master. His main research interests include vulnerability detection and graph deep learning.

胡　菘, 1999 年生. 硕士. 主要研究方向为漏洞检测、图深度学习.



Luo Jiachi, born in 2002. Bachelor. His main research interest includes machine learning.

罗嘉驰, 2002 年生. 学士. 主要研究方向为机器学习.



Wan Wenkai, born in 2001. Master. His main research interests include vulnerability detection and graph deep learning.

万文凯, 2001 年生. 硕士. 主要研究方向为漏洞检测、图深度学习.



Yan Yang, born in 1999. Master. His main research interests include vulnerability detection and machine learning.

闫　阳, 1999 年生. 硕士. 主要研究方向为漏洞检测、机器学习.



Guo Fan, born in 1977. PhD, associate professor, His main research interests include network security, program analysis, and vulnerability mining.

郭　帆, 1977 年生. 博士, 副教授. 主要研究方向为网络安全、程序分析、漏洞挖掘.



Qu Yanwen, born in 1983. PhD, lecturer. His main research interests include graph deep learning and vulnerability detection.

曲彦文, 1983 年生. 博士, 讲师. 主要研究方向为图深度学习、漏洞检测.