计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(5): 1158−1176, 2026

基于成对预测的智能漏洞检测模型可信评估与分析

朱嘉豪[1,2,3,5,6] 魏子超[1,2,3,5,6] 郭靖宜[3] 文 明[1,2,3,5,6] 金 海[1,2,4]

[1]（服务计算技术与系统教育部重点实验室（华中科技大学） 武汉 430074）

[2]（大数据技术与系统国家地方联合工程研究中心（华中科技大学） 武汉 430074）

[3]（华中科技大学网络空间安全学院 武汉 430074）

[4]（华中科技大学计算机科学与技术学院 武汉 430074）

[5]（分布式系统安全湖北省重点实验室（华中科技大学） 武汉 430074）

[6]（湖北省大数据安全工程技术研究中心（华中科技大学） 武汉 430074）

（zcwei@hust.edu.cn）

Towards Trustworthy Evaluation and Analysis of Intelligent Vulnerability Detection Models under Pairwise Prediction

Zhu Jiahao[1,2,3,5,6], Wei Zichao[1,2,3,5,6], Guo Jingyi[3], Wen Ming[1,2,3,5,6], and Jin Hai[1,2,4]

[1]（*Services Computing Technology and System Lab* (*Huazhong University of Science and Technology*), *Ministry of Education*, *Wuhan* 430074）

[2]（*National Engineering Research Center for Big Data Technology and System* (*Huazhong University of Science and Technology*), *Wuhan* 430074）

[3]（*School of Cyber Science and Engineering*, *Huazhong University of Science and Technology*, *Wuhan* 430074）

[4]（*School of Computer Science and Technology*, *Huazhong University of Science and Technology*, *Wuhan* 430074）

[5]（*Hubei Key Laboratory of Distributed System Security* (*Huazhong University of Science and Technology*), *Wuhan* 430074）

[6]（*Hubei Engineering Research Center on Big Data Security* (*Huazhong University of Science and Technology*), *Wuhan* 430074）

Abstract Graph neural networks (GNNs) and code pre-trained models (CodePTMs) have been widely applied in vulnerability detection tasks, achieving the state-of-the-art (SOTA) performance on vulnerability benchmark datasets. However, such datasets (denoted as regular datasets) commonly collect vulnerable and non-vulnerable samples from different sources, resulting in significantly divergent code characteristics. Consequently, existing learning models tend to capture spurious features specific to certain datasets, rather than understanding the actual root causes of vulnerabilities. A robust and reliable vulnerability detection model should be able to make pairwise predictions, that is, recognizing the vulnerable code as vulnerable while the corresponding fixed version as non-vulnerable. However, the capability of existing learning models to make such pairwise predictions is significant and has not been explored extensively. Specifically, we first construct a large-scale real-world dataset containing vulnerable code and the corresponding fixed version. This ensures that positive and negative samples are derived from the same source, thus forming similar distributions in terms of code characteristics. We then evaluate eight mainstream SOTA vulnerability detection models in the rigorous setting to see if they can distinguish vulnerable code from its fixed versions (i.e., paired vulnerability instances). Our results reveal that all models perform poorly in making pairwise predictions when trained on regular datasets. We also find that retraining with these paired instances can improve the models' performance by 17% on average. Furthermore, we highlight the limitations of existing CodePTMs in aggregating local

收稿日期：2025-12-15；**修回日期**：2026-03-12

基金项目：国家自然科学基金项目（U2436207, 62372193）

This work was supported by the National Natural Science Foundation of China（U2436207, 62372193）.

通信作者：文明（mwenaa@hust.edu.cn）

code information (i.e., a key statement concerning vulnerable semantics). Driven by our findings, we devise a new approach and the evaluation results demonstrate that it outperforms the SOTA model UniXcoder by 58.47% towards making pairwise predictions.

Key words software engineering; code security; intelligent vulnerability detection; trustworthy model evaluation; code representation learning

摘 要 图神经网络（graph neural networks,GNNs）和代码预训练模型（code pre-trained models, CodePTMs）已被广泛应用于漏洞检测任务，并在漏洞基准数据集上达到了当前最优（state-of-the-art, SOTA）性能。但这类数据集（称常规数据集）通常从不同来源收集漏洞样本和非漏洞样本，导致代码特征差异显著。因此，现有学习模型往往捕捉到的是特定数据集中的虚假特征，并未真正理解漏洞的根本成因。一个鲁棒且可靠的漏洞检测模型应当能对漏洞修复前后样本做出成对预测，即既能将漏洞代码识别为有漏洞，又能将其对应的修复版本判定为无漏洞。然而，现有学习模型在这种能力上存在显著不足，且尚未得到充分研究。具体而言，首先构建了一个大规模真实世界数据集，其中包含漏洞代码及其对应的修复版本。这确保了正负样本来源一致，从而在代码特征上形成相似分布。随后在严格实验设置下对 8 个主流 SOTA 漏洞检测模型进行评估，检验其能否区分漏洞代码与其修复版本（即配对漏洞实例）。实验结果表明，基于常规数据集进行训练时，所有模型在成对预测任务上表现均不佳。研究发现，使用配对实例重新训练可使模型性能平均提升 17%。此外，揭示了现有 CodePTMs 在聚合局部代码信息（即与漏洞语义相关的关键语句）方面存在的局限性。基于上述发现，提出了一种新方法，评估结果表明，该方法在成对预测任务上相较于当前最优模型 UniXcoder 提升了 58.47%。

关键词 软件工程；代码安全；智能漏洞检测；模型可信评估；代码表示学习

中图法分类号 TP311.13；TP309

DOI: 10.7544/issn1000-1239.202550902 **CSTR:** 32373.14.issn1000-1239.202550902

软件漏洞往往会导致严重后果，如系统崩溃、未授权访问和信息泄露等[1]。近年来，随着深度学习（deep learning, DL）的快速发展，众多基于深度学习的源代码漏洞检测方法受到广泛关注[2-3]。现有方法利用图神经网络（graph neural networks, GNNs）、代码预训练模型（code pre-trained models, CodePTMs），或大语言模型（large language models, LLMs）强大的推理能力和领域知识，实现了当前最优（state-of-the-art, SOTA）性能[4-5]。这些方法利用此类学习模型的强大能力，从包含代码片段及对应标签（即是否存在漏洞）[6-7]的大规模标注数据集中捕捉复杂模式和关联关系，为漏洞检测任务提供了一种可行的解决方案。基于此类方法训练的分类模型能识别漏洞模式，并准确预测新代码样本中是否存在漏洞。

基于 GNN 的方法[7-9]将源代码转换为代码属性图（code property graphs, CPGs）[10]，从而直接为模型提供精准且全面的代码结构和语义信息。CodePTMs（如 CodeBERT[11], UniXcoder[12], Code Llama[13]）是在大量代码任务上预训练的语言模型，比通用 GNNs 具有更先进的代码理解与代码生成能力。当前，CodePTMs 在微软 CodeXGLUE①基准测试平台上取得了漏洞检测的最高准确率（69.29%）。此外，诸多研究基于 CodePTMs[14-16]进行了进一步改进与扩展。至于 LLMs，它们在海量异构数据集上经过无监督预训练而成，具有处理复杂任务的强大能力。近年来，多项研究[17-20]开始采用 LLMs 进行漏洞检测。特别是部分研究[4-5]运用流行的微调技术（如提示学习和思维链[21]）引入与漏洞相关的领域知识，有效提升了 LLMs 在该任务中的性能。

尽管基于 GNN、PTM 和 LLM 的方法在漏洞检测方面展现出良好性能，但近期研究揭示，这些模型在准确性、可靠性和鲁棒性上存在显著局限[22-27]，这表明其对漏洞语义缺乏深层理解。准确性是指模型在已知数据集上的预测正确率，它反映了模型对整体样本的判断能力，常用来衡量分类任务中正确预测的比例；然而，在样本不均衡的情境下，单纯依赖准确性可能导致误导性结果。可靠性通过 $F1$ 值来衡量，强调模型在不同类别间的不均衡样本上的表

① https://microsoft.github.io/CodeXGLUE

现，尤其是在处理正负样本分布失衡时，*F*1 值能够综合考虑精确率和召回率，避免单一指标的偏差。鲁棒性则关注模型在面对极为相似的样本或对抗攻击时的稳定性和应对能力，在漏洞检测任务中，当漏洞样本与非漏洞样本非常相似，鲁棒性的强弱决定了模型是否能够区分它们。例如，现有深度学习模型往往难以区分漏洞代码与其修复后的版本[25]，因为它们倾向于过度拟合虚假特征(如特定代码标识符、代码结构等)，而非真正捕捉漏洞的根本成因[24]。此外，与常规数据集相比，这些模型在真实场景下的性能显著下降[22]，说明模型预测的漏洞标签与其对漏洞语义的实际理解之间存在脱节。传统指标(如*F*1 值)已不足以准确反映模型在理解漏洞语义方面的有效性[28]。因此，需要一种能够更精准评估漏洞语义理解能力的新指标，同时，探索使模型关注漏洞修复相关特征的方法也至关重要。

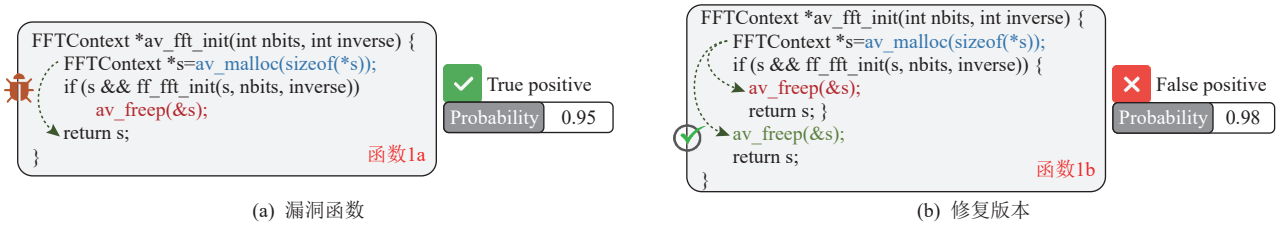
一个能够真正理解漏洞根本成因的模型应该具备一个基本且关键的能力：能够准确区分原始漏洞代码与其修复后的版本(称为配对漏洞实例)。图 1 展示了一组配对漏洞实例，具体而言，图 1(b)在所有可能路径中调用 av_freep 函数，以释放由 av_malloc 分配的内存空间，从而修复了图 1(a)中存在的内存泄漏漏洞。然而，CodeBERT 却以极高的置信度(概率≥0.95)为这 2 个样本分配了相同的漏洞标签。值得注意的是，CodeBERT 对函数 1b 的置信度甚至提高了 0.03。造成这一现象的可能原因是 av_freep 被模型捕获为一种虚假特征：模型虽然识别到了 av_freep 与样本标签之间的表面关联，却未能真正理解添加该函数对代码语义带来的本质改变。

一个鲁棒且可靠的漏洞检测模型应当能准确区分配对实例(如图 1 所示)，即既能识别出漏洞代码为有漏洞，又能识别出其对应的修复版本为无漏洞。在本研究中，将此类结果称为“成对预测”。然而，目前广泛使用的漏洞检测基准数据集(称为常规数据集)，如 Devign[7]，ReVeal[22] 和 BigVul[29]，均未包含配对漏洞实例。具体来说，这些数据集从不同来源(例如不同项目)提取漏洞样本和非漏洞样本，而未考虑 2 类样本的分布情况。大量漏洞检测方法[7,14-16]及实证研究[30-31]均基于此类基准开展，因而无法评估模型是否真正具备成对预测的能力。这一疏忽可能影响研究者对模型在真实世界中检测漏洞能力的全面理解。近期已有研究[25,28,32]注意到该问题，并对现有模型的成对预测能力进行了评估。然而，这些研究对模型性能的分析仍停留在表面，既缺乏基于大规模数据集的全面评估，也未能深入分析性能下降的原因，更未详细探讨不同模型对哪些类型的样本具有更高的预测准确性。此外，目前尚未提出解决上述局限性的实用方案。

本研究旨在填补这一空白。具体而言，首先构建了一个大规模真实世界的数据集 VulFixed，其中包含漏洞代码及其对应的修复版本。此类信息提取自漏洞补丁，这些补丁记录了修复漏洞过程中的代码变更，蕴含丰富的漏洞语义信息。随后在 4 种主流的基于 GNN 的方法和 4 种广泛使用的基于 PTM 的方法上开展实验。首先，本文评估并比较了不同模型实现成对预测的能力；继而通过细致分析成对预测样本的数据特征(如漏洞类型)，并结合模型可解释性分析，深入探究了现有模型在特定场景下表现不佳的原因。

本研究主要获得 4 个重要发现：1)基于现有基准数据集训练时，SOTA 模型难以有效区分漏洞函数和其对应修复版本；2)当训练数据中引入足够比例的配对实例后，模型的性能显著提升，但其最终预测准确率仍然较低，这表明从具有相似模式的样本中学习漏洞特征仍具有挑战性；3)特定类型的样本更易被模型学习，包括某些漏洞类型(如权限提升类漏洞)以及具有较高漏洞复杂性的样本(即修复补丁中变更行数较多的漏洞)；4)基于 PTM 的模型往往根据全局特征进行预测，缺乏对关键局部细节的关注，而漏洞通常仅与代码中的小部分代码相关。

基于上述发现，本文提出 VulGraphPTM 方法，该方法通过融合 GNN 与 PTM 进行漏洞检测，旨在达到



(a) 漏洞函数 (b) 修复版本

Fig. 1 Prediction results of CodeBERT towards a vulnerable function and the repaired version

图 1 CodeBERT 对漏洞函数和修复版本的预测结果

更优的成对预测性能。具体来说通过增强模型对局部信息(尤其是与漏洞语义密切相关的信息)的表达能力来实现该目标。GNN能够提供代码图结构的卷积特征(通过聚合各节点周边信息获得的局部表征),从而弥补PTM因忽略代码图结构而造成的信息损失。通过对齐GNN与PTM,模型能够同时捕捉漏洞的整体特征和局部修改,而局部修改对于准确预测配对漏洞实例至关重要。具体而言,将CodePTM的token映射至代码属性图(code property graph, CPG)的节点上,从而将PTM提供的整体语义理解与GNN提供的局部结构信息进行融合。该方法增强了模型理解和表达关键代码语句的能力。随后采用节点级注意力机制提取关键语句的信息,以减轻无关代码的干扰,从而提升漏洞检测的效果和区分配对实例的能力。本研究在自构建的VulFixed数据集上评估了VulGraphPTM的有效性,重点关注其区分配对漏洞实例的能力。本文将VulGraphPTM与8个现有的SOTA模型进行了比较,评估结果显示,VulGraphPTM的平均*F*1分数达到66.42%,超过了所有基线模型。更重要的是,该模型在36.86%的案例中对配对实例做出成对预测,比最先进的UniXcoder模型的成对预测率(pairwise predictions, PP)高出58.47%。

综上所述,本文的主要贡献可概括如下:

1)原创性。本研究针对配对漏洞实例分类任务进行大规模数据集上的深度分析,揭示了模型性能差异背后的关键因素。通过对模型行为的详细探索,为漏洞检测领域提供了新的思路,尤其是在漏洞修复前后对版本区分的精度与鲁棒性方面的深入洞察。

2)全面评估与对比。本文系统地评估了8种主流基于学习的漏洞检测模型,尤其是在配对漏洞实例分类任务中的表现,并深入分析了模型性能差异的内在原因。通过这项广泛的实验,本文不仅为现有方法的优缺点提供了系统性评价,还为后续的模型改进指明了方向。

3)方法创新。本文提出了VulGraphPTM,一种新的分类模型,专门针对配对漏洞实例的区分任务。这一方法不仅在性能上超越了当前最先进的模型,而且通过新的设计理念解决了传统方法在处理漏洞函数与修复版本时常见的鲁棒性问题,显著提升了分类精度和稳定性。

4)开源贡献。为了促进学术界和工业界的进一步研究和应用,本文将所开发的工具和构建的基准数据集进行开源,相关资源可通过以下链接获取:https://github.com/CGCL-codes/VulGraphPTM。

1 相关工作

本文总结了几项具有代表性的漏洞检测实证研究。Steenhoek等人[31]调查并复现了9种最先进的深度学习模型,并在Devign和MSR这2个广泛使用的漏洞检测数据集上开展研究。他们研究发现不同模型的输出一致性较低。此外,还深入对比了针对特定类型漏洞训练的模型与一次性训练以检测所有漏洞的模型,以及深度学习可能难以处理的程序类型。Gao等人[33]开展了基于LLM的漏洞检测实证研究。他们构建了一个全面的漏洞检测基准数据集VulBench,整合了来自各类夺旗竞赛(capture the flag, CTF)和实际应用的高质量数据。该研究探索了16种LLM在漏洞检测中的性能,以深度学习方法和静态分析工具为基准进行对比,揭示了LLM在漏洞检测领域的潜力。这些相关研究广泛探讨了从传统深度学习到新兴LLM在漏洞检测中的表现。然而,它们的分析局限于由不同来源的正负样本组成的数据集,未考虑漏洞与其对应补丁间的相互作用。直至近期,仅有少数研究关注模型在漏洞对分类任务中的性能,且仅进行浅层讨论而缺乏深入分析。Ding等人[28]探索了5种代码语言模型以及GPT-3.5[34]和GPT-4[35]在配对漏洞实例分类任务中的表现,发现所有模型均表现不佳。他们还发现,改变数据集中漏洞样本比例及引入对比学习[36]并不能改善模型性能。但该研究未深入探究具有不同特征的漏洞对之间的性能差异,也未分析模型表现不佳的根本原因。

本研究针对配对漏洞实例开展了大规模实证研究,从数据特征和模型可解释性等多维度对模型性能进行全面探究,从而为现有研究提供了新的见解。

2 研究设计

本节将介绍实验设计,包括选定的基准数据集、基线模型、实现细节以及目标评估指标。随后将介绍设定的研究问题及相应的实验设置。

2.1 数据集

为验证SOTA模型在配对漏洞实例上的性能,本研究基于BigVul[29]构建了一个数据集VulFixed,其中包含漏洞函数及其对应修复版本。具体而言,先根据提交信息得到BigVul中每个漏洞函数对应的修复函数,随后排除了重复的函数对以及存在明显语法

错误的函数对。

此外，本文用 Li 等人[37]发布的数据集中的实例来扩充 VulFixed。该数据集提取自美国国家漏洞数据库(NVD)①，包含真实世界中的漏洞样本及其修复版本，可进一步提升 VulFixed 多样性。最终获得 14 117 对漏洞函数与对应修复版本，共计 28 234 个样本。

本文引入 2 个常规基准数据集用于对比：Devign[7]与 BigVul[29]。这 2 个数据集被广泛使用于漏洞检测领域[12,19,37-38]，但均不包含修复后的函数。本研究利用这些数据集校准 SOTA 模型的原始性能，并将其与 VulFixed 的性能进行对比。具体而言，Devign[7]从 FFmpeg 和 Qemu 两个开源项目中收集真实世界的漏洞函数，从而确保样本的高质量。BigVul[29]则整合了来自通用漏洞披露(common vulnerability and exposures, CVE)数据库的代码修复提交，覆盖超过 300 个开源项目，使其在数据多样性和规模上成为最大的开源 C/C++漏洞数据集之一。

为抵消数据不平衡对评估结果造成的偏差，本研究进一步对 Devign 和 BigVul 数据集执行样本平衡操作(将正负样本比例控制在 1∶1)。经处理后，Devign 数据集包含 24 920 个代码函数，BigVul 数据集包含 18 236 个样本。在此过程中，还从数据中过滤了代码注释、重复样本对以及存在明显语法错误的样本，获得了 VulFixed 数据集。表 1 展示了这 3 个数据集的摘要信息。为理解现有数据集 Devign、BigVul 和 VulFixed 中正负样本间的特征差异，本文开展了初步研究以比较其相似度。图 2 展示了不同数据集中对立标签样本间的最大相似度分布。具体来说计算了每个样本与其对立标签样本之间的最大 Jaccard 相似度(反之亦然)。更高相似度的样本能使模型在训练过程中学习漏洞代码与修复版本之间的细微差异(即漏洞的本质特征)，从而提升模型性能。如图 2 所示，BigVul 和 Devign 数据集中相似度低于 0.5 的样本比例均超过 50%，其中 BigVul 数据显示 94.51% 的样本相似度低于 0.5。相比之下，VulFixed 数据集中 73.54% 的样本相似度高于 0.9，而 BigVul 和 Devign 的

Table 1 Summary of the Datasets

表 1 数据集汇总

数据集	Vulnerable	Non-Vul	CWEs
Devign	12 460	12 460	
BigVul	9 118	9 118	91
VulFixed	14 117	14 117	105

① https://nvd.nist.gov/

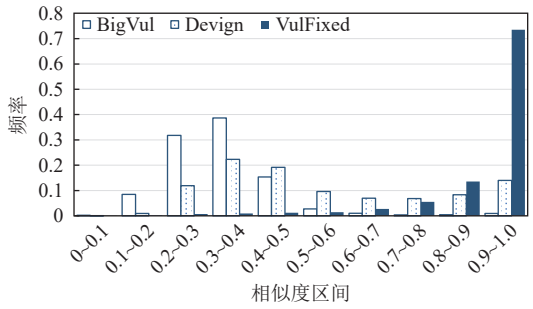


Fig. 2 Distribution of maximum similarity between positive and negative samples (histogram)

图 2 正负样本间最大相似度频率分布直方图

该比例分别仅为 0.95% 和 13.98%。这表明 Devign 和 BigVul 数据集中的正负样本特征存在较大差异，因此，基于这些数据集的评估难以全面反映模型是否真正学习到了漏洞语义，抑或仅仅对数据集中的差异性特征产生了过拟合。

2.2 学习模型选择

在选择基线方法时，本文考虑了多种模型结构与数据处理技术，以拓展研究结果的广度。为获取 SOTA 方法，本文参考了微软公司的 CodeXGLUE 排行榜和近期的一项实证研究[31]。最终从基于 GNN 和基于 PTM 的方法中分别选取了 4 种 SOTA 模型作为基线。下文将详细介绍各基线的详细信息。

2.2.1 基于 GNN 的方法

1) Devign[7]。Devign 从代码函数中提取代码属性图与自然代码序列(nature code sequence, NCS)，构建代码组合图作为输入，采用门控图神经网络(GGNN)[39]学习并聚合代码信息，最终生成针对代码函数的预测。

2) ReVeal[22]。ReVeal 主要由 2 个部分组成：首先通过 GGNN 学习图结构以获得中间表示，随后输入至一个线性分类层学习并生成分类结果。

3) AMPLE[8]。AMPLE 在多层图结构上使用带有多头注意力机制[40]的关系图卷积网络(RGCN[41])来学习中间层信息，继而利用 2 种类型的卷积层捕获语义信息并输出分类结果。

4) ReGVD[16]。ReGVD 利用 GraphCodeBERT[38]来嵌入代码函数文本，从而引入代码先验知识。但由于代码嵌入最终需输入至 GNN 进行学习并输出分类结果，本文仍将 ReGVD 归类为基于 GNN 的方法。

2.2.2 基于 PTM 的方法

VulBERTa 是基于 Transformer 框架的 RoBERTa[42]的扩展模型，它通过在大量代码数据上训练掩码语

言建模(mask language modeling, MLM)任务来获取基础代码知识。随后，VulBERTa 分别与 MLP 和 CNN 连接进行微调，最终形成 2 个版本的检测模型 VulBERTa-MLP 和 VulBERTa-CNN[15]。

1)LineVul[14]。LineVul 是基于 CodeBERT[11] 构建的 Transformer 架构模型，并在 BigVul 数据集上进行了微调。

2)UniXcoder[12]。UniXcoder 是一种基于 Transformer 的模型，其输入为经代码注释和序列化抽象语法树增强的代码表示。UniXcoder 提出了 2 种新的预训练任务：多模态对比学习和跨模态生成，以增强模型对代码语义的理解。

2.3 实现细节

与现有研究保持一致，本文采用 CodeXGLUE[43] 的数据划分策略处理 Devign 数据集。对于 BigVul 和 VulFixed 数据集，按 8∶1∶1 的比例将其随机划分为训练集、验证集和测试集。特别地，实验确保 VulFixed 数据集中每个漏洞函数对应的修复版本被划分到同一子集。此外，为保证实验的公平性和鲁棒性，所有模型均使用相同的划分数据集。本文遵循现有的研究[7,22,31,43] 处理输入。具体而言，对于基于 GNN 的方法，使用 Joern[10] 生成代码属性图(CPGs)；对于基于 PTM 的方法，则直接使用代码文本作为输入。由于 Devign 的作者未公开源代码，本研究中使用的是 Chakraborty 等人[22] 复现的版本①。所有模型均采用其对应论文中提供的默认参数。对于基于 GNN 的模型，学习率设置为 1E−4，批次大小为 128；若 $F1$ 分数在 50 次迭代内未得到提升，则提前终止训练。对于基于 PTM 的模型，学习率设置为 2E−5，批次大小为 8，训练轮次为 10。所有模型均使用 Adam 作为优化器，且为保证结果的稳定可复现性，种子统一设置为 12 345。

2.4 性能指标

本实验采用以下广泛使用的评估指标[8,14,15,24,30,31,44-46]，并从正确性、可靠性和鲁棒性 3 个维度进行深入探讨，以更全面地评估模型性能。

1) 准确率(accuracy, *Acc*)

该指标是衡量模型整体预测准确性的基本指标，计算公式为

$$Acc=\frac{TP+TN}{TP+TN+FN+FP}。$$

其中，*TP* 为真正例，*TN* 为真反例，*FP* 为假正例，*FN* 为假反例。准确率反映了模型在所有样本中正确预测的比例，其值越高表示模型越能够在样本上做出正确的判断。在二分类任务中，尽管准确率是最常用的指标，但其对于样本不均衡问题的敏感性较高，因此需要结合其他指标来综合评价模型性能。

2) 可靠性(reliability)

$F1$ 分数作为可靠性评估的核心指标[7,37,47-48]，能够有效反映在样本不均衡时模型的表现。$F1$ 值是精确率(*Precision*)和召回率(*Recall*)的调和平均值，公式为

$$F1=2\times\frac{Precision\times Recall}{Precision+Recall}。$$

其中，精确率表示被检测出的正样本中，实际为正样本的比例；召回率表示所有正样本中被正确检测出的比例。$F1$ 值综合了这 2 个指标，能更好地衡量模型在处理正负样本不均衡时的可靠性。

3) 鲁棒性(robustness)

本实验特别关注模型在漏洞函数与修复版本的区分能力上，即如何在面对相似或几乎相同的样本时保持鲁棒性。理想的模型应能够有效区分漏洞函数和修复版本，预测漏洞函数为有漏洞，预测修复版本为无漏洞。为此，本文设计了以下 2 个与鲁棒性相关的指标。

①成对预测率(pairwise predictions, *PP*)。该指标衡量模型做出正确成对预测的概率。公式为

$$PP=\frac{BC}{BV+BN+BC+BI}。$$

PP 值越高，表示模型在成对预测中表现越好，能够正确区分漏洞函数与修复版本，鲁棒性越强。其中，4 种预测结果包括：

BV 表示 2 个版本均被预测为有漏洞；

BN 表示 2 个版本均被预测为无漏洞；

BC 表示 2 个版本均被正确预测(即漏洞函数为有漏洞，修复版本为无漏洞)；

BI 表示 2 个版本均被错误预测(即漏洞函数为无漏洞，修复版本为有漏洞)。

②相同预测率(same predictions, *SP*)。该指标衡量模型对配对实例做出相同预测的概率。公式为

$$SP=\frac{BV+BN}{BV+BN+BC+BI}。$$

SP 值越高，表示模型更倾向于对配对实例做出相同的预测，即漏洞函数和修复版本被视为相似或等效，这反映了模型在区分这 2 类实例时的弱鲁棒

① https://github.com/saikat107/Devign

性,性能较差。

2.5 研究问题

本研究旨在通过解答以下研究问题,系统地了解模型在识别漏洞语义方面的性能。

RQ1: 现有模型能否区分漏洞函数与其对应修复版本?现有研究通常采用相对独立的正负实例(例如来源不同)评估模型有效性。然而,这些模型容易过拟合与漏洞语义无关的数据集特定特征,导致在这些数据集上性能虚高。因此,本研究通过在更严格的设置中评估现有模型,检验其是否能实现成对预测,即标记漏洞函数为有漏洞,标记修复版为无漏洞。值得注意的是,本文在现有数据集上训练模型以保持与先前研究的一致性,旨在探究现有模型能否识别漏洞语义。具体而言,分别在 Devign 和 BigVul 上训练模型,随后在 VulFixed 上进行统一测试(因其包含配对漏洞实例,参见 2.1 节)。研究选择 VulFixed 作为测试数据集是基于双重考量: Devign 与 VulFixed 完全源自不同数据源,有助于深入了解不同代码风格对模型学习能力的影响; BigVul 与 VulFixed 采用相似数据源且正样本存在重叠(参见 2.1 节),但负样本有所不同,这有助于实现漏洞修复样本与常规良性样本间的对比。

RQ2: 配对漏洞实例信息能否增强现有模型性能?本文通过在 VulFixed 数据集上进行统一训练与测试来探究该问题,其中每个漏洞的修复版本在模型训练阶段均可获得。此外,为进一步探究漏洞修复良性样本与常规良性样本对模型性能的贡献,本研究通过调整训练集的良性子集中这 2 类样本的比例,构建出一系列具有不同漏洞修复样本占比(0%~100%)的数据集。具体通过用常规良性样本替换 VulFixed 中特定比例的漏洞修复实例来构建这些数据集,并确保所有良性样本无重复以避免数据泄露。本文基于 UniXcoder 训练了一系列模型,并在 BigVul 和 VulFixed 上评估其性能。这一过程用于交叉验证 2 类负样本对模型在 2 个数据集上性能的影响。

RQ3: 哪些类型的漏洞实例更容易被现有模型成对预测?现有模型对不同类型漏洞的成对预测能力可能存在差异。本问题旨在探索哪些类型的漏洞实例可以被现有模型更好地预测。具体从漏洞类型与漏洞复杂度 2 个维度展开分析,分析主要基于 VulFixed 数据集进行。在漏洞类型方面,本文遵循现有工作[31],根据 CWE ID 将实例划分为 7 类,见表 2;漏洞复杂性主要通过修复补丁的大小衡量,该值基于补丁差异的变更行数提取。为确保复杂度评估的准确性,补丁中移除了注释和空行等无关内容。随后从这 2 个维度分析现有模型的性能。

Table 2 Overview of the Vulnerability Categories with Detailed CWE Information

表 2 漏洞类别概览及详细 CWE 信息

漏洞类型	CWE IDs
权限提升	CWE-264,CWE-284,CWE-269…(9)
条件错误	CWE-362,CWE-835,CWE-617,CWE-755
内存错误	CWE-119,CWE-125,CWE-787,CWE-120
输入验证错误	CWE-20,CWE-79,CWE-94…(9)
值错误	CWE-189,CWE-476,CWE-190…(6)
资源错误	CWE-416,CWE-415,CWE-339…(10)
其他	CWE-200,CWE-310,CWE-254…(9)

注:( ) 内的数字表示未列举出来的 CWE 数量。

RQ4: 现有模型在预测过程中是否关注细微的漏洞语义?在本问题中,研究深入分析现有模型在预测过程中的注意力分布,以提供一定程度的模型可解释性。本文主要基于 PTMs 开展分析,因为其自注意力机制有助于获取每个 token 对预测结果的贡献度。具体从行级和 token 级 2 个层面计算每个函数的注意力分数: 在行级别,比较修改行与未修改行之间、漏洞函数与其补丁版本之间的注意力分数,并采用交并比(intersection over union, IoU)分数和肯德尔相关系数(Kendall's tau correlation coefficient)作为这 2 种场景的评估指标。IoU 分数常用于细粒度漏洞检测任务[49-50]以衡量预测区域与漏洞行之间的关联程度; 肯德尔相关系数则用于衡量配对漏洞实例间注意力排序的相似度,取值为 1 表示完全正相关, −1 表示完全负相关。在 token 级别,将所有 token 划分为四大类(即标识符、关键字、运算符和代码结构 token),并考察模型注意力在各类别中的分布情况。

3 实证研究结果

3.1 RQ1: 成对预测性能分析

首先,本研究聚焦于基于常规数据集(即 Devign 和 BigVul)训练的模型,与漏洞检测领域的现有研究保持一致。研究结果为现有工作提供了更加精确且具有互补性的新视角。表 3 展示了各基线模型的性能表现,通过对结果的细致分析,得出以下关键发现:

1)所有选择的 SOTA 模型在成对预测能力方面表现欠佳。具体而言,在 Devign 上训练的模型最高 *PP* 值仅为 7.45%(即 AMPLE 模型),在 BigVul 上该数

Table 3 Performance of Models on Devign, BigVul and VulFixed Test Sets

表 3 Devign, BigVul 和 VulFixed 测试集上的模型性能 %

模型	Devign				BigVul				VulFixed			
	Acc↑	*F*1↑	*SP*↓	*PP*↑	*Acc*↑	*F*1↑	*SP*↓	*PP*↑	*Acc*↑	*F*1↑	*SP*↓	*PP*↑
Devign	49.41	25.81	88.50	5.59	49.95	62.88	93.10	3.29	51.19	46.64	**65.28**	18.07
ReVeal	49.62	25.77	93.32	2.96	49.95	**66.01**	98.25	0.44	51.78	57.21	87.62	7.56
AMPLE	49.51	40.10	**84.12**	**7.45**	50.11	50.43	**85.98**	**7.34**	55.45	51.15	70.76	19.72
ReGVD	49.72	55.27	93.29	3.07	49.56	59.48	94.86	2.13	59.80	59.11	73.40	23.13
GNN-average	49.57	36.74	89.81	4.77	49.89	59.70	93.05	3.30	54.56	53.53	74.27	17.12
VulBERTa-MLP	49.40	52.92	87.89	4.45	50.68	43.81	94.95	2.99	**61.50**	62.83	71.84	23.08
VulBERTa-CNN	**50.66**	51.45	86.01	6.15	**50.76**	49.86	95.11	3.22	59.75	48.89	76.56	21.57
LineVul	50.31	**66.71**	97.68	1.88	50.62	45.62	96.05	2.64	51.54	62.31	77.96	17.55
UniXcoder	49.72	52.99	94.54	2.45	50.74	53.33	97.75	1.86	59.83	**65.85**	73.48	**23.26**
PTM-average	50.02	56.02	91.53	3.73	50.70	48.16	95.97	2.68	58.16	59.97	74.96	21.37
平均	49.79	46.31	90.67	4.25	50.30	53.93	94.51	2.99	56.36	56.75	74.61	19.24

注: “↑”表示该指标为正向指标，即数值越大性能越优。“↓”表示该指标为负向指标，即数值越小性能越优。黑体数值表示最优值。

值为 7.34%(亦为 AMPLE)。相反，大多数模型的 *SP* 得分超过 90%，表明其难以区分漏洞函数与修复版本。漏洞对中的代码变更与漏洞语义之间的直接关联表明，并非所有模型都能真正理解漏洞本质，而可能是基于其他特征生成预测。现有研究在常规数据集上报告出表面高性能(虚高性能)(在 BigVul 上 *F*1 值最高达 91%[14])，这表明仅用常规数据集评估模型性能可能无法充分反映模型对漏洞底层语义理解程度，凸显了像 VulFixed 这样的专用成对数据集的重要性。

2) *F*1 值与 *PP* 指标并非总是呈正相关。实际上，结果显示二者存在一定程度的负相关: 在 Devign 数据集上，LineVul 的 *F*1 分数最高，但 *PP* 值仅为 1.88%，而 AMPLE 虽在 *F*1 值上未超越任何基于 PTM 的模型，却取得了 7.45% 的最高 *PP* 值。在基于 PTM 模型中，VulBERTa-CNN 的 *F*1 分数最低，但其 *PP* 值最高。模型在 BigVul 数据集上的结果也呈现出类似特征。*F*1 值衡量的是模型预测与真实标签的吻合度，但未必能反映模型对漏洞语义的理解能力，因为模型的预测可能并非始终依赖漏洞相关特征。Ding 等人[6]也指出 *F*1 分数并非评估实际场景中漏洞检测能力的理想指标，因其受多重因素影响，且可能高估模型性能。因此，本文主张将 *PP* 值作为漏洞检测评估指标的重要组成部分，相比 *F*1 值它能提供更直观且严谨的模型性能衡量标准。

研究发现 1。主流模型在理解漏洞根本原因方面能力有限，这阻碍了其区分漏洞函数与修复版本。此外，*F*1 分数与 *PP* 指标之间不存在相关性，因此在评估基于学习的漏洞检测模型时，必须将 *PP* 作为 *F*1 指标的补充度量。

3.2 RQ2: 配对实例的性能增强效果

本研究使用 VulFixed 数据集对 8 个模型重新训练，以探究在训练数据中加入配对实例能否提升模型性能。结果汇总于表 3。

首先可得出一个明确结论: 在融入漏洞修复信息后，模型区分配对漏洞实例的能力显著提升。具体而言，与使用 Devign 和 BigVul 训练相比，基于 VulFixed 训练的模型的 *PP* 值显著提升，平均达到 19.24%。其中 ReGVD、VulBERTa-MLP 和 UniXcoder 的 *PP* 提升幅度均超过 20 个百分点(分别提升 21.00 个百分点 (23.13−2.13)、20.09 个百分点 (23.08−2.99) 和 21.40 个百分点 (23.26−1.86))。进一步分析表明，模型开始更加关注配对漏洞实例之间的差异。具体而言，本研究针对 VulFixed 中包含的每一组漏洞配对，获取了模型在多种不同的训练数据集上生成的预测置信度。图 3 展示了基于 GNN 和 PTM 模型的典型结果: 各图中黄线表示对漏洞函数的预测置信度曲线(按升序排列)，蓝线表示对相应修复版本的预测置信度曲线。所有模型均根据特定阈值(通常为 0.5)将预测置信度划为具体标签。因此，成对预测表现为黄线位于阈值上方，且同一 x 轴值对应的蓝线位于阈值下方。结果显示，基于 VulFixed 训练的模型具有更多呈现这种分布的样本点。此外，与其他数据集相比，这 2 条曲线间的预测概率波动明显更大，即基于 VulFixed 训练的模型 2 条曲线围成的面积为 16.983 1，而基于 Devign 和 BigVul 训练的模型对

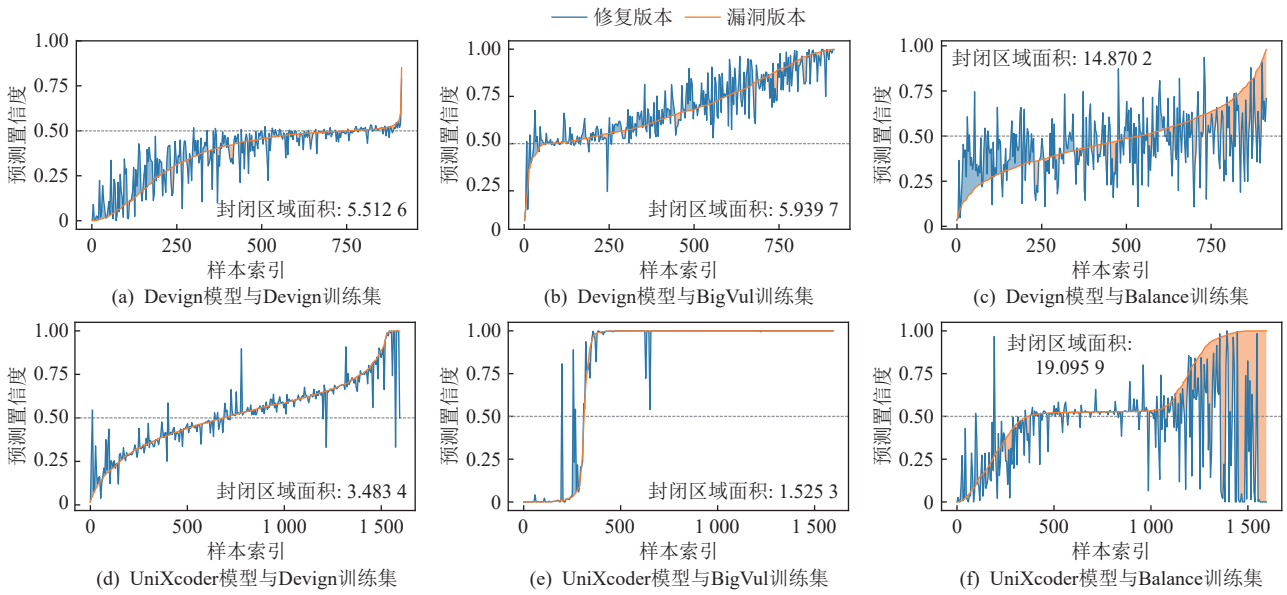


Fig. 3 Prediction probability perturbation curves of baseline models on VulFixed test set

图 3 基线模型在 VulFixed 测试集上的预测概率扰动曲线

应的面积分别仅为 4.498 0 和 3.732 5。该值越大，表明模型越倾向于对配对实例做出不同预测，这意味着模型对漏洞补丁行的关注度更高。

然而，区分漏洞代码与修复版本仍然具有挑战性。因为大多数模型的 *SP* 指标仍高于 70%。相当一部分样本的预测置信度接近阈值，如图 3(c)与图 3(f)所示。此外，模型难以同时在常规样本和配对漏洞实例上取得高性能。如图 4 所示，在由配对漏洞实例与常规样本(即来源不同的漏洞样本和良性样本)混合组成的训练数据集中，配对实例比例越高，模型在常规数据集(即 BigVul 测试集)上的性能越差，同时其在配对数据集(即 VulFixed 测试集)上的性能则越好。当配对实例比例超过 60% 时，这一趋势更为明显。由于 *PP* 能更好地反映模型对漏洞语义的理解能力，在训练过程中，它的优先级应比准确率和

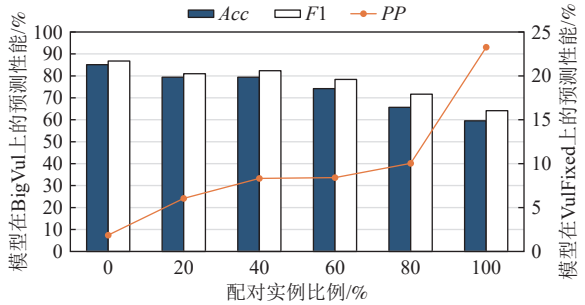


Fig. 4 Performance of UniXcoder trained on various datasets with different portion of paired vulnerability instances

图 4 基于不同比例配对漏洞实例数据集上训练的 UniXcoder 性能

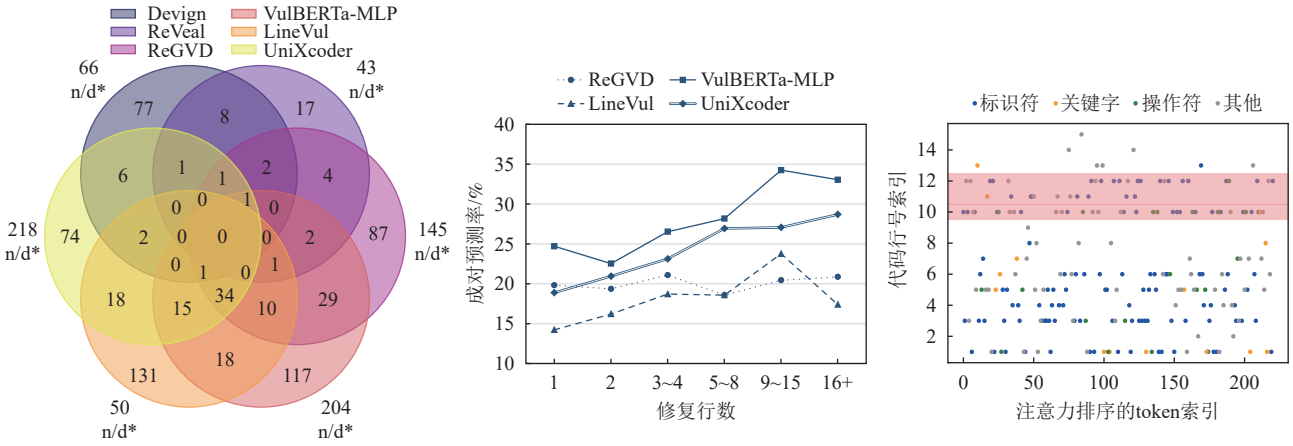
*F*1 分数略高。因此实验必须慎重考虑配对实例的比例。根据图 4 评估结果，若要确保 *PP* 超过 10%，配对实例比例需超过 80%；若希望模型在区分配对漏洞实例的高难度场景中达到最优性能，配对实例比例应为 100%。但即使排除所有常规样本，模型的 *PP* 也只能达到 23%。由此可见，在确保模型对常规样本预测性能不显著下降的前提下，亟需增强其区分配对漏洞实例的能力。

研究发现 2。在训练过程中引入配对漏洞实例能够增强模型区分漏洞函数及其修复版本的能力，因此有必要在训练集中包含足量的漏洞修复样本。

3.3 RQ3: 成对预测实例的深度解析

尽管大多数模型的 *PP* 值相对接近，但其可能提取到了不同的特征。如图 5(a)所示，没有一个被成对预测的样本能得到所有模型一致认可，这意味着寻找成对预测样本间的共同特征极具挑战性。具体而言，基于 GNN 的模型预测结果存在显著偏差，仅有少量样本(约 1%)被所有模型一致正确预测；基于 PTM 的模型表现出更好性能，有 50 个样本(约占总样本 7%)被所有模型正确识别。值得一提的是，通过引入 PTM 编码器，ReGVD 与 PTMs 间的交集显著大于其与 GNN 模型的交集，这表明预训练代码的先验知识对模型学习具有积极影响。因此，后续的分析将主要聚焦于 PTMs 和 ReGVD。

首先，模型对不同漏洞类型样本的性能表现存在差异。表 2 展示了根据 CWE 编号进行的漏洞分类，图 6 展示了 7 种漏洞类型下成对预测的样本占所有



(a) 基于GNN与PTM的检测模型对样本的预测交互情况　(b) 不同修复行数下的*PP*分布　(c) CVE-2018-11645函数中按注意力得分降序排列的token排序分布

Fig. 5　Detailed analysis towards pairwise predicted samples

图 5　对成对预测样本的详细分析

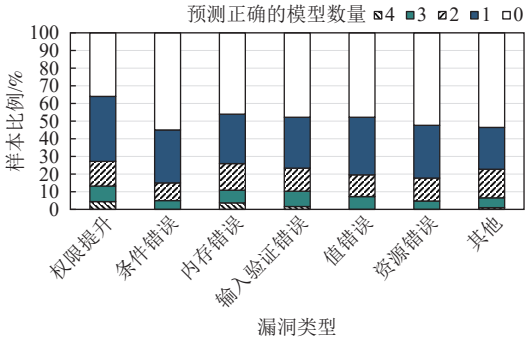


Fig. 6　Proportion of multi-model correct predictions across vulnerability types

图 6　不同漏洞类型中多模型预测正确比例

样本的比例，图例中的不同颜色和数字代表当前漏洞样本被多少个模型共同成对预测了。这里统计的是 ReGVD、VulBERTa-MLP、LineVul 和 UniXcoder 四个模型的结果。表现最佳的 2 个类别是权限提升和内存错误，其样本覆盖率分别达到 63.97% 和 53.94%。其次是输入验证错误和值错误，两者表现相近，覆盖度均为 52.17%。权限提升与权限管理、特权及其他执行访问控制相关的安全功能有关。内存错误主要包含不正确的内存访问，例如越界读取或写入。实际上，这些错误通常被认为难以检测[51-52]，因为它们与程序执行过程中变量/对象的状态紧密相关，且仅在特定条件下触发。对这些漏洞类别的进一步分析表明，模型性能更优可能与代码修复信息相关。对于访问控制漏洞（CWE-264），许多样本呈现相似的修复模式：要么移除权限相关的小型代码段，要么添加条件限制。这使得漏洞样本与其修复版本间形成了一致的模式。同样，内存错误中的缓冲区边界溢出漏洞（CWE-119）也展现出相对统一的修复方式：添加缓冲区边界的条件检查。模型在对漏洞和修复样本进行联合训练时，很可能识别出这些独特特征，并将其作为预测依据。然而，对条件错误类漏洞的分析显示，其修复模式更为多样且样本数量较少，这可能是导致训练效果较差的原因。

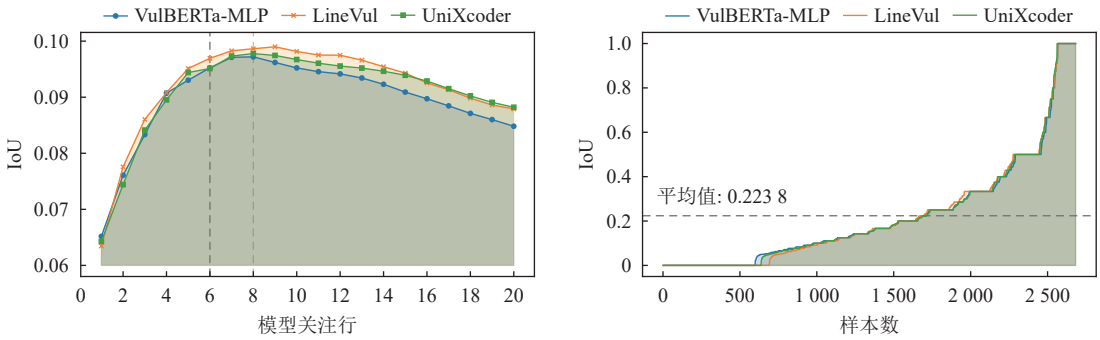
其次，当配对实例的漏洞复杂性（即修复行数）增加时，模型预测性能会有所提升。如图 5(b)所示，当修复行数大于 8 时，几乎所有模型的 *PP* 值均高于修复行数小于 3 的情况。这一整体上升趋势表明，更高的漏洞复杂性能为模型提供更多可学习信息，使其能更好识别与漏洞相关的模式特征。此外，代码变更数量的增加会让漏洞特征更加明显，从而使模型更有效地区分漏洞代码与正常代码。

研究发现 3。具有特定漏洞类型（如权限提升）和更高漏洞复杂性（即变更行数超过 8）的实例，相对更容易被现有学习模型成对预测。

3.4　RQ4: 代码预训练模型的注意力分布

本文从行级和 token 级 2 个粒度开展详细分析，以探究模型关注区域与漏洞语义之间的关系。

在行级层面，本文发现模型对漏洞行的关注极少。图 7(a)展示了模型关注的代码行与补丁行在不同 K 值下的平均 IoU 分数。所有模型的得分均低于 0.10，表明模型的理解与漏洞语义之间存在显著偏差。当 K=8 时，模型整体上达到最大 IoU 值。在评估中，这与模型首次预测到正确漏洞行的平均排名相近。图 7(b)显示，即使仅考虑每个样本的最大 IoU 值，3 个模型的平均性能也仅为 22.38%。所有模型中，IoU 值高于 0.5 的样本占比不足 15%，IoU 值为 0 的样本



(a) 注意力得分Top-*K*行与补丁行间的平均IoU分数 (b) 各样本的最大IoU值分布图

Fig. 7 Overlap between the highest-attention line of model and the vulnerability line

图 7 模型最高注意力行与漏洞行之间的重合情况

占比约为 24%, 这凸显出现有方法在漏洞行预测方面仍然具有不足。对于配对实例, 漏洞行之外的部分保持一致, 这导致模型对漏洞函数及其修复版本的注意力分布几乎相同。表 4 展现了各模型在不同 *K* 值下的平均肯德尔系数, 结果显示几乎所有平均肯德尔系数都接近或高于 0.7, 表明配对实例间的注意力排序具有高度一致性。这说明 PTMs 倾向于基于代码整体特征进行预测, 而忽略了与漏洞相关的细微差异。

Table 4 Average Kendall Coefficient *KD@K* for Top-*K* Attention Lines

表 4 注意力 Top-*K* 行的平均肯德尔系数 *KD@K*

模型	*KD@5*↓	*KD@8*↓	*KD@10*↓	*KD@20*↓
VulBERTa-MLP	0.734	0.709	0.697	0.658
LineVul	0.788	0.765	0.744	0.701
UniXcoder	0.785	0.757	0.744	0.697
VulGraphPTM	**0.673**	**0.649**	**0.631**	**0.628**

注: 采用 Kendall 秩相关检验评估配对漏洞样本注意力排序之间的相关性, 并通过 p 值判断统计显著性 (p<0.05)。"↓"表示该指标为负向指标, 即数值越小性能越优。黑体数值表示最优值。

在 token 级层面, 模型会关注大量与漏洞语义无关的 token, 其中近半数为代码结构 token。研究提取了注意力最高的 76 个 token(该数值为数据集中 8 行代码的平均 token 数), 并计算它们与漏洞行的重叠度。结果发现平均仅有 8 个 token 出现在漏洞行中, 且这些 token 的平均排名为 52 位, 这与在行级别观察到的结果一致。更具体地说, 在这 76 个 token 中, 平均有 44.51% 为标识符(如变量/函数名、常量), 6.94% 为运算符(如逻辑运算符、算术运算符), 5.12% 为关键字(如 if、while), 剩余 43.44% 为与代码结构相关的 token(如括号、逗号、空格)。这表明模型努力理解代码结构特征, 试图确定其与样本标签的密切关联。虽然这些 token 对全面准确理解代码语义十分重要, 但它们对漏洞语义的贡献十分有限。此外, 模型关注的 token 分散在整个函数内, 缺乏对局部区域的关注。图 5(c)展示了在一个成对预测样本 CVE-2018-11645 的整个函数中, 按注意力分数排序的 token 分布情况。理想情况下, 如果模型能识别漏洞语句, 该语句内各 token 的注意力排序应较为接近, 因为它们共同构成了漏洞语义。然而在图 5(c)中并未观察到该结果, 相反, token 分布呈现出分散性与不连续性。以 CVE-2018-11645 为例, 漏洞行中有一条语句 strcmp (pname.iodev->dname, "pipe")。其中 dname 的注意力排名为 1, 而 iodev 排在 155 位, pname 排在 42 位, strcmp 排在 211 位。结果表明, 注意力排名中 2 个相邻 token 在代码行间的平均距离为 11.44 行。同一代码行内所有 token 注意力排序的平均标准差为 80.85。这表明在漏洞检测任务中, PTMs 可能并未真正理解特定代码语句的语义, 而是基于 token 间的分布关系进行推断。因此, 模型可能会基于某个特定 token 改变整体输出结果, 如图 1 所示例子。

研究发现 4。模型难以聚焦局部细节差异(即漏洞行与补丁行), 且更倾向于基于输入代码的全局特征进行预测。模型关注的 token 是分散的, 表明其缺乏对相邻和相关 token 间关系的理解。

4 VulGraphPTM 新方法

本研究系统地评估了现有 SOTA 漏洞检测模型在实现成对预测方面的能力, 结果表明现有模型在这种严格场景下存在显著局限性(研究发现 1)。通过在训练数据集中引入足量的漏洞配对样本(比例超过 80%), 可以在一定程度上提升模型性能, 但模型能达到的最高 *PP* 值仍低于 24%(研究发现 2)。

Risse 等人[25]指出，通过对数据集进行语义保留变换的数据增强方法，可能导致模型过度适应特定变换，对帮助模型捕获漏洞特征收效甚微。也就是说数据增强方法存在局限性。通过对 CodePTMs 注意力分布的进一步分析，本文推测模型的性能瓶颈可能与其特定架构相关。具体而言，注意力权重最高的 token 要么分散分布，要么分布不均衡（研究发现 4），这表明模型的自注意力机制过度关注全局分散特征而忽视局部细节。因此，模型对单条语句语义的理解有限，往往导致模型忽略与漏洞语义相关的关键局部信息，最终使 IoU 分数并不理想，最大值仅为 22.38%（研究发现 4）。

基于 PTMs 的模型通常将输入代码分割为 token 进行学习，因此基础语句总是被分解为更小的组成单元（例如将“*a*=*b*”拆分为“*a*”“=”和“*b*”）。模型只有捕获这些 token 间的关系时才能重构该语句的原始语义。然而，研究发现 4 揭示了现有模型在处理此类局部信息时存在的固有局限性，这促使我们推进模型架构的改进。值得注意的是，类似信息（即赋值语句）可以通过代码属性图（CPG）[10]的结构直接表示。CPG 对构成代码语句的基础元素（如变量、运算符）的层次结构进行编码，并提供丰富的控制流与数据流信息。基于上述考虑和研究结果，本文提出 VulGraphPTM——一个融合 CodePTM 与 GNN 的漏洞分类模型，旨在更好地区分漏洞函数及其修复版本。图 8 展示了 VulGraphPTM 的整体架构。核心思想是利用源代码的图结构信息来弥补 CodePTM 在聚合局部信息方面的不足。其中，CodePTM 学习代码 token 间的语义关系，并提供代码函数文本的全局特征，这正是其优势所在。

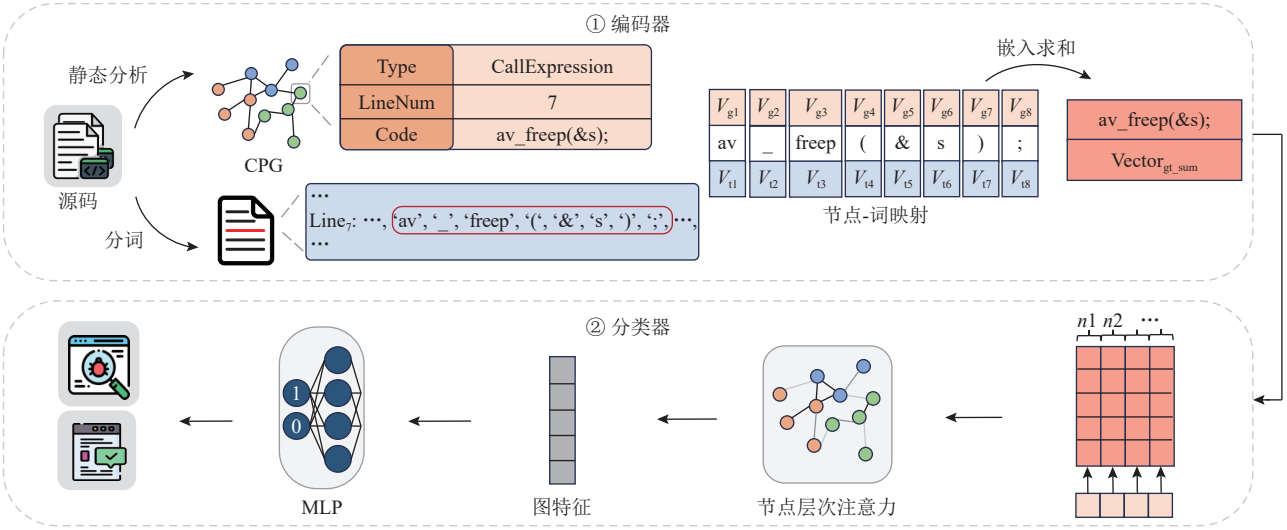


Fig. 8 Overview of VulGraphPTM

图 8 VulGraphPTM 概览

在此基础上，GNN 将重构代码的结构化信息并不断更新图中节点状态，使每个节点能整合相邻节点的局部信息。随后，本研究在图节点层面对齐 CodePTM 与 GNN 的信息，以同时获取全局和局部信息。该过程在代码属性图上进行，形成一个隐藏状态图，其中节点包含融合后的信息，同时保留大量控制流边与数据流边。这一过程通过名为“编码器”的结构实现（即图 8 的上部分）。最终，采用注意力增强的图分类器（即图 8 的下部分）获得最终模型预测结果。本文使用节点级注意力机制捕获最关键的节点信息，减少无关信息引入的噪声。下文将详细介绍这 2 个模块。

4.1 融合 CodePTM 和 GNN 的编码器

图 8 中的上部分直观展示了该编码器的结构。对于输入的代码样本，分别使用 CodePTM 的分词器和静态分析工具，将其编码为代码 token 和代码属性图。具体实现采用 UniXcoder[12]和 JOERN[10]工具。代码 token 和代码属性图分别通过 CodePTM 与 GNN 进行初始化。CodePTM 学习每个 token 相对于所有其他 token 的信息，并聚合全局信息，如式（1）所示。

$$a_{ij}=\frac{\exp\left(\boldsymbol{q}_i\times\boldsymbol{k}_j\right)}{\sum_{k=1}^{n}\exp\left(\boldsymbol{q}_i\times\boldsymbol{k}_k\right)},\tag{1}$$

其中，$\boldsymbol{q}_i$是查询向量，$\boldsymbol{k}_j$是键向量，a_{ij}是 token i对 token j的注意力权重。GNN 则学习每个节点与其相邻节点间的信息，并聚合局部信息，如式（2）所示。

$$h_i^{(t+1)}=\sigma\left(\boldsymbol{W}h_i^{(t)}+\sum_{j\in N(i)}\frac{1}{C_{ij}}\boldsymbol{W}'h_j^{(t)}\right),\tag{2}$$

其中，$h_i^{(t)}$是节点i在第t层的隐藏状态，$N(i)$是节点i的邻居，C_{ij}是归一化常数，$\boldsymbol{W}$和$\boldsymbol{W}'$是权重矩阵，σ是激活函数。研究分别采用了 UniXcoder[12] 和 R-GCN[41] 模型，其中 R-GCN 能沿不同类型的边学习图信息，这与代码属性图的结构高度契合。本文遵循了先前研究[22]的图预处理步骤。经过 CodePTM 与 GNN 处理后，这些代码 token 和代码属性图节点被转换为固定长度的高维向量，从而可通过求和运算实现信息融合。

该方法使用映射函数$M(\boldsymbol{T},\boldsymbol{N})$将每个 token 的隐藏状态聚合到对应节点的隐藏状态，如式(3)所示。

$$M(\boldsymbol{T},\boldsymbol{N}):\mathbb{R}^{d_T}\rightarrow\mathbb{R}^{d_N}\tag{3}$$

其中，d_T和d_N分别是 token 和节点的隐藏状态向量维度，映射函数$M(\boldsymbol{T},\boldsymbol{N})$将 token 的向量映射到图节点的向量空间。对于代码属性图$G(\mathcal{N},\mathcal{E})$，每个节点$n_i\in\mathcal{N}$都具有代码属性，这些属性代表属于某一代码语句的一小段代码片段。通过使用与 CodePTM 相同的分词器，代码属性图的节点可通过代码属性映射到对应的代码 token。为获取具体映射关系，本研究采用两步映射方法：首先利用行号定位同一行的 token。在分词过程中，维护一个从 token 到行号的映射表，该表与图$G(\mathcal{N},\mathcal{E})$中节点$n_i$的行号属性对应。其次，通过子串匹配定位同一行内的特定 token。在定位到的 token 中过滤掉无关 token，保留标识符、关键词和运算符这 3 类 token。此过程即构建映射$M(\boldsymbol{T},\boldsymbol{N})$的过程，其中键代表 token 索引，值对应其关联节点的索引。这种双重信息便于从各自的高维向量中提取目标隐藏状态。随后，对于与某一节点相关联的一组 token，先计算 token 隐藏状态的均值，再将其与节点隐藏状态直接相加。融合后的结构仍为图结构，因其更贴合代码的自然结构，其中保留了节点间的数据流和控制流等关键代码信息。该图的每个节点包含两类信息：来自完整上下文的信息和来自相邻节点的信息。如式(4)所示，融合后的结构将被输入到图分类层以获取最终分类结果。

$$h_i^{\text{fused}}=\alpha h_i^{\text{CodePTM}}+(1-\alpha)h_i^{\text{GNN}},\tag{4}$$

其中，h_i^{CodePTM}是由 CodePTM 提供的节点特征，h_i^{GNN}是由 GNN 提供的节点特征，α是一个超参数，控制 2 种信息源的融合比例。在学习过程中，GNN 与 CodePTM 的参数同步更新。

4.2 节点级注意力增强的图分类器

该分类器以融合后的隐藏状态作为输入并最终生成预测，其中 Readout 函数是最关键的组成部分。Readout 函数是指 GNNs 对图中节点特征进行汇总的过程，用于生成整个图的表示或预测，是图分类任务中的关键步骤。本方法在 Readout 函数中引入节点级注意力机制[38]，以过滤图中大量冗余信息。具体而言，注意力层为图中每个节点分配一个门控网络f_{gate}，该网络量化每个节点对输出的贡献程度，并在整个训练过程中持续更新。每个f_{gate}本质上是一个线性层，其将节点特征映射到该节点的注意力权重。f_{gate}与节点状态的乘积代表该节点对整体图表示的贡献，通过对所有贡献求和即可获得图的最终输出表示。该过程可通过式(5)表示。

$$r^{(i)}=\sum_{k=1}^{N_i}softmax\left(f_{\text{gate}}\left(x_k^{(i)}\right)\right)f_{\text{feat}}\left(x_k^{(i)}\right),\tag{5}$$

其中，f_{feat}表示应用于节点状态的转换函数，此处采用线性映射。注意力层的输出是代码属性图的整体表示。随后使用 MLP[53] 层将整体图表示转换为最终的分类标签，如式(6)所示。

$$\boldsymbol{H}_{\text{global}}=\sum_{i=1}^{N}\alpha_i h_i,\tag{6}$$

其中，h_i是节点i的最终隐藏状态，α_i是节点i的注意力权重，N是图中的节点数量，$\boldsymbol{H}_{\text{global}}$是整体图的表示。

4.3 评估结果

本文主要关注所提出方法在实现成对预测方面的性能表现，因为这与漏洞直接相关。表 5 展示了 VulGraphPTM 及所有其他基线模型在 VulFixed 数据集上的评估结果。根据表中结果，VulGraphPTM 在准确率(*Acc*)、*F*1 分数和成对预测率(*PP*)三项指标上均持续优于现有模型，分别达到最高值 65.45%、66.42% 和 36.86%。值得注意的是，VulGraphPTM 在 *PP* 指标上显著提升，比 SOTA 模型 UniXcoder 高出了 58.47%((36.86−23.26)/23.26)。这一显著进展凸显了 VulGraphPTM 在区分漏洞函数与其修正版本方面的卓越能力。通过融合 PTM 编码的全局信息和 GNN 编码的局部信息，VulGraphPTM 相比基于 PTM 的模型，性能相对提升了 72.48%((36.86−21.37)/21.37)，相比基于 GNN 的模型，性能相对提升了 115.30%((36.86−17.12)/17.12)。从模型无法实现成对预测的角度(即 *BV*、*BN* 和 *BI* 指标，参见 2.4 节)来看，此方法仍展现出强大的竞争力。尽管某些模型单独看来具有较低的 *BV*(如 VulBERTa-CNN 的 *BV* 为 17.00%)

Table 5 Performance Comparison Between VulGraphPTM and Existing State-of-the-Art Models

表 5 VulGraphPTM 与 SOTA 模型的性能对比

模型		Devign		BigVul		VulFixed						平均时间开销/s
		Acc/%↑	*F*1/%↑	*Acc*/%↑	*F*1/%↑	*Acc*/%↑	*F*1/%↑	*BV*/%↓	*BN*/%↓	*BI*/%↓	*PP*/%↑	
GNN	Devign	57.27	37.54	65.48	51.14	51.19	46.64	22.89	42.39	16.65	18.07	0.003
	ReVeal	53.60	63.85	62.55	44.70	51.78	57.21	55.64	31.98	4.82	7.56	**0.001**
	AMPLE	51.82	60.57	64.50	49.85	55.45	51.15	24.48	46.82	7.89	19.72	0.003
	ReGVD	58.97	58.40	67.85	67.65	59.80	59.11	36.90	36.50	4.82	23.13	0.014
	GNN-average	55.42	55.09	65.10	53.34	54.56	53.53	34.98	39.42	8.55	17.12	0.053
CodePTM	VulBERTa-MLP	61.79	58.67	64.04	71.42	61.50	62.83	36.06	35.78	3.97	23.08	0.017
	VulBERTa-CNN	61.31	57.22	68.23	71.15	59.75	48.89	**17.00**	59.56	**2.13**	21.57	0.035
	LineVul	64.75	63.81	**86.63**	**87.25**	51.54	62.31	62.57	**15.39**	5.23	17.55	0.055
	UniXcoder	**67.06**	**64.09**	85.09	86.80	59.83	65.85	39.14	34.34	4.51	23.26	0.034
	PTM-average	63.73	60.95	76.50	79.16	58.16	59.97	38.69	36.27	3.96	21.37	0.028
VulGraphPTM（本文）		55.91	57.95	69.99	62.44	**65.45**	**66.42**	29.03	27.72	6.41	**36.86**	0.019

注：“↑”表示该指标为正向指标，即数值越大性能越优。“↓”表示该指标为负向指标，即数值越小性能越优。黑体数值表示每列的最优值。

和 *BN*（LineVul 的 *BN* 为 15.39%），但它们在其他指标上的较差表现不容忽视（*BN* 高达 59.56%，*BV* 高达 62.57%）。相反，VulGraphPTM 的各项指标反映出其在“避免做出相同预测”方面具有更优异且全面的性能：既不会草率地将相似样本归类为“均存在漏洞”，又能细致审查潜在的漏洞函数。VulGraphPTM 在 *BV*、*BN* 和 *BI* 指标上相对较低的值，正体现了其在实际漏洞检测应用中更高的可靠性。

在确保成对预测准确性的同时，VulGraphPTM 在常规数据集上也保持了与基于 GNN 的模型相当的良好性能。具体而言，在 Devign 数据集上，VulGraphPTM 的 *F*1 分数平均提升 5.20%（(57.95−55.09)/55.09）；在 BigVul 数据集上，*F*1 分数平均提升 17.06%（(62.44−53.34)/53.34）。VulGraphPTM 的性能与同样融合 PTM 和 GNN 结构的 ReGVD 最为接近，它们在 2 个数据集上的 *F*1 分数差异均保持在 8% 以内。与传统 GNN 相比，VulGraphPTM 的优势更为显著，在 Devign 数据集上平均提升 7.33%，在 BigVul 数据集上平均提升 28.57%。然而，结果显示 VulGraphPTM 与基于 PTM 的模型之间仍存在性能差距。考虑到 VulGraphPTM 仍使用 GNN 进行聚合和预测（见图 8），其保留多样化输入样本的能力尚未达到 PTM 的水平。但需要强调的是，更高的 *F*1 分数并不必然意味着对漏洞语义有更好理解[25,28]。例如研究[28]结果显示，即使所有被评估模型的平均 *F*1 分数超过 60%，它们在 *PP* 的性能都下降了 39%~64%。为验证基于 PTM 模型是否过拟合与漏洞无关的文本特征，本文进一步使用在 BigVul 数据集上训练的模型对 Devign 数据集进行预测。结果表明 LineVul 的 *F*1 分数降至 43.41%，显著下降 50.25%（(87.25−43.41)/87.25）。其他模型的 *F*1 分数也大幅下降，其中 UniXcoder 下降 53.92%，ReGVD 下降 34.80%，Devign 下降 74.74%。所有基线模型的性能平均大幅下降 47.76%，这凸显了原始评估中存在的虚高性能。相比之下，VulGraphPTM 的 *F*1 分数仍能保持在 37.61%，比传统 GNN 的平均分数 31.70% 高出 18.64%。这些结果表明 VulGraphPTM 对数据集中的伪特征依赖程度更低。

在进行效率相关的实验分析之前，首先对时间开销的计算方式进行说明。本文将单样本的平均训练时间作为衡量模型计算效率的指标，其计算如式（7）所示。

$$TimeCost = \frac{Training\ Time}{Epochs \times Number\ of\ Training\ Samples} \tag{7}$$

其中，*Training Time* 表示模型完成整个训练过程所消耗的总时间，*Epochs* 为训练轮数，*Number of Training Samples* 表示训练集中样本的总数量。该指标能够消除不同训练轮数和数据规模带来的影响，从而更公平地反映不同模型在单位样本上的计算效率。如表 5 最后一列所示，所有模型的时间开销有所不同，反映出它们在计算效率方面的差异。例如，传统的 GNN 模型（如 Devign）在处理相同数据集时，时间开销仅为 0.003 s，表现出较为高效的计算能力。相比之下，基于 CodePTM 的模型，如 VulBERTa-CNN 和 LineVul，尽管在准确率和 *F*1 得分上有较好表现，但它们的时间开销相对较大，分别为 0.035 s 和 0.055 s，分别比 GNN

模型高出近 12 倍和 18 倍。值得注意的是, VulGraphPTM 模型的时间开销为 0.019 s, 相比于传统 GNN 模型, 表现出较为理想的平衡, 既保证了较高的准确度, 同时也在时间开销上做出了优化, 明显优于 CodePTM 的多个被测试模型。这些结果表明 VulGraphPTM 在保持高效计算的同时, 能够提供有竞争力的性能。后续工作将进一步探讨如何在更大规模的数据集上优化计算效率, 以提高模型的可扩展性和实际应用中的响应速度。最后, 需要注意模型在常规数据集上取得的原始评价指标仍具有重要意义。后续工作将专注于提升 VulGraphPTM 在多样化样本上的性能, 确保其在不同数据中的鲁棒性和适应性。

VulGraphPTM 相对于基线模型的性能提升主要源于其更能准确识别漏洞复杂度(即修复行数)较低的相似样本。图 9 展示了不同修复行数下的平均 *PP* 的变化情况, 清晰表明 VulGraphPTM 在几乎所有修复行数上都持续优于基线模型。这表明 VulGraphPTM 尤其重视漏洞修复信息。当修复行数处于 2~8 时, 性能提升最为显著, *PP* 峰值达到 48.01%。这些样本占总案例的 61.1%, 恰好对应现实世界补丁中最常见的代码修改行数范围, 说明 VulGraphPTM 在真实场景中具有实用价值。此外, 与基线模型相比, VulGraphPTM 在漏洞样本与其修复版本之间展现出更显著的注意力分布差异。表 4 展示了 VulGraphPTM 与基线模型在注意力分数最高的前 *K* 行上的平均肯德尔系数。结果显示, 在前 5 行和前 8 行位置, VulGraphPTM 的肯德尔系数值显著降低。出现这种降低是因为 VulGraphPTM 增强了漏洞行检测能力——漏洞/修复行更频繁地出现在注意力排名前列, 而完全相同的无关行则减少, 使得目标样本行间的文本差异更加明显。

如图 10 所示, 我们通过 2 个高度相似的真实漏洞修复案例, 分析不同模型在细粒度漏洞识别能力上的差异。图 10(a)来自 Linux 内核 process_keys.c, 其修复仅涉及错误返回值语义的调整, 将对-EEXIST 的特殊处理改为直接返回错误码。该变化仅表现为单行条件逻辑修改, 函数结构与控制流基本保持不变, 极易被误判为与非漏洞版本等价。图 10(b)来自 OpenSSL 的 ssl/t1_lib.c, 补丁将多处基于指针加法的边界检查重写为以剩余长度(limit − data)为中心的比较形式, 以避免整数溢出和越界访问风险。尽管涉及多行修改, 其核心仍集中于同一数据流上的边界约束, 函数整体结构高度相似。

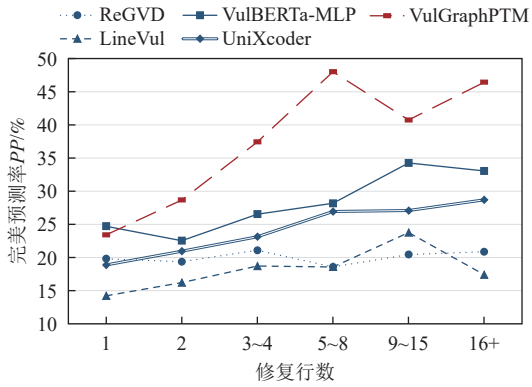


Fig. 9 *PP* under various number of fixed lines

图 9 不同修复行数下的 *PP*

在这 2 个案例中, 基线模型由于过度依赖表层语法和函数级相似性, 未能有效捕捉关键的控制流与数据流约束变化, 从而无法正确识别漏洞版本。相比之下, VulGraphPTM 在代码属性图上对齐 CodePTM 的全局语义与 GNN 的局部结构信息, 使返回路径和边界检查相关节点及其依赖关系得到强化表达, 因此即便在函数高度相似、修复行数较少的情况下, 仍能准确区分漏洞版本与修复版本。

5 有效性威胁

本文的研究结果与结论面临以下多种威胁:

1)可复现性。模型的超参数是影响其结果的关键因素, 也决定了结果在可复现性方面的可靠性。为降低超参数对结果有效性的影响, 本文采用基线模型指定的默认超参数, 并确保同类型模型在随机种子、学习率、batch size 和训练轮数等参数上保持一致。这种方法有助于更公平地比较模型性能。

2)数据质量。标注错误的样本会误导模型学习错误特征, 从而显著影响模型训练的稳定性。已有研究表明, 标签噪声会显著降低模型的泛化能力, 并可能导致训练过程不稳定, 尤其是在深度学习模型中表现更为明显[54-55]。本文基于现有数据集构建 VulFixed 数据集, 其标签准确性取决于原始数据集的准确性(这并非本文的讨论重点)。尽管如此, 为降低错误标签带来的潜在风险, 本研究精心选择数据集, 采用人工标注的 Devign 数据集和广泛使用的 BigVul 数据集。数据集划分遵循 CodeXGLUE 提供的标准划分方案或采用随机划分, 并通过细致的数据验证防止数据泄露。已有研究指出, 遵循标准划分并进行数据泄露检查可显著提升实验结果的可信度与可复现性[56]。

3)漏洞行定位。在许多情况下, 漏洞补丁中的

```
diff --git a/security/keys/process_keys.cb/security/keys/process_keys.c
index b6fdd22205b169..9139b18fc863eb 100644
--- a/security/keys/process_keys.c
+++ b/security/keys/process_keys.c
@@ -194, 11 +199, 9 @@ int install_process_keyring_to_cred(struct cred *new)
 }
 static int install_process_keyring(void)
 {
@@ -212,14 +215,18 @@ static int install_process_keyring(void)
                ret=install_process_keyring_to_cred(new);
                if (ret<0) {
                        abort_creds(new);
-                       return ret!=-EEXIST? ret : 0;
+                       return ret;
                }
```

(a) Linux项目中CVE-2016-1621漏洞样本

```
diff --git a/ssl/t1_lib.c b/ssl/t1_lib.c
index fb64607acf35e..cdac0118caf9d 100644
--- a/ssl/t1_lib.c
+++ b/ssl/t1_lib.c
@@ -1867, 11 +1867, 11 @@ static void ssl_check_for_safari(SSL *s, const unsigned char *data,
- if (data>=(limit-2))
+ if (limit-data<=2)
        return;
    data+=2;
- if (data>(limit-4))
+ if (limit-data<4)
        return;
if (type!=TLSEXT_TYPE_server_name)
        return;
// <15 lines>
- if (data+size>limit)
+ if (limit-data<size)
        return;
    data+=size;
// <11 lines>
- if (data+len1+len2!=limit)
+ if (limit-data!=(int)(len1+len2))
            return;
const size_tlen= sizeof(kSafariExtensionsBlock);
// <15 lines>
- if (data+len!=limit)
+ if (limit-data!=(int)(len))
```

(b) OpenSSL项目中CVE-2016-2177漏洞样本

Fig. 10 Case study on vulnerability detection for similar function, where only VulGraphPTM correctly identifies

图 10 相似函数漏洞检测案例分析（仅 VulGraphPTM 正确识别）

修改行并不等同于触发漏洞的代码行，因为修复操作可能不在漏洞原始位置进行。但是，有两点原因支持使用补丁修改行作为漏洞行的近似替代。首先，目前尚无精确定位触发漏洞行的大型开源数据集，因为这需要投入大量人力进行漏洞定位。在先前研究中，使用修改行作为漏洞行近似是一种标准解决方案[14,37,57-58]。其次，修改行与漏洞行之间存在直接语义关联，因此使用修改行评估模型性能具有合理性。一个鲁棒的模型应对修改行和漏洞行给予相似的关注度。反之，若关注度缺乏相似性，则表明模型对漏洞语义的理解不完整。已有工作也指出，若模型在补丁修改行与真实漏洞行之间表现出显著关注差异，则往往意味着其对漏洞语义的理解仍然不充分[54]。因此，尽管补丁修改行并非漏洞触发行的严格等价替代，本文采用该近似方式与现有主流研究保持一致，使实验结果具备良好的可比性，并在当前数据条件下具有合理性。

6 结　　论

本文系统评估了 8 种 SOTA 漏洞检测模型在配对漏洞实例上进行成对预测的能力，整理并构建了 VulFixed 数据集用于评估这些模型的性能。研究结果表明，现有模型在上述场景下表现欠佳，而在数据集中加入足够数量（超过 80%）的配对实例可显著提升模型性能。研究显示，与特定漏洞类型（即权限提升）相关的样本以及复杂度较高（即修改行数较多）的样本更易被现有模型准确预测。此外，模型倾向

于捕获整个输入上下文中分散的特征,缺乏对局部区域信息的整合能力。

为解决这些局限性,本文提出 VulGraphPTM 模型。该模型融合 PTM 的全局考察与 GNN 的局部信息,增强了模型对漏洞语义的表达能力。在 VulFixed 数据集上的评估结果表明,VulGraphPTM 显著优于所有现有模型。

7 数据开源

本文中开发的工具和构建的基准数据集开源地址为 https://github.com/CGCL-codes/VulGraphPTM。

作者贡献声明:朱嘉豪和魏子超负责论文的整体研究设计与实验实施,包括模型复现、改进方法的实现以及主要实验分析与论文撰写工作;郭靖宜负责实验数据的系统整理、结果校对及全文一致性核查;文明和金海共同确定了本研究的总体方向与技术框架,同时负责论文的修改、校订与定稿。

参 考 文 献

[1] Ghaffarian S M, Shahriari H R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey [J]. ACM Computing Surveys, 2017, 50(4): 56: 1−56: 36

[2] Li Zhen, Zou Deqing, Xu Shouhuai, et al. VulDeePecker: A deep learning-based system for vulnerability detection [C] //Proc of the 25th Annual Network and Distributed System Security Symp. San Diego, California: ISOC 2018, [2025-08-01]. https://doi.org/10.14722/ndss.2018.23158

[3] Lin Guangjun, Wen Sheng, Han Qinglong, et al. Software vulnerability detection using deep neural networks: A survey[J]. Proceedings of the IEEE, 2020, 108(10): 1825−1848

[4] Lu Guilong, Ju Xiaolin, Chen Xiang, et al. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning[J]. Journal of Systems and Software, 2024, 212: 112031

[5] Sun Yuqiang, Wu Daoyuan, Xue Yue, et al. LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning [J]. arXiv preprint, arXiv: 2401.16185, 2024

[6] Chen Yizheng, Ding Zhoujie, Alowain L, et al. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection [C] //Proc of the 26th Int Symp on Research in Attacks, Intrusions and Defenses. New York: ACM, 2023: 654−668

[7] Zhou Yaqin, Liu Shangqing, Siow J, 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. Red Hook, NJ: Curran Associates, Inc., 2019: 10197−10207

[8] Wen Xincheng, Chen Yupan, Gao Cuiyun, et al. Vulnerability detection with graph simplification and enhanced graph representation learning [C] //Proc of the 45th IEEE/ACM Int Conf on Software Engineering. Melbourne, Australia: ACM/IEEE, 2023: 2275−2286

[9] Hu Song, Luo Jiachi, Wan Wenkai, et al. LHG-VD: A learnable hierarchical graph representation approach for vulnerability detection[J]. Journal of Computer Research and Development, 2025, 62(9): 2348−2361(in Chinese)
(胡菘, 罗嘉驰, 万文凯, 等. LHG-VD: 一种可学习的层次化图表示漏洞检测方法[J]. 计算机研究与发展, 2025, 62(9): 2348−2361)

[10] Yamaguchi F, Golde N, Arp D, et al. Modeling and discovering vulnerabilities with code property graphs [C] // Proc of IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2014: 590−604

[11] Feng Zhangyin, Guo Daya, Tang Duyu, et al. CodeBERT: A pre-trained model for programming and natural languages [C] //Proc of the Findings of the Association for Computational Linguistics: EMNLP 2020. Stroudsburg, PA: ACL, 2020: 1536−1547

[12] Guo Daya, Lu Shuai, Duan Nan, et al. UniXcoder: Unified cross-modal pre-training for code representation [C] //Proc of the 60th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2022: 7212−7225

[13] Rozière B, Gehring J, Gloeckle F, et al. Code Llama: Open foundation models for code[J]. arXiv preprint, arXiv: 2308.12950, 2023

[14] Fu M, Tantithamthavorn C. LineVul: A Transformer-based line-level vulnerability prediction [C] //Proc of the 19th Int Conf on Mining Software Repositories. Pittsburgh, PA: ACM/IEEE, 2022: 608−620

[15] Hanif H, Maffeis S. VulBERTa: Simplified source code pre-training for vulnerability detection [C] //Proc of Int Joint Conf on Neural Networks. Piscataway, NJ: IEEE, 2022: 1−8

[16] Nguyen V A, Nguyen D Q, Nguyen V, et al. ReGVD: Revisiting graph neural networks for vulnerability detection [C] //Proc of the 44th IEEE/ACM Int Conf on Software Engineering. Pittsburgh, PA: ACM/IEEE, 2022: 178−182

[17] Fang R, Bindu R, Gupta A, et al. LLM agents can autonomously exploit one-day vulnerabilities [J]. arXiv preprint, arXiv: 2404.08144, 2024

[18] Jensen R I T, Tawosi V, Alamir S. Software vulnerability and functionality assessment using LLMs [J]. arXiv preprint, arXiv: 2403.08429, 2024

[19] Jie Wanqing, Qiu Wangjie, Yang Haofu, et al. Agent4Vul: Multimodal LLM agents for smart contract vulnerability detection[J]. Science China Information Sciences, 2025, 68(6): 160101

[20] Yin Xin, Ni Chao, Wang Shaohua. Multitask-based evaluation of open-source LLM on software vulnerability [J]. arXiv preprint, arXiv: 2404.02056, 2024

[21] Wei J, Wang Xuezhi, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models [C] //Proc of the 35th Int Conf on Neural Information Processing Systems. Red Hook, NJ: Curran Associates, Inc., 2022 : 24824−24837

[22] Chakraborty S, Krishna R, Ding Yangruibo, et al. Deep learning based vulnerability detection: Are we there yet?[J]. IEEE Transactions on Software Engineering, 2022, 48(9): 3280−3296

[23] Du Qianjin, Zhou Shiji, Kuang Xiaohui, et al. Joint geometrical and statistical domain adaptation for cross-domain code vulnerability detection [C] //Proc of the 2023 Conf on Empirical Methods in Natural Language Processing. Stroudsburg, PA: ACL, 2023: 12791−12800

[24] Rahman M M, Ceka I, Mao Chengzhi, et al. Towards causal deep learning for vulnerability detection [J]. arXiv preprint, arXiv: 2310.07958, 2023

[25] Risse N, Böhme M. Uncovering the limits of machine learning for automatic vulnerability detection [C] //Proc of the 33rd USENIX Security Symp. Berkeley, CA: USENIX Association, 2024: 4247−4264

[26] Zhang Huangzhao, Li Zhuo, Li Ge, et al. Generating adversarial examples for holding robustness of source code processing models [C] //Proc of the 34th AAAI Conf on Artificial Intelligence. Menlo Park, CA: AAAI, 2020: 1169−1176

[27] Zhang Jie, Ma Wei, Hu Qiang, et al. A black-box attack on code models via representation nearest neighbor search [C] //Proc of the Findings of the Association for Computational Linguistics: EMNLP 2023. Stroudsburg, PA: ACL, 2023: 9706−9716

[28] Ding Yangruibo, Fu Yanjun, Ibrahim O, et al. Vulnerability detection with code language models: How far are we? [J]. arXiv preprint, arXiv: 2403.18624, 2024

[29] 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/IEEE, 2020: 508−512

[30] Fu M, Tantithamthavorn C, Nguyen V, et al. ChatGPT for vulnerability detection, classification, and repair: How far are we? [J]. arXiv preprint, arXiv: 2310.09810, 2023

[31] Steenhoek B, Rahman M M, Jiles R, et al. An empirical study of deep learning models for vulnerability detection [C] //Proc of the 45th IEEE/ACM Int Conf on Software Engineering. New York: ACM/IEEE, 2023: 2237−2248

[32] Steenhoek B, Rahman M M, Roy M K, et al. A comprehensive study of the capabilities of large language models for vulnerability detection [J]. arXiv preprint, arXiv: 2403.17218, 2024

[33] Gao Zeyu, Wang Hao, Zhou Yuchen, et al. How far have we gone in vulnerability detection using large language models [J]. arXiv preprint, arXiv: 2311.12420, 2023

[34] Ouyang Long, Wu J, Jiang Xu, et al. Training language models to follow instructions with human feedback [C] //Proc of the 35th Int Conf on Neural Information Processing Systems. Red Hook, NJ: Curran Associates, Inc., 2022 : 27730−27744

[35] OpenAI. GPT-4 technical report [J]. arXiv preprint, arXiv: 2303.08774, 2023

[36] Ding Yangruibo, Chakraborty S, Buratti L, et al. CONCORD: Clone-aware contrastive learning for source code [C] //Proc of the 32nd ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2023: 26−38

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

[38] Guo Daya, Ren Shuo, Lu Shuai, et al. GraphCodeBERT: Pre-training code representations with data flow [C/OL] //Proc of the 9th Int Conf on Learning Representations. Austria: OpenReview, 2021[2025-08-01]. https://iclr.cc/virtual/2021/poster/2598

[39] Li Yujia, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks [C/OL] //Proc of the 4th Int Conf on Learning Representations. San Juan, Puerto Rico: OpenReview, 2016 [2025-08-01]. https://doi.org/10.48550/arXiv.1511.05493

[40] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need [C] // Proc of the Annual Conf on Neural Information Processing Systems. Red Hook, NJ: Curran Associates, Inc., 2017: 5998−6008

[41] Schlichtkrull M S, Kipf T N, Bloem P, et al. Modeling relational data with graph convolutional networks [C] //Proc of the 15th Extended Semantic Web Conf. Berlin: Springer, 2018: 593−607

[42] Liu Yinhan, Ott M, Goyal N, et al. RoBERTa: A robustly optimized BERT pretraining approach [J]. arXiv preprint, arXiv: 1907.11692, 2019

[43] Lu Shuai, Guo Daya, Ren Shuo, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation [C/OL] //Proc of the Neural Information Processing Systems Track on Datasets and Benchmarks. Red Hook, NJ: Curran Associates, Inc., 2021 [2025-08-01]. https://doi.org/10.48550/arXiv.2102.04664

[44] Tai Jianwei, Yang Shuangning, Wang Jiajia, et al. Survey of adversarial attacks and defenses for large language models[J]. Journal of Computer Research and Development, 2025, 62(3): 563−588 (in Chinese)
(台建玮, 杨双宁, 王佳佳, 等. 大语言模型对抗性攻击与防御综述[J]. 计算机研究与发展, 2025, 62(3): 563−588)

[45] Qin Zhen, Zhuang Tianming, Zhu Guosong, et al. Survey of security attack and defense strategies for artificial intelligence model[J]. Journal of Computer Research and Development, 2024, 61(10): 2627−2648 (in Chinese)
(秦臻, 庄添铭, 朱国淞, 等. 面向人工智能模型的安全攻击和防御策略综述[J]. 计算机研究与发展, 2024, 61(10): 2627−2648)

[46] Li Nan, Ding Yidong, Jiang Haoyu, et al. Jailbreak attack for large language models: A survey[J]. Journal of Computer Research and Development, 2024, 61(5): 1156−1181 (in Chinese)
(李南, 丁益东, 江浩宇, 等. 面向大语言模型的越狱攻击综述[J]. 计算机研究与发展, 2024, 61(5): 1156−1181)

[47] Powers D M. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation[J]. arXiv preprint arXiv: 2010.16061, 2020

[48] Saito T, Rehmsmeier M. The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets[J]. PLoS ONE, 2015, 10(3): e0118432

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

[50] Li Zhen, Zou Deqing, Xu Shouhuai, et al. VulDeeLocator: A deep learning-based fine-grained vulnerability detector[J]. Journal of IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2821−2837

[51] Serebryany K, Bruening D, Potapenko A, et al. AddressSanitizer: A fast address sanity checker [C] //Proc of the 2012 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2012: 309−318

[52] Sharmeen S, Huda M S, Abawajy J H, et al. An adaptive framework against android privilege escalation threats using deep learning and semi-supervised approaches[J]. Journal of Applied Soft Computing, 2020, 89: 106089

[53] Guo Menghao, Liu Zhengning, Mu Taijiang, et al. Beyond self-attention: External attention using two linear layers for visual tasks[J]. Journal of IEEE Transactions on Pattern Analysis and Machine Intelligence, 2023, 45(5): 5436−5447

[54] Northcutt C, Jiang Lu, Chuang I. Confident learning: Estimating uncertainty in dataset labels[J]. Journal of Artificial Intelligence Research, 2021, 70: 1373−1411

[55] Rolnick D, Veit A, Belongie S, et al. Deep learning is robust to massive label noise[J]. arXiv preprint arXiv: 1705.10694, 2017

[56] Zhang Kechi, Wang Wenhan, Zhang Huangzhao, et al. Learning to represent programs with heterogeneous graphs[C] //Proc of the 30th IEEE/ACM Int Conf on Program Comprehension (ICPC'22). New York: ACM, 378–389

[57] Cheng Xiao, Zhang Guanqin, Wang Haoyu, et al. Path-sensitive code embedding via contrastive learning for software vulnerability detection [C] //Proc of the 31st ACM SIGSOFT Int Symp on Software Testing and Analysis. New York: ACM, 2022: 519−531

[58] Du Xiaohu, Wen Ming, Zhu Jiahao, et al. Generalization-enhanced code vvulnerability detection via multi-task instruction fine-tuning [C] //Proc of the Findings of the Association for Computational Linguistics. Stroudsburg, PA: ACL, 2024: 10507−10521



Zhu Jiahao, born in 2000. Master. His main research interests include code security and vulnerability detection.

朱嘉豪, 2000 年生。硕士。主要研究方向为代码安全、漏洞检测。



Wei Zichao, born in 2000. PhD candidate. His main research interests include code security and automated vulnerability repair.

魏子超, 2000 年生。博士研究生。主要研究方向为代码安全、自动漏洞修复。



Guo Jingyi, born in 2005. Undergraduate. Her main research interest includes code security.

郭靖宜, 2005 年生。本科生。主要研究方向为代码安全。



Wen Ming, born in 1991. PhD, professor. Senior member of CCF. His main research interests include software security, software supply chain security, and big code analysis.

文　明, 1991 年生。博士，教授。CCF 高级会员。主要研究方向为软件安全、软件供应链安全、大规模代码分析。



Jin Hai, born in 1966. PhD, professor. Fellow of CCF. His main research interests include computer architecture, parallel and distributed processing, cloud computing, and big data.

金　海, 1966 年生。博士，教授。CCF 会士。主要研究方向为计算机体系结构、并行与分布式处理、云计算、大数据。