

一种基于程序依赖语义嵌入的可解释性 漏洞检测方法

桑海涯, 尹中旭, 李俊儒, 宗国笑

(信息工程大学, 河南 郑州 450001)

摘要: 当前基于图神经网络的漏洞检测方法难以有效建模代码长距离依赖与深层语义, 而基于大语言模型的方法又缺乏对程序结构的显式感知, 导致漏洞定位与解释能力不足。为此, 提出一种融合程序依赖语义嵌入与解码器推理的漏洞可解释性分析框架(DEPVulDet)。设计专用编码器捕获程序依赖中的控制依赖和数据流语义特征, 通过轻量级线性映射将其对齐至解码器大模型的语义空间; 解码器基于映射后的表示进行漏洞检测, 并引入基于思维链的多轮对话训练机制, 增强模型对漏洞位置、上下文及成因的逻辑推理与解释能力。实验表明, DEPVulDet 的准确率分别达到 69.6% 与 95.7%, $F1$ 分数分别为 71.5% 与 48.1%, 多项指标优于对比方法。

关键词: 漏洞检测; 编码器; 解码器; 可解释性; 程序依赖

中图分类号: TP309

文献标识码: A

文章编号: 1671-0673(2026)01-0081-07

An Explainable Vulnerability Detection Method Based on Program Dependency Semantic Embedding

SANG Haiya, YIN Zhongxu, LI Junru, ZONG Guoxiao

(Information Engineering University, Zhengzhou 450001, China)

Abstract: Current vulnerability detection methods based on graph neural networks struggle to effectively model long-range dependencies and deep semantic features in code, whereas approaches leveraging large language models lack explicit structural awareness of programs, resulting in insufficient capabilities for vulnerability localization and interpretability. An explainable vulnerability analysis framework DEPVulDet, that integrates program dependency semantic embedding with decoder-based reasoning, is proposed. A dedicated encoder is employed to capture control and data-flow semantic features from program dependencies, which is then aligned into the semantic space of a large language model of the decoder via a lightweight linear mapping mechanism. The decoder is used to perform vulnerability detection based on the mapped representations, while a chain-of-thought-inspired multi-turn dialogue training strategy is introduced to enhance the model's logical reasoning and explanatory capability regarding vulnerability locations, contextual dependencies, and root causes. Experimental results show the accuracy of DEPVulDet reaches 69.6% and 95.7% respectively, with its $F1$ -scores standing at 71.5% and 48.1%, exhibiting superior performance over the comparison methods on various evaluation metrics.

Key words: vulnerability detection; encoder; decoder; explainability; program dependency

收稿日期: 2025-11-11; 修回日期: 2025-12-19

作者简介: 桑海涯(2001—), 男, 硕士生, 主要研究方向为逆向工程与漏洞检测。E-mail: 15397591646@163.com

通信作者: 尹中旭(1983—), 男, 副教授, 博士, 主要研究方向为软件逆向与漏洞挖掘。E-mail: yinzhu@163.com

随着软件系统日益复杂,漏洞数量持续攀升,仅2024年新增漏洞已超过4万例,严重威胁软件安全。在此背景下,高效准确的漏洞检测技术已成为保障软件安全的关键防线。

深度学习在漏洞检测领域展现出巨大潜力。当前主流方法分为两类:1)基于图神经网络(Graph Neural Networks, GNN)的方法^[1-2]能有效建模程序依赖图(Program Dependence Graph, PDG)和控制流图(Control Flow Graph, CFG)等结构化表示。研究表明,其性能显著优于基于词法单元(Token)的序列分析方法^[3-4]。2)基于大语言模型(Large Language Model, LLM)的方法^[5]则通过编码器(如CodeBERT)或生成式模型(如Llama)学习代码语义特征。然而,GNN方法^[6-7]在处理复杂图时难以建模长距离依赖关系,对深层语义理解有限。LLM方法^[8-9]虽有较强语义理解能力,但缺乏对程序结构的显式建模,难以聚焦漏洞相关的局部依赖。此外,现有方法^[10-11]普遍缺乏对漏洞成因的合理归因与精准定位能力。

针对上述问题,本文提出编码器—解码器协同工作的漏洞检测方法,通过专用编码器捕获程序语义和依赖信息,利用轻量级映射转换为与大模型语义空间匹配的表示。同时,设计基于思维

链的漏洞解释推理机制,使模型能生成逻辑连贯的漏洞成因分析。

综上所述,本文的主要贡献如下:

1)设计了编码器、解码器协同工作的方法,其中,编码阶段专注于捕获代码语义和程序依赖信息,解码阶段基于编码结果完成漏洞检测与解释。

2)提出基于静态分析的漏洞线索提取算法,在开源漏洞数据集上构建高质量训练样本,采用多任务自我指导微调方法提升模型对漏洞的解释能力。

3)实现了完整的漏洞检测框架DEPVulDet,在真实漏洞代码数据集上的实验表明,本方法在检测准确率和可解释性方面均优于现有主流方法。

1 本文方法

DEPVulDet架构如图1所示,主要包含基于编码器—解码器的漏洞检测方法和基于思维链的增强漏洞解释训练两大部分。前者通过程序依赖编码器捕获语义信息,经轻量级映射对齐至大模型语义空间,实现漏洞识别;后者将漏洞样本转化为多轮对话数据,引入代码片段与成因分析,通过参数高效微调(Parameter-Efficient Fine-Tuning, PEFT),使解码器具备检测与解释双重能力。

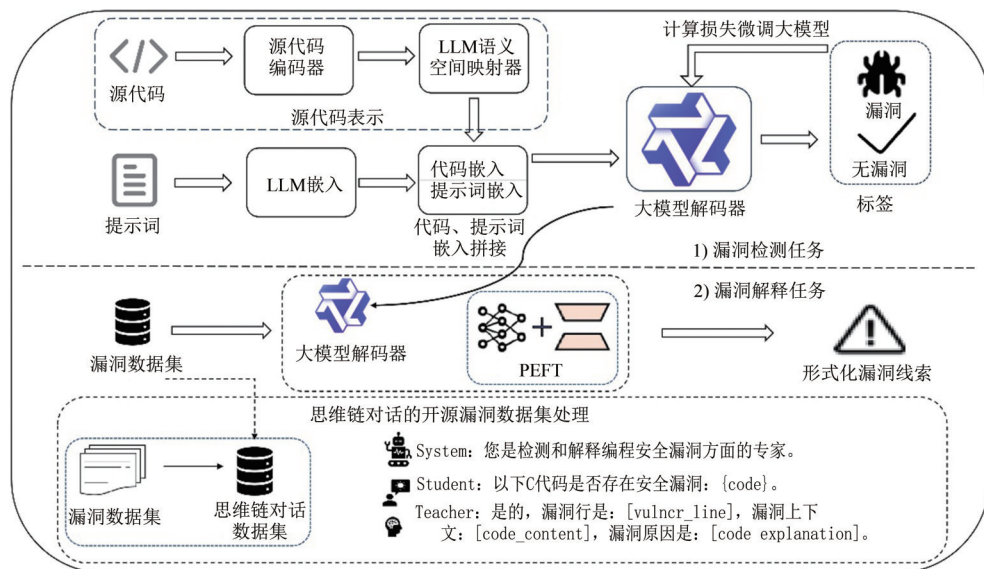


图1 DEPvulDet架构图

1.1 基于程序依赖的源代码编码器

本文提出一种编码器与解码器协同工作的漏洞检测与解释框架。在该框架中,解码器选用自然语言处理领域表现优异的Qwen2模型,而编码器则基于BERT架构^[12]设计,专注于捕获代码中的程序依赖语义信息。本节将详细阐述编码器的结构

设计、训练方法以及其与解码器之间的语义映射机制。

表1展示了常见漏洞类型与其相关程序结构之间的对应关系。结果表明,在13类典型漏洞中,10类(SQL注入、XSS(Cross-Site Scripting)、缓冲区溢出、内存泄漏、UAF(Use After Free)、整数溢出、

不安全参数、空指针解引用、整数类型安全问题、除零漏洞)通过数据流图(Data Flow Graph, DFG)建立语义关联,3类(缺少权限检查、条件竞争、资源泄露)依赖CFG边进行传播。值得注意的是,所有高危漏洞(共7类)均与DFG或CFG存在直接映射关系,其中DFG占比约为71.43%,凸显数据流分析在漏洞检测中具有关键作用。基于这一发现,本文提出的编码器通过学习数据流与控制流依赖,实现对程序依赖语义信息的有效嵌入。

表1 与漏洞相关的结构及其严重性

漏洞类型	相关边	危险等级
SQL注入	DFG	高
XSS	DFG	高
缓冲区溢出	DFG	高
内存泄漏	DFG	高
缺少权限检查	CFG	高
条件竞争	CFG	高
UAF	DFG	高
整数溢出	DFG	中
资源泄露	CFG	中
不安全参数	DFG	中
空指针解引用	DFG	低
整数类型安全问题	DFG	中
除零漏洞	DFG	中

在先前的研究中,文献[13]提出了数据流边预测任务和节点对齐任务,增强了对代码中数据流语义的捕获能力。具体来说,边预测任务通过随机掩码数据流中20%的节点及其连接边,迫使模型学习变量之间“值从哪里来”的语义依赖关系;数据流节点对齐任务则通过掩码变量节点与源代码标记之间的关联,训练模型建立数据流图与源代码之间的精确映射,从而识别数据流中的特定变量及其在代码中的位置。在此基础上,本文进一步提出语句级控制依赖预测任务,以捕获程序中的控制依赖语义。首先,使用静态代码分析工具Joern提取函数的语句级PDG。PDG中的控制依赖边表示控制依赖关系。将源代码以语句级别编码为矩阵 $\mathbf{G}^c \in \mathbf{R}^{m_c \times m_c}$ (m_c 表示PDG中的语句级控制依赖图节点数量), $G_{ij}^c \in \{1,0\}$ 表示节点 j 是否依赖于节点 i 。解析每个节点的Token序列,并对这些Token的上下文嵌入进行平均,得到节点的向量 $\hat{\mathbf{h}}_i$,然后将其输入到多层感知机(Multi-Layer Perceptron, MLP),再由ReLU函数进行激活得到最终的表示 $\mathbf{h}_i$:

$$\mathbf{h}_i = \text{ReLU}[\text{MLP}(\hat{\mathbf{h}}_i)] \quad (1)$$

使用双线性层来预测两个节点(j 是否依赖于 i)之间的控制依赖关系:

$$p_{ij} = \sigma(\mathbf{h}_i^T \mathbf{W}_c \mathbf{h}_j + \mathbf{b}_c) \quad (2)$$

式中: $\mathbf{W}_c$ 和 $\mathbf{b}_c$ 是可训练参数; $\sigma(\cdot)$ 是激活函数。最后根据预测概率和真实标签 $\mathbf{G}^c$ 来计算交叉熵,作为控制依赖预测的损失函数:

$$L = \frac{1}{|\mathbf{G}^c|} \sum_{ij} [-G_{ij}^c \log_2 p_{ij} - (1 - G_{ij}^c) \log_2 (1 - p_{ij})] \quad (3)$$

通过联合训练数据流边预测任务、数据流节点对齐任务、语句级控制依赖预测任务及BERT原有的掩码语言模型任务,编码器能够全面捕获程序的依赖语义信息。

为实现代码嵌入与自然语言提示的有效融合,本文设计了一种轻量级映射机制,将编码器输出的代码嵌入转换至目标大语言模型的语义空间。首先,将源代码输入至编码器,并提取最后一个隐藏层状态 $\mathbf{x}_s$,通过一个可训练矩阵 $\mathbf{W}$ 将 $\mathbf{x}_s$ 转换为矩阵 $\mathbf{H}_s$:

$$\mathbf{H}_s = \mathbf{W} \cdot \mathbf{x}_s \quad (4)$$

转换后的代码嵌入 $\mathbf{H}_s$ 与自然语言提示的词嵌入拼接,共同输入至LLM解码器进行后续处理。该映射机制参数量小,在降低模型复杂度的同时,有效保持了代码与自然语言在语义上的一致性。

1.2 解码器漏洞检测监督训练

解码器采用生成式大模型,负责接收上述自然语言嵌入和代码嵌入的拼接,并对其解码完成漏洞检测。本节详细阐述所提出的监督微调策略,旨在优化解码器在漏洞检测任务中的性能表现。

首先将自然语言(“Please check the following code for any potential vulnerabilities”)通过解码器的词嵌入层进行编码,得到自然语言嵌入表示 $\mathbf{E}_{\text{prompt}}$ 。随后,将源代码通过编码模型嵌入并经映射模型转换,得到代码语义嵌入 $\mathbf{E}_{\text{code}}$ 。将二者进行拼接操作,形成解码器的完整输入表示:

$$\mathbf{X}_{\text{input}} = [\mathbf{E}_{\text{prompt}}; \mathbf{E}_{\text{code}}] \quad (5)$$

本文采用参数高效微调与量化低秩适配(Quantized Low-Rank Adaptation, QLoRA)^[14]相结合的方法。QLoRA通过NF4四值量化技术降低模型计算复杂度,同时在推理过程中保持模型性能。在LoRA框架^[15]下,仅需训练少量轻量级适配器参数,即可实现模型的有效微调,大幅减少了训练资源消耗。

图2展示了LoRA的核心思想。左侧表示冻结的预训练权重矩阵 $W \in \mathbf{R}^{d \times d}$, 右侧为可训练的低秩适配器矩阵 $A \in \mathbf{R}^{r \times d}$ (随机高斯初始化) 和 $B \in \mathbf{R}^{d \times r}$ (初始全为0)。初始 x 维度为 d , 分别与 Wx 和 Bx 相加, 最终输出 h 。该设计确保仅 A 和 B 参与梯度更新, 预训练权重始终被保护。

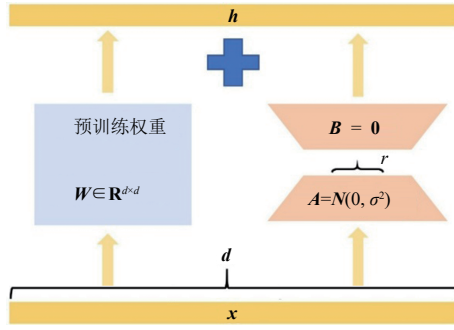


图2 LoRA 高效微调的核心处理

针对漏洞分类任务, 定义交叉损失熵函数驱动模型优化:

$$L = -\frac{1}{N} \sum_{i=1}^N [y_i \log_2(p_i) + (1 - y_i) \log_2(1 - p_i)] \quad (6)$$

式中: $y_i \in \{0, 1\}$ 为真实标签 (0 表示无漏洞, 1 表示有漏洞); p_i 为模型预测存在漏洞的概率, 计算方式为

$$p_i = \frac{e^{z_i, \text{Yes}}}{e^{z_i, \text{Yes}} + e^{z_i, \text{No}}} \quad (7)$$

式中: (z_i, Yes) 和 (z_i, No) 分别标识模型输出 “Yes” 和 “No” 的对数几率。该损失函数通过最大化正确分类概率, 引导模型在代码语义空间中学习区分漏洞与非漏洞样本的决策边界, 从而提升漏洞检测的准确性和鲁棒性。

1.3 增强漏洞解释推理训练

为提升模型对漏洞成因的深层理解与解释能力, 本文设计了一种基于思维链的漏洞解释推理

训练机制。该机制通过系统化构建漏洞线索与多轮对话数据, 引导模型学习从代码特征到漏洞成因的逻辑推理过程, 从而生成具有准确性和逻辑连贯性的漏洞解释。

定义漏洞线索: $\text{Vul_clue} = \{\text{vul_line}, \text{vul_context}, \text{vul_explanation}\}$, 其中, vul_line 、 vul_context 、 vul_explanation 分别是漏洞行、漏洞上下文和漏洞解释。漏洞上下文是与漏洞行相关的上下文, 代码片段主要是数据依赖和控制依赖。

算法1展示了如何提取漏洞函数的漏洞线索。首先, 分析补丁文件, 通过差异分析来确定漏洞行。随后, 使用静态分析工具 Joern 提取目标函数的 PDG。在 PDG 中, 从漏洞行节点出发, 使用广度优先搜索算法 (Breadth First Search, BFS) 遍历 k 跳范围内的关联节点, 确定与漏洞相关的数据依赖和控制依赖。最后, 从 CVE 等漏洞数据库中检索专业的漏洞技术解释, 形成完整的漏洞线索三元组。

算法1 漏洞线索提取算法

输入: 漏洞函数 vulner_fun 、补丁 Patch、跳数 k

输出: 漏洞线索 Vul_clue

1. 初始化: $\text{Vul_clue} \leftarrow \emptyset$
2. $\text{vul_lines} \leftarrow$ 从 patch 中查找删除行
3. $\text{PDG} \leftarrow \text{Joern}(\text{vul_fun})$
4. for vul_line in vul_lines do:
5. $\text{related_lines} \leftarrow \text{BFS}(\text{PDG}, \text{vul_line}, k)$
6. $\text{vul_context.append}(\text{related_lines})$
7. end for
8. $\text{vul_explanation} \leftarrow$ 查找 CVE 漏洞信息
9. return Vul_clue

图3展示了漏洞函数 parse_metadata 的漏洞线索提取过程。

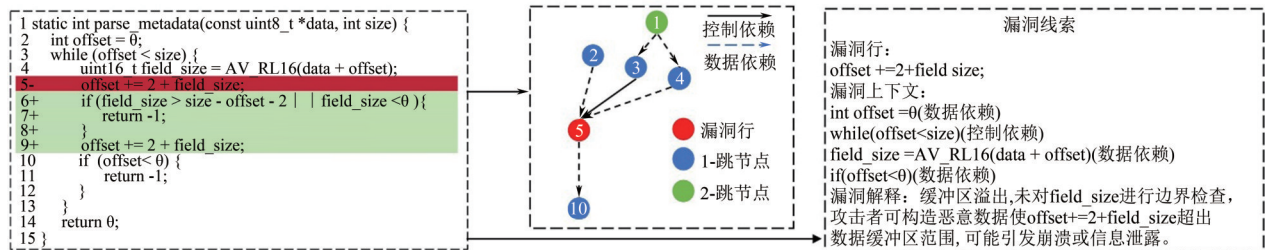


图3 漏洞线索提取过程

在开源漏洞数据集上, 基于前述算法提取结构化的漏洞线索。受思维链推理范式的启发, 本文将漏洞代码及其对应的漏洞位置、上下文依赖关系与成因分析转化为多轮对话形式的训练样本, 如图4所示。引导大语言模型逐步推理解析漏

洞相关语义信息, 从而提升其对漏洞模式的理解与解释能力。构建的对话数据以系统提示为起始, 明确设定模型角色为具备代码安全领域专业知识的分析主体, 随后通过一系列层次化提问, 驱动模型依次识别漏洞代码片段、定位关键缺陷位

置,并最终生成包含上下文依赖路径与根本成因的详细解释。

1	Round 0 = {
2	role: “系统”,
3	content: “您是检测和定位编程安全漏洞的专家,可以帮助回答漏洞问题”,
4	},
5	Round 1 = {
6	role: “学生”,
7	content: “以下代码是否存在安全漏洞: {漏洞代码}”,
8	role: “老师”
9	content: “Yes, {vul_clue.vul_explanation}”,
10	},
11	Round 3 = {
12	role: “学生”,
13	content: “请定位下漏洞行和上下文”,
14	role: “老师”,
15	content: “漏洞行是 {vul_clue.vul_lines}, 漏洞上下文是 {vul_clue.vul_context}”
16	}

图 4 多轮对话数据格式

为充分利用上述生成的对话数据集,本文采用多任务自指导微调^[16]策略。训练阶段,模型生成的完整响应序列与真实标注的对话内容进行对齐,通过对比二者差异计算损失。针对多任务学习中常见的任务不平衡问题,设计一种改进的加权损失函数:

$$L(\theta) = \min_{\theta} \left(\frac{1}{N} \sum_{i=1}^N \frac{\sum_{j=1}^{M_i} \sum_{k=1}^{T_{ij}} (-\log_2(p_0(T_{ijk})))}{\sum_{j=1}^{M_i} T_{ij}} \right) \quad (8)$$

式中: N 表示任务总数; M_i 表示第 i 个任务的样本数; T_{ij} 是第 i 个任务的第 j 个样本的损失函数中设计的有效Token的计数; T_{ijk} 是第 i 个任务的第 j 个样本的损失函数中设计的第 k 个有效Token。式(8)计算所有任务的平均损失。

2 实验分析

2.1 数据集

本文采用 FFmpeg+Qemu 和 Big-Vul^[17]两个数据集进行实验,验证 DEPVulDet 方法的有效性。FFmpeg+Qemu 数据集由 FFmpeg 与 Qemu 两个 C 语言开源项目构成,包含 11 673 个漏洞样本和 13 942 个非漏洞样本。Big-Vul 数据集涵盖 2002—2019 年 348 个真实软件项目,包含 11 834 个漏洞样本和 253 096 个非漏洞样本。

2.2 评估指标

为全面评估 DEPVulDet 在漏洞检测任务中的性能,本文选择了以下 4 个常用的评价指标:准确率($R_{accuracy}$)、精确率($R_{precision}$)、召回率(R_{recall})和 F1 分

数($F1$)。这些指标能够从不同角度衡量模型的分

$$R_{accuracy} = \frac{N_{TP} + N_{TN}}{N_{TP} + N_{TN} + N_{FP} + N_{FN}} \quad (9)$$

$$R_{precision} = \frac{N_{TP}}{N_{TP} + N_{FP}} \quad (10)$$

$$R_{recall} = \frac{N_{TP}}{N_{TP} + N_{FN}} \quad (11)$$

$$F1 = \frac{2 \times R_{precision} \times R_{recall}}{R_{precision} + R_{recall}} \quad (12)$$

式中: N_{TP} 为真实漏洞且被正确预测为漏洞的样本数; N_{TN} 为真实非漏洞且被正确预测为非漏洞的样本数; N_{FP} 为真实非漏洞但被错误预测为漏洞的样本数; N_{FN} 为真实漏洞但被错误预测为非漏洞的样本数。

2.3 软件与硬件环境

本文实验环境配置如下:

1)硬件平台采用配备 Intel(R) Xeon(R) Silver 4410Y 处理器(24 核心,2.80 GHz 主频)、128 GB DDR5 系统内存以及 NVIDIA A6000 GPU(48 GB GDDR6 显存)的高性能计算服务器。

2)软件环境方面,操作系统采用 Ubuntu 22.04 LTS 长期支持版本,深度学习框架选用 PyTorch 2.1.0,配合 CUDA 12.1 并行计算平台以充分发挥 GPU 计算能力,模型构建与调用基于 Transformers 库 4.42.0 版本。

2.4 实验结果与分析

2.4.1 对比试验

为全面评估本文提出的 DEPVulDet 方法的性能优势,将其与当前主流的基于深度学习的漏洞检测方法和基于大语言模型的漏洞检测方法进行系统性对比。对比方法包括:Devign^[18]、SySeVR^[3]、AMPLE^[1]、Text CNN^[19]、GraphCodeBert、CodeBert、PDBERT^[20]、CodeT5^[21]以及 CodeLLama-7B^[22]等代表性模型。实验结果如表 2 和表 3 所示。

表 2 在 FFmpeg+Qemu 数据集的对比实验结果 单位: %

模型	准确率	精确率	F1 分数	召回率
Devign	56.8	56.5	57.9	64.6
SySeVR	47.8	46.0	51.6	58.8
AMPLE	62.1	55.6	66.9	83.9
Text CNN	60.8	59.0	58.3	75.7
GraphCodeBert	64.8	64.3	58.9	54.3
CodeBert	62.3	61.5	54.0	48.2
CodeT5	63.3	58.6	63.2	68.8
CodeLLama	53.4	46.2	11.2	6.3
DEPVulDet	69.6	62.2	71.5	84.1

表 3 在 Big-Vul 数据集的对比实验结果 单位: %

模型	准确率	精确率	F1 分数	召回率
Devign	92.7	30.6	20.9	15.9
SySeVR	90.1	30.9	19.3	14.0
AMPLE	93.1	29.9	32.1	34.5
Text CNN	92.9	33.4	36.4	27.0
GraphCodeBert	91.5	70.2	40.7	28.7
CodeBert	90.8	61.2	36.5	26.0
CodeT5	92.6	70.4	46.9	35.2
CodeLLama	93.6	59.7	40.0	31.4
DEPVulDet	95.7	77.0	48.1	35.0

实验结果表明,DEPVulDet 在两个基准数据集的 4 项关键评估指标上总体取得最优性能,展现出显著综合优势。GNN 方法(如 AMPLE 和 Devign)整体优于 Token 序列方法(如 SySeVR),但 Text CNN(基于 Token 序列)表现优于 Devign,这是因为其利用大规模预训练语言知识,而 Devign 需从零训练。编码模型(GraphCodeBert、CodeBert)显著优于解码模型(Codellama-7B),归因于前者双向注意力机制能全面捕获代码结构与语义信息;然而,编码模型可解释性较差,难以清晰解释漏洞根本原因。

本文提出的 DEPVulDet 方法融合了编码模型对代码结构的理解优势与解码模型对代码语义的表达能力。具体而言,DEPVulDet 设计了专门的编码模块来捕获程序依赖语义信息,经过轻量级线性映射后输入至解码器完成漏洞检测任务。表 2 的实验结果进一步验证了该设计的有效性,表明 DEPVulDet 在各项评估指标上均取得了显著提升,实现了结构理解与语义表达的有机统一,为漏洞检测提供了新的技术思路。

2.4.2 消融实验

本文在 FFmpeg+Qemu 和 Big-Vul 两个数据集上对 DEPVulDet 漏洞检测系统进行了消融实验。通过分别移除编码器模块、解码器模块以及增强漏洞解释推理训练模块,系统评估各关键组件对整体漏洞检测性能的贡献程度。DEPVulDet_{no_encode}为移除编码器模块的变体,保留解码器和增强漏洞解释推理模块;DEPVulDet_{no_exp}移除解码器模块的变体,仅保留编码器,其输出直接通过 MLP 和 Softmax 函数进行分类决策;DEPVulDet_{no_exp}为移除增强漏洞解释推理训练模块的变体,保留完整的编码器—解码器架构。

消融实验结果如表 4 和表 5 所示。实验表明,不同模块对漏洞检测性能影响显著。移除编码器(DEPVulDet_{no_encode})后, F1 值在 FFmpeg+Qemu 和

Big-Vul 数据集上分别降低 18.2 和 18.9 个百分点,证明编码器对捕获代码语义特征至关重要。仅保留编码器(DEPVulDet_{encode})时, F1 值分别下降 29.2 和 12.7 个百分点,凸显解码器在上下文整合与长程依赖分析中的关键作用。移除增强漏洞解释推理模块(DEPVulDet_{no_exp})对基础检测影响较小,准确率仅下降 13.8 和 2.5 个百分点,但该模块显著提升漏洞成因解释能力。综上所述,消融实验表明,编码器和解码器均为 DEPVulDet 的关键组件,但其相对重要性因数据集特性而异。在 FFmpeg+Qemu 数据集上,解码器对性能的贡献更突出(移除后 F1 值下降 29.2 个百分点);而在 Big-Vul 数据集上,编码器的作用相对更关键(移除后 F1 值下降 18.9 个百分点)。这表明不同数据集的代码特性与漏洞模式可能影响模型组件的重要性。此外,增强漏洞解释推理模块虽对基础检测精度的提升有限,但能显著增强系统的可解释性,具有重要实用价值。

表 4 在 FFmpeg+Qemu 数据集上的消融实验结果 单位: %

模型	准确率	精确率	F1 分数	召回率
DEPVulDet _{no_encode}	46.0	67.7	53.3	43.9
DEPVulDet _{encode}	41.7	69.2	42.3	30.4
DEPVulDet _{no_exp}	55.8	59.6	44.7	35.8
DEPVulDet	69.6	62.2	71.5	84.1

表 5 在 Big-Vul 数据集上的消融实验结果 单位: %

模型	准确率	精确率	F1 分数	召回率
DEPVulDet _{no_encode}	85.1	58.4	29.2	19.5
DEPVulDet _{encode}	89.3	63.8	35.4	24.6
DEPVulDet _{no_exp}	93.2	70.2	42.0	30.1
DEPVulDet	95.7	77.0	48.1	35.0

2.4.3 可解释性实验对比

为评估 DEPVulDet 的漏洞解释能力,本文选取表 6 所示的代表性漏洞案例,邀请 5 位具有 3 年以上漏洞分析经验的安全专家,对 DEPVulDet 与具有解释能力的深度学习方法及大语言模型进行系统性对比。评估主要从 3 个维度展开:漏洞定位准确性(漏洞行识别)、漏洞上下文相关性(相关代码片段识别)以及漏洞成因解释的合理性。专家对各方法的输出进行评分,分别计算各维度的准确率均值,作为最终评估指标。

本文通过专家评估实验,系统验证了 DEPVulDet 在漏洞可解释性方面的卓越性能。如表 7 所示,DEPVulDet 在漏洞行识别、上下文相关性和漏

洞原因的解釋3个核心维度上均取得显著优势,准确率分别达到 92.6%、91.4% 和 91.4%,显著优于 Llama-13B、DeepSeek 等对比方法。这一结果充分证实了 DEPVulDet 框架在融合程序依赖语义嵌入与解码器推理机制方面的有效性,不仅实现了高精度的漏洞检测,更通过基于思维链的多轮对话训练机制,显著提升了模型对漏洞位置、上下文依赖关系及漏洞原因的逻辑推理与解释能力。

表 6 可解释性研究漏洞案例集合

库	安全漏洞
LIBTIFF	CVE-2018-7456、CVE-2018-8905、CVE-2018-17000、CVE-2017-7598、CVE-2016-10267
Libxml2	CVE-2016-1837、CVE-2017-5969、CVE-2017-9047
Tcpdump	CVE-2014-8767、CVE-2014-8768、CVE-2014-8769
Libpcrc	CVE-2017-7186、CVE-2017-7244、CVE-2017-7246

表 7 DEPVulDet 与其他方法的解释性实验对比 单位:%

方法	漏洞行识别	上下文相关性	漏洞原因解释
Llama-13B	88.5	85.7	84.2
Qwen-13b	81.4	70.0	71.4
Vul-rag	71.4	70.0	74.2
DeepSeek-13b	80.0	78.5	72.8
DEPVulDet	92.6	91.4	91.4

3 结束语

提出了一种基于编码器与解码器协同工作的漏洞检测方法,其中编码器依托程序依赖信息捕获代码结构特征,重点提取数据流与程序依赖信息,并通过轻量级映射机制将编码结果投影至解码器语义空间,由解码器实现漏洞检测与解释任务。为进一步增强模型能力,使其不仅能够有效完成漏洞分类,还能准确识别漏洞所在行、关联上下文并生成合理解释,本文设计了增强型漏洞解释推理训练任务。总体而言,DEPVulDet 弥补了基于 GNN 方法在代码语义理解方面的不足,以及大语言模型对程序结构信息不敏感的缺陷,为漏洞检测研究提供了融合结构感知与语义理解的新思路。

4 参考文献:

[1] WEN X C, CHEN Y P, GAO C Y, et al. Vulnerability detection with graph simplification and enhanced graph representation learning[C]//Proceedings of 2023 IEEE/ACM 45th International Conference on Software Engineer-

ing. Piscataway, USA: IEEE, 2023:2275-2286.

[2] ZHANG G D, YAO T Y, QIN J W, et al. CodeSAGE: a multi-feature fusion vulnerability detection approach using code attribute graphs and attention mechanisms[J]. Journal of Information Security and Applications, 2025, 89: No.103973.

[3] LI Z, ZOU D Q, XU S H, 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.

[4] 杜江,魏强,王奕森,等.DeepClap:一种基于残差对齐网络的漏洞跨模态搜索方法[J].信息工程大学学报, 2025, 26(4):478-484.

[5] LEKSSAYS A, MOUHCINE H, TRAN K, et al. LLMx-CPG: context-aware vulnerability detection through code property graph-guided Large Language Models[DB/OL]. (2025-07-22) [2025-12-08]. <https://arXiv.org/abs/2507.16585>.

[6] QUAN V L A, PHAT C T, VAN NGUYEN K, et al. XGV-BERT: leveraging contextualized language model and graph neural network for efficient software vulnerability detection[J]. The Journal of Supercomputing, 2025, 81(6):No.750.

[7] WEN X C, GAO C Y, GAO S Z, et al. SCALE: constructing structured natural language comment trees for software vulnerability detection[C]//Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. New York, USA: ACM, 2024:235-247.

[8] LU G L, JU X L, CHEN X, et al. GRACE: empowering LLM-based software vulnerability detection with graph structure and in-context learning[J]. Journal of Systems and Software, 2024, 212:No.112031.

[9] JIANG Z H, SUN W F, GU X Y, et al. DFEPT: data flow embedding for enhancing pre-trained model based vulnerability detection[DB/OL]. (2024-10-24) [2025-06-07]. <https://arXiv.org/abs/2410.18479>.

[10] CAI Z L, CAI Y W, CHEN X, et al. CSVD-TF: cross-project software vulnerability detection with TrAdaBoost by fusing expert metrics and semantic metrics[J]. Journal of Systems and Software, 2024, 213:No.112038.

[11] VASILAKIS N, BENETOPOULOS A, HANDA S, et al. Supply-chain vulnerability elimination via active learning and regeneration[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2021:1755-1770.

[12] DEVLIN J, CHANG M W, LEE K, et al. Bert: pre-training of deep bidirectional transformers for language understanding[DB/OL].(2018-10-11)上网时间[2025-07-07]. <https://doi.org/10.48550/arXiv.1810.04805>.