软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(6):2543−2561 [doi: 10.13328/j.cnki.jos.006849]

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

基于图神经网络的切片级漏洞检测及解释方法*

胡雨涛[1,2,3], 王溯远[1,2,3], 吴月明[1,2,3], 邹德清[1,2,3], 李文科[2,3], 金 海[1,4]

[1](大数据技术与系统国家地方联合工程研究中心 (服务计算技术与系统教育部重点实验室 华中科技大学), 湖北 武汉 430074)

[2](分布式系统安全湖北省重点实验室, 湖北 武汉 430074)

[3](华中科技大学 网络空间安全学院, 湖北 武汉 430074)

[4](华中科技大学 计算机科学与技术学院, 湖北 武汉 430074)

通信作者: 吴月明, E-mail: wuyueming21@gmail.com

摘 要: 随着软件的复杂程度越来越高, 对漏洞检测的研究需求也日益增大. 软件漏洞的迅速发现和修补, 可以将漏洞带来的损失降到最低. 基于深度学习的漏洞检测方法作为目前新兴的检测手段, 可以从漏洞代码中自动学习其隐含的漏洞模式, 节省了大量人力投入. 但基于深度学习的漏洞检测方法尚未完善, 其中, 函数级别的检测方法存在检测粒度较粗且检测准确率较低的问题, 切片级别的检测方法虽然能够有效减少样本噪声, 但仍存在以下两方面的问题: 一方面, 现有方法大多采用人工漏洞数据集进行实验, 因此其在真实环境中的漏洞检测能力仍然存疑; 另一方面, 相关工作仅致力于检测出切片样本是否存在漏洞, 而缺乏对检测结果可解释性的考虑. 针对上述问题, 提出基于图神经网络的切片级漏洞检测及解释方法. 该方法首先对 C/C++源代码进行规范化并提取切片, 以减少样本冗余信息干扰; 之后, 采用图神经网络模型进行切片嵌入得到其向量表征, 以保留源代码的结构信息和漏洞特征; 然后, 将切片的向量表征输入漏洞检测模型进行训练和预测; 最后, 将训练完成的漏洞检测模型和待解释的漏洞切片输入漏洞解释器, 得到具体的漏洞代码行. 实验结果显示: 在漏洞检测方面, 该方法对于真实漏洞数据的检测 $F1$ 分数达到 75.1%, 相较于对比方法提升了 41.2%–110.4%; 在漏洞解释方面, 该方法在限定前 10%的关键节点时, 准确率可达 73.6%, 相较于两种对比解释器分别提升了 8.9%和 24.9%, 且时间开销分别缩短了 42.5%和 15.4%. 最后, 该方法正确检测并解释了 4 个开源软件中 59 个真实漏洞, 证明了其在现实世界漏洞发掘方面的实用性.

关键词: 漏洞检测; 深度学习; 图神经网络; 人工智能可解释性

中图法分类号: TP311

中文引用格式: 胡雨涛, 王溯远, 吴月明, 邹德清, 李文科, 金海. 基于图神经网络的切片级漏洞检测及解释方法. 软件学报, 2023, 34(6): 2543–2561. http://www.jos.org.cn/1000-9825/6849.htm

英文引用格式: Hu YT, Wang SY, Wu YM, Zou DQ, Li WK, Jin H. Slice-level Vulnerability Detection and Interpretation Method Based on Graph Neural Network. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2543−2561 (in Chinese). http://www.jos.org.cn/1000-9825/6849.htm

Slice-level Vulnerability Detection and Interpretation Method Based on Graph Neural Network

HU Yu-Tao[1,2,3], WANG Su-Yuan[1,2,3], WU Yue-Ming[1,2,3], ZOU De-Qing[1,2,3], LI Wen-Ke[2,3], JIN Hai[1,4]

[1](National Engineering Research Center for Big Data Technology and System (Key Laboratory of Services Computing Technology and System, Ministry of Education, Huazhong University of Science and Technology), Wuhan 430074, China)

* 基金项目: 国家自然科学基金(62172168); 湖北省重点研发计划(2021BAA032)
本文由“软件可信性与供应链安全前沿进展”专题特约编辑向剑文教授、郑征教授、申文博研究员、常瑞副教授、田聪教授推荐.
收稿时间: 2022-09-05; 修改时间: 2022-10-10; 采用时间: 2022-12-14; jos 在线出版时间: 2023-01-13

[2](Hubei Key Laboratory of Distributed System Security, Wuhan 430074, China)
[3](School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)
[4](School of Computer Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract: As software becomes more complex, the need for research on vulnerability detection is increasing. The rapid discovery and patching of software vulnerabilities is able to minimize the damage caused by vulnerabilities. As an emerging detection method, deep learning-based vulnerability detection methods can learn from the vulnerability code and automatically generate its implied vulnerability pattern, saving a lot of human effort. However, deep learning-based vulnerability detection methods are not yet perfect; function-level detection methods have a coarse detection granularity with low detection accuracy; slice-level detection methods can effectively reduce sample noise, but there are still the following two aspects of the problem: On the one hand, most of the existing methods use artificial vulnerability datasets for experiments, and the ability to detect vulnerabilities in real environments is still in doubt; on the other hand, the work is only dedicated to detecting the existence of vulnerabilities in the slice samples and the lack of interpretability of the detection results. To address above issues, this study proposes a slice-level vulnerability detection and interpretation method based on the graph neural network. The method first normalizes the C/C++ source code and extracts slices to reduce the interference of redundant information in the samples; secondly, a graph neural network model is used to embed the slices to obtain their vector representations to preserve the structural information and vulnerability features of the source code; then the vector representations of slices are fed into the vulnerability detection model for training and prediction; finally, the trained vulnerability detection model and the vulnerability slices to be explained are fed into the vulnerability interpreter to obtain the specific lines of vulnerability code. The experimental results show that in terms of vulnerability detection, the method achieves an *F*1 score of 75.1% for real-world vulnerability, which is 41.2%–110.4% higher than the comparative methods. In terms of vulnerability interpretation, the method can reach 73.6% *accuracy* when limiting the top 10% of critical nodes, which is 8.9% and 24.9% higher than the other two interpreters, and the time overhead is reduced by 42.5% and 15.4%, respectively. Finally, this method correctly detects and explains 59 real vulnerabilities in the four open-source software, proving its practicality in real-world vulnerability discovery.

Key words: vulnerability detection; deep learning; graph neural network (GNN); explainable AI

网络空间安全已成为国家安全的关键组成部分和重要保障. 近年来, 网络黑客勒索、僵尸网络攻击、用户信息泄露等各种网络空间安全事件频发, 给国家安全以及用户的人身财产造成了严重威胁. 漏洞是各种网络空间安全事件发起的根源, 软件系统作为网络空间的一个重要组成部分, 其本身的漏洞给网络空间带来了严重的安全威胁. 针对源代码进行漏洞检测, 是发现软件漏洞的主要途径之一[1], 且具有天然优势: 一方面, 从源代码中可以获得丰富的语义信息, 便于表示软件的漏洞特征; 另一方面, 目前, 开源软件得到了广泛且大量的应用, 其中的漏洞也随之广泛传播, 因此, 针对源代码检测具有明显的优势和意义. 随着机器学习, 特别是深度学习的兴起, 最近的研究开始利用深度学习模型进行源代码漏洞的检测[2−4], 其因为不依赖专家经验而减少了人工干预的主观性, 因此能够取得较低的误报率和漏报率.

现阶段, 基于深度学习的漏洞检测方法按照样本粒度可以分为函数级别和切片级别. 其中,

- 函数级别面向整个函数进行漏洞检测, 其优势在于函数能够涵盖较为全面的漏洞信息, 但同时也会引入大量漏洞无关的噪声语句, 导致检测的准确率欠佳;
- 切片级别的漏洞检测优化了样本粒度, 即去除了样本中漏洞无关的噪声语句, 进而使得模型能够学习到更为精准的漏洞特征. 然而, 已有的切片级漏洞检测工作主要依赖人工构造的漏洞数据集, 如 software assurance reference dataset (SARD)[5], 以证明其方法的有效性, 因此, 其在真实环境中的漏洞检测能力仍然存疑.

此外, 相关工作仅致力于检测出切片样本是否存在漏洞, 而缺乏对检测结果可解释性的考虑. 由于深度学习模型内部涉及到大量复杂的特征计算, 其所学习到的漏洞特征难以被安全分析人员所理解, 导致检测结果无法令人信服. 而模型的可解释性不但是决定用户是否能够信任检测模型的关键因素, 同时也能够帮助用户进一步分析漏洞成因, 进而快速修补漏洞.

针对上述问题, 本文提出了一种基于图神经网络的切片级漏洞检测及解释方法. 该方法首先针对 C/C++ 源代码进行规范化并提取切片, 以减少样本冗余信息干扰; 之后, 采用图神经网络模型进行切片嵌入得到其向量表征, 以保留源代码的结构信息和漏洞特征; 然后, 将切片的向量表征输入漏洞检测模型进行训练和预

测; 最后, 将训练完成的漏洞检测模型和待解释的漏洞代码输入漏洞解释器, 得到具体的漏洞代码行.

为了验证本文方法对于真实漏洞检测和解释的有效性, 本文采用真实漏洞数据集进行全部实验. 在漏洞检测方面, 本文选取了 3 个基于规则的漏洞检测工具(Checkmark[6]、FlawFinder[7]和 RATS[8])和 4 个基于深度学习的漏洞检测方法(TokednCNN[9]、StatementLSTM[10]、SySeVR[11]和 Devign[12])作为对比工具. 实验结果表明: 本文方法的 *F*1 分数可达 75.1%, 相较于其他漏洞检测方法提升了 41.2%–110.4%. 在漏洞解释方面, 本文选取了两种先进的图神经网络解释器(GNNExplainer[13]和 PGExplainer[14])作为对比解释器. 实验结果表明: 本解释方法在限定前 10%的关键节点时, 准确率可以达到 73.6%, 相较于对比方法分别提升了 8.9%和 24.9%, 且在时间开销方面分别缩短了 42.5%和 15.4%. 最后, 为了检验本文方法在现实世界漏洞发掘方面的实用性, 本文选取了 4 个真实的开源软件进行测试, 并成功检测和解释了开源软件中 59 个真实漏洞. 实验结果表明: 本文方法能够有效检测真实软件漏洞, 并提供可靠的检测结果解释.

本文的主要贡献如下:

(1) 提出了一种基于图神经网络的切片级漏洞检测方法. 通过提取代码切片, 有效降低了代码冗余, 减少了无关语句的干扰. 同时, 图神经网络可以更好地保留代码结构和漏洞语法语义信息, 使其能够针对复杂的真实漏洞数据达到较好的检测效果;

(2) 利用并改进了一种图神经网络解释器, 使通用的图神经网络解释器更适用于漏洞解释任务. 本文通过改进 GNNExplainer, 使其能够快速且准确地输出漏洞代码行, 增强了基于深度学习漏洞检测方法的可信度, 且有助于研究人员分析漏洞成因并修复漏洞;

(3) 本文方法扫描了 4 个开源软件的 3 312 657 个切片, 并通过人工分析和与已知漏洞模式匹配的方式验证检测和解释结果. 实验结果表明, 其正确检测并解释了 59 个真实漏洞, 证实了本文方法在现实世界漏洞发掘方面的实用性.

本文第 1 节介绍基于深度学习漏洞检测及解释的相关工作. 第 2 节介绍本文所需基础知识, 包括静态程序分析和图神经网络的相关概念. 第 3 节介绍本文基于图神经网络的切片级漏洞检测及解释方法的具体实现. 第 4 节通过对比实验验证本文方法在漏洞检测、漏洞解释和实际应用这 3 个方面的有效性. 最后总结全文, 并展望未来工作.

1 相关工作

本节将对本文涉及的相关工作进行介绍, 主要包括基于深度学习的漏洞检测和模型可解释性这两方面.

1.1 基于深度学习的漏洞检测相关工作

近年来, 由于深度学习模型强大的建模能力和特征学习能力而被应用于漏洞检测领域. 基于深度学习的漏洞检测方法可以按照样本粒度和模型类型这两个角度进行分类.

基于深度学习的漏洞检测方法根据待处理样本的粒度可以分为函数级别和切片级别.

- 在函数级别: Lin 等人[15]针对软件跨项目的特点, 首次采用深度学习模型在函数级别进行检测漏洞; Feng 等人[16]设计了一种基于树的漏洞检测器, 其首先用静态分析提取函数的抽象语法树, 然后应用前序遍历搜索将树转换为序列, 最后将该序列输入双向门控循环单元(biphasic gated recurrent unit, BGRU)模型进行训练和测试; Wang 等人[17]提出了 FUNDED 漏洞检测模型, 其通过提取代码的抽象语法树和程序控制依赖图, 并输入图神经网络完成漏洞检测; Wu 等人[18]将待测函数转换为图片后, 送入卷积神经网络进行漏洞检测;
- 为降低样本噪声对检测结果的影响, 研究人员优化了样本粒度, 并提出了更细粒度的切片级漏洞检测方法. 例如: Li 等人[19]提出的 VulDeePecker 首先针对待测程序中的敏感 API 提取程序切片, 然后利用双向长短期记忆网络(bidirectional long short-term memory, BiLSTM)训练漏洞检测器; Zou 等人[20]首次引入了代码注意力这一概念, 在程序切片构建时考虑程序控制依赖关系, 以包含更加全面的漏洞语义和语法信息, 并完成多类型的漏洞检测任务; Li 等人[11]提出的 SySeVR 对漏洞切片方法进一步加

以完善, 即在敏感 API 的基础上补充了数组使用、指针使用和整数使用作为漏洞切片的基准点. 该方法使得漏洞切片数量大幅提升, 更有利于深度学习模型的训练.

基于深度学习的漏洞检测方法根据模型类型可以分为基于文本模型和基于图模型两种方式.

- 基于文本的漏洞检测方法主要采用卷积神经网络模型和循环神经网络模型, 其思想是将代码看作普通文本进行向量表征, 并将向量输入深度学习模型进行检测. 例如: Russell 等人[9]提出的 TokenCNN 首先提取代码令牌序列, 然后使用卷积神经网络进行表征学习, 并将其输入到全连接层中进行分类, 以得到漏洞检测结果; Duan 等人[21]提出的 VulSniper 通过提取代码属性图并转换为特征张量送入 BiLSTM 模型完成漏洞检测. 然而, 代码不同于普通文本, 其具有更为丰富的结构和语义信息, 因此, 将代码看作文本处理的方式会在一定程度上损失代码信息;
- 代码表征图能够充分表征代码结构和语义[22], 因此, 研究人员提出将代码表征图输入图神经网络模型进行漏洞检测. 例如: Zhou 等人[12]采用通用的图神经网络模型进行漏洞检测, 其模型包含的图卷积模块可以有效地从函数的图表征中学习到函数级别的漏洞特征; Cheng 等人[23]提出了 DeepWukong, 其利用图神经网络(graph neural network, GNN)实现了面向代码程序依赖图的子图的 C/C++函数过程间漏洞检测. 但是过程间的漏洞检测难以应用于真实场景, 原因在于真实软件的函数间调用关系的复杂性使得调用链的长度难以得到控制, 从而导致图过于复杂而难以进行有效检测.

函数级漏洞检测的优势在于函数能够覆盖相对完整的漏洞特征, 但同时也会引入较多与漏洞无关的噪声语句. 尤其是在真实世界的漏洞检测场景下, 函数包含的代码行数过多, 样本中存在过多的噪声会对深度学习模型造成干扰, 使其无法学习到精确的漏洞特征而影响检测结果的准确性. 因此, 为了保证本系统对于真实漏洞检测任务的有效性, 本文选择构建切片粒度的漏洞检测模型. 此外, 根据 Chakraborty 等人[24]的调研结果, 图比文本能够更好地表达代码的结构和语义信息, 因此在漏洞检测领域, 基于图模型的检测方式效果明显优于基于文本模型的检测方式. 因此, 本文采用代码切片(即程序依赖图子图)训练图神经网络模型, 以实现对真实漏洞的准确检测.

1.2 基于深度学习漏洞解释的相关工作

目前, 图神经网络的可解释性工作可以分为基于扰动的解释方法、基于特征的解释方法、基于模型的解释方法和基于分解的解释方法这 4 种类型. 其中, 基于模型的解释方法是面向模型而设计的解释方法, 其他类型的解释方法均是面向单个实例提供检测结果的解释. 在漏洞解释任务中, 研究人员更倾向于了解每一个漏洞的漏洞原理而非理解深度学习模型本身的运作机制. 因此, 面向实例的解释方法更适用于漏洞检测的解释任务.

基于模型的解释方法侧重于对模型整体进行解释, 即对深度学习模型本身提供解释方案, 以对模型的工作方式提供高层次的见解和一般性的理解, 从而方便研究人员有针对性地对模型进行优化. 例如: Yuan 等人[25]提出训练一个图生成器, 其生成的图模式可以对深度图模型进行解释, 即根据已训练图模型的反馈, 使用策略梯度对图生成器进行训练. 基于分解的解释方法通过将原始模型预测分解为若干项来衡量输入特征的重要性, 这些项被视为相应输入特征的重要性分数. 例如: Schnake 等人[26]提出的 GNN-LRP 解释器研究了图神经网络中不同相关步长(记录了层与层之间消息传递的路径)的重要性, 通过将分数分配给不同的步长, 并从其对应节点上获得分数进行解释. 但其计算复杂度过高, 导致实际使用时受到一定的限制. 基于特征的解释方法依靠梯度或隐藏图特征表示不同输入特征的重要性(其中, 梯度或特征值越高, 表示其重要性越高), 被广泛运用于图像和文本任务. 例如: Selvaraju 等人[27]提出的 Grad-Cam 解释器是一种基于图像分类器的解释模型, 该方法利用梯度表示不同输入特征的重要程度. 该方法可以拓展应用到图模型以衡量不同节点的重要性, 其关键思想是, 结合隐藏特征图和梯度来指示节点重要性. 但是, 基于特征的解释方法在面向图神经网络解释时, 对图神经网络的结构有特殊要求, 从而不具有普适性. 相反地, 基于扰动的解释方法由于不受限制于模型的网络结构而被广泛用于解释图神经网络模型. 例如: Ying 等人[13]提出的 GNNExplainer 解释器可以从图的结构和节点角度来对任意图神经网络生成解释, 探寻与预测结果相关性最大的子图结构, 以实现对预测的解

释; Luo 等人[14]提出的 PGExplainer 解释器的主要思想与 GNNExplainer 相似, 但其在对图神经网络模型解释过程前需要进行解释器的参数训练, 以实现同时对多个实例从模型的全局视角进行解释, 并输出对模型预测结果起关键作用的子图结构; Yuan 等人[28]提出的 SubgraphX 模型旨在解释图神经网络模型在预测过程中重要子图的作用, 具体来说, 其将预训练的待解释模型和图实例数据, 与蒙特卡罗树搜索法相结合输出图实例中的重要子图.

目前, 基于深度学习的漏洞解释工作尚处于起步探索阶段, 相关工作较少. Li 等人[29]提出的 IVDetect 工具利用图神经网络解释器 GNNExplainer 实现函数级漏洞检测结果解释. 不同于已有的 IVDetect 的解释工作, 本文将使用并改进 GNNExplainer 对切片级的漏洞检测结果进行解释, 一方面, 切片含有较少的噪声信息, 能够获得更好的解释效果; 另一方面, 被解释对象为切片(程序依赖图子图), 因其包含的节点数相较完整的程序依赖图较少, 故可提高解释效率.

2 基础知识

本节将对本文涉及理论的基础知识进行介绍, 主要包括静态程序分析和图神经网络的相关概念.

2.1 程序静态分析相关概念

程序静态分析方法主要基于代码抽象语法树、控制流图、程序依赖图等代码表征图, 相关概念介绍如下.

- 抽象语法树(abstract syntax tree, AST): AST 是一种树形结构的源代码抽象表示, 其从根节点开始, 将代码依次分解为代码块、语句、声明、表达式等. AST 主要用于描述代码的语法结构, 是构成其他代码表征图的基础;
- 控制流图(control flow graph, CFG): CFG 是一种有向图, 其节点表示函数语句, 边则表示相邻语句间运行的先后关系, 描述了程序在执行过程中可能经过的所有运行路径;
- 数据依赖: 设 A 和 B 是两个代码语句, 若 A 语句中计算得到的变量在 B 语句中使用, 则称 B 语句数据依赖于 A 语句;
- 控制依赖: 设 A 和 B 是两个代码语句, 若 B 语句能否执行取决于 A 语句执行的结果, 则称 B 语句控制依赖于 A 语句;
- 程序依赖图(program dependence graph, PDG): PDG 是一种带有标记的有向多重图, 其基于 AST 的节点构建, 节点表示代码语句, 边表示相邻节点存在数据依赖或控制依赖关系;
- 代码属性图(code property graph, CPG): CPG 将 AST、CFG 和 PDG 整合至一种数据结构, 其包含源代码中的所有语法特征和语义特征;
- 代码切片: 代码切片由彼此之间具有数据依赖或控制依赖关系的语句构成[19].

2.2 图神经网络

传统的卷积神经网络和循环神经网络在提取欧氏空间数据(如自然语言)的特征方面取得了巨大的成功, 但其处理非欧式空间的图数据(如社交网络)时表现不佳. 而图神经网络在处理非欧式空间的图数据的表达能力上, 与传统深度学习模型相比具有明显优势, 因此, 图神经网络在推荐系统、社交网络分析和交通预测等领域被广泛应用. 图神经网络模型的工作原理为: 首先, 构建相邻节点之间需要传递的消息; 之后, 收集邻居节点相关的消息; 然后, 更新节点表示. 其中, 信息传递工作主要是通过取出每个节点及其邻居节点的信息; 聚合函数聚合以上所有信息; 最后, 更新函数对节点信息进行更新, 从而捕获节点间相互依赖的关系, 更好地学习图结构数据的相应特征.

图神经网络模型根据具体解决的任务级别可以分为如下 3 种.

- 图级别任务: 图级别任务的目标是预测整个图的属性. 例如, 本文漏洞检测即为一个图级别任务, 其目的是判断输入的代码表征图是否存在漏洞;
- 节点级任务: 节点级任务的目标是预测图中每个节点的类型. 例如, 社交关系图中判断某个节点的所

属阵营;

- 边级任务: 边级任务的目标是预测每条边的属性. 例如: 在目标检测的语义分割任务中不仅需要识别每个目标的类型, 还需要预测各个目标之间的关系.

目前, 图神经网络根据模型类型主要分为以下 3 种[30].

- 图卷积网络(graph convolutional network, GCN): GCN 将卷积运算推广到图数据, 其核心思想在于学习一个函数映射, 通过该映射, 可以聚合图中节点及其邻居节点特征, 从而生成该节点的新表示. 卷积运算又可以具体分为基于谱的方法和基于空间的方法, 其中, 基于谱的方法在训练时需要将全图加载入内存, 因此在处理复杂图时性能较差; 由于基于空间的图卷积方法的中心节点、感受域和聚合函数不确定, 导致相互制约依赖, 实际使用效果不佳;
- 图注意力网络(graph attention network, GAT): GAT 在 GNN 上引入注意力机制, 在聚合过程使用注意力关注对任务有影响的邻居节点信息, 进而为节点分配权重, 使得 GNN 重点关注与任务相关的节点和边, 以提升模型效果. 但其增加了计算邻居之间注意权重的时间开销和内存消耗, 同时, 注意力机制的实际作用又与网络初始化相关;
- 图循环网络(graph recurrent network, GRN): GRN 将图转换为序列, 使用长短期记忆网络(long short-term memory, LSTM)或门控循环单元(gated recurrent unit, GRU)等循环神经网络作为架构训练. 相比于其他 GNN 模型, GRN 在信息传递过程中使用门控机制, 可以提高在整个图上的长期信息传播能力及模型表征能力. 本文使用的门控图神经网络(gated graph neural network, GGNN)模型即为 GRN 类型下的一种网络结构.

3 基于图神经网络的切片级漏洞检测及解释方法

为了解决现有漏洞检测模型在真实漏洞数据集上效果差且无法给出检测结果解释的问题, 本文提出并实现了一种基于图神经网络的切片级漏洞检测及解释方法. 如图 1 所示为本文方法的整体框架图, 该方法由数据预处理模块、漏洞检测模块和漏洞解释模块这 3 个模块组成. 系统的输入是待测软件的源代码, 输出是目标程序切片中是否存在漏洞以及具体的漏洞语句.

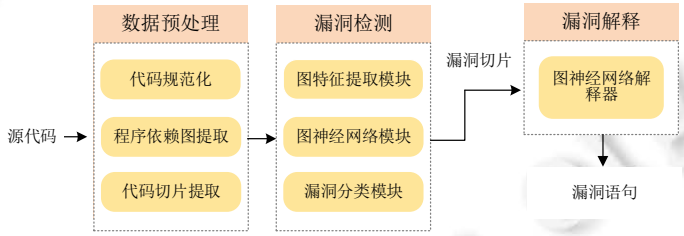

图 1 基于图神经网络的切片级漏洞检测及解释方法整体框架

3.1 数据预处理

3.1.1 代码规范化

程序源代码含有丰富的语义信息, 便于表示漏洞特征. 因此, 本系统面向源代码进行漏洞检测. 但同时, 源代码中代码注释、复杂的变量和函数命名等与代码语义无关的信息会干扰深度学习模型的训练和预测. 为了降低代码中无关信息的干扰, 本系统首先对源代码进行代码规范化处理. 考虑到真实代码较为复杂难以展示, 因此, 图 2 所示源代码来自于人工数据集 SARD.

如图 2 中代码规范化部分所示, 代码规范化共包含如下具体 3 个步骤.

步骤 1: 去除代码中的注释信息(例如/**/);

步骤 2: 将用户自定义的变量名映射为统一的变量名(例如 *VAR*1);

步骤 3: 将用户自定义函数名映射为统一函数名(例如 *FUNC*1).

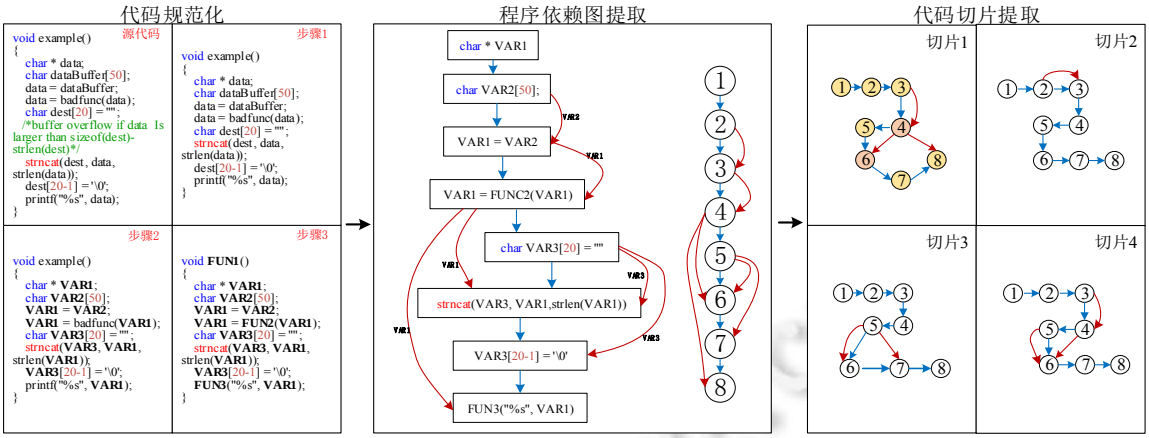

图 2 数据预处理样例

3.1.2 程序依赖图提取

代码表征图是一种有效的代码表征方式, 可以直观地体现代码的语义、语法和结构信息. 其中, 程序依赖图是一种由数据依赖关系和控制依赖关系连接抽象语法树节点的数据结构, 其不同于抽象语法树、控制流图等仅包含语法或控制流等单一代码信息的图表征方式, 且不同于代码属性图包含过多未精简的代码信息而影响模型的检测效率. 程序依赖图是一种高效且较为全面的代码图表征方式, 且能够有效地表征多种类型漏洞[24]. 因此, 本文选择程序依赖图作为源代码的表征方式.

本文利用开源的静态代码解析工具 Joern[31]对源代码进行程序依赖图提取. 选择该工具的原因在于, 其无需预编译即可完成代码解析和程序依赖图生成工作, 这意味着其适用于任意粒度的待测代码, 极大程度地降低了用户的使用成本. 如图 2 程序依赖图提取部分所示, 其中, 黑色框表示代码行对应的节点, 蓝色边表示控制依赖关系, 红色边表示数据依赖关系, 红色边旁的变量表示数据依赖的变量名称. 为了更简洁地展示后续工作, 本文用编号替换程序依赖图节点中的代码, 如图 2 程序依赖图提取的右半部分所示.

3.1.3 代码切片提取

一方面, 函数级样本中包含大量漏洞无关的噪声语句会干扰模型学习漏洞特征, 进而影响检测效果; 另一方面, 真实软件的代码语句较为复杂, 因此其程序依赖图中包含节点和边的数量过于庞大, 导致训练和解释过程需要大量的时间和内存开销. 为避免上述问题, 本文的漏洞检测和解释过程均面向切片级别以去除漏洞无关信息, 从而提高检测和解释效果并降低开销. 具体而言, 本文采用 Li 等人[19]提出的漏洞切片概念作为切片生成的指导思想. 其工作最后总结得出, 软件漏洞主要由指针、数组、表达式运算、敏感 API 函数所在位置引入, 并称其为漏洞关注点. 如常见的数组越界、整数溢出、空指针、API 函数错误使用等类型漏洞, 均由以上 4 类漏洞关注点所致. 在程序代码中, 与漏洞关注点存在数据依赖或者控制依赖关系的语句集合构成一个可能存在漏洞的程序切片; 反之, 其他语句被视为会干扰模型训练的漏洞无关语句. 不同于 Li 等人[19]构造的文本切片, 本文提取程序依赖图的子图作为切片, 即选取上述 4 类漏洞关注点作为程序切片的基准点后, 保留与其存在数据及控制依赖关系的节点和边, 以生成程序依赖图子图. 具体来说, 生成代码切片可以分为 3 个步骤.

(1) 漏洞关注点选取. 通过遍历程序依赖图节点, 选取符合 4 类漏洞关注点的代码元素, 并记录该节点为切片基准点. 具体来说: 通过在标识符声明节点中匹配“[”字符来确定数组元素; 通过在标识符声明节点中匹配“*”字符来确定指针元素; 通过正则表达式规则来匹配表达式运算节点; 通过 Li 等人[19]提供的敏感 API 列表进行敏感 API 节点匹配. 如图 2 中程序依赖图提取部分所示, 1 号节点存在指针元素 *VAR*1, 2 号和 5 号节点分别存在数组元素 *VAR*2 和 *VAR*3, 6 号节点存在敏感 API 元素

strncat. 因此, 以上节点被选定为程序切片的基准点;

(2) 程序切片生成. 从切片基准点出发, 分别执行前向及后向切片以生成程序切片. 具体而言, 在程序依赖图上, 以漏洞关注点所在节点为起点, 分别追溯前向、后向的控制依赖边和数据依赖边, 并记录涉及到的节点和边, 直到不再出现新增节点和边为止. 根据上述步骤得到的程序依赖图子图即为一个程序切片. 因其只包含与漏洞关注点具有依赖关系的节点和边, 故在保留源代码结构信息的同时, 排除了图中漏洞无关的信息. 图 2 中代码切片提取部分展示了从该源代码的程序依赖图中提取的部分切片, 切片 1–切片 4 分别为以 *VAR*1、*VAR*2、*VAR*3 和 strncat 为切片基准点获得的切片;

(3) 程序切片标注. 本文依赖漏洞的补丁信息对切片进行标注, 即: 包含该漏洞补丁中删减行的切片被标注为有漏洞切片, 反之被认为是无漏洞切片. 根据其提供的漏洞信息, 漏洞补丁的删减行为源代码的第 11 行, 即对应程序依赖图中的 6 号节点. 根据标注规则, 由于图 2 的切片 1–切片 4 均包含 6 号节点, 因此均被标注为有漏洞切片.

3.2 漏洞检测

本文采用图神经网络模型进行漏洞检测. 图神经网络模型自动学习输入图的漏洞特征, 并对其特征向量进行分类以检测代码是否含有漏洞, 其主要分为图特征提取模块、图神经网络模块和漏洞分类模块, 如图 3 所示.

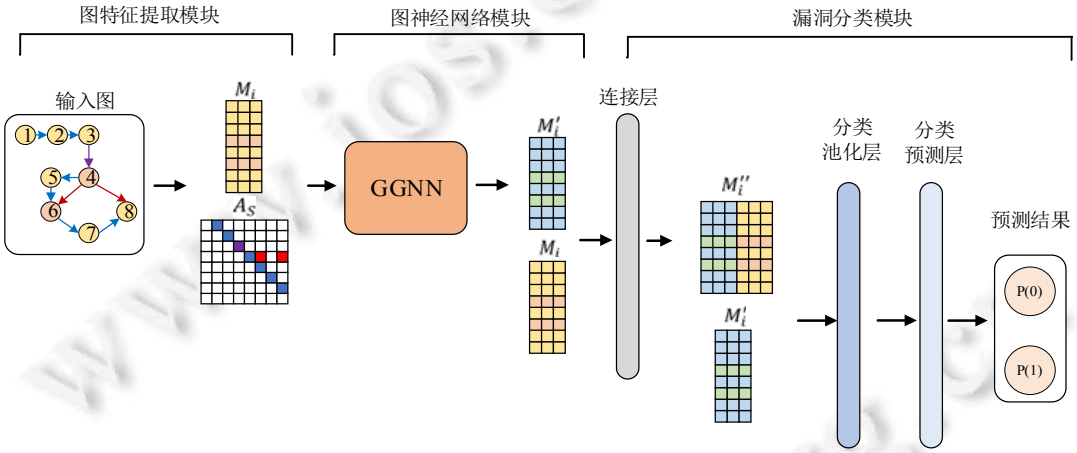

图 3 漏洞检测模型

3.2.1 图特征提取模块

由于本文抽取的代码切片是抽象图格式的文件(程序依赖图子图), 无法直接输入图神经网络漏洞检测模型, 因此需要提取图的关键特征以获得图特征向量. 程序依赖图子图包含两个维度的特征, 即节点内的代码特征(下文简称为节点特征)和图结构特征.

针对节点特征, 本文对节点内代码进行嵌入表征. 具体地, 将每个节点中的代码视为一个句子, 并分解句子为令牌列表后, 嵌入得到一个固定长度的向量. 本文使用 word2vec 模型[32]实现节点嵌入, 其采用一种分布式表示的思想, 将令牌映射为整数再转换为一个固定长度的向量. 该方法在训练嵌入速度上, 相对于传统嵌入工具而言速度更快且具有泛用性, 因此在文本挖掘领域中被广泛使用. 具体来说, 本文将所有切片中的令牌列表训练生成一个预训练的 word2vec 模型, 再使用该预训练模型对全部节点进行向量嵌入. 即: 将预处理得到的切片输入预训练的 word2vec 模型, 输出其形状为 $m\times n$ 的特征矩阵 M_i, 其中, M_i 表示切片的节点数, n 表示嵌入向量的维度. 考虑到实际检测效果和性能, 本文设置 n 为 100 维. 如图 3 的输入图所示, 图中共有 8 个节点, 因此, 其节点特征向量 M_i 的维度为 8×100.

针对图结构特征, 本文对图中的边关系进行嵌入表征. 每一条边可被视为一个三元组(起始节点,终止节

点,边类型). 其中, 起始节点和终止节点均能够从程序依赖图中直接获取, 边类型分为数据依赖边和控制依赖边. 如图 3 的输入图所示, 其具有 10 条边, 其中包括 3 条数据依赖边和 7 条控制依赖边(红色边表示数据依赖边, 蓝色边表示控制依赖边, 紫色边表示既有数据依赖边又有控制依赖边), 输出矩阵 A_S 即表示图结构特征矩阵.

3.2.2 图神经网络模块

由于本文将源代码转化为具有数据依赖关系和控制依赖关系的图结构数据, 而图神经网络有助于进一步聚合传递更新以便更好地捕捉图的结构和语义信息, 因此, 本文采用图神经网络对图样本作进一步的特征嵌入. 目前, 深度学习领域提出了多种基于聚合邻域信息的图神经网络模型[33−35]. 其中, GGNN 增强了网络的长期记忆能力, 比传统的图神经网络模型更加深入[35], 更适用于处理既有语义也有图结构的数据. 因此, 本文选用 GGNN 为最终的图神经网络模型.

GGNN 的原理在于: 聚合节点及其邻居节点的信息, 将聚合后的节点和当前节点一起送入 GRU 以得到下一时刻的当前节点, 按此过程重复, 经过若干个时间步迭代, 会生成所有节点的最终节点特征. 如图 3 图神经网络模块所示, 输入图特征 $g_i(M_i,A_S)$后, GGNN 通过嵌入每个节点及其邻域, 将其转换为一个长 $m\times n'$的切片特征矩阵 M_i', 其中, n'为设置的最终切片特征大小, 本文设置为 200 维, 则图中的特征矩阵 M_i' 维度为 8×200.

具体来说, 对于图中的每一个节点 v_u, 初始化节点向量 $h_u^{(1)}=[m_u^{\top},0]^{\top}$, 即复制 v_u 节点的特征向量并额外填充 0. 设 T 为邻域聚合的总时间步数, 为了得到整个图传播的信息, 对于每个时间步长 $t\leqslant T$, 所有节点根据所依赖的边进行通信, 即:

$$a_u^{(t)}=A_u^{\top}(W_u[h_1^{(t)^{\top}},...,h_m^{(t)^{\top}}]+b) \tag{1}$$

其中, W_u 表示可训练的参数, b 为偏差, $A_u^{\top}$ 表示节点 v_u 对应于 A_S 的邻接矩阵, $a_u^{(t)}$ 为当前节点和相邻节点间通过边的相互作用所得结果. 最后, 通过聚合函数 AGG 聚合节点 v_u 的信息并与上一个时间步合并, 从而获取该节点的新状态, 即:

$$h_u^{(t+1)}=GRU(h_u^{(t)},AGG(\{a_u^{(t)}\})) \tag{2}$$

3.2.3 漏洞分类模块

漏洞分类模块的核心思想在于: 选择与漏洞特征相关的特征集合, 以完成图级别的漏洞样本分类任务. 先前的工作[36]提出: 在图卷积层后使用一个分类池化层(SortPooling), 从而实现对图卷积层输出特征的排序, 即可输入传统的神经网络对其进行训练, 以提取切片向量嵌入中的有用特征. 因此在本文中, 节点特征通过 GGNN 层学习, 进而使用一维卷积和全连接层学习与图分类任务相关的特征, 以便更有效地进行分类. 具体来说, 本文定义分类池化层 $\tau(M)$的表述如下式:

$$\tau(M)=MaxPool(Relu(BN(Conv(M)))) \tag{3}$$

其中, *Conv* 表示卷积层, *BN* 表示 *BatchNorm* 层, *Relu* 表示激活函数, *MaxPool* 表示最大池化层, M 表示一个特征矩阵.

如图 3 漏洞分类模块所示, 本文将切片的节点特征矩阵 M_i 和相应切片特征矩阵 M_i' 连接成一个新的矩阵 M_i'', 分别对 M_i' 和 M_i'' 执行 τ分类池化操作, 得到输出 Y_1 和 Y_2; 接着, 将 Y_1 和 Y_2 分别送入输出维度为 2 的全连接层中; 最后对两个输出值相乘求出平均后进行 Sigmoid 分类, 从而得到预测, 即图 3 所示的分类预测层, 如下式所示:

$$P=Sigmoid(Avg(Liner(Y_1)\cdot Liner(Y_2))) \tag{4}$$

其中,

- *Avg* 表示平均操作;
- *Liner* 表示全连接层;
- P 是输出的二分类结果, 由 2 个维度组成. 其中, 第 1 个维度表示结果无漏洞的概率, 第 2 个维度表示结果有漏洞的概率.

最后, 模型采取两者中较大的概率作为漏洞分类的最终结果输出. 表 1 给出了本文在训练模型中使用到

的具体参数信息. 本文模型使用交叉熵损失函数 CrossEntropyLoss 纠正错误的分类, 同时使用学习率为 0.000 1 且权重衰退为 0.001 的 Adam[37]优化算法, 以训练第 3.2.2 节公式(1)中图神经网络模块的参数 W_u 和 b. 获得训练好的模型后, 再用其判断新的代码切片是否存在漏洞.

表 1 模型训练相关参数设置

参数	设置值
损失函数	CrossEntropyLoss
优化算法	Adam
学习率	0.000 1
权重衰退	0.001
批量大小	8
训练轮数	500

3.3 漏洞解释

本文在漏洞解释部分使用并改进了 GNNExplainer 图神经网络解释器[13], 从而对漏洞检测模型输出的检测结果提供了细粒度的解释, 即具体的漏洞代码行, 以帮助研究人员完成漏洞成因分析及进一步的漏洞修复.

GNNExplainer 是一种针对 GNN 模型设计的基于扰动的解释方法, 它不依赖具体的图神经网络模型, 具有很强的泛用性. 其根本思想是: 完成一个最大化互信息的优化任务, 对输入的实例图生成相应的边掩码进行更新训练, 最终根据预测结果的变化输出图中的关键信息作为解释结果. 在掩码更新训练过程中, 每次训练可以得到一个包含关键信息的新图, 将新图输入到已训练的 GNN 模型中, 根据检测结果评估边掩码的重要程度, 同时继续重复训练以降低损失, 并以找到一个损失最小的关键信息为训练的终点. 最后, 解释器通过边掩码得到重要边, 并根据重要边抽取出重要子图作为模型对于该实例的解释.

具体来说, GNNExplainer 的目标是从原图 $G_W(M_i,A_S)$中学习一个边掩码 E_M 以获取关键子图 G_S 作为解释结果. 其出发点是最大化子图 G_S 和原图 G_W 互信息 MI, 即保证子图尽可能地涵盖原图的重要信息, 如下式所示:

$$\mathrm{Max}(H(Y\mid G=G_S))=-E_{Y\mid G_S}\log P(Y\mid G=G_S) \tag{5}$$

其中, Y 表示漏洞检测模型预测的结果. 显然, 熵 $H(Y)$对于完成预训练的模型而言是一个常数, 故上式可等价为最小化条件熵 $H(Y|G=G_S)$, 根据条件熵定义, 有下式:

$$\mathrm{Min}(H(Y\mid G=G_S))=-E_{Y\mid G_S}\log P(Y\mid G=G_S) \tag{6}$$

即表示, 在解释结果为 G_S 时, 预测结果 Y 尚存在多大程度的不确定性. 然而, 上式无法直接优化.

GNNExplainer 在特殊情况下将 G_S~g 视为一个随机的图变量, 因此上式变为

$$\mathrm{Min}_g(E_{G_S\sim g}H(Y\mid G=G_S)) \tag{7}$$

GNNExplainer 进一步利用 Jenson 不等式以及凸性假设, 将上式转换为

$$\mathrm{Min}_g(H(Y|G=E_g[G_S])) \tag{8}$$

其中, $E_g[G_S]$可被边掩码所代替. 因此, 解释器最后训练的目标实际为边掩码. 训练完成后, 输出的程序依赖图关键子图 G_S 作为解释结果. 从上述描述可知: GNNExplainer 通过边掩码进行重要边的排序, 进而生成重要子图. 然而, 在漏洞检测任务中, 研究人员更倾向于知道是哪些代码行(即图节点)存在漏洞, 而非程序依赖图中的哪些数据控制依赖边存在漏洞. 因此, 事实上并不需要额外生成重要子图的操作. 为了解决上述问题, 本文为 GNNExplainer 设计并增加了一个节点排序算法, 使其能够按节点重要性排序并输出图中的关键节点, 使得解释器更适用于漏洞检测任务, 可以明显提高解释准确率, 并降低原方法的时间开销(减少了解释器生成子图和无关掩码的计算过程). 具体方法是: 首先, 将边掩码按重要程度排序; 其次, 为边掩码对应的边节点累加权重, 其权重为边掩码的大小; 最后, 对每个节点按权重大小进行排序, 即可得到按重要程度排序的节点序列. 具体来说: 对于图中的节点 i, 其权重大小用 $N_M^{(i)}$ 表示(初始所有节点权重均为 0). 用 EI_l 表示含有节点 i 的边号, 则 $E_M^{(EI_i)}$ 表示边号为 EI_i 的边掩码大小. 那么, 节点 i 的权重更新公式可以表示为

$$N_M^{(i)}=N_M^{(i)}+E_M^{(EI_i)} \tag{9}$$

本文的漏洞解释模型如图 4 所示, 解释器的输入为漏洞检测器检测出的漏洞切片, 解释器通过扰动输入切片的边掩码以生成新图, 并观察漏洞检测模型对新图的检测结果, 从而获得该边掩码的重要分数, 最后将边掩码的重要分数映射成为节点的重要分数并按照分数排序输出. 其中, 分数越高的节点则被认为更有可能是一个漏洞代码行, 即对漏洞检测结果的解释.

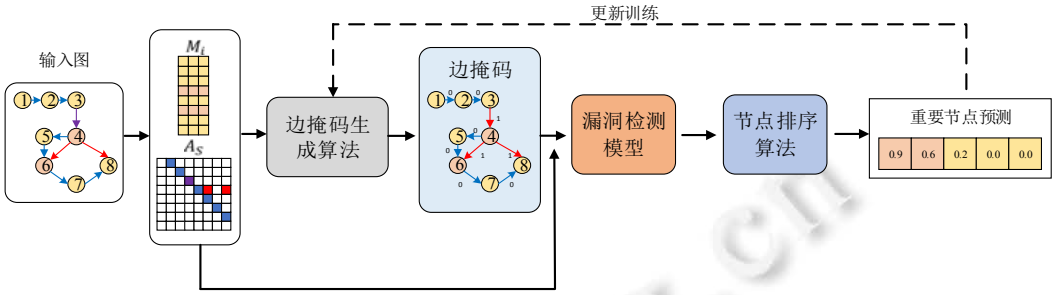

图 4 漏洞解释模型

4 实验分析

本节主要对具体实验环节进行阐述: 首先, 对实验准备加以说明, 包括实验数据集、实验具体实施方法以及实验评估指标; 其次, 引入 3 个研究问题, 以验证本文方法的实验效果; 最后, 对实验结果进行分析.

4.1 实验准备

(1) 实验数据集

本文在广泛使用的漏洞数据集 Big-Vul[38]的子集上进行了全部实验. 一方面, Big-Vul 数据集是一个高质量的真实漏洞数据集, 其涵盖了从 2002–2019 年间 348 个真实开源项目的全部 CVE (common vulnerabilities and exposures[39])条目, 共有 11 834 个漏洞函数和 253 096 个无漏洞函数; 另一方面, 它涵盖了修补前后的完整代码以及具体的漏洞修补位置, 以便于进行切片标注. 因此, 本文选择 Big-Vul 作为实验数据集.

需要说明的是: Big-Vul 数据集爬取了含有具体漏洞补丁的源文件中的所有函数, 因此补丁前的函数被标注为有漏洞函数, 补丁后的函数被标注为无漏洞函数, 对于没有任何修改且仍然保留的函数则标注为无漏洞. 在实际预处理过程中, 切片标注依赖于漏洞补丁中的修改行信息, 导致部分数据由于无法标注而需要过滤. 因此, 本文最终使用了 Big-Vul 数据集中的 4 030 个有漏洞函数和 4 716 个无漏洞函数, 并从中获得了 16 260 个有漏洞的切片和 16 301 个无漏洞的切片.

(2) 具体实施

本文实验设备内存大小 128 GB, 配有 16 核的英特尔 Xeon 处理器和英伟达 Quadro GTX 5000 系列 16 GB 显卡. 系统各个阶段分别采用 Joern[31]、word2vec[32]和 PyTorch[40]等工具实现. 在漏洞检测部分, 本文根据切片样本的数量将数据集按照 8:1:1 随机划分为训练集、验证集和测试集. 在漏洞解释部分, 由于只有漏洞切片需要被提供漏洞解释, 因此本文随机选择了测试集中预测正确的 1 399 个有漏洞切片输入解释器中进行解释.

(3) 评估指标

为评估漏洞检测模型的有效性, 本文采用常见的评价模型准确程度的指标对漏洞检测模型进行评估[11,17].

- 真正例(true positive, TP): 正确预测为有漏洞的样本数量;
- 真负例(true negative, TN): 正确预测为无漏洞的样本数量;
- 假正例(false positive, FP): 错误预测为有漏洞的样本数量;
- 假负例(false negative, FN): 错误预测为无漏洞的样本数量;
- 准确率(accuracy): $accuracy=(TP+TN)/(TP+TN+FP+FN)$;
- 召回率(recall): $recall=TP/(TP+FN)$;

- 精确率(precision): $precision=TP/(TP+FP)$;
- $F1=2\times precision\times recall/(precision+recall)$.

为评估解释模型的有效性, 本文借鉴 IVDetect[29]的准确率评价指标对漏洞解释模型进行评估. 具体而言: 若输出的节点中包含漏洞补丁修改行, 则认为解释正确; 反之, 则认为解释错误. 解释结果的准确率为解释正确的样本数量在全部样本数量的比值.

4.2 实验结果与分析

本文所设计的实验用于回答以下 3 个研究问题.

RQ1: 本文方法在源代码漏洞检测方面的表现如何?

RQ2: 本文方法对于漏洞检测结果的解释效果如何?

RQ3: 本文方法是否可以检测并解释真实软件漏洞?

4.2.1 RQ1: 本文方法在源代码漏洞检测方面的表现如何?

为回答该研究问题, 本文将所提方法与几种先进的漏洞检测工具进行比较. 对比方法包括 3 个基于规则的漏洞检测工具(Checkmark[6]、FlawFinder[7]和 RATS[8])和 4 个基于深度学习的漏洞检测方法(TokednCNN[9]、StatementLSTM[10]、SySeVR[11]、Devign[12]). 实验结果见表 2.

表 2 与其他漏洞检测工具对比

类别	工具或方法	精确率(%)	召回率(%)	*F*1 分数(%)
基于规则方法	Checkmark	40.4	31.9	35.7
基于规则方法	FlawFinder	39.6	33.4	36.2
基于规则方法	RATS	41.6	39.1	40.3
基于令牌方法	TokenCNN	43.2	54.3	48.1
基于语句方法	StatementLSTM	48.9	57.5	52.9
基于函数级方法	Devign	51.8	54.7	53.2
基于切片级方法	SySeVR	49.7	53.8	51.7
基于切片级方法	本文方法	70.5	80.3	75.1

基于规则的方法是将专家预定义的漏洞模式与源码进行匹配的漏洞检测方法. 如表 2 中精确率、召回率和 *F*1 分数所示, 商用静态漏洞检测器(Checkmark)和静态分析系统(FlawFinder 和 RATS)的检测效果均不理想. 例如, Checkmark 的召回率只有 31.9%, 说明其对于真实世界的漏洞仅能成功检测 31.9%. 这些工具依赖于人类专家制定的规则和模式, 通过词法分析对目标程序的代码进行模式匹配以检测漏洞. 但是, 专家定义的漏洞模式难以覆盖复杂的真实环境, 导致其对于真实漏洞的检测效果不佳. 不同于基于规则的方法, 本文方法使用神经网络模型自动化地挖掘并学习训练样本中的漏洞模式, 在节省人工开销的同时, 达到了更低的漏报率和误报率.

基于令牌的方法是将整个源代码片段视为一段自然语言文本, 并按照文本的分词方法将代码文本拆解为令牌序列后进行模型训练. 其中, TokenCNN 是该类方法中的代表性工作. 其首先通过词法分析将源代码转换为令牌序列, 然后将其嵌入为向量, 最后将向量表征输入卷积神经网络模型进行漏洞预测. 因为 TokenCNN 将源代码视为纯文本, 且直接将源代码按照文本处理的方式进行模型的训练和预测, 缺乏对源代码语义和结构信息的考虑从而损失了代码中的大量信息, 因此导致检测效果不佳.

基于语句的方法类似于基于令牌的方法, 其直接将每行代码进行嵌入代替了基于令牌方法先分词后嵌入的解决方案. 例如: StatementLSTM 将每行代码视为一个自然语言句子, 将其嵌入固定长度的向量后输入 LSTM 模型训练漏洞检测器. 其相较于基于令牌的方式避免了因分词带来的语义损失, 所以检测效果相对更好. 但是基于语句的方法同样是从文本角度处理代码, 仍未在根源上解决代码语法语义损失的问题.

基于函数级的方法是将整个函数当作处理对象进行模型训练. 例如: Devign 首先使用复杂程序分析提取一个完整函数的代码属性图, 接着使用通用的图神经网络检测漏洞. 通过完整函数的代码属性图表征, 使得训练样本能够包含关于代码的全面语义和句法信息. 然而, 函数的代码属性图一方面包含了大量漏洞无关的

节点和边干扰模型训练; 另一方面, 代码属性图还包含了抽象语法树、程序控制流图、程序依赖图等多种数据结构, 复杂的图结构使得模型难以学习到精确的漏洞特征, 导致模型效果欠佳.

基于切片的方法是指先将待测程序通过程序切片提取出程序关键语义, 进而对关键语义相关的语句或节点集合进行训练和测试. 其中, SySeVR 是最具代表性的切片级方法之一. 其对目标程序执行切片以生成代码切片, 然后将其嵌入为相应的向量表征. 与其他级别的方法不同, 切片级方法相当于对样本进行了预先提纯, 即删除样本中的噪声语句. 然而, SySeVR 将代码切片视为一段自然语言文本, 因此代码行之间的依赖关系不会在其切片中捕获. 相反, 本文方法生成的切片是一个程序依赖图子图, 其保留了漏洞相关的语句间完整的语义和代码结构, 因此达到了更佳的检测结果.

综上, 本文方法采用源代码切片(程序依赖图子图)的方式训练图神经网络漏洞检测器, 实验结果表明, 其针对真实的漏洞检测可以取得 75.1%的 *F*1 分数. 相较于其他漏洞检测工具, *F*1 分数提高了 41.2%–110.4%, 具有明显的优势.

4.2.2 RQ2: 本文方法对于漏洞检测结果的解释效果如何?

为回答该研究问题, 本文将所提方法的解释效果与两种先进的图神经网络通用解释器(GNNExplainer[13]和 PGExplainer[14])进行了比较. 表 3 展示了本文改进 GNNExplainer 方法与原 GNNExplainer 和 PGExplainer 的解释效果对比情况. 其中, 原 GE 表示 GNNExplainer, 原 PE 表示 PGExplainer, 本文方法则指本文改进的 GNNExplainer. 表 3 的第 1 行分别表示解释器输出待解释图中 2%–20%的节点作为解释结果的准确率.

表 3 与其他图神经网络解释器准确率对比(%)

方法	$ACC_{2\%}$	$ACC_{4\%}$	$ACC_{6\%}$	$ACC_{8\%}$	$ACC_{10\%}$	$ACC_{12\%}$	$ACC_{14\%}$	$ACC_{16\%}$	$ACC_{18\%}$	$ACC_{20\%}$
原 GE	25.4	40.0	50.3	58.8	67.6	73.8	77.5	81.6	85.1	87.1
原 PE	26.2	41.3	48.9	54.5	58.9	62.8	65.6	67.7	69.4	70.6
本文方法	29.7	47.4	57.5	66.0	73.6	78.4	82.9	85.6	88.3	91.1

由表 3 的实验结果可以看出: 本文改进后的 GNNExplainer 方法相比于原 GNNExplainer 和 PGExplainer 在实际漏洞解释中准确率分别提升了 3.8%–18.5%和 13.4%–29.0%, 其中, 前 10%节点的准确率高达 73.6%, 前 20%节点的准确率高达 91.1%. 其原因在于:

- 原 GNNExplainer 是以关键边的角度构建关键子图后作为解释结果输出, 然而其作为一个通用的图神经网络并非为漏洞样本设计, 因此在根据重要边构造出最小联通子图的过程中, 会对漏洞信息造成一定损失; 而本文方法直接将边掩码的重要程度分数映射给节点, 减少了中间步骤, 因此较原先的 GNNExplainer 解释效果更佳, 且更贴近研究人员分析漏洞成因时倾向于获得哪些代码行是漏洞代码的研究需求;
- 不同于 GNNExplainer 对于单个样本直接进行解释, PGExplainer 在对图神经网络模型进行解释之前需要大量样本进行解释器的参数训练, 以实现同时对多个实例从模型的全局视角进行解释, 且其解释的准确性依赖于解释器的训练效果; 而本文的实验数据均为真实漏洞数据, 数据具有高度复杂性, PGExplainer 难以应对大量复杂实例的解释器训练, 因此导致解释效果欠佳. 相较之下, GNNExplainer 对单个实例进行单独的训练和解释, 使得单个实例损失较小, 解释准确率更高.

由实验结果可知: 随着节点数目的增加, 解释语句覆盖漏洞行的准确率越高. 但同时, 解释结果输出的节点数量越大, 意味着研究人员需要分析更多的潜在漏洞行, 因此会带来较大的人工负担. 由表 3 的结果可知: 当解释输出节点数大于 10%时, 解释准确率的增长开始缓慢. 同时, 考虑到为避免带来过多的无关解释语句, 本文最终以前 10%节点作为最后解释器的节点输出数量.

本文对所提方法和对比解释器的时间开销进行了实验统计, 见表 4.

实验随机选取了 1 399 个待解释样本进行解释, 训练轮数默认均取 100 轮. 对比原 GNNExplainer, 本文方法减少了解释器生成子图和无关掩码的计算过程, 因此相比于原方法, 时间开销明显降低. 由于 PGExplainer 需要事先训练解释器, 训练完成的解释器再对单个实例依次进行解释, 因此其时间开销主要用于解释器训练.

而本文方法针对每个实例同时进行训练和解释, 因此总时间开销明显低于 PGExplainer, 解释效率更高.

表 4 与其他图神经网络解释器时间开销对比

解释方法	训练时间(s)	解释时间(s)	总时间(s)	平均时间(s)
原 GE	–	136 764.1	136 764.1	97.8
原 PE	92 401.2	468.7	92 869.9	66.4
本文方法	–	78 605.5	78 605.5	56.2

综上, 本文通过改进 GNNExplainer, 能够精确且快速地定位漏洞语句, 一方面有助于提高漏洞检测模型的可信性; 另一方面, 便于研究人员分析漏洞成因并对漏洞语句有针对性地加以修补. 相比于已有的图神经网络通用的解释器, 本文方法在准确度和时间开销方面均具有明显优势.

4.2.3 RQ3: 本文方法是否可以检测并解释真实软件漏洞?

为了回答该研究问题, 本文对常见的开源软件进行了漏洞检测和解释, 以检验本文方法在现实世界漏洞发掘方面的实用性. 具体来说, 本文实验将 4 个目标程序的开源 C/C++代码作为检测目标, 待测软件包含 liabv、Xen、OpenSSL 和 httpd.

表 5 给出了目标软件的细节描述, 其中共计目标函数 286 039 个, 并从中获取切片 3 312 657 个. 实验具体过程是: 首先, 将预处理后的切片输入训练好的漏洞检测模型进行漏洞检测; 接着, 漏洞切片被输入解释器以获得重要程度前 10%的关键节点对应的漏洞行. 由于本文方法作为静态检测手段无法直接给出漏洞在程序运行中的触发信息, 即面向 NVD 申请新的漏洞号(CVE-ID)时需要提交的漏洞触发的程序运行依据, 因此, 实验通过人工分析以验证本文方法对于真实软件的检测和解释结果是否正确. 具体地, 人工分析的过程采用两种验证方法: 首先, 通过与 national vulnerability database (NVD)[41]已有漏洞的漏洞模式对比, 从而确定本文方法是否检测出真实漏洞, 并核实漏洞解释结果的准确性; 其次, 通过人工分析漏洞成因的方式加以验证.

表 5 本文选择的开源软件细节

开源产品名称及版本	文件数量	函数数量	代码行数	切片数量
libav-0.8.21/9.21/11.12/12.3	4 983	37 682	2 043 116	489 866
openssl-1.0.0/1.1.1/3.0.0	3 195	24 512	1 187 467	311 302
xen-4.12.0/4.14.0/4.15.1	14 829	207 447	8 131 195	2 281 917
httpd-2.0.35/2.2.14/2.4.51	1 338	16 398	611 578	229 572
总计	24 345	286 039	11 973 356	3 312 657

本节对一个 Xen 中关于 USB 仿真程序的漏洞实例进行详细描述, 图 5 给出了一个切片 API#2 从切片生成到漏洞解释的完整过程. 考虑到实际的切片图占用篇幅较大, 这里使用节点对应的代码行展示切片内容.

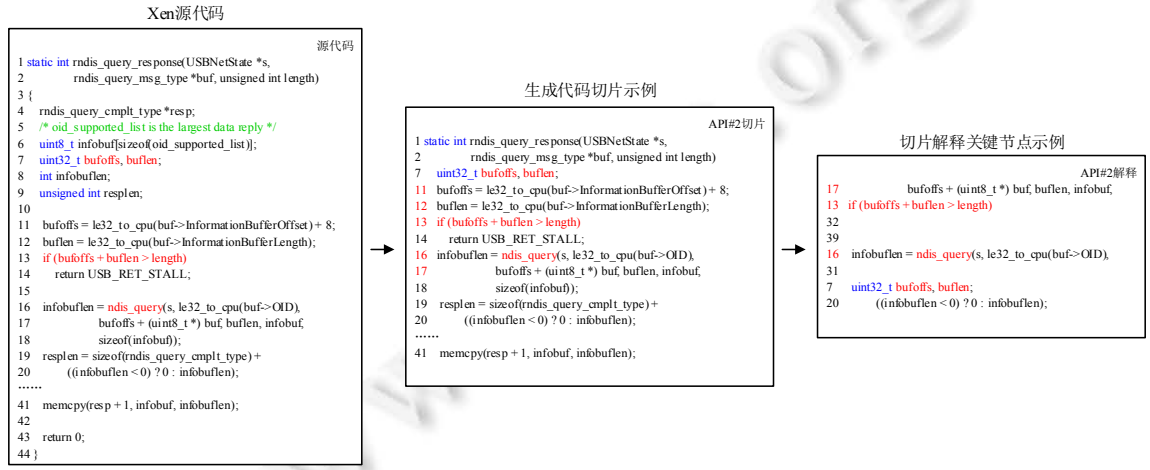

图 5 Xen 真实漏洞检测解释实例分析

根据分析, 该函数中存在一个整数溢出漏洞. 该函数允许本地来宾用户通过远程 NDIS 控制消息包造成拒绝服务攻击. 具体来说, 代码第 7 行声明了两个整数 *bufoffs* 和 *buflen*, *buffoffs* 和 *buflen* 在第 11 行和第 12 行接收用户输入数值后被赋值, 分支条件为 *bufoffs*+*buflen*>*length*(第 13 行). 当 *buffoffs* 和 *buflen* 的数值被攻击者故意构造为极大值时会导致整数溢出, 从而使得该分支条件不被满足, 即无法正常返回. 在函数进一步调用第 16–18 行函数时导致 QEMU 进程崩溃, 进而主机内存信息泄露. 因此, 与 *buffoffs* 和 *buflen* 存在数据和控制依赖关系的代码行, 包括第 7 行、第 11–13 行、第 16 行以及第 17 行, 被认为是漏洞相关行. 如图 5 所示, 仅 API#2 切片的解释结果就包含正确的漏洞行(第 17 行、第 13 行、第 16 行). 事实上, 原函数的 2–3 个切片的解释结果能够覆盖全部的漏洞引入位置和触发位置, 但由于篇幅限制, 该漏洞的全部切片和解释结果将公布在本文的代码仓库中[42].

根据与已知漏洞模式匹配的验证方式, 本文方法从待测样本中至少检测到了 54 个新漏洞, 对应于 NVD 中 36 种已知的漏洞模式. 表 6 的第 1 列和第 2 列显示了本文方法检测到的漏洞的详细信息, 包括所在软件名称及版本和漏洞所在文件路径, 第 3–5 列表示与该漏洞存在相似漏洞模式的已知 NVD 漏洞信息, 包含匹配漏洞模式的 CVE 号、报告漏洞的软件名称、漏洞发布时间和漏洞修补情况. 其中, 21 个漏洞已经在该软件的后续版本中被默默修补或者删除, 33 个漏洞仍未进行修补. 对于未进行修补的漏洞, 本文为了保护软件安全, 对其详细信息进行了模糊处理, 并已向其开发厂商提交了漏洞报告和漏洞补丁建议. 根据人工分析漏洞成因的验证方式, 本文方法从待测样本中至少检测到了 5 个新漏洞. 表 7 为新漏洞模糊处理后的漏洞信息, 我们已将该漏洞提交给 NVD 审核并通知厂商及时修复, 待 NVD 审核完成以及漏洞成功修复后, 再在本文的代码仓库中公布详细的漏洞信息.

表 6 通过与已知漏洞匹配模式验证的漏洞列表

目标软件名称及版本	漏洞文件所在路径	匹配的漏洞模式	报告漏洞软件	发布时间	最新版本是否修复
libav-0.8.21	libavcodec/dsputil.c	CVE-2013-7010	FFmpeg	20131208	修复
	libavcodec/**.c	CVE-2013-****	FFmpeg	20131208	未修复
	libavcodec/error_resilience.c	CVE-2011-3941	FFmpeg	20111001	修复
	libavcodec/**.c	CVE-2015-3395	FFmpeg	20150421	未修复
libav-9.21	libavcodec/**.c	CVE-2013-****	FFmpeg	20131208	未修复
	libavcodec/**.c	CVE-2015-****	FFmpeg	20150421	未修复
	libavcodec/aac_parser.c	CVE-2016-7393	FFmpeg	20160909	修复
	libavcodec/wmalosslessdec.c	CVE-2012-2795	FFmpeg	20120519	修复
	libavcodec/**.c	CVE-2013-****	FFmpeg	20130107	未修复
	libavcodec/dfa.c	CVE-2017-9992	FFmpeg	20170628	修复
	libavcodec/**.c	CVE-2013-****	FFmpeg	20130107	未修复
	libavformat/mov.c	CVE-2016-3062	FFmpeg	20160309	修复
libav-11.12	libavcodec/wmalosslessdec.c	CVE-2012-2795	FFmpeg	20120519	修复
	libavcodec/**.c	CVE-2013-****	FFmpeg	20130107	未修复
	libavformat/**.c	CVE-2017-****	FFmpeg	20170726	未修复
	libavformat/**.c	CVE-2017-****	FFmpeg	20170831	未修复
	libavcodec/h264.c	CVE-2015-3417	FFmpeg	20150424	不存在
	libavcodec/**.c	CVE-2013-****	FFmpeg	20131208	未修复
libav-12.3	libavformat/**.c	CVE-2017-****	FFmpeg	20170726	未修复
	libavcodec/wmalosslessdec.c	CVE-2012-2795	FFmpeg	20120519	修复
	libavcodec/**.c	CVE-2013-****	FFmpeg	20130107	未修复
	libavformat/**.c	CVE-2016-****	FFmpeg	20170201	未修复
	libavformat/**.c	CVE-2018-****	FFmpeg	20180723	未修复
openssl-1.0.0	crypto/asn1/d2i_pr.c	CVE-2015-3195	OpenSSL	20150410	修复
	crypto/asn1/a_d2i_fp.c	CVE-2016-2109	OpenSSL	20160129	修复
	crypto/bn/bn_exp.c	CVE-2016-0702	OpenSSL	20151216	修复
xen-4.12.0	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160302	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160223	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160112	未修复
	qemu-xen-traditional/**.c	CVE-2016-****	QEMU	20160306	未修复
	qemu-xen/hw/usb/**.c	CVE-2018-****	QEMU	20180911	未修复

表 6 通过与已知漏洞匹配模式验证的漏洞列表(续)

目标软件名称及版本	漏洞文件所在路径	匹配的漏洞模式	报告漏洞软件	发布时间	最新版本是否修复
xen-4.12.0	qemu-xen/hw/9pfs/9p.c	CVE-2018-19364	QEMU	20181119	不存在
	qemu-xen/hw/scsi/**.c	CVE-2018-****	QEMU	20181030	未修复
	qemu-xen/hw/scsi/**.c	CVE-2016-****	QEMU	20160523	未修复
	qemu-xen/hw/ppc/**.c	CVE-2018-****	QEMU	20181105	不存在
	qemu-xen/slirp/**.c	CVE-2019-****	QEMU	20190124	未修复
xen-4.14.0	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160302	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160223	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160112	未修复
	qemu-xen-traditional/**.c	CVE-2016-****	QEMU	20160306	未修复
	qemu-xen/hw/scsi/**_.c	CVE-2016-****	QEMU	20160523	未修复
	qemu-xen-traditional/hw/i8254.c	CVE-2015-3214	QEMU	20150410	未修复
xen-4.15.1	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160302	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160223	未修复
	qemu-xen-traditional/hw/**.c	CVE-2016-****	QEMU	20160112	未修复
	qemu-xen-traditional/**.c	CVE-2016-****	QEMU	20160306	未修复
	qemu-xen/hw/scsi/**.c	CVE-2016-****	QEMU	20160523	未修复
httpd-2.0.35	lib/xmltok_impl.c	CVE-2009-3720	Expat	20091016	修复
	lib/xmltok_impl.c, xmltok.c	CVE-2016-0718	Expat	20151216	修复
	server/scoreboard.c	CVE-2012-0031	Apache	20111207	修复
httpd-2.2.14	http/byterange_filter.c	CVE-2011-3192	Apache	20110819	修复
	filters/mod_include.c	CVE-2009-1195	Apache	20090331	修复
	arch/win32/mod_isapi.c	CVE-2010-0425	Apache	20100127	修复
	lib/xmltok_impl.c, xmltok.c	CVE-2016-0718	Expat	20151216	修复

表 7 通过漏洞成因分析验证的漏洞列表

目标软件名称及版本	漏洞文件所在路径	漏洞函数名
libav-0.8.21/9.21/11.12/12.3	libavcodec/**.c	ff_vorbis_**
xen-4.12.0/4.14.0/4.15.1	qemu-xen-traditional/hw/**.c	vmsvga_**
xen-4.14.0	qemu-xen/hw/9pfs/**.c	proxy_**
xen-4.15.1	xen/common/libfdt/**.c	fdt_**
openssl-3.0.0-beta2	crypto/conf/**.c	ossl_**

综上所述, 本文方法成功检测并解释了 4 个开源产品中的 59 个新漏洞. 其中, 与现有漏洞模式匹配的新漏洞共计 54 个, 人工分析漏洞成因确认新漏洞 5 个. 实验结果表明: 本文方法能够有效检测真实软件漏洞, 并提供可靠的检测结果解释.

5 总结与展望

针对已有漏洞检测模型对于真实漏洞的检测效果不佳且无法给出具体漏洞语句的问题, 本文提出了基于图神经网络的切片级漏洞检测解释方法. 该方法首先针对漏洞关注点提取代码切片, 并采用图神经网络和改进的图神经网络解释器对代码切片实现了一个高效、准确的漏洞检测和解释方法. 本文方法在真实漏洞数据集上的测试结果达到 75.1%的 $F1$ 分数, 相较于现有的漏洞检测模型提高了 41.2%–110.4%, 实验结果表明本文方法可以有效检测真实漏洞. 在对漏洞检测结果的解释方面, 本文改进的图神经网络解释器在解释准确率上比其他两个通用的图神经网络解释器分别提升了 8.9%和 24.9%, 时间开销分别缩短了 42.5%和 15.4%. 实验结果表明: 本文方法对于真实漏洞能够准确并高效地输出漏洞代码行, 即能够为漏洞检测模型提供可信的检测结果解释. 最后, 本文方法扫描 4 个开源软件的千万行代码, 成功检测并解释了 36 种与已知漏洞模式相匹配的真实漏洞, 证明了本文方法在实际应用方面的有效性.

但是, 目前的工作仍具有一定的局限性.

- 在漏洞检测方面, 并非全部漏洞都是由文中提到的 4 类漏洞关注点引入, 导致部分样本由于不存在漏洞关注点而无法成功提取切片, 进而无法进行漏洞检测. 因此, 未来我们会补充并完善漏洞关注点的种类, 或采用函数级和切片级漏洞检测相结合的方法, 以使更多的样本能够成功得到检测;

- 其次, 本文在漏洞解释部分改进了现有的通用图神经网络解释器 GNNExplainer 作为漏洞解释器, 但其对于漏洞样本的特征考虑仍然不够详尽. 未来我们会继续深入分析漏洞样本特征, 然后根据漏洞特征设计一种面向漏洞样本的扰动方法并构造解释器;
- 最后, 本文在对已有真实产品进行漏洞检测解释时, 通过人工分析将检测出的漏洞与已知漏洞模式一一比对以确认检测结果的准确性, 该过程由 4 位有相关经验的从业者耗时 1 个月完成. 对于本文方法检测到的未匹配到已知模式的其他漏洞, 则被认为是可能存在的零日漏洞. 对于这类漏洞, 可以采用定向模糊测试技术生成 PoC 以验证其是否真实存在, 并提交 PoC 和漏洞触发信息以申请漏洞号. 但是现阶段的模糊测试方法对于该数量级漏洞进行验证将消耗巨大的计算资源, 因此, 未来我们拟设计一种轻量级的定向模糊测试器, 以支持本文检测的零日漏洞的后续 PoC 生成等漏洞验证工作.

在未来工作中, 我们将继续提升漏洞检测模型的检测效果和解释效果, 更好地辅助专家进行漏洞挖掘和分析工作.

References:

[1] Li Z, Zou DQ, Wang ZL, Jin H. Survey on static software vulnerability detection for source code. Chinese Journal of Network and Information Security, 2019, 5(1): 1−14 (in Chinese with English abstract).

[2] Chen ZX, Zou DQ, Li Z, Jin H. Intelligent vulnerability detection system based on abstract syntax tree. Journal of Cyber Security, 2020, 5(4): 1−13 (in Chinese with English abstract).

[3] Wang J, Kuang HY, Li RL, Su YF. Software source code vulnerability detection based on CNN-GAP interpretability model. Journal of Electronics & Information Technology, 2022, 44(7): 2568−2575 (in Chinese with English abstract).

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

[5] Software assurance reference dataset. https://samate.nist.gov/SRD/index.php

[6] Checkmarx. 2022. https://www.checkmarx.com/

[7] Flawfinder. 2022. http://www.dwheeler.com/flawfinde/r

[8] Rough audit tool for security. 2022. https://code.google.com/archive/p/rough-auditing-tool-for-security/

[9] Russell R, Kim L, Hamilton L, Lazovich T, *et al*. Automated vulnerability detection in source code using deep representation learning. In: Proc. of the 17th IEEE Int'l Conf. on Machine Learning and Applications (ICMLA 2018). 2018. 757−762.

[10] Lin G, Xiao W, Zhang J, *et al*. Deep learning-based vulnerable function detection: A benchmark. In: Proc. of the 21st Int'l Conf. on Information and Communications Security (ICICS 2019). 2019. 219−232.

[11] Li Z, Zou D, Xu S, *et al*. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. on Dependable and Secure Computing, 2021, vol. abs/1807.06756.

[12] Zhou Y, Liu S, Siow J, *et al*. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Annual Conf. on Neural Information Processing Systems (NIPS 2019). 2019. 10197−10207.

[13] Ying Z, Bourgeois D, You J, *et al*. Gnnexplainer: Generating explanations for graph neural networks. In: Proc. of the 33rd Annual Conf. on Neural Information Processing Systems (NIPS 2019). 2019. 32.

[14] Luo D, Cheng W, Xu D, *et al*. Parameterized explainer for graph neural network. In: Proc. of the 34th Annual Conf. on Neural Information Processing Systems (NIPS 2020), Vol.33. 2020. 19620−19631.

[15] Lin G, Zhang J, Luo W, *et al*. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2017). 2017. 2539−2541.

[16] Feng H, Fu X, Sun H, *et al*. Efficient vulnerability detection based on abstract syntax tree and deep learning. In: Proc. of the IEEE INFOCOM 2020-IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS 2020). 2020. 722−727.

[17] Wang H, Ye G, Tang Z, *et al*. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. on Information Forensics and Security, 2020, 16: 1943−1958.

[18] Wu Y, Zou D, Dou S, *et al*. VulCNN: An image-inspired scalable vulnerability detection system. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE 2022). 2022. 1−12.

[19] Li Z, Zou D, Xu S, Ou X, *et al*. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proc. of the 25th Network and Distributed System Security Symp. (NDSS 2018). 2018. 1−15.

[20] Zou D, Wang S, Xu S, *et al*. μVulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. on Dependable and Secure Computing, 2021, 18(5): 2224−2236.

[21] Duan X, Wu J, Ji S, *et al*. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In: Proc. of the 28th Int'l Joint Conf. on Artificial Intelligence (IJCAI 2019). 2019. 4665−4671.

[22] Yamaguchi F, Golde N, Arp D, *et al*. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of the 35th IEEE Symp. on Security and Privacy (S&P 2014). 2014. 590−604.

[23] Cheng X, Wang H, Hua J, *et al*. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. on Software Engineering and Methodology, 2021, 30(3): 1−33.

[24] Chakraborty S, Krishna R, Ding Y, *et al*. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. on Software Engineering, 2021. [doi: 10.1109/TSE.2021.3087402]

[25] Yuan H, Tang J, Hu X, *et al*. XGNN: Towards model-level explanations of graph neural networks. In: Proc. of the 26th ACM SIGKDD Int'l Conf. on Knowledge Discovery & Data Mining (SIGKDD 2020). 2020. 430−438.

[26] Schnake T, Eberle O, Lederer J, *et al*. Higher-order explanations of graph neural networks via relevant walks. 2020. [doi: 10.48550/arXiv.2006.03589]

[27] Selvaraju RR, Cogswell M, Das A, *et al*. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In: Proc. of the IEEE Int'l Conf. on Computer Vision (ICCV 2017). 2017. 618−626.

[28] Yuan H, Yu H, Wang J, *et al*. On explainability of graph neural networks via subgraph explorations. In: Proc. of the 38th Int'l Conf. on Machine Learning (ICML 2021). 2021. 12241−12252.

[29] Li Y, Wang S, Nguyen TN. Vulnerability detection with fine-grained interpretations. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE 2021). 2021. 292−303.

[30] Wu B, Liang X, Zhang SS, Xu R. Advances and applications in graph neural network. Chinese Journal of Computers, 2022, 45(1): 35−68 (in Chinese with English abstract).

[31] Open-source code querying engine for C/C++. 2020. https://joern.io/

[32] Wolf L, Hanani Y, Bar K, *et al*. Joint word2vec networks for bilingual semantic representations. Int'l Journal of Computational Linguistics and Applications, 2014, 5(1): 27−42.

[33] Schlichtkrull M, Kipf TN, Bloem P, *et al*. Modeling relational data with graph convolutional networks. In: Proc. of the 15th European Semantic Web Conf. (ESWC 2018). 2018. 593−607.

[34] Veličković P, Cucurull G, Casanova A, *et al*. Graph attention networks. arXiv:1710.10903, 2017.

[35] Li Y, Tarlow D, Brockschmidt M, *et al*. Gated graph sequence neural networks. arXiv:1511.05493, 2015.

[36] Zhang M, Cui Z, Neumann M, *et al*. An end-to-end deep learning architecture for graph classification. In: Proc. of the 32nd AAAI Conf. on Artificial Intelligence (AAAI 2018). 2018. 4438−4445.

[37] Kingma DP, Ba J. Adam: A method for stochastic optimization. In: Proc. of the 3rd Int'l Conf. on Learning Representations (ICLR 2015). 2015. http://arxiv.org/abs/1412.6980

[38] Fan J, Li Y, Wang S, *et al*. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories (MSR 2020). 2020. 508−512.

[39] Common vulnerabilities & exposures. https://cve.mitre.org/

[40] Tensors and dynamic neural networks in python with strong GPU acceleration (PyTorch). 2022. https://pytorch.org

[41] National vulnerability database. https://nvd.nist.gov/

[42] Vuldetexp. https://github.com/Stwsyburg/Vuldetexp

附中文参考文献:

[1] 李珍, 邹德清, 王泽丽, 金海. 面向源代码的软件漏洞静态检测综述. 网络与信息安全学报, 2019, 5(1): 1−14.

[2] 陈肇炫, 邹德清, 李珍, 金海. 基于抽象语法树的智能化漏洞检测系统. 信息安全学报, 2020, 5(4): 1−13.

[3] 王剑, 匡洪宇, 李瑞林, 苏云飞. 基于 CNN-GAP 可解释性模型的软件源码漏洞检测方法. 电子与信息学报, 2022, 44(7): 2568−2575.

[4] 段旭, 吴敬征, 罗天悦, 杨牧天, 武延军. 基于代码属性图及注意力双向 LSTM 的漏洞挖掘方法. 软件学报, 2020, 31(11): 3404−3420. http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

[30] 吴博, 梁循, 张树森, 徐睿. 图神经网络前沿进展与应用. 计算机学报, 2022, 45(1): 35−68.

胡雨涛(1997－), 女, 博士生, CCF 学生会员, 主要研究领域为漏洞检测, 代码分析, 克隆检测.

王溯远(2000－), 男, 硕士生, 主要研究领域为机器学习, 漏洞检测, 软件工程.

吴月明(1993－), 男, 博士, CCF 学生会员, 主要研究领域为移动安全, 软件供应链安全, 人工智能安全, 恶意软件分析, 漏洞分析, 克隆代码审计.

邹德清(1975－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为云计算安全, 网络攻防与漏洞检测, 软件定义安全与主动防御, 大数据安全与人工智能安全, 容错计算.

李文科(2000－), 男, 硕士生, 主要研究领域为软件工程, 漏洞检测.

金海(1966－), 男, 博士, 教授, 博士生导师, CCF 会士, IEEE 会士, ACM 终身会员, 主要研究领域为计算机系统结构, 虚拟化技术, 集群计算, 网格计算, 并行与分布式计算, 对等计算普适计算, 语义网, 存储与安全.