软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3134−3150 [doi: 10.13328/j.cnki.jos.007341] [CSTR: 32375.14.jos.007341]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于函数间结构特征关联的软件漏洞检测方法*

邱少健[1,2], 程嘉濠[1], 黄梦阳[1], 黄 琼[1,2]

[1](华南农业大学 软件学院, 广东 广州 510642)

[2](广州市智慧农业重点实验室, 广东 广州 510642)

通信作者: 黄琼, E-mail: qhuang@scau.edu.cn

摘 要: 漏洞检测是软件系统安全领域的关键技术. 近年来, 深度学习凭借其代码特征提取的卓越能力, 在漏洞检测领域取得了显著进展. 然而, 当前基于深度学习的方法仅关注于代码实例自身的独立结构特征, 而忽视了不同漏洞代码间存在的结构特征相似关联, 限制了漏洞检测技术的性能. 针对这一问题, 提出了一种基于函数间结构特征关联的软件漏洞检测方法 (vulnerability detection method based on correlation of structural features between functions, CSFF-VD). 该方法首先将函数解析为代码属性图, 并通过门控图神经网络提取函数内的独立结构特征. 在此基础之上, 利用特征之间的相似性构建函数间的关联网络并构建基于图注意力网络进一步提取函数间关联信息, 以此提升漏洞检测的性能. 实验结果显示, CSFF-VD 在 3 个公开的漏洞检测数据集上超过了当前基于深度学习的漏洞检测方法. 此外, 在函数内各独立特征提取的基础上, 通过增加 CSFF-VD 中函数间关联特征提取方法的实验, 证明了集成函数间关联信息的有效性.

关键词: 漏洞检测; 代码结构特征; 函数关联; 图注意力网络

中图法分类号: TP311

中文引用格式: 邱少健, 程嘉濠, 黄梦阳, 黄琼. 基于函数间结构特征关联的软件漏洞检测方法. 软件学报, 2025, 36(7): 3134–3150. http://www.jos.org.cn/1000-9825/7341.htm

英文引用格式: Qiu SJ, Cheng JH, Huang MY, Huang Q. Software Vulnerability Detection Method Based on Correlation of Structural Features Between Functions. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3134–3150 (in Chinese). http://www.jos.org.cn/1000-9825/7341.htm

Software Vulnerability Detection Method Based on Correlation of Structural Features Between Functions

QIU Shao-Jian[1,2], CHENG Jia-Hao[1], HUANG Meng-Yang[1], HUANG Qiong[1,2]

[1](College of Software, South China Agricultural University, Guangzhou 510642, China)

[2](Guangzhou Key Laboratory of Intelligent Agriculture, Guangzhou 510642, China)

Abstract: Vulnerability detection is a critical technology in software system security. In recent years, deep learning has achieved remarkable progress in vulnerability detection due to its outstanding ability in code feature extraction. However, the existing deep learning-based methods only concentrate on the independent structural features of code instances, overlooking the structural feature similarities and correlations among different vulnerable codes, which limits the performance of vulnerability detection technology. To address this issue, this study proposes a vulnerability detection method based on correlation of structural features between functions (CSFF-VD). This method first parses functions into code property graphs and the independent structural features within functions are extracted by using gated graph neural networks. On this foundation, an association network among functions is constructed based on feature similarity, and a graph

* 基金项目: 国家自然科学基金 (62272174); 广东省基础与应用基础研究基金 (2022A1515110564); 广州市智慧农业重点实验室项目 (201902010081)
本文由“新兴软件与系统的可信赖性与安全”专题特约编辑向剑文教授、陈厅教授、杨珉教授、周俊伟教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-15; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-17

© 中国科学院软件研究所 http://www.jos.org.cn

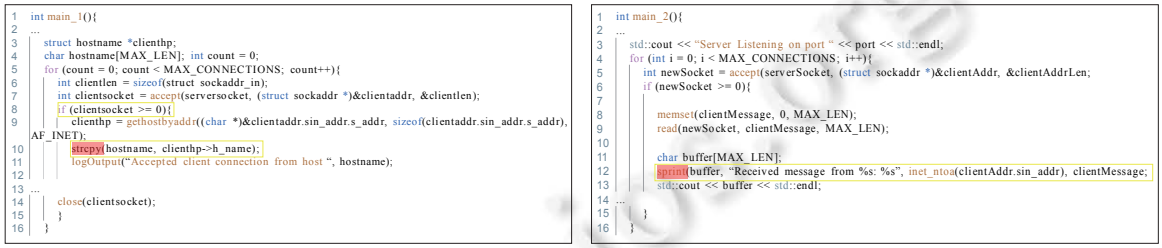
attention network is utilized to further extract the correlation information between functions, thus improving the performance of vulnerability detection. Experimental results show that CSFF-VD surpasses the current deep learning-based vulnerability detection methods on three public vulnerability detection datasets. In addition, based on the extraction of independent features within the function, this study proves the effectiveness of integrating the correlation information between functions by adding experiments on the inter-function correlation feature extraction method in CSFF-VD.

Key words: vulnerability detection; code structural feature; correlation of function; graph attention network

随着软件规模和复杂度的不断增加, 软件中出现漏洞的概率也在增大[1−3]. 软件漏洞是指软件程序或系统中的缺陷或弱点, 攻击者可以利用这些漏洞窃取数据或对系统和用户造成损害. 软件的漏洞通常源于软件架构设计不当、代码缺陷、执行错误和软件供应链问题等多种原因[4]. 通过分析软件源代码进行漏洞检测和预警, 可以帮助开发者提升软件的安全性和可靠性[5−7].

传统的软件漏洞检测方法主要依赖于专家制定的规则, 通过检测代码是否匹配这些规则以判断软件是否存在漏洞[8−10]. 然而, 传统基于规则匹配的方法容易产生漏洞的误报和漏报, 且预定义的规则难以及时适配新出现的漏洞类型. 近年来, 随着人工智能的兴起, 学者们开始利用深度学习技术从代码样本中提取漏洞特征, 并基于这些特征对软件源代码进行检测, 在一定程度上提升了漏洞检测技术的性能. 目前, 基于深度学习的漏洞检测方法可根据不同的代码表示输入, 分为基于代码文本表示的漏洞检测方法和基于代码图结构表示的漏洞检测方法[11]. 前者将代码视作文本序列, 使用深度学习模型从代码文本中提取语义特征, 并以此构建漏洞检测模型. 后者则将代码转换为抽象语法树、数据流图和控制流图等表示形式, 后采用图神经网络从多样的代码表示中提取代码结构特征, 并以此构建漏洞检测模型.

然而, 上述基于深度学习的漏洞检测方法主要关注于代码实例本身的特征, 忽略了不同代码实例之间潜在的结构相似性关联. 以两个函数检测为例, 如图 1(a) 所示, main_1 函数中存在 CWE-120 漏洞 (buffer copy without checking size of input), 即在第 8 行处未对用户传入的“clienthp->h_name”进行边界检查, 且在第 10 行服务器处理客户端请求时, 未使用安全的字符串拷贝, 导致“clienthp->h_name”的长度超过“hostname”的容量时会出现缓冲区溢出的漏洞, 该漏洞可能被攻击者用于执行任意代码. 在图 1(b) 中的 main_2 函数中, 第 12 行代码在未对用户输入信息进行输入验证就使用 sprintf 读取外部输入, 若客户端发送恶意的特制输入, 此处可被用于执行任意代码, 引发 CWE-134 漏洞 (use of externally-controlled format string). 上述两个函数的漏洞触发方式虽然不同但代码结构模式相似, 且其都未对用户输入进行充分的过滤和验证. 这种函数间结构近似的关联信息在现有研究中没有得到的充分利用, 漏洞检测模型的性能仍有提升空间.



(a) CWE-120 漏洞样例 (b) CWE-134 漏洞样例

图 1 函数间的结构特征关联

针对上述问题, 本文面向软件的函数漏洞检测粒度, 提出了一种基于函数间结构特征关联的漏洞检测方法 (vulnerability detection method based on correlation of structural features between functions, CSFF-VD), 在提取代码函数内独立结构特征的基础上, 构建并挖掘函数间的结构相似关联信息, 从而提升漏洞检测性能. 具体地, CSFF-VD 首先构建函数的代码属性图表示, 将其输入到残差门控图神经网络[12]中进行函数内独立结构特征提取. 然后, 在各函数以独立结构特征向量表示的基础上, 使用欧氏距离计算函数独立结构特征间的相似度以构建函数节点间的有向边, 从而建立函数间结构关联. 最后, 采用能够减少关联噪声的注意力图神经网络提取函数间关联特征, 并利

© 中国科学院软件研究所 http://www.jos.org.cn

用该关联特征构建漏洞检测模型. 本文在 FFmpeg、Qemu 和 Chrome+Debian 这 3 个广泛使用的软件漏洞数据集上评估了 CSFF-VD, 以验证其有效性. 实验结果表明, 与 5 种先进的漏洞检测方法相比, CSFF-VD 在 *F-measure*、*Precision*、*Recall* 和 *FNR* 这 4 个指标上, 均能获得更优异的结果.

本文的贡献总结如下. (1) 本文提出了一种基于函数间结构特征关联的软件漏洞检测方法 CSFF-VD. 在提取函数内独立结构特征的基础上增加了函数间结构关联的信息, 以此提升漏洞检测的性能. 本文在 3 个漏洞检测数据集上评估了提出方法和基线方案的性能, 结果表明, 本文所提出的 CSFF-VD 在指标性能上超过了其他先进方法的水平. (2) 本文提出的函数间关联挖掘方法, 能即插即用于各类函数内独立特征提取方法上, 用于提升漏洞检测模型的性能. 对此, 第 4 节中研究问题 2 的实验结果表明了集成函数间关联特征提取方法的有效性.

本文第 1 节介绍漏洞检测的研究现状. 第 2 节介绍相关技术背景. 第 3 节介绍本文构建的基于函数间结构特征关联的软件漏洞检测方法模型. 第 4 节阐述我们的实验设计. 第 5 节展示实验结果并进行分析. 第 6 节根据实验结果进行进一步讨论. 最后在第 7 节总结全文.

1 漏洞检测研究现状

软件漏洞检测的目标是通过对源代码进行分析, 判断软件代码是否存在漏洞, 指导开发人员进行修复从而保障软件的安全性和可靠性. 软件漏洞检测的传统方法主要依赖于专家制定的规则进行模式匹配, 通过检测代码的编码方式进而判断代码是否存在某一漏洞[13,14]. 但基于规则和模式匹配的方法容易产生误报和漏报, 且预定义的规则和专家知识难以快速响应新出现的漏洞类型. 此外, 传统的软件漏洞检测也普遍使用自动化测试工具, 针对特定软件代码模块进行渗透测试, 但该方法需要测试人员具有充足经验, 且所需人力较多, 耗时过长[15,16]. 随着人工智能的发展, 特别是深度学习的兴起, 近年来出现了许多使用深度神经网络模型提取源代码特征, 用于漏洞检测的研究. 现阶段, 按照代码表示方式划分, 基于深度学习的漏洞检测方法主要包含两类: 基于代码文本表示的漏洞检测和基于代码图结构表示的漏洞检测[11].

基于代码文本表示的漏洞检测将代码视作文本序列输入至深度网络中. Louati 等人[17]使用 Word2Vec 对软件源码进行嵌入, 并使用双向 LSTM 进行特征提取和检测. 文献 [18] 先对代码文本中库和 API 调用部分进行切片并拼合, 随后对片段进行清洗和嵌入, 最后使用双向 LSTM 进行特征提取. Lin 等人[19]使用多种词嵌入模型和多种主流神经网络 (如 TextCNN、LSTM 等) 进行搭配, 对漏洞代码进行检测并对比不同方式的优劣. 文献 [20] 先对代码中函数调用相关代码进行切片, 随后使用 BERT 模型对代码进行嵌入, 最后使用结合了自注意力的 LSTM 进行特征提取. 上述基于代码文本表示的漏洞检测方法主要对文本词义特征进行提取, 代码语义信息和依赖信息会有损失, 导致模型无法做出准确判断[11]. 基于代码图结构表示的漏洞检测方法近年来受到广泛关注, 其将代码转换为抽象语法树、控制流图、数据流图、代码属性图[21]等形式作为神经网络的输入, 从而进行特征提取和漏洞检测. 该方法相较于基于代码文本表示的方法可更完全地保留代码语义信息, 同时可突出展现如控制流、数据流、代码依赖等信息. 在基于图结构表示的漏洞检测研究中, Zhang 等人[22]使用数据流图对源代码表示后通过遍历数据流图多条路径获得该路径的节点序列, 使用预训练模型和卷积神经网络对路径节点序列提取特征, 最终结合多条路径特征进行漏洞检测. 文献 [23] 先提取源代码行级文本词义特征, 后用该词义特征对代码依赖图中各行对应的节点进行初始化, 最后使用带特征注意力的图卷积模型提取特征进行漏洞检测. Wen 等人[24]使用代码属性图对程序进行表示后, 根据自定义规则进行图结构的简化, 减少图内漏洞无关信息, 从而降低噪声带来的干扰, 最后使用不同尺寸的卷积核进行特征提取. Steenhoek 等人[25]为控制流图的节点赋予自定义向量, 表示每个节点的抽象数据流信息, 并使用门控图序列网络的消息聚合和更新函数模拟数据流分析中信息传播过程, 获得图的向量表示用于漏洞检测. Cheng 等人[26] 使用代码结构图对源代码进行表示, 并将其解构为执行路径. 随后使用启发式路径选择策略识别固定数量的关键路径以排除无关代码, 最后融合两种深度神经网络提取的增强特征表示, 用于检测漏洞.

上述基于深度学习的漏洞检测方法主要关注于代码实例 (如某个函数) 本身的特征, 忽略了不同代码实例之间可能存在的结构相似性关联. 实际上, 不同函数实例间可能存在结构相似性, 即不同代码会因相似的实现方式, 产生类似的漏洞. 基于此, 本文以代码函数作为漏洞检测的粒度, 提出了一种基于函数间结构特征关联的软件漏洞

© 中国科学院软件研究所 http://www.jos.org.cn

检测方法 CSFF-VD, 在代码函数内独立结构特征提取的基础上, 构建函数间的相似关联, 并使用具有降噪机制的图注意力网络挖掘函数间的所蕴含的关联信息[27], 从而提升漏洞检测模型的性能.

2 技术背景

2.1 代码属性图表示

在程序分析领域, 目前已有多种代码表示方法用于推理程序中的特征. 其中, 代码属性图 (code property graph, CPG) 作为一种新颖的源代码表示方法[21], 结合了抽象语法树 (abstract syntax tree, AST)、控制流图 (control flow graph, CFG) 和程序依赖图 (program dependence graph, PDG) 等经典程序表示, 形成一种联合的代码表示结构. CPG 的设计可有效辅助源代码中漏洞的发现, 其构建过程包括以下几个步骤.

(1) 代码 AST 表示解析

AST 通常是编译器生成的一种中间表示, 它为其他代码表示方法提供了基础架构. AST 精确编码了语句和表达式的嵌套结构, 以生成程序. AST 是一种有序树, 其中, 非叶子节点表示操作符 (例如加法或赋值), 叶子节点表示操作数 (例如常量或标识符). 尽管 AST 适用于简单的代码表示, 并被广泛用于代码语义相似度计算, 但由于其不能显式表示控制流和程序依赖关系, AST 并不适用于更复杂的代码分析, 例如检测死代码或未初始化的变量.

(2) 代码 CFG 表示解析

CFG 描述了代码语句的执行顺序及特定执行路径所需的条件. CFG 中的节点被称为基本块, 代表代码中在分支出现前的一段连续语句, 基本块之间通过有向边连接, 以表示控制的转移. CFG 广泛应用于安全领域, 例如检测已知恶意应用的变种和指导模糊测试工具. 此外, CFG 已成为逆向工程中的标准代码表示方法, 有助于程序理解. 然而, CFG 只揭示了程序的控制流, 未提供数据依赖相关的信息, 因此其无法有效挖掘因数据交互产生的漏洞.

(3) 代码 PDG 表示解析

PDG 显式表示了语句之间的依赖关系, 其中节点代表程序中的语句, 边则表示这些节点之间的依赖关系. PDG 中有两种类型的边: 数据依赖边和控制依赖边. 数据依赖边表示一个语句中定义的变量如何影响后续语句中使用的变量; 控制依赖边表示某个语句的执行依赖于前面的条件. PDG 的主要优势在于它能够同时捕捉程序的控制流和数据流信息, 这使得它在程序分析和优化中具有广泛的应用.

(4) 代码 CPG 表示生成

通过将 AST、CFG 和 PDG 整合为一个联合的数据结构, 形成代码属性图 CPG. 合并的关键思想是, AST、CFG 和 PDG 中的每个节点都对应源代码中的一个语句或谓词, 因此这些表示可以自然地连接在一起. 具体地, CPG 中的节点集 V 等于 AST 中的节点集合 V_{AST}, 边集合 E 等于 AST 中的边集合 E_{AST}、CFG 中的边集合 E_{CFG} 以及 PDG 中的边集合 E_{PDG} 的并集, 即 $E = E_{\text{AST}} \cup E_{\text{CFG}} \cup E_{\text{PDG}}$. 通过这种方式, 代码属性图能够综合利用抽象语法树、控制流图和程序依赖图的优势, 为高效分析和发现代码中的漏洞提供了支持.

2.2 图注意力网络

图注意力网络 (graph attention network, GAT) 是一种用于处理图数据的深度学习模型, 通过注意力机制自适应地捕捉节点之间的关系, 从而在图数据上实现高效的特征提取[28]. GAT 的核心思想是引入多头注意力机制, 以在节点邻居 $\mathcal{N}_u$ 之间分配不同的重要性权重. 具体来说, 对于图 $G(V,E)$, 注意力层接收一组节点特征 $H = \{h_1, h_2, \ldots, h_N\}$, 其中 $h_i \in \mathbb{R}^F$, F 表示特征维度, N 表示节点的个数. 输出特征为表示 $H^* = \{h_1^*, h_2^*, \ldots, h_N^*\}$, 计算过程如下.

GAT 首先计算图 G 中每一个节点 u 与其邻居节点 $v \in \mathcal{N}_u$ 之间的注意力分数 $e_{uv} = f(Wh_u, Wh_v)$, 随后通过 *Softmax* 函数对注意力分数进行归一化得到注意力系数 a_{uv}, 该结果反映了节点 v 对节点 u 的重要性:

$$a_{uv} = \frac{\exp(e_{uv})}{\sum_{k \in \mathcal{N}_u} \exp(e_{uk})} \tag{1}$$

根据注意力系数对邻居节点的特征进行加权求和, 获得更新后的节点 u 表示:

© 中国科学院软件研究所 http://www.jos.org.cn

$$h_u^* = ReLU\left(\sum_{v\in N_u} a_{uv} W h_v\right) \tag{2}$$

其中, W 为更新节点特征的权重矩阵, $ReLU$ 为非线性激活函数. 通过引入自适应的注意力机制, GAT 为图中节点的特征分配不同的权重, 从而增强了模型的表达能力和鲁棒性. GAT 在许多图数据处理和分析任务中表现优异, 成为图表示学习领域的重要模型之一.

3 基于函数间结构特征关联的软件漏洞检测方法 CSFF-VD

本节中介绍了基于函数间结构特征关联的漏洞检测方法 CSFF-VD 的技术框架, 该框架旨在利用函数内独立结构特征之间的结构相似性来增强漏洞检测的效果. 如图 2 所示, 该框架包括 3 个主要阶段: ① 将函数解析为代码属性图并生成函数独立结构特征; ② 构建函数间结构特征相似关联图, 并从中提取关联特征; ③ 应用分类技术进行漏洞检测.

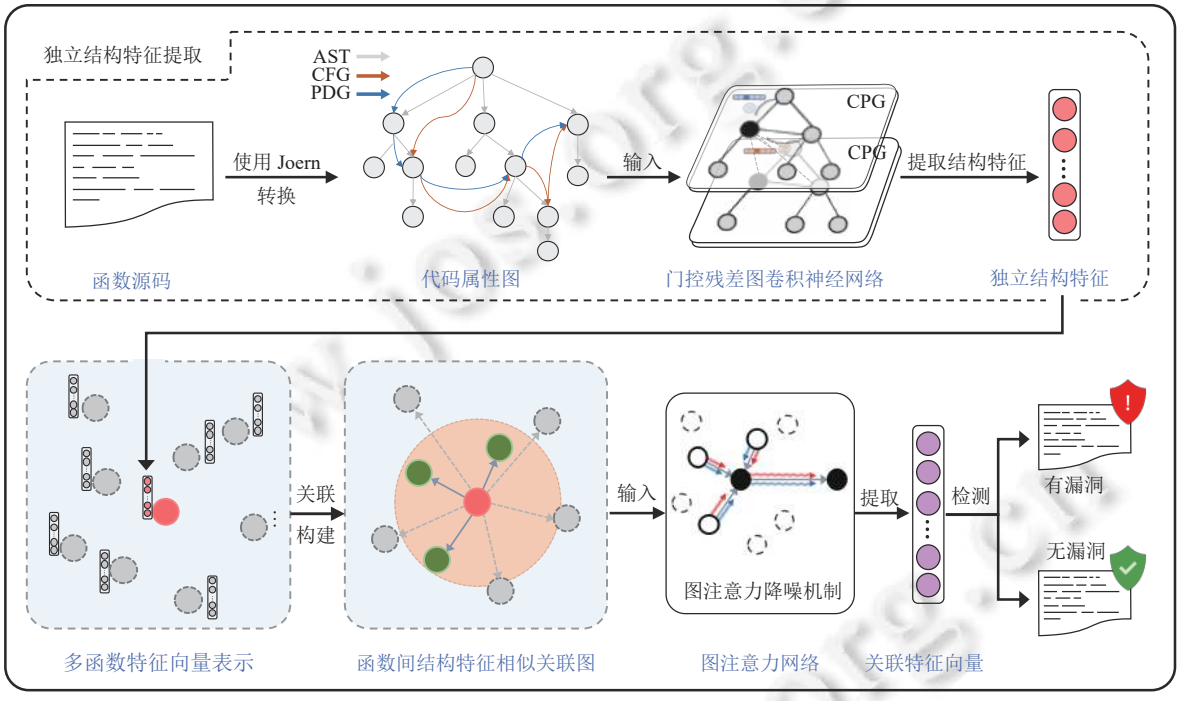


图 2 基于函数间结构特征关联的软件漏洞检测方法 CSFF-VD

首先, CSFF-VD 构建函数的代码属性图表示, 并将其输入到神经网络中进行函数内独立结构特征的提取. 通过这种方式, 可以有效地捕捉每个函数的独立结构特征, 为后续的特征关联挖掘奠定基础. 随后, 在各函数以独立结构特征向量表示的基础上, CSFF-VD 通过计算函数间独立结构特征的欧氏距离, 来构建函数节点间的有向边, 从而形成函数间的结构特征相似关联图, 作为注意力图神经网络的输入, 从而充分挖掘函数间的结构相似性信息, 以提高漏洞检测的准确性和全面性. 最后, CSFF-VD 使用图注意力网络对所生成的函数间结构特征相似关联图提取关联特征, 用于漏洞检测模型的构建.

3.1 函数独立结构特征生成

不同于自然语言, 代码作为一种特殊的文本, 包含丰富的结构信息. 在 CSFF-VD 框架的第 1 步, 我们对每个函数进行解析, 以生成其独立的结构特征. 具体而言, 我们利用静态代码分析工具 Joern[29]对每个函数进行解析, 并将其转换为代码属性图 CPG. 代码属性图将抽象语法树 AST、控制流图 CFG 和程序依赖图 PDG 整合为一个联合的代码图表示. 其中, AST 提供语法元素之间的层次关系, CFG 提供控制流信息, PDG 则表明数据之间和语句之间

© 中国科学院软件研究所 http://www.jos.org.cn

的相互依赖关系. 工具 Joern 解析这 3 种图结构并将其整合为一种表示方式, 记作 $\mathcal{G}_{\text{CPG}}(\mathcal{V}_{\text{CPG}},\mathcal{E}_{\text{CPG}})$. 在该图中, 节点集合 $\mathcal{V}_{\text{CPG}}$ 表示代码语句或语法元素, 边集 $\mathcal{E}_{\text{CPG}}$ 则描述各种类型的信息流. 为了从 $\mathcal{G}_{\text{CPG}}$ 生成函数结构特征, 我们实现了一个基于残差门控图神经网络的特征生成模块. 该步骤选择残差门控图神经网络的原因是其计算复杂度比传统图注意力网络更低, 且比图卷积神经网络有更强的表示能力[12,28]. 该模块首先使用 Word2Vec 将每个节点嵌入到一个 d 维特征向量 $\alpha^i\in\mathbb{R}^d$, $i\leqslant n$ 中, 所有节点共同形成特征矩阵 $\mathcal{M}=[\alpha^1,\alpha^2,\ldots,\alpha^n]^{\mathrm{T}}\in\mathbb{R}^{n\times d}$, 其中, n 表示节点个数. 为了增强 $\mathcal{G}_{\text{CPG}}$ 中节点信息的上下文理解, 本文采用基于门控循环单元 (gated recurrent unit, GRU) 的聚合函数 $f_{\text{agg-GRU}}$, 捕捉代码结构上下文. 该函数表示如下:

$$\mathcal{M}^{\ell+1}=f_{\text{agg-GRU}}\left(\mathcal{A}\mathcal{M}^{\ell},\mathcal{M}^{\ell}\right) \tag{3}$$

其中, $\mathcal{A}$ 表示 $\mathcal{G}_{\text{CPG}}$ 的邻接矩阵, $\mathcal{M}^{\ell}$ 表示第 ℓ 层的节点特征矩阵. 更新后的节点表示 $\mathcal{M}^{\ell+1}$ 通过 $f_{\text{agg-GRU}}$ 聚合邻近节点的信息 $\mathcal{A}\mathcal{M}^{\ell}$ 以及输入特征 $\mathcal{M}^{\ell}$ 得到. 随后, 使用平均池化层整合各层所有节点的特征, 并将其连接形成函数结构特征表示 $x\in\mathbb{R}^d$.

在漏洞数据集中, 我们为每个函数生成独立结构特征, 构成函数独立结构特征 $\mathcal{X}=\{x^1,x^2,\ldots,x^N\}$. 函数独立结构特征生成模块使用数据集的子集进行训练. 需要注意的是, 整个 CSFF-VD 框架在所有阶段使用数据集的同一个子集进行训练, 从而避免信息泄露.

3.2 函数间关联特征提取

在本节中, CSFF-VD 利用了函数间独立结构特征的相似性, 通过提取关联特征来更有效地识别潜在的安全漏洞. 具体来说, 如图 3 所示, 该过程主要分为两个步骤: 函数间结构特征相似关联图构建和关联特征提取.

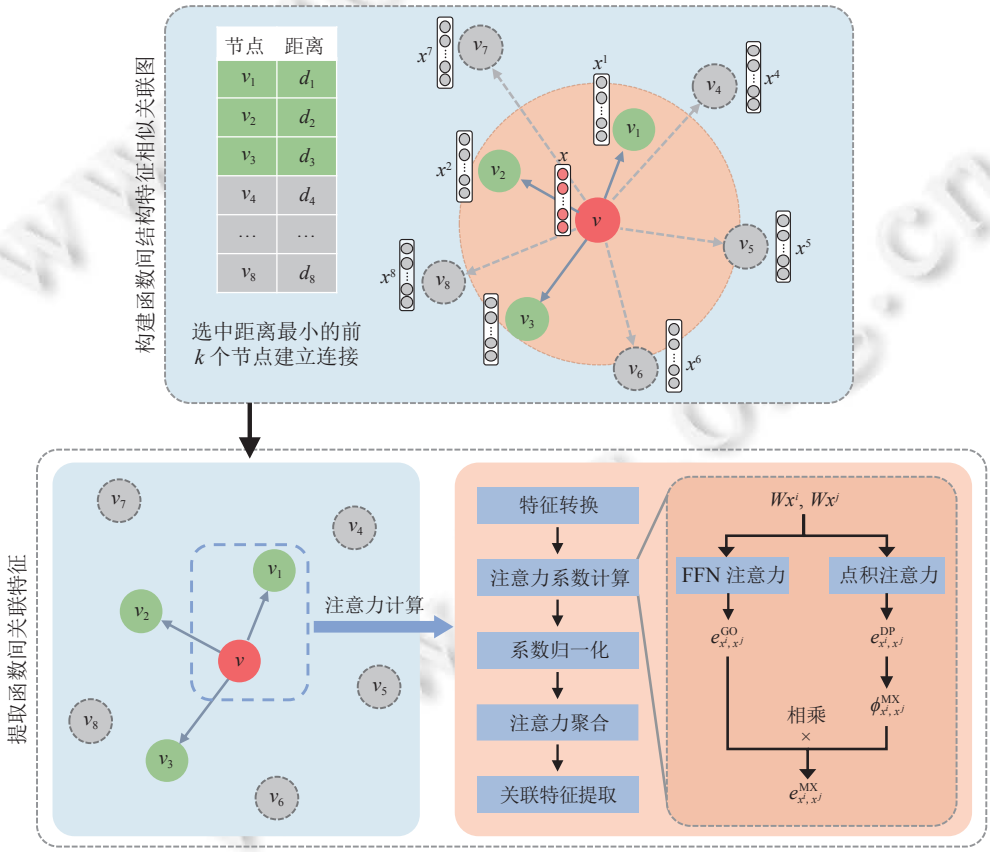


图 3 函数间结构特征相似关联构建和函数间关联特征提取方法

© 中国科学院软件研究所 http://www.jos.org.cn

(1) 函数间结构特征相似关联图构建

在第 3.1 节中, CSFF-VD 为每个函数生成了独立结构特征 $x \in \mathbb{R}^d$, 所有函数共同构成特征集合 $\mathcal{X} = \{x^1, x^2, \ldots, x^N\}$. 根据独立结构特征 x 之间的相似性距离构建函数之间链接, 从而识别具有相似特征的漏洞以及在不同函数之间发现差异. 如图 3 所示, 我们将每个函数视作相似关联图中的节点, 迭代时将每个节点视作质心, 并计算其与其他节点之间的欧氏距离 $d(x^i, x^j) = \| x^i - x^j \|_2$ 来确定每个节点特征最相似的 K 个邻居节点. 然后将每个节点与其 K 个邻居节点使用单向边链接起来, 构成函数间的相似关联图 $G = (V, E)$, 其中 V 表示节点集合, E 表示边集合. 本部分在训练时, 需读取数据集中训练集样本的函数结构特征表示作为相似关联图中的其他节点. 在对训练集样本进行检测时需要再次将待测样本读入, 此时由于待测样本已在前一步骤被读入, 训练时需将 K 设置为 $K+1$ (待测样本被重复读入时最近邻为其本身). 本部分在验证和测试时, 步骤与训练时相似, 需先读取训练集样本作为相似关联图的其他节点, 但由于验证集与测试集不存在被重复读入的情况, K 保持不变.

(2) 函数间关联特征提取

在依赖函数间的相似性来构建关联图的同时, 也引入了不必要的噪音关联. 因此, 准确计算函数之间关联重要性是挖掘函数间信息的关键. 我们构建了一个基于图注意力网络的提取模块来挖掘函数间的关联特征, 我们结合传统图注意力网络的原始 (GAT's original, GO) 注意力分数计算方式和点乘 (dot product, DP) 注意力分数计算方式来提升模型的表现. 在计算函数 i 和函数 j 之间的关联重要性时, 首先应用线性变换矩阵 W 对节点 x^i 和 x^j 的特征进行转换, Wx^i 和 Wx^j 分别计算 GO 注意力分数 e_{ij}^{GO} 和 DP 注意力分数 e_{ij}^{DP}.

$$e_{ij}^{\mathrm{GO}} = a^{\mathrm{T}} \left[Wx^i \,\|\, Wx^j \right] \tag{4}$$

$$e_{ij}^{\mathrm{DP}} = (Wx^i)^{\mathrm{T}} \cdot \left(Wx^j \right) \tag{5}$$

其中, a 是可学习的权重向量, 符号 $\|$ 表示向量拼接操作. 随后, 我们将这两个注意力分数进行融合, 得到融合注意力系数: $e_{ij}^{\mathrm{MX}} = e_{ij}^{\mathrm{GO}} \cdot \sigma\left(e_{ij}^{\mathrm{DP}}\right)$, σ 为 *Sigmoid* 函数, 定义为 $\sigma(x) = \dfrac{1}{1+\exp(-x)}$, 最后通过 *Softmax* 函数对注意力系数进行处理, 得到函数 i 和函数 j 之间的注意力系数归一化表示:

$$\alpha_{ij}^{\mathrm{MX}} = \frac{\exp\left(e_{ij}^{\mathrm{MX}}\right)}{\sum_{k \in \mathcal{N}(i)} \exp\left(e_{ik}^{\mathrm{MX}}\right)} \tag{6}$$

我们使用计算得到的注意力系数对函数节点进行更新, 节点的更新过程可以表示为:

$$x^{i^*} = \rho\left(\sum\nolimits_{j \in \mathcal{N}(i)} \alpha_{ij}^{\mathrm{MX}} W x^j \right) \tag{7}$$

其中, $\rho(\cdot)$ 是非线性激活函数 *ReLU*, $j \in \mathcal{N}(i)$ 表示节点 i 的邻居节点的集合. 为了进一步提炼节点表示, CSFF-VD 还通过关联图自监督损失进一步对模型进行优化. 具体来说, 我们计算函数 i 和函数 j 之间的边缘存在概率 $\phi_{ij,\mathrm{MX,DP}} = \sigma\left(e_{ij}^{\mathrm{DP}}\right)$, 通过使用边缘预测的二分类交叉熵损失函数 L_E 来减少无关边缘带来的噪音影响:

$$L_E = -\frac{1}{|E \cup E^-|} \sum\nolimits_{(j,i) \in E \cup E^-} \left(\mathbb{I}_{(j,i)=1} \log \phi_{ij} + \mathbb{I}_{(j,i)=0} \log(1 - \phi_{ij}) \right) \tag{8}$$

其中, E 是真实存在的边集合, E^- 表示通过负采样策略生成的不存在的负样本边集合, $|E \cup E^-|$ 表示正负边的总数集合, $\mathbb{I}_{(j,i)=1 \text{ or } 0}$ 为指示函数, 当 (j,i) 为真实边或负样本边存在时取值为 1, 否则取值为 0.

通过以上步骤, CSFF-VD 构建了一个函数间关联特征提取网络, 能够有效地挖掘函数间的关联信息, 提升模型在检测相似漏洞和发现函数差异方面的性能. 这种方法不仅利用了函数自身的结构信息, 还充分利用了函数之间的相似性和关联性, 为漏洞检测提供了更加全面和精确的特征表示.

3.3 漏洞检测

CSFF-VD 框架的主要目标是确定某个函数是否存在漏洞, 最终输出是函数存在漏洞的概率, 构成标准的二元分类任务. 在模型训练阶段, 我们使用交叉熵损失函数 L_V 测量模型预测概率与实际标签之间的差异指导 CSFF-VD 的优化. 具体来说, 交叉熵损失函数 L_V 定义为:

© 中国科学院软件研究所 http://www.jos.org.cn

$$L_V = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)) \tag{9}$$

其中, N 是训练样本的数量, y_i 是第 i 个样本的漏洞标签, $\hat{y}_i$ 是模型预测的概率. 目标是最小化这种交叉熵损失. 至此, 总的训练损失包括漏洞分类损失 L_V, 自监督图注意力损失 L_E 和 L2 正则化项, 表示为:

$$L = L_V + \lambda_E L_E + \lambda_2 \| W\|_2 \tag{10}$$

其中, λ_E 和 λ_2 是权重系数, 分别控制自监督图注意力损失 L_E 和 L2 正则化项对总损失的贡献. 通过结合这些损失项, CSFF-VD 框架不仅能够准确地检测出代码中的漏洞, 还能通过图注意力机制更好地捕捉函数之间的关联信息, 并利用正则化项防止模型过拟合. 这样的方法确保了模型在训练过程中能够平衡各个损失项的影响, 从而在实际应用中表现出更高的鲁棒性和准确性.

最终, CSFF-VD 提供了一种漏洞检测框架, 用于提高软件系统的安全性. 如算法 1 所示, 我们提供了 CSFF-VD 的伪代码表示, 以便帮助读者更深入地理解框架的技术细节. 在步骤 1 中, 我们解析函数以生成函数内独立结构特征. 步骤 2 中, CSFF-VD 利用函数间独立结构特征之间的相似性构建关联图, 并利用图注意力网络挖掘关联信息. 最后一步, 输出函数漏洞存在概率进行检测判别.

算法 1. 基于函数间结构特征相似关联的漏洞检测 CSFF-VD.

Input: 代码函数集合 $S = \{s^1, s^2, \ldots, s^N\}$;

Output: 漏洞检测结果 D.

Step 1. 代码函数独立结构特征生成

1. for 每一个函数 s in S do
2. 将函数 s 解析为抽象语法树 $\mathcal{G}_{\text{AST}}$, 控制流图 $\mathcal{G}_{\text{CFG}}$, 程序依赖图 $\mathcal{G}_{\text{PDG}}$.
3. 整合 $\mathcal{G}_{\text{AST}}$、$\mathcal{G}_{\text{CFG}}$、$\mathcal{G}_{\text{PDG}}$ 到一个联合的代码属性图表示 $\mathcal{G}_{\text{CPG}}$.
4. end for
5. 训练基于残差门控图神经网络的函数独立结构特征生成模块.
6. 将 $\mathcal{G}_{\text{CPG}}$ 作为输入生成函数独立结构特征集合 $\mathcal{X} = \{x^1, x^2, \ldots, x^N\}$.

Step 2. 代码函数间结构特征相似关联图构建

7. 初始化函数间相似关联图 $G = (V, E)$.
8. for 每一个函数特征 x in X do
9. 计算 x 与其他函数特征间的欧氏距离 $d(x^i, x^j) = \| x^i - x^j\|_2$.
10. 将 x 与距离最近的 top-k 邻居特征链接起来, 构建相似边 e.
11. 将 e 加入边集 E, 利用 x 初始化节点 v 并加入节点集 V.
12. end for
13. 构建基于注意力机制的关联特征提取网络 $\text{Model}_{\text{GAT}}$.
14. for 每一个函数节点 v in $G = (V, E)$ do
15. 计算其与相邻节点的注意力分数 e_{ij}^{MX} 注意力系数 α_{ij}^{MX}.
16. 更新节点表示.
17. end for
18. 通过漏洞分类损失和自监督图注意力损失训练模型, 提取关联特征.

Step 3. 函数漏洞检测

19. 对关联特征进行分类表示, 输出漏洞概率 D.

© 中国科学院软件研究所 http://www.jos.org.cn

4 实验设计

4.1 研究问题

为了评估基于函数间结构特征关联的漏洞检测方法 CSFF-VD 的有效性, 我们探讨了以下两个关键研究问题 (research question, RQ).

● RQ1: CSFF-VD 在漏洞检测任务中的有效性如何? 本研究问题旨在通过与当前最先进的方法进行比较, 全面评估 CSFF-VD 在漏洞检测方面的性能. 具体而言, 我们将在 3 个不同的漏洞数据集上进行实验, 通过比较 *Precision*、*Recall* 、*F-measure* 和 *FNR* 这 4 个关键性能指标, 以量化 CSFF-VD 的有效性.

● RQ2: CSFF-VD 中函数间的结构特征相似关联如何影响漏洞检测的性能? 本研究问题旨在深入分析 CSFF-VD 中函数间的相似关联信息对提升漏洞检测效率的具体贡献. 为解答此问题, 我们采用了消融实验方法, 以评估函数间关联信息对漏洞检测表现的影响, 从而揭示其在提升检测准确性和效率方面的作用.

4.2 数据集

为了验证 CSFF-VD 模型的性能, 本研究选取了 3 个广泛认可的漏洞数据集: FFmpeg、Qemu 及 Chrome+Debian, 作为实验数据集. 表 1 详细列出了这 3 个数据集的样本规模及其对应的漏洞率. 具体而言, FFmpeg 与 Qemu 数据集均由 C 语言编写的开源软件组成, 属于平衡漏洞数据集, 可用于准确评估模型性能. 其中, 多媒体处理框架 FFmpeg 的漏洞率为 51.1%, 而开源虚拟机及仿真器 Qemu 的漏洞率为 42.6%. 另一方面, Chrome+Debian 数据集源自 Linux Debian 内核与 Chromium 项目, 其漏洞率较低, 仅为 9.8%, 属于不平衡数据集, 可用于考察模型应对类别不平衡问题的能力. 这些数据集的样本数和漏洞率各不相同, 以便全面评估 CSFF-VD 模型在多样化软件环境下的适应性与稳健性.

表 1 实验数据集

数据集	描述	样本数量	有漏洞样本数量	无漏洞样本数量	漏洞率 (%)
FFmpeg	开源多媒体框架, 用于处理视频、音频、字幕等	9768	4981	4788	51.1
Qemu	开源虚拟化软件, 用于硬件虚拟化和仿真	17549	7479	10070	42.6
Chrome+Debian	包含 Chrome 浏览器和 Debian 操作系统的代码库	22734	2240	20494	9.8

4.3 对比方法

在实验评估中, 我们将 CSFF-VD 与 5 种最先进的漏洞检测方法进行了比较, 对比方法详细描述如下.

(1) Devign. Devign 是一种通过整合抽象语法树 (AST)、控制流图 (CFG)、数据流图 (DFG) 和代码序列表示, 并利用图门控网络进行漏洞检测的方法, 它通过融合多种代码表示方式来捕捉代码结构和语义信息, 从而提高漏洞检测的准确性[30].

(2) CodeBERT. CodeBERT 是一种基于编程语言和自然语言的双峰预训练模型, 它利用 BERT 技术进行代码表示的提取, 通过同时学习编程语言和自然语言注释特征, 能够更好地理解代码上下文, 从而提升漏洞检测效果[31].

(3) Reveal. Reveal 是一种通过控制流程图 (CPG) 和图门控神经网络 (GGNN) 进行嵌入, 并使用合成少数过采样技术 (SMOTE) 算法解决样本不平衡问题的检测方法, 它提升了模型的泛化性[11].

(4) VulCNN. VulCNN 是一种通过将程序依赖图 (PDG) 转换为图像表示, 并利用卷积神经网络 (CNN) 提取代码特征进行漏洞检测的方法, 它通过图像处理技术来分析代码结构, 利用 CNN 的强大特征提取能力来识别漏洞[32].

(5) VulBG. VulBG 是一种通过程序切片提取行为信息, 构建代码行为图以提高漏洞检测性能的方法, 它利用代码行为的底层关联, 构建更全面的代码表示, 从而增强检测效果[33].

(6) CSFF-VD. CSFF-VD 使用残差图门控图神经网络对代码属性图 (CPG) 进行嵌入, 后为待测样本与嵌入空

© 中国科学院软件研究所 http://www.jos.org.cn

间内其他样本构建函数间结构特征相似关联图, 利用样本间关联性来结合待测样本本身信息与其他样本的信息一同用于检测. 其构建结构特征关联图的机制可有效提升漏洞检测性能和模型泛化能力.

实验的对比涵盖了多类方法: 基于文本的 CodeBERT、基于图结构的 Devign 和 Reveal、基于代码可视化的 VulCNN 和基于代码行为的 VulBG. 通过与这些方法进行比较, 我们可以综合评估 CSFF-VD 的函数特征提取机制对提升漏洞检测性能的有效性.

4.4 参数设置和实验环境

为确保实验的公正性与可重复性, 本研究在所有对比方法中统一采用相同的数据分割策略. 具体而言, 数据集按照 8:1:1 的比例被随机划分为训练集、验证集和测试集. 本研究所提出的 CSFF-VD 模型采用 Adam 优化器进行参数优化, 其中, *Gamma* 参数设定为 0.9, 权重衰减设置为 5×10^{-4}, 训练周期 (epoch) 设定为 200, 初始学习率设定为 0.01, 特征提取的维度设定为 256, 邻居节点数量设定为 5. 实验在一台高性能计算机上进行, 该计算机配置有英特尔 i7-12700 CPU, 64 GB 内存, 以及 10 GB 显存的 NVIDIA 3080 GPU, 运行环境为 Ubuntu 20.04 操作系统.

4.5 评价指标

为了评估检测性能, 本文采用了漏洞检测研究中广泛使用的指标 *Precision*、*Recall* 、*F-measure* 和 *FNR*. 在漏洞检测任务中, 可能出现以下 4 种结果: ① 真正例 (true positive, *TP*): 将实际存在漏洞的实例正确检测为有漏洞. ② 假正例 (false positive, *FP*): 将实际无漏洞的实例错误检测为有漏洞. ③ 假负例 (false negative, *FN*): 将实际存在漏洞的实例错误检测为无漏洞. ④ 真负例 (true negative, *TN*): 将实际无漏洞的实例正确检测为无漏洞. 我们可以通过这 4 种结果计算 3 种评价指标, 计算方法如下:

$$FNR = \frac{FN}{TP + FN} \tag{11}$$

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

$$Recall = \frac{TP}{TP + FN} \tag{13}$$

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{14}$$

其中, *FNR* 衡量 *FN* 实例在所有实际有漏洞实例中的比例, 可以反映模型对有漏洞样本的遗漏情况. 越高的 *FNR* 表示模型对漏洞的遗漏情况更多, 模型的表现越差. *Precision* 衡量 *TP* 实例在所有分类为有漏洞的实例中所占的比例, 可以反映模型对有漏洞样本分类的准确性. *Recall* 衡量 *TP* 实例在所有实际有漏洞实例中的比例, 可以反映模型对无漏洞样本的误判情况. 越高的 *Recall* 表示模型对无漏洞样本的误判情况越少. *F-measure* 是 *Precision* 和 *Recall* 的调和平均值, 可以对 *Precision* 和 *Recall* 这两个指标进行有效的平衡. *FNR* 指标的最佳值为 0, 最差值为 100. *Precision*、*Recall* 和 *F-measure* 这 3 个评价指标的最佳值为 100, 最差值为 0. 通过这些指标, 我们可以全面评估 CSFF-VD 在检测漏洞方面的性能, 从而为模型的有效性和实用性提供量化支持.

5 实验结果与分析

5.1 RQ1: CSFF-VD 在漏洞检测任务中的有效性如何?

为回答这一研究问题, 我们将 CSFF-VD 与 5 种对比方法在 3 个数据集上进行了性能比较, 验证了 CSFF-VD 在漏洞检测中的有效性. 对比结果如表 2 所示, 对比方法中的第 1 行表示该方法的对应评价指标, 第 2 行开头的“+”或“–”分别表示 CSFF-VD 与其对比后上升或下降的幅度, 最佳性能用粗体标出. 总的来说, CSFF-VD 在 3 个数据集上的 *F-measure* 值分别为 67.8%、59.8% 和 48.8%. 这些结果相较于其他方法分别提高了 8.3%–30.6%、7.0%–21.3% 和 33.6%–92.1%. *F-measure* 值的显著提升表明, CSFF-VD 在 *Precision* 和 *Recall* 之间实现了良好的平衡, 是一种稳健的漏洞检测方法.

从其他指标观察, 在 *FNR* 方面, CSFF-VD 在 3 个数据集上的值分别为 11.7%、35.0% 和 45.9%. 除了在 Chrome+

© 中国科学院软件研究所 http://www.jos.org.cn

Debian 数据集上略高于 VulBG 之外, 相较于其他方法降低了 33.5%–76.0%、14.8%–26.6% 和 6.3%–36.8%, 其低 *FNR* 体现了其减少漏报的能力. 在 *Precision* 方面, CSFF-VD 在 3 个数据集上的值分别为 55.0%、55.5% 和 44.4%. 这些值相较于其他方法提高了 0.2%–8.7%、0.5%–22.8% 和 68.2%–94.7%, 其高 *Precision* 突显了其减少误报的能力. 在 *Recall* 方面, CSFF-VD 在 3 个数据集上的值分别为 88.3%、65.0% 和 54.1%. 除了在 Chrome+Debian 数据集上略低于 VulBG 之外, 相较于其他方法提高了 7.2%–72.4%、10.4%–24.3% 和 6.1%–97.4%. 尽管如此, CSFF-VD 的 *Recall* 和 *FNR* 依然具有竞争力, 能够有效地检测出漏洞, 确保检测过程中较少的漏报.

表 2　CSFF-VD 与其他漏洞检测方法的性能比较 (%)

方法	FFmpeg				Qemu				Chrome+Debian			
	F-measure	*Precision*	*Recall*	*FNR*	*F-measure*	*Precision*	*Recall*	*FNR*	*F-measure*	*Precision*	*Recall*	*FNR*
Devign	51.9 +15.9	52.2 +2.8	52.1 +36.2	47.9 −36.2	53.7 +6.1	53.1 +2.4	53.4 +11.6	46.6 −11.6	28.4 +20.4	24.1 +20.3	28.7 +25.4	71.3 −25.4
CodeBERT	53.0 +14.8	54.9 +0.1	51.2 +37.1	48.8 −37.1	54.1 +5.7	55.2 +0.3	52.3 +12.7	47.7 −12.7	25.4 +23.4	23.7 +20.7	27.4 +26.7	72.6 −26.7
Reveal	62.6 +5.2	50.6 +4.4	82.4 +5.9	17.6 −5.9	49.3 +10.5	45.2 +10.3	54.0 +11.0	46.0 −11.0	26.3 +22.5	24.4 +20.0	28.6 +26.1	71.4 −26.1
VulCNN	54.2 +13.6	51.2 +3.8	57.7 +30.6	42.3 −30.6	55.1 +4.7	52.3 +3.2	58.2 +6.8	41.8 −6.8	31.5 +17.3	22.8 +21.6	51.0 +3.1	49.0 −3.1
VulBG	57.5 +10.3	52.8 +2.2	62.1 +26.2	37.9 −26.2	55.9 +3.9	53.2 +2.3	58.9 +6.1	41.1 −6.1	36.5 +12.3	26.4 +18.8	**59.3** −5.2	**40.7** +5.2
CSFF-VD	**67.8**	**55.0**	**88.3**	**11.7**	**59.8**	**55.5**	**65.0**	**35.0**	**48.8**	**44.4**	54.1	45.9
W/T/L	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	5/0/0	4/0/1	4/0/1

表 2 的最后一行总结了 Win/Tie/Loss (W/T/L) 结果, 即 CSFF-VD 在每个数据集的各个指标上与每种基准方法的比较. 例如, FFmpeg 数据集上 *F-measure* 值的 W/T/L 结果 5/0/0 表明, CSFF-VD 在所有 5 种基准方法中表现最佳, 没有平局或失利. 总体而言, CSFF-VD 在大多数漏洞检测任务中表现优越, 多个指标上取得了 5/0/0 的 W/T/L 结果, 仅在两个指标上略低 (4/0/1). 这些在不同指标和数据集上的一致胜利进一步证明了 CSFF-VD 的稳健性和有效性. 综上所述, CSFF-VD 在漏洞检测任务中不仅在 *F-measure* 值上表现出色, 而且有效降低了模型的 *FNR*, 同时在 *Precision* 和 *Recall* 上也均有显著提升, 显示出其在减少误报和漏报方面的强大能力. 这些结果充分证明了 CSFF-VD 作为一种漏洞检测方法的优越性.

5.2　RQ2: CSFF-VD 中函数间的结构特征相似关联如何影响漏洞检测的性能?

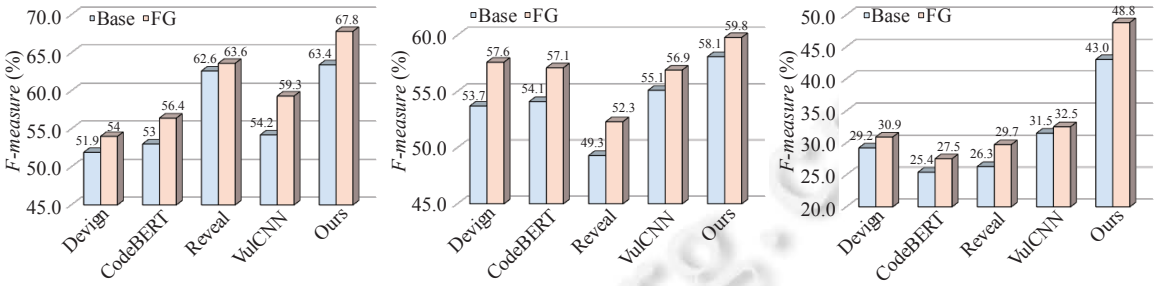
为了探究这一研究问题, 我们研究了函数间特征相似性对漏洞检测方法性能的贡献. 本文比较了两组漏洞检测方法. 具体来说, 我们选择了 4 种现有基于函数独立结构特征的漏洞检测方法: Devign、CodeBERT、Reveal 和 VulCNN 以及 CSFF-VD 的独立特征提取模块, 总共 5 种函数独立结构特征提取方法作为第 1 比较组, 表示为“Base”. 然后, 我们通过构建函数间相似关联来提取关联特征对这些方法进行扩展, 形成第 2 比较组, 表示为“+FG” (例如 Devign+FG).

图 4 展示了 3 组柱状图, 直观地展示了独立结构特征方法“Base”及其“+FG”扩展在 3 个数据集上的 *F-measure* 值的比较结果. 如图 4 所示, 针对现有方法的局限性, 本文提出构建函数间的相似关联挖掘关联特征以进一步增强漏洞检测. 结果表明, 我们的方法显著改进了 5 种独立结构特征提取方法, 在 FFmpeg 上提高了 1.6%–9.4%, 在 Qemu 上提高了 2.9%–7.3%, 在 Chrome+Debian 上提高了 3.2%–13.5%.

此外, 为了进一步验证“+FG”的性能提升是源自函数特征间的相似性关联构建, 而不是随机信息量的增加, 我们设计了一组对比实验. 具体而言, Intra 方法代表仅依赖独立结构特征提取模块进行检测的基准方法; RandomK 方法表示通过随机选取 K 个样本构建图关联的方法 (为了避免随机选取 K 个样本带来的效果差异, 我们进行了 50 次的随机选择并取平均值作为比较); CSFF-VD 则为本文提出的创新方法, 通过结构特征相似性构建关联图以挖掘更丰富的漏洞信息. 表 3 列出了上述 3 种方法的 *F-measure* 值. 可以观察到, RandomK 方法的 *F-measure* 值

© 中国科学院软件研究所 http://www.jos.org.cn

在 3 个数据集中分别为 61.2%、48.9% 和 32.3%, 比 Intra 方法分别下降 3.4%、15.8% 和 24.9%, 基于该对比结果, RandomK 方法引入的随机关联样本, 可能会带来噪声信息, 反而降低了代码特征提取的有效性; 而 CSFF-VD 相较于 Intra 分别提升了 6.9%、2.9% 和 13.5%, 与 RandomK 方法效果截然相反, 说明 CSFF-VD 带来的性能提升是源于结构特征相似关联信息的增益而不是随机信息量的增加. 总的来说, 整合函数间相似关联样本信息可以提升模型的检测效果.



(a) FFmpeg 数据集上的 RQ2 实验结果 (b) Qemu 数据集上的 RQ2 实验结果 (c) Chrome+Debian 数据集上的 RQ2 实验结果

图 4 基于函数间特征相似性提取关联特征对漏洞检测性能的提升

表 3 3 种对比方法在漏洞检测任务中的 *F-measure* 值 (%)

对比方法	FFmpeg	Qemu	Chrome+Debian
Intra	63.4	58.1	43.0
RandomK	61.2	48.9	32.3
CSFF-VD	67.8	59.8	48.8

6 拓展讨论

6.1 CSFF-VD 对最具威胁性漏洞的检测效果

为了评估 CSFF-VD 在现实世界漏洞检测中的有效性, 我们在国际上公开的最具威胁性漏洞榜单 (common weakness enumeration top, CWE Top) 中进行了实验. 这些漏洞可被攻击者利用从而进行未经授权的系统访问、窃取敏感数据甚至破坏系统. 我们从 BigVul 数据集[34]中选择属于 CWE Top 的函数, 并按 CWE ID 对它们进行分组, 构建测试数据集. 如表 4 所示, 我们收集了 15 种不同的 CWE 类型相关的函数. 该表总结了每种 CWE 类型漏洞的描述以及我们收集到的函数数量.

表 4 CWE Top 类型漏洞描述检测结果

CWE排名	ID	官方漏洞描述	VulBG	CSFF-VD	Vuln
1	CWE-787	Out-of-bounds Write	80	**91**	136
2	CWE-79	Improper Neutralization of Input During Web Page Generation	**1**	0	1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command	1	**1**	5
4	CWE-416	Use After Free	109	**117**	190
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command	**5**	4	11
6	CWE-20	Improper Input Validation	291	**341**	658
7	CWE-125	Out-of-bounds Read	231	**259**	535
9	CWE-352	Cross-site Request Forgery	0	**1**	1
11	CWE-862	Missing Authorization	1	**2**	3
12	CWE-476	NULL Pointer Dereference	113	**132**	207
15	CWE-502	Deserialization of Untrusted Data	0	**1**	1
16	CWE-77	Improper Neutralization of Special Elements used in a Command	**4**	3	10

© 中国科学院软件研究所 http://www.jos.org.cn

表 4 CWE Top 类型漏洞描述检测结果 (续)

CWE排名	ID	官方漏洞描述	VulBG	CSFF-VD	Vuln
19	CWE-918	Server-side Request Forgery	1	**1**	2
23	CWE-94	Improper Control of Generation of Code	1	**1**	3
24	CWE-863	Incorrect Authorization	**2**	1	4
—	—	合计	840	**956**	1 767

我们使用 Chrome+Debian 数据集作为训练数据集, 因为其缺陷率与 BigVul 类似. 为了保证实验的公平性, 测试数据集去除了与 Chrome+Debian 中重叠的数据. 然后, 我们使用本次训练中具有最高 *F-measure* 值的模型来评估其准确性. 由于在之前的实验结果中, CSFF-VD 的 *Recall* 性能略微低于 VulBG, 因此我们也对 VulBG 进行评估作为额外一组比较方法, 最佳结果用粗体标出. 实验结果表明, 在属于 CWE Top 的 1 767 个漏洞中, CSFF-VD 检测到了 956 个, 检测率达到 54.1%, 与 VulBG 相比提高了 13.9%. 检测 top-1 和 top-5 CWE 漏洞的准确率分别为 66.9% 和 62.4%, 也比 VulBG 显示出显著的优势. 表中的最佳结果以粗体显示, 表明 CSFF-VD 在检测大多数现实项目漏洞方面表现更好. CSFF-VD 针对 CWE Top 漏洞的综合评估显示了其处理现实安全威胁的卓越能力.

6.2 CSFF-VD 在最新发布漏洞上的检测效果

为了评估 CSFF-VD 在最新发布漏洞中的检测效果, 我们针对 FFmpeg Security[35]发布的 2023–2024 年新增漏洞案例进行实验. 这些案例来自 FFmpeg 开源项目, 类型涵盖堆缓冲区溢出、越界读取等实际场景中常见的漏洞类型, 详细信息如表 5 前 4 列所示.

表 5 2023–2024 年 FFmpeg 项目新增漏洞及各方法检测情况

漏洞编号	漏洞类型	补丁Commit号前8位	文件名	是否检出			
				CodeBERT	VulCNN	CSFF-Base	CSFF-VD
CVE-2024-28661	堆缓冲区溢出	66b50445	speexdec.c	—	√	√	√
CVE-2024-7055	越界访问	d0ce2529	pnmdec.c	√	√	√	√
CVE-2024-7272	越界访问	a937b3c5	swresample.c	—	—	√	√
CVE-2023-47342	越界访问	e4d5ac8d	rtsp.c	—	—	—	—
CVE-2023-47343	非法内存读取	0f6a3405	mov.c	—	—	—	√
CVE-2023-47344	越界访问	f7ac3512	jpegxl_parser.c	√	√	√	√

本节中, 我们使用 RQ1 中所构建的 CodeBERT 方法 (基于令牌) 与 VulCNN 方法 (基于程序依赖图和卷积神经网络)、RQ2 中所构建的 Intra 方法 (基于代码属性图的独立结构特征) 以及本文提出的 CSFF-VD 方法, 对表 5 中 FFmpeg 的新漏洞案例进行检测. 检测结果如表 5 的后 4 列所示, CodeBERT 仅检出 2 个越界访问漏洞; VulCNN 检出 2 个越界访问和 1 个堆缓冲区溢出漏洞, Intra 检出 3 个越界访问、1 个堆缓冲区溢出漏洞; 本文提出的 CSFF-VD 可检出 5 个漏洞, 其在前 3 个方法检出漏洞的基础上, 额外发现了 1 个因整数溢出导致的非法内存读取漏洞. 上述结果表明, CSFF-VD 在 FFmpeg 最新漏洞案例上的检测能力优于其他方法. 我们认为, CSFF-VD 方法通过函数间结构特征关联的构建, 增强了代码特征提取的有效性, 进而提升了模型的漏洞检测能力.

6.3 模型训练时间和推理成本分析

在本节中, 我们讨论 CSFF-VD 的训练时间、推理时间以及模型参数量. 表 6 列出了 CSFF-VD 与其他 5 种漏洞检测方法的比较结果, 为了保证实验的公平性, 所有比较实验均在同一台高性能计算机上进行. 结果显示, CSFF-VD 的训练时间约为 20 min, 推理时间不到 5 s, 相较于 CodeBERT 模型, 训练时间和推理时间约是原来的 1/40, 与 Devign、Reveal、VulCNN 和 VulBG 方法相比, 训练时间和推理成本也得到显著提升. 此外, CSFF-VD 的模型参数量较少, 因此它适合部署在计算资源有限的平台, 例如无法使用大参数量模型的计算机和边端设备等.

© 中国科学院软件研究所 http://www.jos.org.cn

表 6 模型在不同数据上的训练时间和推理时间

数据集	训练时间			推理时间			参数量 (MB)
	FFmpeg	Qemu	Chrome+Debian	FFmpeg	Qemu	Chrome+Debian	
Devign	2 h 8 min	2 h 39 min	3 h 11 min	4.1 s	4.3 s	5.5 s	4.38
CodeBERT	20 h 15 min	26 h 37 min	34 h 5 min	5 min	5 min	6 min	475
Reveal	7 h 28 min	8 h 12 min	9 h 20 min	2.3 s	2.4 s	3.1 s	2.14
VulCNN	50 min	56 min	1 h 18 min	5.1 s	5.4 s	8.2 s	2.6
VulBG	56 min	1 h 20 min	1 h 57 min	7.3 s	8.7 s	11 s	24.1
CSFF-VD	18 min	21 min	29 min	3.7 s	3.9 s	4.6 s	11.7

6.4 不同的相似性度量方法对实验的影响

本节讨论 CSFF-VD 中, 用于构建函数间结构特征关联图的相似性度量方法选择对 CSFF-VD 性能的影响, 我们使用不同的度量方法作为变量, 比较马氏距离、余弦相似度、曼哈顿距离和欧氏距离在 FFmpeg、Qemu 和 Chrome+Debian 这 3 个数据集下的性能. 结果如表 7 所示, 在 FFmpeg 和 Qemu 这两个平衡数据集上, 使用欧氏距离时 *FNR* 和 *Recall* 指标为最优, *F-measure* 和 *Precision* 仅比最好的马氏距离低一些. 在 Chrome+Debian 不平衡数据集上, 使用欧氏距离在各项指标上皆能达到最优. 因此, 在多个数据集下综合评估所测得的性能后, CSFF-VD 选择了欧氏距离作为构建函数间结构特征关联图的相似性度量方法.

表 7 模型在不同相似性度量方法下性能比较 (%)

度量方法	FFmpeg				Qemu				Chrome+Debian			
	F-measure	*Precision*	*Recall*	*FNR*	*F-measure*	*Precision*	*Recall*	*FNR*	*F-measure*	*Precision*	*Recall*	*FNR*
马氏距离	68.1	56.6	85.4	14.6	60.3	56.4	64.9	35.1	21.5	45.6	14.1	85.9
余弦相似度	51.5	58.7	45.8	54.2	58.1	59.8	56.6	43.4	35.5	37.1	40.0	60.0
曼哈顿距离	66.1	56.4	79.6	20.4	56.6	61.4	52.6	47.4	40.4	38.8	42.3	57.7
欧氏距离	67.8	55.0	88.3	11.7	59.8	55.5	65.0	35.0	48.8	44.4	54.1	45.9

6.5 用于构造函数间结构特征关联的邻居节点数 *K* 对实验的影响

本节讨论 CSFF-VD 中用于构造函数间关联结构的邻居节点数量 *K* 的参数设置, 以达到最佳结果. 由于我们关注 *K* 参数对模型性能的影响, 因此在本实验中仅使用 *K* 作为变量, 其他网络参数保持不变. 我们使用 *F-measure* 值在 3 个数据集上评估 CSFF-VD 的性能: FFmpeg、Qemu 和 Chrome+Debian. 后文图 5 显示了 CSFF-VD 在 3 个数据集上具有不同 *K* 值的性能. 当 *K* 设置为 5 时, FFmpeg 和 Qemu 数据集上的性能达到峰值. 此外, 对于 Chrome+Debian, 当 *K* 设置为 4 时, *F-measure* 值最佳. 实验结果表明, CSFF-VD 在构造函数间结构特征关联图时, 将邻居节点数量 *K* 设置 5 通常会取得最佳结果. 这一发现体现了选择适当邻居节点数量 *K* 提取关联特征并增强漏洞检测性能的重要性.

6.6 有效性威胁

在本节中, 我们讨论对方法有效性的几个潜在威胁.

(1) 数据预处理时间. 本文采用 CPG 对样本进行表示. 由于 CPG 在生成时需解析程序的 AST 和 PDG 等信息, 其生成过程存在耗时较长的问题. 在实际检测推理的场景中, 针对单个待测样本生成 CPG 的平均耗时约为 3 s.

(2) 方法实现过程中可能存在的差别. 本文使用在线提供的源代码在实验中实现了一些比较方法 (例如 Devign 和 Reveal). 对于不提供源代码的对比方法, 我们通过严格遵循原文中提供的方法细节来实现. 尽管如此, 我们的实现可能与原始比较方法存在细微差别.

(3) 数据集的局限性. 同时本文所使用的 FFmpeg 和 Qemu 数据集都来自 Devign 数据集, Croft 等人[36]指出该数据集中存在标签不准确、重复样本等问题. 相较于其他常见的数据集, Devign 数据集受影响较小, 实验结果仍具备参考价值.

© 中国科学院软件研究所 http://www.jos.org.cn

(4) 漏洞检测评估指标的选择. 本文选择软件漏洞检测研究中常用的 *F-measure*、*Precision*、*Recall* 和 *FNR* 作为评估指标, 这些指标已基本满足了常见的模型性能对比需要. 但选择其他不同的指标可能会产生不同的结果.

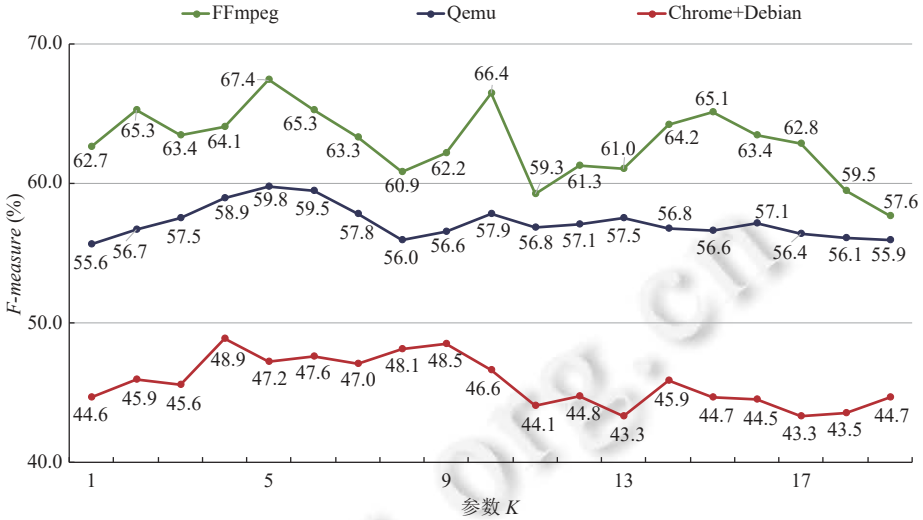


图 5 不同 *K* 值下, CSFF-VD 在 3 个数据集上的 *F-measure* 值表现

7 总 结

本文分析了现有基于深度学习的软件漏洞检测方法及其忽略了不同代码实例之间可能存在结构相似性关联的问题, 并提出了一种基于函数间结构特征关联的软件漏洞检测方法 CSFF-VD. 方法通过构建函数的代码属性图, 并将其输入残差门控图神经网络中提取独立的结构特征. 在此基础上, 计算函数间的相似度, 形成函数间的有向边, 从而建立函数间的结构关联. 最终, 通过注意力图神经网络减少关联噪声, 提取关联特征并构建漏洞检测模型. 为验证 CSFF-VD 的有效性, 本文在 3 个广泛使用的软件漏洞数据集上进行了评估. 实验结果表明, 与 5 种最先进的漏洞检测方法相比, CSFF-VD 在 *F-measure*、*Precision* 、*Recall* 和 *FNR* 这 4 个指标上均表现出更优异的性能. 此外, 通过将函数间关联特征提取的方法应用于函数独立结构特征的方法中, 漏洞检测的性能在实验数据集上均得到了提升, 表明了集成函数间关联挖掘方法的有效性. 本研究的数据和源代码可在 GitHub 上获取, 网址为 https://github.com/insoft-lab/CSFF-VD.

未来的工作将着重于改进函数独立结构特征的表示方式和相似度计算方法, 以更好地表达函数语义结构, 以及提高构建函数间关联的效果. 我们也计划在实际的企业级软件项目中测试该方法的有效性, 以推动 CSFF-VD 在工业界的应用.

References:

[1] Amoroso E. Recent progress in software security. IEEE Software, 2018, 35(2): 11–13. [doi: 10.1109/MS.2018.1661316]

[2] Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Poshyvanyk D. Fusion: A tool for facilitating and augmenting Android bug reporting. In: Proc. of the 38th Int'l Conf. on Software Engineering Companion. Austin: ACM, 2016. 609–612. [doi: 10.1145/2889160.2889177]

[3] Zhang B, Gao Y, Wu JY, Wang N, Wang Q, Ren JD. Approach to predict software vulnerability based on multiple-level *N*-gram feature extraction and heterogeneous ensemble learning. Int'l Journal of Software Engineering and Knowledge Engineering, 2022, 32(10): 1559–1582. [doi: 10.1142/S0218194022500620]

[4] Zhao QC, Huang C, Dai LH. VULDEFF: Vulnerability detection method based on function fingerprints and code differences. Knowledge-based Systems, 2023, 260: 110139. [doi: 10.1016/j.knosys.2022.110139]

[5] Zaharia S, Rebedea T, Trausan-Matu S. Machine learning-based security pattern recognition techniques for code developers. Applied Sciences, 2022, 12(23): 12463. [doi: 10.3390/app122312463]

© 中国科学院软件研究所 http://www.jos.org.cn

[6] Iannone E, Guadagni R, Ferrucci F, De Lucia A, Palomba F. The secret life of software vulnerabilities: A large-scale empirical study. IEEE Trans. on Software Engineering, 2023, 49(1): 44–63. [doi: 10.1109/TSE.2022.3140868]

[7] Le THM, Chen H, Babar MA. Deep learning for source code modeling and generation: Models, applications, and challenges. ACM Computing Surveys (CSUR), 2020, 53(3): 62. [doi: 10.1145/3383458]

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

[9] Shao SH, Gao Q, Ma S, Duan FY, Ma X, Zhang SK, Hu JH. Progress in research on buffer overflow vulnerability analysis technologies. Ruan Jian Xue Bao/Journal of Software, 2018, 29(5): 1179–1198 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5504.htm [doi: 10.13328/j.cnki.jos.005504]

[10] Díaz G, Bermejo JR. Static analysis of source code security: Assessment of tools against SAMATE tests. Information and Software Technology, 2013, 55(8): 1462–1476. [doi: 10.1016/j.infsof.2013.02.005]

[11] Chakraborty S, Krishna R, Ding YRB, Ray B. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. on Software Engineering, 2022, 48(9): 3280–3296. [doi: 10.1109/TSE.2021.3087402]

[12] Bresson X, Laurent T. Residual gated graph ConvNets. arXiv:1711.07553, 2018.

[13] Johnson B, Song Y, Murphy-Hill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? In: Proc. of the 35th Int'l Conf. on Software Engineering. San Francisco: IEEE, 2013. 672–681. [doi: 10.1109/ICSE.2013.6606613]

[14] Qasem A, Shirani P, Debbabi M, Wang LY, Lebel B, Agba BL. Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. ACM Computing Surveys, 2022, 54(2): 25. [doi: 10.1145/3432893]

[15] Tyagi Y, Shekhar S, P A, Bhardwaj S. SEEMA: An automation framework for vulnerability assessement and penetration testing. In: Proc. of the 2nd Int'l Conf. on Vision Towards Emerging Trends in Communication and Networking Technologies. Vellore: IEEE, 2023. 1–5. [doi: 10.1109/ViTECoN58111.2023.10157032]

[16] Chen K, Lu H, Fang BX, Sun YB, Su S, Tian ZH. Survey on automated penetration testing technology research. Ruan Jian Xue Bao/Journal of Software, 2024, 35(5): 2268–2288 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7038.htm [doi: 10.13328/j.cnki.jos.007038]

[17] Louati A, Gasiba T. Source code vulnerability detection using deep learning algorithms for industrial applications. In: Proc. of the 2nd Int'l Conf. on Ubiquitous Security. Zhangjiajie: Springer, 2022. 161–178. [doi: 10.1007/978-981-99-0272-9_11]

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

[19] Lin GJ, Xiao W, Zhang J, Xiang Y. Deep learning-based vulnerable function detection: A benchmark. In: Proc. of the 21st Int'l Conf. on Information and Communications Security. Beijing: Springer, 2020. 219–232. [doi: 10.1007/978-3-030-41579-2_13]

[20] Li N, Zhang HY, Hu ZH, Kou G, Dai HD. Automated software vulnerability detection via pre-trained context encoder and self attention. In: Proc. of the 12th EAI Int'l Conf. on Digital Forensics and Cyber Crime. Springer, 2022. 248–264. [doi: 10.1007/978-3-031-06365-7_15]

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

[22] Zhang JW, Liu ZX, Hu X, Xia X, Li SP. Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans. on Software Engineering, 2023, 49(8): 4196–4212. [doi: 10.1109/TSE.2023.3286586]

[23] Li Y, Wang SH, 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. Athens: ACM, 2021. 292–303. [doi: 10.1145/3468264.3468597]

[24] Wen XC, Chen YP, Gao CY, Zhang HY, Zhang JM, Liao Q. Vulnerability detection with graph simplification and enhanced graph representation learning. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2275–2286. [doi: 10.1109/ICSE48619.2023.00191]

[25] Steenhoek B, Gao HY, Le W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 16. [doi: 10.1145/3597503.3623345]

[26] Cheng JX, Chen YZ, Cao YZ, Wang HP. A vulnerability detection framework by focusing on critical execution paths. Information and Software Technology, 2024, 174: 107517. [doi: 10.1016/j.infsof.2024.107517]

[27] Kim D, Oh A. How to find your friendly neighborhood: Graph attention design with self-supervision. arXiv:2204.04879, 2022.

[28] Veličković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. arXiv:1710.10903, 2018.

[29] Joern. 2024. https://github.com/joernio/joern

© 中国科学院软件研究所 http://www.jos.org.cn

[30] Zhou YQ, Liu SQ, Siow J, Du XN, Liu Y. *Devign*: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proc. of the 33rd Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 10197–10207.

[31] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, 2020. 1536–1547. [doi: 10.18653/v1/2020.findings-emnlp.139]

[32] Wu YM, Zou DQ, Dou SH, Yang W, Xu D, Jin H. VulCNN: An image-inspired scalable vulnerability detection system. In: Proc. of the 44th Int'l Conf. on Software Engineering. Pittsburgh: ACM, 2022. 2365–2376. [doi: 10.1145/3510003.3510229]

[33] Yuan B, Lu YF, Fang YL, Wu YM, Zou DQ, Li Z, Li Z, Jin H. Enhancing deep learning-based vulnerability detection by building behavior graph model. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 2262–2274. [doi: 10.1109/ICSE48619.2023.00190]

[34] Fan JH, Li Y, Wang SH, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. Seoul: ACM, 2020. 508–512. [doi: 10.1145/3379597.3387501]

[35] FFmpeg Security. 2024. https://www.ffmpeg.org/security.html

[36] Croft R, Babar MA, Kholoosi MM. Data quality for software vulnerability datasets. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering. Melbourne: IEEE, 2023. 121–133. [doi: 10.1109/ICSE48619.2023.00022]

附中文参考文献:

[8] 刘剑, 苏璞睿, 杨珉, 和亮, 张源, 朱雪阳, 林惠民. 软件与网络安全研究综述. 软件学报, 2018, 29(1): 42–68. http://www.jos.org.cn/1000-9825/5320.htm [doi: 10.13328/j.cnki.jos.005320]

[9] 邵思豪, 高庆, 马森, 段富尧, 马骁, 张世琨, 胡津华. 缓冲区溢出漏洞分析技术研究进展. 软件学报, 2018, 29(5): 1179–1198. http://www.jos.org.cn/1000-9825/5504.htm [doi: 10.13328/j.cnki.jos.005504]

[16] 陈可, 鲁辉, 方滨兴, 孙彦斌, 苏申, 田志宏. 自动化渗透测试技术研究综述. 软件学报, 2024, 35(5): 2268–2288. http://www.jos.org.cn/1000-9825/7038.htm [doi: 10.13328/j.cnki.jos.007038]

邱少健(1990－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为智能软件工程.

黄梦阳(1998－), 男, 硕士生, 主要研究领域为软件可靠性.

程嘉濠(1999－), 男, 硕士生, 主要研究领域为机器学习, 软件漏洞检测.

黄琼(1982－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为密码学与信息安全.

© 中国科学院软件研究所 http://www.jos.org.cn