软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(2):625−654 [doi: 10.13328/j.cnki.jos.006696]

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

基于深度学习的源代码缺陷检测研究综述*

邓 枭[1,2], 叶 蔚[2], 谢 睿[2,3], 张世琨[2]

[1](北京大学 软件与微电子学院, 北京 100871)

[2](软件工程国家工程研究中心(北京大学), 北京 100871)

[3](北京大学 信息科学技术学院, 北京 100871)

通信作者: 叶蔚, E-mail: wye@pku.edu.cn

摘 要: 源代码缺陷检测是判别程序代码中是否存在非预期行为的过程, 广泛应用于软件测试、软件维护等软件工程任务, 对软件的功能保障与应用安全方面具有至关重要的作用. 传统的缺陷检测研究以程序分析为基础, 通常需要很强的领域知识与复杂的计算规则, 面临状态爆炸问题, 导致检测性能有限, 在误报漏报率上都有较大提高空间. 近年来, 开源社区的蓬勃发展积累了以开源代码为核心的海量数据, 在此背景下, 利用深度学习的特征学习能力能够自动学习语义丰富的代码表示, 从而为缺陷检测提供一种新的途径. 搜集了该领域最新的高水平论文, 从缺陷代码数据集与深度学习缺陷检测模型两方面系统地对当前方法进行了归纳与阐述. 最后对该领域研究所面临的主要挑战进行总结, 并展望了未来可能的研究重点.

关键词: 深度学习; 缺陷检测; 代码表征

中图法分类号: TP311

中文引用格式: 邓枭, 叶蔚, 谢睿, 张世琨. 基于深度学习的源代码缺陷检测研究综述. 软件学报, 2023, 34(2): 625–654. http://www.jos.org.cn/1000-9825/6696.htm

英文引用格式: Deng X, Ye W, Xie R, Zhang SK. Survey of Source Code Bug Detection Based on Deep Learning. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 625–654 (in Chinese). http://www.jos.org.cn/1000-9825/6696.htm

Survey of Source Code Bug Detection Based on Deep Learning

DENG Xiao[1,2], YE Wei[2], XIE Rui[2,3], ZHANG Shi-Kun[2]

[1](School of Software and Microelectronics, Peking University, Beijing 100871, China)

[2](National Engineering Research Center for Software Engineering (Peking University), Beijing 100871, China)

[3](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract: Source code bug (vulnerability) detection is a process of judging whether there are unexpected behaviors in the program code. It is widely used in software engineering tasks such as software testing and software maintenance, and plays a vital role in software functional assurance and application security. Traditional vulnerability detection research is based on program analysis, which usually requires strong domain knowledge and complex calculation rules, and faces the problem of state explosion, resulting in limited detection performance, and there is room for greater improvement in the rate of false positives and false negatives. In recent years, the open source community's vigorous development has accumulated massive amounts of data with open source code as the core. In this context, the feature learning capabilities of deep learning can automatically learn semantically rich code representations, thereby providing a new way for vulnerability detection. This study collected the latest high-level papers in this field, systematically summarized and explained the current methods from two aspects: vulnerability code dataset and deep learning vulnerability detection model. Finally, it summarizes the main challenges faced by the research in this field, and looks forward to the possible future research focus.

Key words: deep learning; vulnerability detection; code representation

* 收稿时间: 2022-01-05; 修改时间: 2022-02-27; 采用时间: 2022-04-01

1 引 言

随着越来越多的行业使用软件作为其业务载体, 代码已经成为支撑社会正常运转的最基本元素之一, 软件的安全性问题也正在成为当今社会的根本性和基础性的问题. 软件质量已经直接影响了人们的日常生活. 美国国家标准与技术研究院的一项技术报告指出, 软件缺陷每年对美国造成的经济损失高达 600 亿美元[1].

然而, 由于传统的缺陷检测需要专业人员花费较多的人力, 其在软件开发过程中一直占用了较高的成本. 实证研究表明, 缺陷检测与修复耗费成本占软件开发所有成本的 50%–70%[2]. 因此, 人们对自动化缺陷检测的需求日益增长.

随着开源软件的蓬勃发展, 研究者们能够获取的代码量与缺陷信息越来越多. 成立于 2008 年的开源项目托管网站 GitHub, 截至 2020 年, 已有超过 5 000 万名用户、1 亿个开源项目和 2 亿的提交请求, 覆盖了超过 4.28 亿个文件. 截至 2021 年 9 月, 缺陷名录网站 NVD 已经收录了 171 178 条缺陷数据. 这些代码数据为基于学习的缺陷检测研究提供了充分的数据基础.

近年来, 大规模的数据和硬件方面不断提升的计算能力, 使得深度学习技术在图像处理、语音识别、自然语言处理等领域的多项任务上取得了突破性的进展, 促成了人工智能的第三次浪潮. 于是, 研究者们也逐渐开始在代码领域应用深度学习来尝试替代传统的研究方法, 期望通过深度学习来解决现有方法无法解决的一些问题.

经过近 5 年的探究, 深度学习对缺陷代码特征挖掘的能力得到了一定的验证, 也吸引了越来越多的研究者从传统缺陷检测方法转向使用深度学习. 同时, 深度学习方法应用于源代码缺陷检测在数据集构建和模型设计方面依然面临众多挑战. 本综述即聚焦于基于深度学习的源代码缺陷检测技术研究.

1.1 术语定义

软件源代码安全作为软件安全的重要研究点, 其核心内容是对软件源代码进行缺陷检测. 源代码缺陷即程序代码中存在的某种破坏正常运行能力的问题、错误或隐藏的功能缺陷, 会导致软件产品在某种程度上不能满足用户的需要[3]. 源代码漏洞是源代码缺陷的一种, 可导致攻击者利用的缺陷称为漏洞[4]. 当前很多研究特定于漏洞, 因此在总体概述中, 我们将两者统称为“缺陷”, 在特定上下文中会区分“缺陷”与“漏洞”.

1.2 传统缺陷检测技术概述

源代码缺陷检测是检查并发现软件系统中存在缺陷的主要手段之一, 也是成熟的工业软件在开发过程中必不可少的一道工序. 其通过利用统计工具对软件代码进行各维度度量上的审计, 或利用分析工具分析软件的执行过程来查找并定位软件的设计错误、编码缺陷、运行故障. 从运行模式上看, 早期的缺陷检测技术可依据是否需要运行待检测程序划分为静态分析方法与动态分析方法: 静态分析方法一般应用于软件的开发编码阶段, 其无需运行软件, 而是通过扫描源代码分析词法、语法、控制流和数据流等信息来发现缺陷; 动态分析方法则一般应用于软件的测试运行阶段, 在软件程序运行过程中, 通过分析动态调试器中程序的状态、执行路径等信息来发现缺陷. 在该领域几十年的发展历程中, 当前已有许多公开的缺陷检测工具, 如: Coverity[5]、Klocwork[6]以及 Cobot[7]等典型的通用缺陷检测工具, 通过对程序源代码进行静态分析与规则判别来检测系统缺陷; KLEE[8]、S2E[9]、Mayhem[10]等动态工具采用符号执行这种动态分析方法进行缺陷检测; libFuzzer[11]、Radamsa[12]以及 AFL[13]等动态工具采用基于模糊测试的动态分析方法排查错误. 同时, 也有相关研究尝试通过动静态相结合的分析方法来提高检测效率和准确率. 如: angr[14]集成了静态分析和动态符号执行, 能够实现自动化分析二进制文件; SAGE[15]结合使用了模糊测试和符号执行, 并将动态符号执行应用在 x86 架构的程序分析中. 近年来, 为了提高缺陷检测的效率, 也出现了一些自动化或半自动化的缺陷挖掘工具, 比较有代表性的如: Bochspwn[16]对内核层采用污点追踪, 从而检测用户层泄露数据的行为; Digtool[17]针对 Windows 系统, 可自动化捕获程序执行过程中触发的缺陷; 谷歌团队推出的 Syzkaller[18]针对 Linux 内核进行无监督、覆盖引导的模糊测试; RapidScan[19]缺陷扫描器则通过自动化执行 nmap、dnsrecon 和 wafw00f 等多个安全扫描工具, 通过多个工具扫描结果的融合共同发现缺陷.

尽管拥有较长的研究历史, 传统的缺陷检测方法依然存在由其工作模式带来的难以避免的不足. 如传统的静态分析方法往往依赖于专家人工构造缺陷模式, 随着软件与缺陷复杂性的增加, 人工构造成本和难度过高, 且人的主观性导致不同专家对缺陷的理解不一, 这些都会严重影响误报率和漏报率. 动态分析方法中, 监测目标程序的崩溃是模糊测试发现漏洞的重要依据之一, 因此测试效果严重依赖于输入种子的质量, 而测试用例的自动生成与变异存在较大的偶然性, 其存在测试冗余、测试攻击面模糊、难以发现访问控制漏洞和设计逻辑错误等问题[20]. 动态分析方法中的符号执行方法虽然能以较少的测试用例覆盖更多的程序路径, 从而挖掘复杂软件更深层次的缺陷, 但仍然存在路径爆炸、约束求解难、内存建模与并行处理复杂等问题[21], 单独处理大型软件系统时仍存在较大困难[22]. 因此, 虽然上述早期缺陷检测方法已在各类小型规模软件的缺陷检测任务中取得了一定成果, 但在实际的代码工业环境中应对大型复杂软件系统以及变化多样的新型缺陷时, 通常无法满足需求.

1.3 基于传统机器学习的缺陷检测

针对大型复杂软件系统的缺陷检测问题, 研究者一方面尝试通过优化和改造现有方法来突破传统缺陷检测方法在检测能力上的瓶颈, 另一方面尝试探索新的智能化软件缺陷检测方法. 直观上认为, 大部分缺陷相关信息可以通过代码分析得到, 即挖掘软件缺陷的能力与分析代码数据的能力紧密相关, 而基于学习的方法非常适合于从海量数据中发现和学习规律. 理论研究中, Hindle 等人[23]利用统计学的方法将编程语言和自然语言进行比较, 研究结果表明, 两者具有非常相似的统计学特性, 甚至编程语言更加规整, 从而提出了代码的“自然说”假设. 受到这个假设的启发, 研究人员逐渐意识到运用统计学习方法对代码中蕴含的规律进行分析和泛化的可行性.

早期基于学习的缺陷检测研究主要使用机器学习方法, 如 VCCFinder[24]针对数据形态设计人工特征并进行统计, 得到具有区分性的特征项, 在对样本特征值进行 One-Hot 编码后, 使用支持向量机对其进行分类. 但该类方法在效率与效果上均存在较大的不足: 一方面, 大多数机器学习方法依然需要专家人工构造特征作为输入, 其检测能力受限于特征工程的质量高低; 另一方面, 当前的机器学习方法在实际的检测效果上存在较高的误报率, 难以达到实际应用的需求. 究其根本, 主要原因是机器学习模型对深层特征的挖掘能力有限.

1.4 基于深度学习的缺陷检测

得益于深度神经网络在图像识别、自然语言处理等任务上取得的巨大成功, 人们发现, 深度学习方法在挖掘深层特征上更加具有优势. 一些研究者开始尝试将其引入源代码缺陷检测任务. 当前的结果显示, 深度学习在缺陷检测任务上同样具有传统方法和机器学习方法无法比拟的优势. 并且, 由于其研究历史相对短暂, 还有相当深入的空间供研究者探究.

使用深度学习进行缺陷检测任务, 核心构成即为两部分: 缺陷代码数据集与深度学习缺陷检测模型. 缺陷代码数据集是深度学习模型的学习基础与特征来源, 其需要拥有足够规模的代码数据来对缺陷代码与非缺陷代码进行表征. 深度学习缺陷检测模型则需要根据代码的特性设置合适的网络结构, 以充分挖掘缺陷代码数据集中的特征, 从而得到区分缺陷代码与非缺陷代码的能力.

本文广泛收集了当前将深度学习应用于源代码缺陷检测的相关研究. 我们使用“vulnerability/defect/bug”等关键词查询了谷歌学术搜索、IEEE Xplore、ACM Digital Library、Springer、DBLP、arXiv 以及中国知网 CNKI 等搜索引擎和数据库, 共查询到 908 条结果; 随后, 通过“code/detection”等关键词筛查文章摘要, 保留了其中 237 篇; 再由 2 人逐一对查询结果的标题、摘要进行人工审查, 过滤不使用深度学习方法的无关内容, 在出现分歧时共同进行二次讨论; 最后根据文献的引用情况进行查缺补漏. 最终, 我们一共搜集到 149 篇基于深度学习进行源代码缺陷检测的相关文章, 时间覆盖 2017 年–2021 年 11 月, 其详细的年份分布情况如图 1-1 所示.

可以看到, 从 2017 年起, 深度学习开始被引入到源代码缺陷检测领域. 近年来, 该领域论文的发表数量呈逐年增加的趋势. 该数据表明, 使用深度学习进行缺陷检测的研究热度在不断增加.

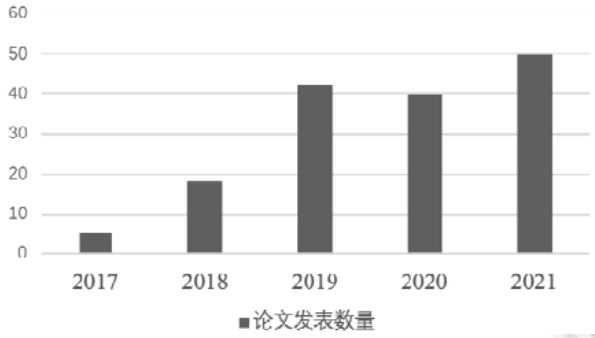

图 1-1 不同年份论文发表分布

本文在介绍深度学习缺陷检测技术时优先选择了 CCF 分级较高的论文，包括 CCF-B 及以上和部分高质量的 CCF-C 级论文. 最终，本文选取了 67 篇深度学习源代码缺陷检测相关论文进行详细介绍，其中相当一部分论文来自所涉及领域的高质量会议和期刊，例如 USENIX Security 会议(4 篇)、CCS 会议(2 篇)、OOPSLA 会议(2 篇)、NeurIPS 会议(2 篇)、IJCAI 会议(2 篇)、ESEC/FSE 会议(1 篇)、ACL 会议(1 篇)、ICML 会议(1 篇)、TDSC 期刊(4 篇)、TOSEM 期刊(2 篇)、TSE 期刊(1 篇)、Proc. IEEE 期刊(1 篇)、《软件学报》(1 篇)等.

1.5 相关综述

事实上，在 2019 年底，李韵等人[25]就已经对基于机器学习的软件漏洞挖掘方法进行了分类与分析，其重点关注机器学习方法，且主要关注 2010–2019 年的文献. 而该领域中基于深度学习的方法作为近年的热门方向，其相关文献主要发表于 2019–2021 年间，因此本综述与前人总结在技术关注点与文献分布上均有较大的差异，可以作为前文的补充与进一步的延申扩展. 此外，还有一些针对缺陷检测技术的英文综述，如 Chakraborty 等人[26]和 Lin 等人[27]的研究，与本文的方向相近. 但此类综述往往只关注深度模型，而本文在覆盖更新研究点的同时，对数据集方面也进行了更为细致的分析.

本文从源代码缺陷数据集构造与深度学习缺陷检测模型这两个技术模块对当前研究进行了分类归纳与整理，并对目前该研究领域亟待解决的问题与未来可能的研究方向进行了阐述.

2 缺陷代码数据集的构造

由于使用深度学习进行缺陷检测的研究近几年才兴起，因此有别于一些经典、成熟的深度学习任务拥有统一、公认的数据集，当前在缺陷检测领域并没有一个用来统一评测的缺陷代码数据集. 当前的研究往往是自行构建一个缺陷代码数据集，作为衡量模型性能的数据基础. 这就导致各个研究使用不同的自建数据集. 然而，在不同数据集上依靠数值指标来横向比较不同模型的优劣是不现实的，因此，当前该领域的研究往往会回避模型之间的比较问题. 不过目前为止，还没有研究针对性地指出数据集不统一带来的问题. 同时，自建缺陷代码数据集的构建过程也存在较多的模糊或近似的问题，甚至存在一些不合理的操作，这些都会严重地影响模型在其上的训练表现. 因此，本文将数据集的构造方法作为基于深度学习的缺陷检测中一个重要的研究点，着重关注这一部分的研究现状与存在的问题.

不同于图像处理、自然语言处理等任务拥有成熟、通用的数据集，使用深度学习进行缺陷检测面临着巨大的数据集方面的问题. 究其根本，主要有两个核心难点.

(1) 在已知的真实项目中，有缺陷的代码数量有限. 缺陷本身就是重要资源，有些甚至是涉及到更高层安全的战略资源. 当前，许多研究花费了大量精力进行标注，但是不会公开或者仅部分公开；代码开源且已被公开确认收录的缺陷数量少，且对齐关系模糊. 例如 NVD 缺陷库中的数据，至今已有 17 万余条，但经过验证，能够对齐到具体代码的只有 5 300 余条，细分至不同程序语言则数量更加有限，难以支撑深度模型的训练；商业工具自行构造并收集的测试数据集，由于其巨大的商业价值

不会轻易公开;

(2) 人工标注或自动化生成的难度和成本极高. 对于某些领域, 人工标注数据的成本是容易的, 例如图片、文本、语音, 大部分标注任务普通人即可完成, 这就使得其标注的成本相对低廉, 可以通过众包的方式完成. 但是对于专业的程序员, 识别缺陷也是极其费时耗力的工作.

因此, 缺陷代码数据集作为深度学习缺陷检测任务的基础, 具有很高的重要性, 但同时构造难度较大.

2.1 缺陷代码数据集的构造方法分类

构造缺陷代码数据集, 从流程上可以分为 3 个主要技术环节: 缺陷条目的获取、缺陷代码的抽取与处理、处理后样本的标注. 因此, 对应每一个技术环节, 我们按照数据来源、样本粒度与标签来源对其进行分类, 如图 2-1 所示.

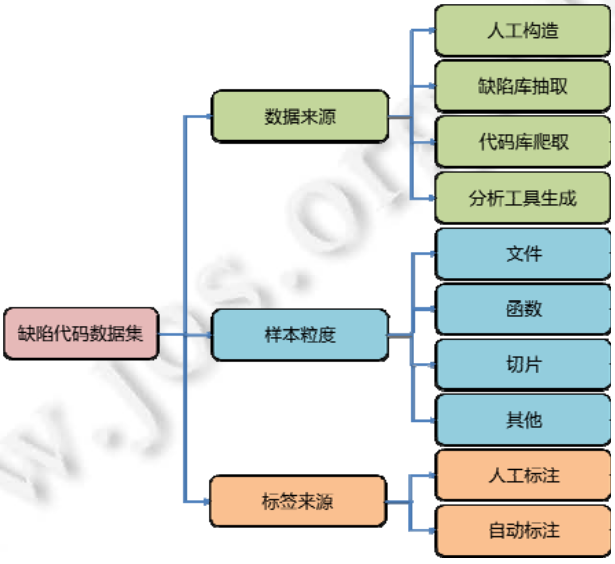

图 2-1 缺陷代码数据集的构造方法分类

构造缺陷代码数据集, 首先需要获取缺陷数据, 即找到与缺陷相关的代码. 从数据来源上划分, 当前缺陷数据的来源主要分为 4 类: 人工构造、缺陷库抽取、代码库爬取与分析工具生成.

(1) 人工构造即通过人工规则将正确代码转化为缺陷代码;

(2) 缺陷库抽取即以公开的缺陷数据库为入口, 抽取指定的缺陷类型条目, 并提取对应的缺陷代码;

(3) 代码库爬取即直接检索代码仓库, 从中爬取出具有缺陷特征的代码提交;

(4) 分析工具生成即通过缺陷分析工具对源代码进行缺陷检测, 将检测结果作为其是否含有缺陷的标签.

当确认了代码中包含缺陷后, 如何选择合适的粒度成为关键问题. 粒度过大则可能带来较多噪声与冗余, 粒度过小则可能使得缺陷特征丢失. 同时, 不同的粒度适用的网络结构也不同, 良好的粒度与网络组合能够使得模型更好地捕获缺陷特征. 当前主要的样本粒度为文件级、函数级、切片级及其他类别.

(1) 文件级主要是将项目进行文件级拆分构成数据集;

(2) 函数级大多数先获得文件数据, 再将其按照函数划分为单个样本;

(3) 切片级需要对代码进行解析, 并根据设定规则进行切片.

当得到了处理后的样本后, 需要对其进行是否含有缺陷的标注.

(1) 若缺陷样本是通过人工规则生成的, 则在生成时即已拥有标签, 可以自动完成标注;

(2) 对于从缺陷库中抽取的缺陷数据, 若采用与数据源相同的缺陷粒度, 则往往可以直接获得其标签,

可以自动完成标注;

(3) 若需要对缺陷进行更细粒度的处理, 则需要研究者自行进行细粒度的标注工作. 当前, 获取标签的方式主要分为自动标注与人工标注两种.

为了便于介绍, 下面我们使用数据来源作为分类维度, 对当前的数据集构造方法进行介绍.

2.1.1 基于人工构造

在软件开发过程中, 为了测试软件的可靠性, 往往会人工撰写一些测试样例用于测试. 这些测试样例因此成为了天然的缺陷样本数据.

Sard (software assurance reference dataset)[28]是美国国家标准与技术研究院发布的软件防护参考数据集, 旨在为用户、研究者与软件安全工具开发人员提供已知的安全缺陷来对安全工具与方法进行测试评估. 该数据集收集了来自工业生产、人工构造与学生撰写的测试用例, 涵盖 C/C++、Java 等 7 种语言, 当前, 该数据集共包含测试用例 177 184 条. Sard 中的代码按照 Good (不含缺陷)、Bad (包含缺陷)与 Mix (含有缺陷及补丁)进行分类, 并对缺陷部分明确标注了缺陷触发位置. 虽然 Sard 数据集中的测试用例来源不尽相同, 但其均为人工撰写用于测试的基础样例, 其缺陷流程较为简单, 复杂度较低, 与真实的代码缺陷存在一定差距.

OWASP Benchmark[29]项目是一个综合的 Java 测试样例集, 专为软件漏洞分析工具的自动评估而设计. 当前, OWASP Benchmark 测试套件有两个主要版本(V1.1 和 V1.2), 包含 11 个类别的 Web 应用程序漏洞的数千个标记测试用例. 其中, V1.1 中拥有更多的测试用例, 但用例不可执行也不可利用, 且正负样本不平衡. V1.2 对这些问题进行了改进, 其用例全部可执行、可利用, 且正负样本均衡.

在研究初期, 为了尽快获得缺陷数据, 很多缺陷代码数据集是从缺陷库中直接抽取出来的.

Xu 等人[30]直接使用 Sard 数据集提供的数据, 其选择了其中缓冲区错误(buffer error, CWE-119)与资源管理错误(resource management error, CWE-399)这两个类型, 共计 23 185 个程序, 其中, 11 093 为缺陷程序, 占比 47.8%.

Duan 等人[31]也使用 Sard 数据集进行测试, 他们为了探究如何更好地捕捉缺陷代码与非缺陷代码中的细小差距, 因此 Sard 数据集正好可以满足该需求. 最终, 他们选择了缓冲区错误(CWE-119)和资源管理(CWE-399)这两个类型, 按照函数级进行收集, 共计 28 049 个函数, 其中, 缺陷函数为 10 561 个, 占比 37.7%. 该数据集未开源.

Cao 等人[32]使用了 Sard 数据集作为数据来源, 并同样选取了其中 CWE-399 和 CWE-119 这两类共计 11 397 个文件, 并按照函数进行分割与打标. 最终, 该数据集包含 63 828 个函数, 其中, 缺陷函数为 46 337 个, 占比 72.6%.

Saccente 等人[33]在 Sard 的基础上构建了一个 Java 语言的函数级缺陷代码数据集. 其首先选取了原数据集中类型下数量超过 100 的 29 个 CWE 缺陷类型, 共 44 495 个文件. 在预处理过程中, 其将对应文件按函数级分割, 然后使用分词器 Javalang 进行分词, 以作为向量形式投入模型. 该数据集已开源(https://gitlab.com/TUSoftwareEngineering/vulnerabilitylocalization-using-machine-learning).

Feng 等人[34]从 Sard 中提取了 3 个级别的针对 C/C++语言的函数级缺陷代码数据集. 其中, 针对 Buffer Overflow 的 BO 数据集包括 CWE-121 与 CWE-122, 用于测试模型对最常见的单一类型缺陷的检测能力; 针对多个类型的 5K 数据集包括样本数量大于 5 000 的全部类型; 最后是整个 Sard 数据集用于完整的评估模型. 所有的数据均按照函数进行分割, 最终的完整数据集包含 269 985 个函数, 其中, 缺陷函数为 89 071 个, 占比 33%. 该数据集未开源.

为了满足图神经网络的输入需求, Ghaffarian 等人[35]构建了一个由多种代码图组成的缺陷代码数据集. 该数据集基于 Java 语言, 其从 OWASP Benchmark v1.2 中提取了 7 个类型的测试样例共 1 698 条, 其中, 缺陷样例为 932 条, 占比 54.9%. 该数据集并未开源. 但作者开源了其跨平台代码解析工具 PROGEX[36], 其用于标准化地对程序生成各种图表示, 以便与后续的模型作快速的数据准备. 同时, 该工具还支持将各种生成图以不同的文本格式存储, 例如 DOT、JSON、GML 等.

虽然从测试样例中抽取样本较为快捷, 但其数量毕竟有限, 无法提供更大规模的缺陷样本. 为此, 一些研究者尝试自行构造缺陷样本, 其通过自行设定的规则, 对正确代码变异来得到缺陷代码.

为了利用代码中变量与函数命名上的自然语言信息, Pradel 等人[37]构建了一个基于命名的缺陷数据集. 针对构造数据集时存在的诸多困难, 他们选择使用代码转换的方式, 将非缺陷代码改写为缺陷代码, 从而极大地减少了构造数据集所需的人力与计算资源. 具体来说, 其使用3种变异规则对代码进行修改, 包括交换函数参数、修改二元操作符、修改二元运算中的操作数. 这些操作都是在 AST 的基础上进行的. 由于其默认大部分的代码均为正确代码(指的是不包含上述3种错误模式), 因此转化前的代码即为非缺陷代码, 转化后的即为缺陷代码. DeepBugs 的数据集由 150 000 个 JavaScript 文件生成, 这些文件均由开源工程收集并经过了清洗与去重, 一共包含 6 860 行代码. 在经过上述变异操作后, 最终该数据集包含 16 634 458 个样本. 该数据集已开源(https://github.com/michaelpradel/DeepBugs).

Allamanis 等人[38]同样采用了人工规则自动构造缺陷数据的方法. 其在 DeepBug[37]规则的基础上, 增加了变量替换与常数替换两种构造规则. 其通过这一方法对 PyPI 中下载量前 4 000 名的 Python 项目进行了变换, 构造了针对 Python 语言的缺陷代码数据集. 该数据集已开源(https://www.microsoft.com/en-us/download/103554).

Choi 等人[39]选择直接生成缺陷代码段, 其自行构建了一个针对缓冲区溢出类型的函数级 C 语言缺陷代码数据集. 代码生成规则中采用与 Sard 相同的缓冲区访问函数和初始化方法, 以保持至少相同级别的任务复杂性. 该数据集中每个构造的样本代码段均为一个无返回值函数, 其由 3 个阶段构成: 初始化变量、缓冲区分配和缓冲区访问. 在缓冲区的使用上, 分为直接调用、使用 API 调用、利用变量分配和利用 API 调用与变量重划分共 4 个不同难度级别. 该数据集最终包含 10 000 个样本, 其中, 缺陷样本占比 50.1%. 不过, 该数据集中的样本构造规则与逻辑较为简单: 其长度均限定在 8–30 行, 远低于实际应用场景中的待测函数代码长度; 其对缓冲区的调用限定为 4 种模式, 均较为直接, 在程序复杂度上远低于实际应用场景. 该数据集已开源(https://github.com/mjc92/buffer_overrun_memory_networks).

总的来说, 基于人工构造的方法通过特定的规则可以快速地生成大量的样本. 然而, 由此构建的缺陷代码数据集只能表征构造规则, 这会导致训练出的深度模型只局限于对特定规则的识别, 而偏离其“缺陷检测”的原始目标. 因此, 当前通用类型检测的系统不再人工构造缺陷数据样本.

2.1.2 基于缺陷库抽取

当前, 有一些机构或组织对软件开发中发现的缺陷进行了收集、整理与发布, 因此通过该类缺陷库即可获得缺陷条目入口. 但是该类缺陷库大多只有部分条目能够链接到缺陷对应的源代码, 因此基于缺陷库抽取缺陷数据的方法大多需要先筛选出其中对齐源代码成功的条目, 再进行后续的处理.

NVD (national vulnerability database)[40]是美国国土安全部下属的联邦计算机应急准备小组维护的国家漏洞数据库, 是当前较为权威的源代码漏洞数据库. 该库实时同步 CVE 发布的条目, 在 CVE 基础上提供安全评分、缺陷影响评级、修复信息、缺陷查找等功能, 并通过人工审核的方式将缺陷与原项目相关信息进行链接. 由于其数据相对较为齐全, 人工对齐源码出处的质量相对较高, 因此该缺陷库成为当前最常用的缺陷抽取源.

Lin 等人[41]使用函数作为缺陷代码的粒度, 其选择了 6 个知名工程(LibTIFF、LibPNG、FFmpeg、Pidgin、VLC Media Player、Asterisk), 通过 NVD 检索其相关缺陷, 并人工定位缺陷所在函数, 剩余未被标记为缺陷函数的即作为非缺陷函数加入数据集. 最终, 该数据集包含 32 988 个函数, 其中, 缺陷函数为 457 个, 占比 1.4%, 不平衡的情况较严重. 该数据集已开源(https://github.com/DanielLin1986/TransferRepresentationLearning).

Li 等人[42]提出了基于 C/C++语言的切片级缺陷代码数据集. 其检索了 NVD 与 Sard 中缓冲区错误(CWE-119)与资源管理错误(CWE-399)这两类缺陷条目, 选择其中 19 个开源 C/C++项目的相关缺陷. 其定义了代码片段(code gadgets), 即语义上相互关联的几行代码, 并用它来表示程序, 以使其避免函数级表示会带来的无关噪声信息. 其首先在源代码中按照分析工具 Checkmarx 的危险函数标准查找库函数与 API 函数调用作为危险入口, 将其语句中的变量作为起点进行切片, 并按照自然代码顺序进行组合形成 code gadget. 该切片只考

虑数据依赖关系. 在标注是否含有缺陷时, 其依靠代码片段中是否含有修复时被删除或修改的语句来进行判断: 若有, 则标注为缺陷片段. 最终, 该数据集包含 61 638 个代码片段, 其中, 17 725 个为缺陷片段, 占比 28.8%. 不过, 其中仅有 1.4% (840 个)来自于真实项目缺陷(NVD). 该数据集已开源(https://github.com/CGCL-codes/VulDeePecker).

为了支持多分类缺陷检测任务, Zou 等人[43]在 VulDeePecker[42]数据集上进行了扩充, 增加了控制依赖作为切片依据, 并将缺陷类型也加入其中, 而不仅仅记录其是否含有缺陷. 其μVulDeePecker 数据集在切片时同时进行前后向的切片, 并同时考虑数据依赖与控制依赖. 该数据集最终包含 181 641 个代码片段, 其中, 43 119 个为缺陷片段, 占比 23.7%. 其共涉及 40 个 CWE 三级分类. 但是, 该数据集的缺陷样本存在类型信息, 而非缺陷样本均标记为非缺陷类型, 不存在类型信息, 可能会影响模型的判断. 并且, 该数据集中来自 NVD 的真实缺陷占比不到 1%. 该数据集已开源(https://github.com/muVulDeePecker/muVulDeePecker).

由于 VulDeePecker[42]数据集只考虑了以危险库/API 函数调用作为入口的缺陷条目作为样本, 因此只能提供两种缺陷类型的样本, 其覆盖范围较窄. Li 等人[44]在其基础上进行了改进, 使其同时兼顾语法特征与语义特征, 从而更全面地表征缺陷以覆盖更多的缺陷类型. 该数据集 SySeVR 同样从 NVD 与 Sard 数据库中抽取缺陷, 其首先根据 Checkmarx 的危险变量规则提出了 4 种危险入口(库/API 函数调用、数组使用、指针使用、算数表达式), 从代码中的危险入口开始切片. 与 VulDeePecker 数据集根据语句中的标识符进行切片不同的是, SySeVR 数据集是在语句的程序依赖图基础上进行切片, 从而利用控制依赖与数据依赖信息完善缺陷特征. SySeVR 的标签方法与 VulDeePecker 数据集相同. 最终, SySeVR 数据集包含 420 627 个代码片段, 其中, 缺陷片段为 56 395 个, 占比 13.4%. 该数据集已开源(https://github.com/SySeVR/SySeVR).

缺陷代码与其修复版本同时在数据集中可以帮助模型更好地学习到其中的差异, 并且能够保证数据集正负样本的平衡. 为此, Xiao 等人[45]构建了一个针对 C/C++的缺陷代码与修复数据集. 其选定了 10 个不同领域的开源工程, 从 NVD 和项目代码提交中检索相关缺陷与修复, 根据修复的位置确定缺陷函数. 该数据集只包含修复在单个函数内部的. 最终, 该数据集包含 25 377 对缺陷与其修复. 不过, 该数据集未开源.

Nikitopoulos 等人[46]构建了一个多语言的缺陷代码数据集 CrossVul. 其人工审查了 NVD 提及的 5 877 个 Github 提交, 确认其链接的可用性, 并标记了缺陷及其对应补丁所在文件, 并将提交信息也作为数据一同保存. 该数据集最终涉及 1 675 个项目, 包括 27 476 个文件(缺陷与非缺陷各半), 涵盖 40 种编程语言, 涉及 168 个 CWE 类型, 每一条缺陷都记录了其对应的 CVE ID 与其来源链接. 不过, 该数据集只是文件级数据, 并没有进一步细化到函数甚至切片, 因此, 在缺陷检测任务中使用可能需要较大工作量的细化标注工作. 该数据集已开源(https://doi.org/10.5281/zenodo.4734050).

Lin 等人[47]提出了一个针对深度学习缺陷函数检测的 C 语言评测集. 其根据 NVD 与 CVE 的描述信息, 通过人工的方式将其对齐到 GitHub 代码库, 并标注缺陷函数与缺陷文件. 其默认最新版本的工程中其他文件均为非缺陷文件, 因此将新版本中未被标记的文件和其中函数作为非缺陷样本. 其标注了 9 个开源工程, 共计 5 780 个文件 60 768 个函数, 其中, 缺陷函数为 1 471 个, 占比 2.4%. 但是, 该数据集没有考虑到缺陷类型等信息.

为了测试跨域学习方法的效果, Liu 等人[48]提出了一个针对 C/C++的缺陷代码数据集. 其选用了 VulDeePecker 数据集中的 CWE-119 与 CWE-399 这两类, 用来测试缺陷类型间的跨域. 其另外根据 NVD 与 CVE 信息人工标注了 3 个开源项目(LibTIFF、FFmpeg、LibPNG)的函数级缺陷数据, 每个项目均抽取了 9 种缺陷类型, 用来测试项目间的跨域. 非缺陷样本都是从项目中未被标记为缺陷样本的函数中抽取的. 该数据集已开源(https://github.com/wolong3385/SVD-Source).

很多缺陷代码数据集在构造时都基于一个假设: 工程代码除了已知的缺陷以外没有其他缺陷. 因此, 设计者将被标记为缺陷样本以外的部分当作非缺陷样本对待. 然而事实是, 新缺陷都是从这些“非缺陷样本”中发现的. 因此, 这种假设会带来许多噪声从而影响模型的判别能力.

为此, Jimenez 等人[49]放弃了缺陷样本以外的代码来避免这个问题, 他们只使用缺陷样本和其对应的修改

后版本作为样本来构造数据集. 他们提出了一个自动的可扩展框架, 并由此构建了缺陷代码数据集 VulData7. 其通过 NVD 作为缺陷入口, 选取 4 个 C 语言开源项目(Linux Kernel、Wireshark、OpenSSL、SystemD)的条目, 自动将其对齐到项目代码库及相应提交. VulData7 记录了每条缺陷的报告信息(描述、CVE 编号、CWE 编号、CVSS 严重程度)、影响版本列表、修复提交和修复前后文件. 当前版本的 VulData7 包含 2 809 条缺陷, 其中, 1 598 条附带了修复信息. 其更新机制会不断同步 NVD 以不断维护数据集. 该数据集已开源(https://github.com/electricalwind/data7).

Clemente 等人[50]从缺陷跟踪网站 Bugzilla、Mozilla Foundation Security Advisory (MFSA)和 Computer Vulnerability Exposure site (CVE)上提取信息构建缺陷数据集. 其将搜索范围限定在 Mozilla 的 C++项目 Firefox 上, 根据网站的报出提取出缺陷相关文件. 最终, 该数据集包含 395 个文件, 其中有 200 个缺陷样本, 占比 50.6%. 该数据集未开源.

除了著名的通用缺陷库以外, 许多大型软件开发商都会安排安全团队自行维护其软件产品的安全记录, 这些安全记录也可以为构建缺陷代码数据集提供可靠的信息.

Alexopoulos 等人[51]提出了一种自动构建缺陷数据集的方法. 他们从 Debian 安全团队维护的缺陷库 Debian Security Advisories (DSAs)中进行信息挖掘. 首先, 从 DSA 的缺陷报告中可以直接提取出报出的缺陷与对应的 NVD 链接, 同时还包括其 CWE 缺陷类型与严重等级; 随后, 使用官方提供的快照功能下载每个月的工程源代码; 再通过开源工具 PKGDIFF 对相邻版本的修改信息进行生成, 从而定位出缺陷报告是在何时被引入. 该数据集最终选用了 7 个 Debian 项目, 分别是 Linux kernel、Firefox、Chromium、PHP、OpenJDK、Thunderbird 和 Wireshark. 不过, 由于该研究还未完全完成, 该数据集未开源.

Fan 等人[52]提出了一个针对 C/C++语言的函数级缺陷代码数据集 Big-Vul. 其通过爬取 CVE 数据库和相关源代码仓库中的 GitHub 链接, 定位了 348 个 GitHub 项目, 并对齐到每个缺陷的修复提交. 其将函数在修复前后两个版本与 CVE 描述信息一并保存, 记录了包括 CVE ID、严重程度、描述等 21 个特征值. 项目中的其他非缺陷相关函数作为正确函数也加以保留. 最终, Big-Vul 数据集包含 3 754 个代码缺陷, 覆盖 91 个缺陷类型, 共含有 11 823 个缺陷函数与 253 096 个非缺陷函数, 缺陷函数占比 4.5%. 该数据集已开源(https://github.com/ZeoVan/MSR_20_Code_Vulnerability_CSV_Dataset).

Li 等人[53]从 tera-PROMISE 缺陷数据库中抽取针对 Java 语言的文件级缺陷代码数据集. 其选择了 7 个开源 Java 项目包含的缺陷条目, 每个项目都选择了连续的两个版本. 从其对应的 GitHub 仓库中爬取了源文件, 并按照报出缺陷位置对文件进行标注. 最终, 该数据集包含 3 290 个文件, 其中, 缺陷文件为 1 152 个, 占比 35%. 该数据集未开源.

Zhang 等人[54]同样从 tera-PROMISE 数据集中抽取缺陷数据, 其选择了 12 个 Java 语言的开源项目, 共计 5 194 个文件, 其中, 缺陷文件为 1 617 个, 占比 31.1%. 该数据集同时还包含了 20 种传统代码度量信息.

Ponta 等人[55]将 NVD 和 50 余个项目自身的安全记录网站作为缺陷条目入口, 在 4 年的时间内监控其安全报告的情况, 通过人工的方式记录缺陷相关提交(commit)与修复相关提交. 截至 2019 年, 该数据集共记录了 624 个缺陷的修复, 涉及 1 282 个提交, 跨越 205 个 Java 项目. 该数据集已开源(https://github.com/SAP/vulnerability-assessment-kb/tree/master/MSR2019).

当前, 从 NVD 等缺陷库中抽取缺陷条目已成为较为通用的做法, 其已经成为构建缺陷代码数据集的重要数据来源. 这就意味着源缺陷库的质量会极大地影响所构建数据集的质量. 事实上, 通过细致的比对, 人们逐渐注意到该类缺陷库的信息质量存在一些问题. 例如, 在多个缺陷来源的对比中(如 NVD 与 CVE)会出现缺陷信息不一致的情况. 如果对缺陷库中的条目不加验证即作为标答(当今普遍做法), 会给缺陷数据集的构建带来噪声, 继而极大地影响在此基础上训练的深度学习模型的检测性能. 为此, Dong 等人[56]提出了一个自动化系统 VIEM 来检测完全标准化的 NVD 数据库与非结构化 CVE 描述及其引用的漏洞报告之间的不一致信息. 其使用了自然语言处理任务中的命名实体识别与关系抽取技术, 对非结构化信息进行提取并与结构化信息比较, 从而大规模地自动检测不一致信息. 其对过去 20 年中 78 296 个 CVE ID 与 70 569 条缺陷报告进行了比对,

只有 59.82%的条目可以严格匹配.

然而, 该不一致性的鉴别只针对缺陷影响的项目版本不一致性. 事实上, 由于公开缺陷库都是由人工的方式进行维护, 其许多信息都可能存在错误/不一致, 例如对应的原提交链接等. 这些都会给以此为基础构造的缺陷代码数据集带来影响, 也需要进行正确性的确认.

除了数据的冲突, 另一种影响数据集数量的问题是数据的缺失. 当前, 由于需要大量的缺陷代码数据, 使用混合数据来源已成为一种常用的扩增数据集的做法. 但是, 混合数据来源容易带来的问题是属性值的不全, 这是由各个缺陷库的数据存储结构所导致的. 即使是同一数据源的缺陷数据, 也可能有数据不全的情况. 例如, 在 NVD 缺陷库中, 有些条目没有 CVE 缺陷类型. 这种数据的缺失会影响数据集的数量与质量, 需要对其进行补全.

Rostami 等人[57]针对缺陷代码数据中数据缺失的问题设计了一个机器学习框架, 从数据完整的条目中学习, 从而对缺失项进行预测. 该方法能够较好地预测缺失的类型数据, 但对于无类型数据, 该框架无法补全.

Gonzalez 等人[58]同样针对 NVD 数据集中分类信息补全标注的问题, 提出了一个自动缺陷标注的方法. 其仅仅使用缺陷条目的 CVE 描述作为依据, 在对描述信息进行清洗并使用 TF-IDF 表示后, 使用 6 种分类器(朴素贝叶斯、决策树、支持向量机、随机森林、AdaBoost 支持向量机、多数投票), 按照 NVD 漏洞描述规则(vulnerability description ontology)中的 19 个类型标准来对其进行分类. 实验结果显示: 多数投票的分类效果最佳, 准确率达到 74.5%. 但考虑到训练成本和时间, 使用支持向量机是该任务的最优选择. 不过, 该方法同样无法对选择型数据进行补全.

因此, 对于非选择型数据的缺失, 如何对其加以补全而不是简单丢弃, 是构造一个样本数量足够多的数据集的可行方向, 一些生成式的方法值得探究.

总的来说, 基于缺陷库抽取的方法利用了现有的人工审查后的缺陷资源, 因此在缺陷来源上可靠性较高. 使用该类方法构造的缺陷代码数据集, 其主要针对的是具有 CVE 编码的真实软件缺陷, 可以认为其样本较为贴近缺陷检测理想的使用场景. 但是, 由于缺陷库需要相关人员的审查, 因此即使经过数十年的积累(如 NVD), 现有的缺陷库中可被用于抽取样本的缺陷条目数量仍然极为有限.

2.1.3 基于开源代码库爬取

从缺陷库抽取缺陷数据较为可靠, 但缺陷库中的可用数据量较少. Bhandari 等人[59]的实证研究表明, NVD 代码库中的 171 178 条缺陷数据中, 只有 5 300 余条能够链接到对应的提交代码, 对齐率不足 3.2%. 而 GitHub 等开源代码库中拥有海量的代码提交, 其中拥有大量的缺陷提交与修复提交, 这些均可被挖掘作为缺陷条目. 由于人工成本等原因, 很多缺陷相关提交并未与 CVE 关联, 甚至没有显式的文字描述信息. 因此, 如何判别缺陷相关提交、如何挖掘代码库中的非标记缺陷条目, 成为了扩增缺陷代码数据集的重要方向.

一种自动获取缺陷数据的路线是启发式搜索.

Karampatsis 等人[60]聚焦于在单语句中可完成修改的缺陷, 构造了针对 Java 语言的单语句缺陷代码数据集 ManySStuBs4J. 其选定最热门的 1 000 个开源 Java Maven 项目, 从其 GitHub 仓库中爬取历史提交, 通过安全相关关键词(error、bug、fix 等)对其进行筛选, 并只保留在单语句中对缺陷进行修复的提交. ManySStuBs4J 数据集最终包含 153 652 条单语句修复缺陷. 该数据集已开源(https://doi.org/10.5281/zenodo.3653444).

为了测试对跨函数缺陷的检测效果, Li 等人[61]从开源网站挖掘了一个函数级 Java 语言缺陷代码数据集. 其使用关键词匹配的启发式方法从 8 个著名开源 Java 项目的仓库中爬取缺陷修复报告, 将修复前后的整个项目下载, 通过缺陷修复时的添加与删除操作定位缺陷函数. 不过, 其并未对标签的情况进行人工验证. 最终, 该数据集包含 497 万个函数, 其中, 182 万个为缺陷函数, 占比 36.7%. 虽然该数据集被用来测试跨函数的检测效果, 但其在样本的标注中只对缺陷函数进行了标注, 而并未将其相关函数也生成标注. 该数据集已开源(https://github.com/OOPSLA-2019-BugDetection/OOPSLA-2019-BugDetection).

启发式搜索的规则较为简单, 因此准确性难以得到保障, 可能会为数据集带来较多噪声. 为此, 研究者们开始探究通过学习的方式进行更精确的判断与挖掘.

Zhou 等人[62]分别训练了 6 个基础分类器(随机森林、朴素贝叶斯、*K* 近邻等), 并通过逻辑回归来对其分类结果进行整合, 从而自动判断代码提交或问题报告是否与缺陷相关. 该系统仅对提交的自然语言部分进行分类, 而不考虑代码部分的信息. 该系统可以实时监控项目提交, 并不断向数据集中补充样本. 其从 GitHub、JIRA、Bugzilla 中挖取 Java、Python、Ruby、JavaScript、Objective C 和 Go 语言共计 8 546 个项目的提交信息与问题报告. 最终, 其提交的数据集中包含 12 409 个提交, 其中, 1 303 个为缺陷提交, 占比 10.5%. 问题报告数据集中包含 24 188 个问题报告, 其中, 1 905 个为缺陷报告, 占比 7.9%. 其安全团队通过两轮的标注对数据标签进行验证. 不过, 该数据集未开源.

Sabetta 等人[63]也尝试自动判别代码提交以筛选出安全相关提交, 其在考虑自然语言信息的基础上, 也将源代码的修改加入参考. 其构建了两个分类器, 分别依靠提交中的自然语言信息与代码修改信息进行安全判断. 对于提交日志信息, 其使用词袋模型对其进行编码, 使用支持向量机进行分类; 对于代码修改信息, 其将源代码修改视为自然语言中的文本撰写, 因此直接使用标准的文本分类方法对其进行操作. 其只使用代码中的标识符名称序列来表示代码, 同样使用词袋模型加以表示, 并使用支持向量机进行分类. 实验结果显示: 该方法对安全相关提交的分类精确度有所提高, 但 *F*1 值为 0.64, 还有较大空间.

Wang 等人[64]提出了一种自动的数据挖掘与标注流程. 该数据集从 Sard、NVD 及 GitHub 中挖掘缺陷相关提交(commit). 对于 GitHub 中的无标签数据, 其首先挑选高星值的项目爬取只修改单个文件的提交, 然后使用 5 种机器学习分类器进行共同投票, 包括 SVM、LR、KNN、RF、GB, 以判断该提交是否引入了缺陷. 这些分类器是用 3 000 个人工标记的提交训练的. 为了保证分类器判断的准确性, 5 个分类器投票结果高度一致的提交才会被保留. 最终, 这些提交会按照函数级进行分割. 该数据集最终获得了 150 950 条函数级的代码样本, 语言涵盖 C、Java、PHP 和 Swift, CWE 缺陷类型达到 30 种. 该数据集已开源(https://github.com/HuantWang/FUNDED_NISL).

上述部分研究使用了机器学习方法进行自动数据挖掘, 鉴于使用深度学习进行文本分类已经取得了较好的效果, 使用深度学习对代码库中的提交进行自动判断也是一个值得探究的研究点.

此外, 也有通过人工方式从开源代码库中挖掘缺陷数据.

Zhou 等人[65]为了获得更加精确标注的缺陷代码数据集, 使用了纯人工标注的方式. 他们雇佣了一批安全人员对开源 C/C++项目 Linux Kernel、QEMU、Wireshard 和 FFmpeg 进行标注. 首先对关键词初筛后的安全相关提交(commit)进行人工判断是否与缺陷修复相关, 然后将这些提交分割为函数级别的代码段. 该数据集共使用 4 名安全研究员, 耗时 600 工时来完成两轮的标注与交叉验证. 最终, 该数据集共收纳 58 965 个样本函数, 其中, 缺陷相关函数为 27 652 条, 占比 46.9%, 正负样本相对均衡. 该数据集目前开源了 2 个工程(FFmpeg 与 Qemu).

Cheng 等人[66]构建了一个混合来源的缺陷代码数据集, 其从 Sard 缺陷库与 2 个开源软件(lua、redis)中抽取缺陷条目. 对于 Sard 数据集, 其只选取最常见缺陷类型 TOP 10 的条目. 对于开源项目, 其通过管检测检索安全相关提交后再使用人工审查进行打标, 只选取缺陷修复提交前后的版本作为样本, 该标注工作花费 3 人共 720 小时. 在收集到函数级与文件级数据后, 根据系统 API 调用作为起点进行前后向切片生成. 切片标签由其是否包含缺陷相关语句决定, 即: 若切片包含至少一条缺陷语句, 则将其标记为“缺陷”. 在开源数据中, “缺陷语句”是根据修复提交的语句操作来判断的. 为了方便与函数级缺陷检测模型进行对比, 所有数据还按照函数级进行了上述标记. 最终, 该数据集包含 140 670 个切片, 其中, 缺陷切片为 44 521 个, 占比 31.6%. 此外, 该数据集中 Sard 部分占比较大, 为 98.3%. 该数据集已开源(https://github.com/DeepWukong/DeepWukong).

相对于自动方法, 通过人工方式挖掘需要大量的人力成本, 且其标注效率有限.

总的来说, 基于开源代码库爬取的方法可以得到大量的备选缺陷样本, 但是爬取出的样本是否为缺陷以及是哪种类型的缺陷都较难判断: 自动化的方法正确率难以保障, 而人工判断的方法则在效率上制约了数据集的大小. 此外, 由于爬取代码库时大多采用启发式的方法, 其爬取的缺陷样本在形态上具有不确定性: 既可能包含具有 CVE 编号的真实、复杂缺陷, 也可能包含项目普通缺陷甚至简单编码错误. 这种缺陷样本的多

样性, 会对深度模型的设计提出较高的要求.

2.1.4 基于静态分析工具生成

还有一种以相对低的成本快速获取大量缺陷数据的方法, 即利用静态分析工具对代码进行缺陷检测, 从而得到缺陷样本.

Russell 等人[67]从 Juliet、Debian Linux 和 GitHub 中选择了共 12 874 380 个 C/C++函数段, 经过去重筛选后, 对无标签的函数段(Juliet 带有标签)使用 3 种开源静态分析工具 Clang、Cppchecke 和 Flawfinder 进行缺陷检测. 这 3 种分析工具拥有不同的检测粒度, 通过投票的方式来共同对代码段进行是否含有缺陷的标注. 最后, 通过人工的方式将分析工具报出的缺陷与 CWE 类型一一匹配对应. 最终, 该函数级缺陷代码数据集包含 1 286 262 条样本, 其中, 缺陷样本为 87 804 条, 占比 6.8%, 其类型对应到 CWE 中有 149 个类型. 不过, 该数据集并未开源.

Dam 等人[68]构建了一个文件级的缺陷代码数据集. 他们使用静态分析工具(由于保密协议未透露)对开源项目 Tizen 进行了检测, 并只选取资源泄露相关警告. 该数据集基于 C 语言, 最终包含 8 118 个源文件, 其中, 2 887 个为有缺陷文件, 达到 35.6%. 不过, 该数据集并未开源.

总的来说, 使用静态分析工具可以利用其分析能力生成较大数据量的标注数据. 但是静态分析工具判别能力有限, 其误报率高等缺陷在实践中已被广泛证实, 因此, 使用静态分析工具得到的判别结果作为标准答案在可靠性上没有保证. 此外, 使用静态分析工具构造样本, 其数据集的表征能力上界即为分析工具的检测能力, 在此数据集上训练出的模型, 其效果只会低于直接使用分析工具, 因此, 使用深度模型意义不大. 而且, 由于当前静态分析工具大多关注相对(CVE)简单的缺陷类型和代码编码规范, 因此由其构造的数据集价值相对于真实工业缺陷来说偏低.

2.2 当前数据集构造方法存在的问题

为了便于对比, 我们对第 2.1 节中梳理的数据集构造方法进行列表展示. 对于每一种数据来源, 挑选至多 3 项代表性工作, 详细列举其各项统计信息, 包括其缺陷数据来源、拥有的缺陷类型(其中, “–”表示未区分类型)、数据样本的粒度、数据集总样本数、缺陷样本数、缺陷样本占比以及数据集是否开源.

统计结果见表 1, 可以看到: 当前的缺陷代码数据集在各项技术环节均存在较大的差异, 如不同的语言、粒度等. 对于从代码库爬取的缺陷数据集, 普遍存在没有缺陷类型这一问题. 对于当前绝大多数数据集, 均存在正负样本不均衡的问题. 有接近半数的数据集未进行开源, 给其他研究者进行模型复现与横向对比带来了困难.

表 1 数据集统计信息小结

数据来源	文献	语言	缺陷类型数	样本粒度	总样本数	缺陷样本数	缺陷样本比例(%)	是否开源
人工构造	[31]	C/C++	2	函数	28 049	10 561	37.7	否
	[35]	Java	7	文件	1 698	932	54.9	是
	[37]	JavaScript	3	代码片段	16 634 458	8 317 229	50.0	是
缺陷库抽取	[42]	C/C++	2	切片	61 638	17 725	28.8	是
	[43]	C/C++	40	切片	181 641	43 119	23.7	是
	[53]	Java	–	文件	3 290	1 152	35.0	否
代码库爬取	[61]	Java	–	函数	4 973 000	1 824 000	36.7	是
	[62]	Go 等 6 种	–	提交	12 409	1 303	10.5	否
	[65]	C/C++	–	函数	58 965	27 652	46.9	部分
分析工具生成	[67]	C/C++	149	函数	1 286 262	87 804	6.8	否
	[68]	C	–	文件	8 118	2 887	35.6	否

此外, 我们在表 2 中按照每个技术环节的技术路线列举了至多 3 项代表性的工作, 并评估了每一类技术路线的优缺点.

从数据来源上看, 不同的数据来源意味着其样本复杂程度的不同. 例如, 在缺陷库抽取中广泛使用的 NVD 抽取数据与人工构造中广泛使用的 Sard 测试集数据: 前者来源于真实的工业代码, 其复杂程度较高, 且

模块化程度高, 各组件之间的依赖关系与调用关系复杂; 后者来源于人工撰写的测试用例, 其复杂程度低, 且主要针对缺陷的表示, 而大多忽略其真实的功能, 因此为独立程序片段, 几乎没有组件之间的长程依赖与复杂的调用关系. 从实际应用的角度看, 缺陷检测系统需要具备对真实工业代码的检测能力, 即理想的缺陷代码数据集应该来源于 NVD 等工业代码而非人工撰写的测试用例. 但工业代码缺陷数据的获取难度与成本远大于测试用例, 需要更加完善的样本提取技术与大量不可避免的人工审查.

从数据粒度上看, 不同的数据粒度在完整度与冗余度上存在差距. 文件级数据可以包含较多路径, 但不可避免地带来大量的无关噪声. 函数级数据在划分时成本极低, 且在检测过程中不需要对被测代码进行分析(切割为函数即可). 然而, 函数级数据基于的朴素假设是: 缺陷发生在函数的范围内. 这在大规模工业代码中显然是难以支持的, 特别是在模块化的情况下, 数据的声明与使用往往不存在于单一函数中. 切片级数据可以支持跨函数的缺陷路径, 但切片操作需要在构建数据集与检测被测代码时均对代码进行数据流、控制流等分析, 对资源的需求较大, 时间成本也较高.

从标注方式上看, 不同的标注方式使得标签的精度与获取成本存在差异. 安全相关人员的人工标注能够保证标签的质量, 但其获取成本较高, 且标注速度较慢, 达到深度学习模型需要的数据规模需要长期的积累. 使用启发式方法自动化地进行标注可以快速、大量地生成标注, 然而其标注精度难以达到理想的标注效果.

表 2 数据集构造方法小结

技术环节	技术路线	文献	优点	缺点
数据来源	人工构造	[31,35,37]	成本低, 数量大	构造规则单一, 复杂度低
	缺陷库抽取	[42,43,53]	复杂度高, 贴近工业场景, 人工确认	成本高, 可挖掘数量少
	代码库爬取	[61,62,65]	复杂度高, 贴近工业场景	成本高, 缺陷类型缺失
	分析工具生成	[67,68]	成本低, 数量大	受限于工具精度
样本粒度	文件	[35,53,68]	天然划分, 成本低	粒度较粗, 冗余信息较多
	函数	[31,61,65]	天然划分, 成本低	不支持跨函数缺陷, 冗余信息多
	切片	[42,43,66]	支持跨函数缺陷, 冗余信息少	计算资源开销较高, 速度较慢
	其他	[37,55,62]	–	–
标签来源	人工标注	[62,65,66]	标注精度高	成本极高, 速度慢
	自动标注	[42,43,53]	成本低, 速度快	标注精度低

下面我们对一些具体问题进行阐述.

2.2.1 不包含完整缺陷路径

许多数据集[31−34,41,45,64−67]是函数级数据集, 或者是先选定函数再在其内部进行切片操作. 但是对于函数的选择往往是根据缺陷补丁位置(diff 修改的函数)来决定的, 而补丁位置并不一定与缺陷在同一函数中, 这样会导致缺陷样本不包含完整的触发路径. 例如图 2-2 所示, 该代码示意图包含 2 个函数 *A* 和 *B*. 缺陷在函数 *B* 被触发, 但是由于缺陷修复被写在了靠前的函数 *A*, 即与函数 *B* 的触发位置不在同一个函数中, 所以最终触发缺陷的语句不包含在数据集中(只包含 diff 所在的函数). 因此, 若将函数 *A* 作为缺陷样本, 则其不是一个完整的缺陷路径. 这种情况在模块化较为显著的大型项目中更为突出.

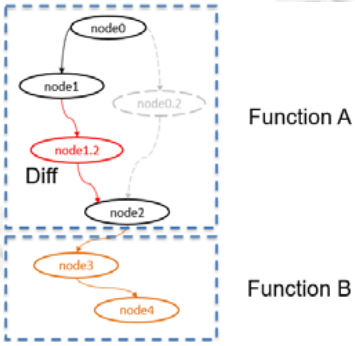

图 2-2 不包含完整缺陷路径的代码示意图

若缺陷样本不包含完整的缺陷流程，特别是不包含最终的缺陷触发位置，则在此基础上进行的特征挖掘将失去意义. 因此，当抽取缺陷样本时，应该考虑到跨函数的问题，而不应直接将代码范围限定在函数内部.

2.2.2 正负样本设置不合理

观察一些数据集的构造方法我们发现，一些数据集在构造正样本(缺陷样本)与负样本(非缺陷样本)时不是针对同一个操作. 例如[42]: 将 diff 所在的函数标记为缺陷函数，而其余的未被 diff 影响的函数标记为非缺陷函数. 事实上，每个函数的功能是不同的，其包含的操作也不相同. 例如: 对于 CWE-119 类型下的缺陷函数，其往往是负责处理缓冲区事务，因此包含较多缓冲区操作，而其他函数并不一定有(且大概率没有)如此多缓冲区操作. 因此，将 CWE-119 类型缺陷函数以外的函数标为非缺陷函数，可能会使得模型退化成对是否含有缓冲区操作或是否含有较多缓冲区操作的识别，而不是对缓冲区操作是否正确的识别. 因此，有必要将正负样本限定在相同的操作行为中. 例如，使用修改前后的代码作为正负样本，就可以避免这类问题.

2.2.3 横向比较问题

当前，基于深度学习的缺陷检测任务分为不同的粒度，如文件级缺陷检测、函数级缺陷检测、切片级缺陷检测. 对于针对不同粒度的检测系统，存在无法横向比较的问题，原因是不同粒度的标答模式不好界定. 如图 2-2 所示，若缺陷路径由函数 *A* 到函数 *B*，那么对于函数级缺陷检测，需要单独对函数 *A* 与函数 *B* 标注，此时对两者标注“缺陷”或“非缺陷”都是有争议的.

3 基于深度学习的源代码缺陷检测模型

如果将代码简单认为是一个文本符号序列，则使用深度学习模型对代码段进行缺陷判断类似于文本分类问题. 但是由于代码本身的特性与缺陷检测任务的特性，其中包含许多技术难点. 因此，本节将列举一些主要的技术难点，介绍当前针对这些难点的研究方法.

3.1 代码表征

由于代码具备各个维度的特征，例如局部的文本共现特征与长程的数据、控制依赖特征，因此，如何对代码进行表征成为了设计深度学习模型需首要考虑的问题. 当前的方法主要通过序列、树和图对代码进行表征，其常用的对应网络结构如图 3-1 所示.

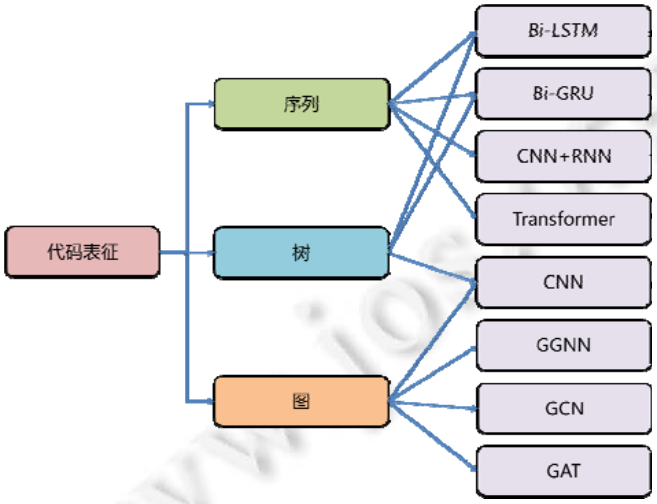

图 3-1 代码表征的分类与应用模型

下面将依次对其相关研究进行介绍.

3.1.1 基于序列的表征

由于深度学习在自然语言处理上的成功，研究者很容易想到将其方法应用到代码语言上，即将代码通过

序列的方式进行表征.

针对现有漏洞检测系统依赖人类专家定义特征、经常出现高漏报的问题, Li 等人[42]提出了 VulDeePecker, 这是一个基于深度学习的漏洞检测系统. 在输入上, 其使用代码片段作为输入粒度, 因此对于输入代码均需检测危险函数调用作为入口并进行切片组合. VulDeePecker 选择了循环神经网络(RNN)模型, 并使用双向长短期记忆(bi-LSTM)来解决梯度消失与前后向依赖问题, 因为程序函数调用的参数可能会受到较早或较晚语句的影响, 其使用 Word2Vec 将输入的代码片段转化为向量表示投入 Bi-LSTM 网络进行训练. Bi-LSTM 网络并未作特殊设计. 实验结果表明: VulDeePecker 总体检测的 *F*1 值达到 90.5, 显著高于使用传统方法(基于模式与基于代码相似度)的缺陷检测系统, 并且可以大幅降低误报率. 不过, 该模型并未开源.

由于 VulDeePecker 只使用了 Bi-LSTM, Li 等人[44]进一步测试了多种模型构架, 包括 GRU、DBN、MLP 等. 其最终确认双向门控循环单元(bi-GRU)拥有更好的检测效果. 基于 Bi-GRU 的 SySeVR 模型在 4 个真实的工程(Libav、Seamonkey、Thunderbird、Xen)上检测出 15 个缺陷, 其中, 7 个为尚未发觉的缺陷. 不过, 该模型并未开源.

Xu 等人[30]首先对程序根据调用情况进行切片以保留相关语句, 然后通过正规化去除命名影响, 最后将其转化为向量形式作为模型输入. 在分类模型上, 其将 CNN 与 RNN 结合成一个网络, 称为“Contextual LSTM”. 具体来说, 其首先通过 CNN 来挖掘局部特征, 然后投入 LSTM 来挖掘前后的依赖特征. 实验结果显示: CLSTM 在 Sard 数据集上, *F*1 可高达 0.97, 高于对比的 CNN 与 LSTM 模型, 但提高不显著.

借鉴自然语言处理中的情感分析任务, Russell 等人[67]设计了一个基于表示学习的源代码分类器. 其首先将源代码按照词法进行嵌入表示, 在特征提取阶段, 通过多层的卷积来挖掘局部特征或通过循环迭代来捕捉前后的依赖关系. 随后, 通过池化层和全连接层得到最终的函数表示. 由于表示学习已经学习到了函数的特征, 因此最后对函数的缺陷检测就可以通过将其表示直接投入随机森林分类器即可完成. 实验结果显示: 使用表示学习连接随机森林分类器的方法比单纯使用词袋模型连接随机森林的效果提高较多, 其检测效果也显著高于选择的 3 种静态分析工具(Clang/Flawfinder/Cppchecker). 不过, 该模型并未开源.

Choi 等人[39]借鉴了自动问答领域的模式来判断缓冲区调用是否安全. 其使用了记忆网络的框架, 将缓冲区调用语句作为查询语句, 将缓冲区相关的初始化与分配语句作为需要查询的相关内容, 最终帮助模型进行安全性的判断. 具体来说, 其首先将代码按照词进行嵌入编码, 将待查询的上下文语句通过矩阵表示为记忆值模块与记忆地址模块, 再通过注意力机制, 使用查询(缓冲区调用)语句得到相关记忆语句的排序, 最终通过检索出的相关语句进行访问操作是否安全的判断. 实验结果显示: 该模型在 4 种复杂度级别的缓冲区操作的判断中均高于传统的 CNN 与 LSTM 模型, 在复杂度较高的情况下优势更加显著. 不过, 该模型并未开源.

事实上, 代码中包含了许多自然语言上的信息, 例如变量和函数的命名等. 这些信息可以指示该变量或函数的功能与属性, 这种提示信息在拥有标准命名规则的大型工程中尤为明显. Pradel 等人[37]就利用了这样的信息, 依靠深度学习, 仅仅针对命名信息进行缺陷检测. 其通过 Word2Vec 对命名进行了嵌入式的向量表示, 以便模型挖掘其语义上的信息. 在分类模型上, 其选用了最基础的前馈神经网络, 没有进行特殊的设计. 实验结果显示, DeepBugs 模型在数据集上的检测准确率超过 89%. 在针对真实代码的检测中, 人工审查其 150 个缺陷报告后, 确认其中 102 个为正确的缺陷报出, 判断精度达到 68%, 证明了即使使用人工构造的缺陷数据集也可以赋予模型挖掘真实缺陷的能力. 在效率上, 其单个文件的检测时常不超过 20 ms, 也显示了其可用性. 但是, 该模型的挖掘能力局限于生成缺陷代码的变异模式, 而不具备较好的扩展性. 该模型现已开源 (https://github.com/michaelpradel/DeepBugs).

Saccente 等人[33]提出了原型工具 Project Achilles, 其使用 LSTM 网络对 Java 代码作函数级缺陷检测. 不同于其他方法使用单一模型对待测样本进行判断, Project Achilles 针对其训练数据中的每一个类型都训练了一个基于 LSTM 的分类器, 因此会对待测样本是否是某个类型的缺陷都进行一次评估. 这样, Project Achilles 不仅可以判断待测样本是否有缺陷, 同时也能直接得到该缺陷的类型. Project Achilles 按 token 序列来读入函数代码, 在 LSTM 模型上, 其并未作特殊修改. 尽管一些类型只有 100 多个文件的数据作为训练样本, Project

Achilles 仍然在 29 个 CWE 类型中的 24 个上达到了超过 90%的准确率. 该模型已开源(https://gitlab.com/TUSoftwareEngineering/vulnerabilitylocalization-using-machine-learning).

除了传统的分类模型, 有些研究还尝试了新的模型架构, 例如使用 seq2seq 的方法进行缺陷检测.

为了只使用缺陷代码和其对应的修改后代码进行缺陷检测, Grag 等人[69]提出了一个基于机器翻译模型的检测系统. 其将缺陷代码视为源语言, 将其修复版本视为目标语言, 通过基于 RNN 的编码器与解码器来学习其中的差异与转化方式. 同时, 还使用修复代码作为源语言, 同一修复代码作为目标语言, 来使模型不对正确的代码做转化. 最终在检测时, 若翻译模型未对输入代码进行修改, 则认定其为无缺陷的正确代码. 实验结果显示: 该方法在无噪声和有噪声的情况下检测效果均好于传统特征挖掘模型, 在有噪声条件下效果更显著. 不过, 其未与深度学习模型进行比较, 也未开源.

传统的缺陷检测方法针对不同的缺陷类型预设了不同的缺陷模式, 通过匹配的方式, 在判断代码是否含有缺陷的同时, 也可以得到其所属的缺陷类型信息. 而基于深度学习的缺陷检测系统往往只对代码做二分类, 即判断其是否含有缺陷, 而无法提供其缺陷类型, 这给开发人员带来了一些不便因素.

为此, Zou 等人[43]提出了第一个多分类的深度学习缺陷检测系统μVulDeePecker. 该系统定义了代码注意力, 即: 在输入的代码片段中选定符合缺陷语法特征规则的语句子集, 从而更好地挖掘缺陷的局部特征. 该缺陷语法特征规则是通过人工总结得到的, 包括库/API 函数调用中参数的定义语句、控制语句及包含库/API 函数调用的语句. 其将代码片段与对应的代码注意力片段分别编码并向量化, 同时作为该段代码的表征投入 Bi-LSTM 进行多分类任务的训练. 实验结果显示: μVulDeePecker 模型在多分类任务上的效果好于单独将 VulDeePecker 模型修改为多分类的版本, 在个别类型上效果尤为显著. 引入的控制依赖对模型有较大的性能提升. 该模型未开源.

近年来, 大规模预训练语言模型(如 BERT[70])在自然语言领域展现出的优良效果为程序语言研究者们提供了一种新的程序表示思路, 人们也开始探究基于 Transformer 在程序语言上构建预训练模型, 其中部分研究将缺陷检测作为下游任务进行了初步的探究.

考虑到相关人员编写代码时会通过附加自然语言的注解与其他人员沟通, 一些研究者试图利用自然语言来帮助更好地挖掘与建模程序语言. Feng 等人[71]提出了 CodeBERT, 该预训练模型使用程序语言与自然语言共同预训练. 其使用了两个目标任务进行训练, 分别为掩码语言建模和替换令牌检测. 为了测试代码预训练模型在非训练任务的可用性, Zhou 等人[72]将 CodeBERT 在实时缺陷检测任务上进行了实验(值得说明的是: 实时缺陷检测任务同时将提交代码与提交的自然语言信息作为输入来判断该提交是否引入了缺陷, 与传统的缺陷检测任务只输入代码存在一定差异). 实验结果表明, 简单地将 CodeBERT 替换原模型的编码器即可接近 SOTA 的性能, 这表明了代码预训练模型的语义捕获能力没有局限在其训练任务中. 该实验已开源(https://github.com/Xin-Zhou-smu/Assessing-generalizability-of-CodeBERT).

不过, 实时缺陷检测任务虽然与源代码缺陷检测任务同为判断代码是否有缺陷, 但两者的输入存在差异. 因此, 代码预训练语言模型在缺陷检测任务上的效果还需要进一步地实验证实.

Ahmad 等人[73]提出了一种能够执行广泛的程序-自然语言的理解-生成任务(program and language understanding and generation)的预训练模型 PLBART, 其通过去噪自动编码对大量 Java 和 Python 函数以及相关的自然语言文本进行了预训练. 具体来说, 其在 StackOverflow 上抓取大量问题、答案与代码段, 投入与 BART[74]相同的结构, 并引入了 3 种噪声: 令牌掩码、令牌删除与令牌填充使得模型能够推理语言语法和语义, 并同时学习连贯地生成语言. 为了检测 PLBART 对未见代码语言的理解能力, 其在基于 C/C++语言的缺陷检测任务上进行了测试. 但是, 只对比了其他的语言模型, 并未与针对缺陷检测任务的深度学习模型进行比较. Ahmad 等人表示: 当前, 基于图的缺陷检测模型效果最好, 使用缺陷检测任务只是用来探究预训练语言模型在未见任务与未见语言上对程序语义的性能. 该模型已开源(https://github.com/wasiahmad/PLBART).

由此可见, 预训练代码语言模型在挖掘程序语义上具备一定的潜力, 但当前的研究还未统一在源代码缺陷检测的场景下横向对比深度学习模型与预训练语言模型的性能差异, 未来还需要进一步地深入探究. 总的

来说, 基于序列的表征较为直观, 且不需要对待测代码进行额外的分析处理, 在实现上较为简便. 但是不同于自然语言, 代码语言具有更强的结构性与局部性, 其上下文的依赖关系更为复杂且距离更长, 因此, 单纯地使用序列作为代码表征会损失大量的代码结构特征. 由于基于序列的表征存在的这类问题, 后期研究大多不再使用序列作为表征方式.

3.1.2 基于树的表征

在对代码进行分析时, 除了以序列的形式以外, 最常用的就是抽象语法树(AST)的形式.

Li 等人[53]提出了基于卷积神经网络的 DP-CNN 模型. 该模型针对文件级缺陷检测, 并将抽象语法树作为代码原始表征以获取更多的结构信息. 其首先将文件代码解析为抽象语法树, 然后从中选择代表性节点构成表示该文件的向量, 转化为词嵌入形式后投入卷积神经网络模型进行特征的学习, 最终学习得到的表示特征与一些传统特征混合投入逻辑回归分类器进行分类. 在 DP-CNN 系统中, 其选择的代表性节点是人工设定的, 包括方法调用和类实例创建的节点、声明节点与控制流节点. 为了解决缺陷样本少的问题, DP-CNN 在训练阶段会重复使用缺陷样本. 实验结果显示: 该方法在其数据集上的检测高于使用深度置信网络的方法, 但整体检测性能有限, $F1$ 值仅为 0.6. 该模型未开源.

Lin 等人[41]使用了树结构作为代码的表征. 具体地, 其使用代码解析工具 CodeSensor 将函数代码转化为 AST, 并使用深度优先遍历将其转化为序列表示; 随后进行截断和补零使其成为等长语句; 最后, 使用 Word2Vec 将其转化为向量形式. 在模型上, 该方法使用 Bi-LSTM 模型来捕获缺陷特征, 模型结构未作特殊修改. 在训练完成后, 将具有少量标签的目标代码输入训练好的模型以获得其表示并判断缺陷. 实验结果显示, 使用表示学习的方法在 TOP 10 缺陷函数的检测效果显著优于基于代码度量的方法. 但是, 该模型的整体误报率较高, 使其可用性不佳.

Dam 等人[68]使用了基于树结构的 LSTM 网络来对通过 AST 表征的代码进行缺陷检测, 具体地, 该模型首先将一个源文件清洗并解析为抽象语法树, 每一个节点按照其类型名称进行嵌入式表示, 将每一个节点都投入 LSTM 单元从而表征整个代码文件, 最终通过传统分类器(逻辑回归与随机森林)对整个源文件的 AST 特征向量表示进行缺陷判断. 实验结果显示: 该模型在工程内的数据集下的缺陷检测 $F1$ 值达到 0.9 以上, 但在跨工程的数据集下 $F1$ 值只有 0.5.

针对现有方法无法较好地处理跨函数缺陷的问题, Li 等人[61]提出了一种结合使用上下文和注意力神经网络的方法. 其首先基于 AST 对函数代码进行向量表征作为局部上下文, 再对 AST 上的路径节点使用程序依赖图和数据流作为全局上下文, 以将待测函数与可能导致错误代码的其他相关函数连接起来. 通过增加对上下文的表示, 减少因为局部代码相似度而导致的误报. 同时, 为了避免上下文的引入使得代码相似度匹配过于严苛, 其使用注意力机制来增加缺陷路径的权重, 从而保证召回率. 在与基于规则的检测工具和基于挖掘的方法比较的过程中, 该方法均大幅优于对比方法, 但整体效果有限: 在难度相对低的项目内检测中, $F1$ 值为 0.64. 该模型已开源(https://github.com/OOPSLA-2019-BugDetection/OOPSLA-2019-BugDetection).

在对代码进行表示时, 传统方法会使用定长向量, 这就需要对代码内容进行截断(代码过长)或填充(代码过短), 这会带来信息的丢失或冗余. Feng 等人[34]提出了一种基于抽象语法树的数据处理方法来提取所有句法特征并减少数据冗余, 其在双向门控循环单元(bi-GRU)网络上应用包填充(pack-padded)方法来训练可变长度数据而无需对变量截断和填充. 具体来说, 其首先将程序源代码解析为 AST, 并按照函数节点进行切割, 得到按照函数分割的 AST. 然后遍历整个 AST, 并将所有用户定义的名称映射到固定的预设名称模式, 以消除不同命名带来的差异. 为了能够作为深度神经网络的输入, 其通过前序遍历将 AST 转换为节点序列, 并使用 Word2Vec 将节点序列映射为向量表示. 在网络模型上, 该方法使用双向门控循环单元来应对函数级表示带来的长向量. 其在模型结构上没有特殊改动, 所不同的是, 其使用了包填充方法来应对可变长度向量, 即针对不同的输入长度进行记录并同时输入模型, 而不需要将向量长度统一化. 实验结果显示, 该模型在 3 种数据集划分下均达到了 0.82 以上的 $F1$ 值. 相对于开源分析工具 Rats 和 Flawfinder, 该模型具有更高的准确率, 同时误报率也更低. 不过, 该模型未开源.

总的来说，相较于基于序列的表征方式，基于树的表征可以更好地显式表征代码中的结构信息，在传统缺陷检测方法中也大多会选用抽象语法树作为分析对象．但事实上，除了抽象语法树能够表示的结构信息以外，源代码本身还具备数据流、控制流等多种不同维度的特征，而这些特征在缺陷检测的过程中恰恰是较为关键的部分．因此，单纯地使用抽象语法树作为代码表征是不够的．

3.1.3 基于图的表征

基于序列或树的缺陷检测在学习全面的程序语义以表征真实源代码漏洞高度的多样性和复杂性方面具有很大的局限性．程序语言不同于自然语言，其更具有结构性与层次性，并且具有抽象语法树、数据流、控制流等多种不同维度的表示模式．根据缺陷类型的不同，需要考虑的维度也不同，单纯地使用平面序列或语法树来表征代码会严重限制模型覆盖各种缺陷的能力．

由于图神经网络与代码的各种属性图可以较好地适配，因此逐渐成为人们在缺陷检测任务上的热点模型．其中，如何构建图以建模表征代码是最关键的探究点．

在代码场景中，缺陷代码和非缺陷代码有时相似度较高，例如仅在单个保护数值或边界条件上有细微差异，而在其他操作上完全一致．这类细小的差异往往难以被模型捕获．针对这个问题，Duan 等人[31]提出了 VulSniper 模型，其使用 attention 机制来捕获代码的关键部位．具体来说，其首先从源代码生成程序属性图(code property graph)以尽可能地保留更多信息，随后将程序属性图转化为一个 144 维的特征向量作为模型输入，并通过 attention 机制来计算不同节点的权重，最后通过全连接层进行二分类判断代码是否有缺陷．其中，attention 部分分别使用多个一维卷积和一维卷积的转置来实现自底向上和自顶向下的结构．通过一维卷积，感受野逐渐扩大以获得周围和全局信息．通过一维卷积的转置，将高级特征缩放到与输入相同的大小，以便将注意力权重应用于输入．VulSniper 针对的是函数级的缺陷检测任务，其在 Sard 数据集中缓冲区错误(CWE-119)和资源管理(CWE-399)这两个缺陷类型上的 $F1$ 值达到了 80.6%和 73.3%，远超传统方法．不过，该模型并未开源．

Zhou 等人[65]提出的 Devign 模型基于 AST 来表征代码，并将不同层级的控制流、数据依赖、自然代码序列显式编码为异构边的联合图，每种类型表示与相应表示相关的连接．这种综合表示方法有助于捕获尽可能广泛的漏洞类型和模式，并能够通过图神经网络学习更好的节点表示．在模型上，Devign 使用带有 Conv 模块的门控图神经网络．有别于使用全部节点进行分类，Conv 模块可以在学习到的丰富节点表示中提取有用的特征用于图级分类．实验结果显示: Devign 在缺陷检测任务的效果上显著高于之前方法；同时，其在 112 个真实项目的缺陷函数中检测准确率达到 74%，显示了其在现实应用上的可能性．不过，该模型并未开源．

Feng 等人[75]同样使用图来表征程序代码，并通过 GNN 来对其进行分类，并探究了不同模块的不同设置带来的影响．其测试使用了不同属性图表征(AST、CFG、CPG)、不同编码方式(Bag-Of-Words、Word2Vec、随机)和不同 GNN 学习方法(DiffPool、Set2Set、DGCNN)．实验初步结果显示: AST+Word2Vec+DiffPool 是当前效果最好的组合，相较于静态分析工具(Cppchecker、Clang、Flawfinder)有较大的性能优势．

Ghaffarian 等人[35]设计了一种定制的程序中间图表示，其使用抽象语法树(abstract syntax tree, AST)、控制流图(control flow graph, CFG)和程序依赖图(program dependence graph, PDG)作为信息来源．在图的基础上，对节点和边的文本信息(变量名、类型)使用 TF-IDF 进行向量化．在图模型的选择上，其选用了图卷积模型(graph convolutional network, GCN)与图注意力模型(graph attention network, GAT)，其在网络上没有作特殊修改．实验结果显示: 图卷积模型在 7 个缺陷类型上的 $F1$ 值都显著高于其他方法，但图注意力模型在跨项目检测上的效果更好．同时，相关实验还指出: 对程序进行表征的向量程度不应过小，否则会影响图神经网络的性能．与一些研究会对代码进行规范化处理(替代变量名等)不同，该研究通过实验特意指出，规范化操作会影响模型辨别效果．不过，该研究的代码并未开源．

传统方法使用序列或无类型图来表征代码作为输入，这会丢失很多代码结构上的控制与依赖信息．Wang 等人[64]提出了 FUNDED，它在抽象语法树的基础上，人工定义了 8 种关系，包括数据依赖关系、控制依赖关系、守护关系、跳转关系、运算关系、序列邻接关系、最后使用关系、最后词法使用关系，以将抽象语法树

扩增为一个有向多图. FUNDED 使用这 9 种关系(本身抽象语法树的关系)得到的邻接矩阵来表征整个函数代码段, 从而使模型能够获得更多的语义、控制等信息. 该模型使用门控图神经网络(GGNN)对表征函数的邻接矩阵进行分类, 其在自建数据集上获得了较好的效果, 在真实项目的检测测试上性能也超过了之前的模型. 该模型已开源(https://github.com/HuantWang/FUNDED_NISL).

Cheng 等人[66]也利用图来对代码片段进行表征, 其提出了 DeepWukong. 具体地, 其首先生成代码的控制依赖图和数据依赖图, 使用分析工具 SVF 从其中找到系统 API 调用作为关键节点并抽取切片, 使用 Doc2Vec 将清洗后的代码转化为向量形式, 将得到的切片图向量表示投入 3 种图卷积网络(GCN、GAT、*k*-GNNs)进行分类. 通过图的形式, 既保留了结构信息(边)也保留了文本信息(节点向量). 在人工数据和真实数据的部分实验结果显示: 不论使用哪种图卷积网络, DeepWukong 都能拥有较好的辨别性能. 该模型已开源(https://github.com/DeepWukong/DeepWukong).

总的来说, 基于图的表征相对于序列或树能够显式表征更多维度的代码特征, 当前被认为是更具前景的代码表征模式. 但基于图的表征也存在一些问题, 例如在基于图的表征模式下, 相同节点往往会被合并, 代码语言的自然顺序也会丢失, 这类信息丢失问题都是在构建基于图的表征时需要考虑的.

3.1.4 基于其他表征

除了采用序列、树或图的方式来对代码进行表征以外, 对代码进行其他形式的转化也是一种思路.

Cao 等人[32]提出了带有傅里叶变换的深度卷积 LSTM 神经网络 FTCLNet 用于漏洞检测. 具体地, 其使用离散傅里叶变换方法将代码空间转换为频域, 并将卷积神经网络与长短期记忆网络结合起来, 以捕捉频域上的局部和全局特征, 再通过反向傅里叶变化将其转化回代码空间, 同时加入注意力机制来调整权重, 最后使用全连接层进行预测. 实验结果显示: FTCLNet 的检测效果高于 VulDeePecker 等深度方法, 并显著高于 Flawfinder 等检测工具.

对源代码进行分析, 涉及到路径可达等问题, 这对模型的求解能力提出了很高的要求, 也影响了模型的性能. 一个解决方案是通过编译, 借助编译器将模型转化为汇编代码, 再交给深度学习进行后续的缺陷判断. Pechenkin 等人[76]使用双向 LSTM 对向量化的汇编代码进行缺陷判别. 汇编代码的操作更加细化, 但同时会导致更大的代码量, 使得对长程依赖的代码结构更加难以捕捉.

一些语言特性会导致直接对源代码进行缺陷检测较为困难, 如 C/C++中宏的跨文件使用. 这时, 利用编译器对源代码进行预处理可以减缓这些问题. Li 等人[77]提出了 VulDeeLocator, 其将源代码转化为中间代码再进行缺陷检测, 并使用粒度细化来缩小定位的范围. VulDeeLocator 读入中间代码的向量表征与定位缺陷的矩阵表示, 其在标准的 Bi-GRU 模型中增加了 3 层: 通过乘法层完成注意力获取; 通过 *k*-max 池化层与平均池化层来完成粒度细化. 在 200 个随机挑选的真实工程文件中, VulDeeLocator 检测出了 18 个缺陷, 其中 16 个为已报出缺陷, 2 个为静默修复缺陷. 不过, 该模型并未开源. 此外, 将代码转化为中间语言需要对代码进行编译, 这在大规模程序的应用场景下成本较高, 其易用性低于直接针对源代码进行操作. 该模型已开源(https://github.com/VulDeeLocator/VulDeeLocator).

3.2 可解释性

虽然在实验结果上, 基于深度学习的缺陷检测已经展现了较好的效果, 甚至能够超过一些传统的检测方法, 但是深度学习方法只能提供结果, 而不能提供解释性的指导. 相比之下, 基于领域专家编写的漏洞检测规则得到的检测结果能够更好地提供这些解释. 此外, 解释性的缺失也使得从业人员无法判断基于深度学习的缺陷检测器学到了什么知识. 因此, 深度学习可解释性是一个重要的研究课题, 它可以帮助人类深入了解软件漏洞的起因、检测, 并对缺陷的确认与修复提供帮助.

An 等人[78]认为: 将函数或文件作为缺陷粒度过于宽泛, 其包含了过多的冗余信息. 而且即使判断了包含缺陷后, 也需要花费很多的人力去定位缺陷的具体位置. 因此, 切片才是合适的粒度, 且模型需要提供更多的可解释性. 为此, 他们提出了 AVDHRAM, 即基于层次表示和注意力机制的自动缺陷检测模型. 其将程序分为了 5 个层次, 分别是程序、函数、切片、语句、字符. 通过与 SySeVR 同样的切片起始点开始切片, 在经过

符号化与向量化后, 通过使用层次注意力网络(hierarchical attention network)来学习通过字符表示语句与通过语句表示切片. 其中, 注意力机制还可以用来进行可视化的表示, 用以表明对模型判断指导最多的部分, 从而辅助人工验证. 实验结果显示: AVDHRAM 的检测效果显著好于静态分析工具, 与 SySeVR 性能相当.

Zou 等人[79]也针对可解释性作了一些初步的尝试. 他们提出了一个可解释框架, 以通过模型对代码段的缺陷判断抽取出判断规则. 该框架的核心内容是识别对做出特定预测贡献最多的少量 token, 通过这些 token 构建一个决策树规则, 从而帮助该领域人员的理解与判断. 该框架应用在 VulDeePecker 与 SySeVR 模型上的结果显示: 该框架确实可以识别重要特征, 对人工验证检测器的结果起到了辅助的作用. 但是, 该解释性框架也存在一些问题, 例如: 无法解释为什么模型认为某些 token 的重要性更高; 生成的规则是半自动化总结的, 而不是完全自动化地进行解释; 框架针对每一个特别的预测进行解释规则的生成, 而不能生成适用于解释其他示例的全局规则. 这些问题都是日后工作亟待解决的.

Mao 等人[80]提出了一个基于注意力机制与双向 RNN 的可解释性缺陷检测模型, 其首先将源代码按照函数转化为 AST, 使用一个堆叠的双向 LSTM 网络来学习其表示, 再投入基于注意力机制的双向 RNN 模型进行分类. 在 NVD 与 Sard 抽取出的数据集测试下, 其获得了较好的分类效果, 但是对于可解释性的效果, 作者并未提出定量的评估方法.

为了使模型能够更精细地告知代码中哪些部分是与缺陷相关的, Li 等人[81]提出了一个基于图网络的可解释的缺陷检测器 IVDetect (interpretable vulnerability detector), 用来丰富模型的输出内容. 传统方法会将缺陷函数的整体作为输入, 而 IVDetect 会对其中语句进行“缺陷相关”和“上下文”的区分. 其首先构建缺陷判别模型 FA-GCN, 将源代码以程序依赖图的形式输入, 在编码时加入其数据依赖与控制依赖上下文; 分类主体是用于图分类的图卷积网络 GCN, 并使用了特征注意力机制. IVDetect 使用 GNNExplainer 来提供解释性结果, 其将 FA-GNN 模型、其判断结果和函数 PDG 作为输入, 通过在 PDG 中寻找一个最小子图, 使其在 FA-GNN 中得到的分数最接近原 PDG. GNNExplainer 通过对边进行遮盖并观察模型判断来达到对边重要性的评估. 实验结果显示: 在函数级缺陷检测上, IVDetect 相对前人方法有大幅度的提升; 在缺陷相关语句的辨别中, IVDetect 有 67%可以在 TOP 5 列表中准确命中.

该模型已开源(https://github.com/vulnerabilitydetection/VulnerabilityDetectionResearch).

但是对于可解释性问题, 最大的难点在于难以定量评估其解释性效果. Li 等人[81]使用了 Reveal[26]、Fan[52]、Devign[65]的数据集, 均为函数级缺陷代码数据集. 共包含 197 551 个函数, 其中, 22 278 个为缺陷函数, 占比 11.3%. 其中, Fan 数据集中包括对缺陷修复的记录, 因此被用于评估缺陷相关语句的判断效果. 但是, 修复语句与缺陷相关语句并不等价, 简单地将修复所影响的语句作为解释性语句的标准答案, 合理性不高.

事实上, 相较于缺陷检测中的标准答案, 可解释性由于其具备一定的主观因素, 其标准答案的制订更加困难. 因此, 为了评估可解释性的优劣, 还需要一个能够被广泛接受的评价标准.

3.3 泛化能力

在缺陷检测的过程中, 不同项目之间或不同语言之间的缺陷存在分布上的差异, 因此, 基于深度学习的方法需要使用域适应学习的方法来使得模型具备跨项目与跨语言的泛化能力. 深度域适应学习鼓励模型学习源数据和目标数据的新表示, 以最大限度地缩小它们之间的差异. 源数据和目标数据通过生成器映射到联合特征空间, 通过最小化其分布之间的差异, 使得两者在联合空间中得以更好地链接.

Nguyen 等人[82]利用深度领域适应来进行缺陷检测, 提出了 CDAN (code domain adaptation network)模型, 其使用序列代码的向量表示作为输入, 通过双向 RNN 作为生成器来生成源数据和目标数据在联合空间中的表示, 通过判别器对两者表示进行区分, 以不断地减少两者表示的差距, 最终通过在有标签源数据上训练的分类器进行分类. 由于在纳什均衡点, 源与目标在联合空间中是相同的, 因此可以将训练好的分类器对目标进行预测. 在 CDAN 基础上, 其提出了半监督的模型 SCDAN (semi-supervised code domain adaptation network), 使用条件交叉熵和光谱图使其满足平滑假设和聚类假设. 实验结果显示: 使用 CDAN 迁移框架, 能够显著提高模型在未标注数据上的预测性能.

当前, 深度学习模型在缺陷检测任务方面能够取得较好的效果, 但其训练集和测试集是同分布的. 事实上, 当应用到实际项目中时, 待测样本往往与训练样本是不同分布的. 例如待测样本和训练样本来源于不同的工程, 或者是不同的缺陷类型. 为此, Liu 等人[48]提出了 CD-VulD, 即跨域软件缺陷检测, 以解决缺陷预测模型的跨域迁移问题. 具体地, CD-VulD 首先将程序转化为 token 序列并转化为数字型向量, 然后通过基于 Bi-LSTM 的深度特征模型来学习高层序列表示, 通过矩阵转移学习框架来缩小源域和目标域的分布差异以学习跨域表示, 最终学到的跨域表示被用来训练分类器进行分类. 该模型针对的是函数级缺陷检测. 实验结果显示, 使用跨域学习得到的代码表示能够显著提高机器学习和深度学习分类器的检测效果. 在跨域问题上, CD-VulD 在类型上的迁移和项目间的迁移都显示出明显优势, 这种优势在源域与目标域使用不同代码表征方式时依然存在甚至更加突出. 该模型并未开源.

上述研究主要通过迁移方法提高项目间的泛化能力, 而语言之间的泛化与迁移同样应予以重视.

Hua 等人[83]复现了针对 C/C++语言的缺陷检测系统 VulDeePecker 与 SySeVR, 以测试其对 Java 代码中相对较多的单语句缺陷的检测效果. 其在单语句缺陷数据集 ManySStuBs4J 上的检测准确度从原来的超过 90%降低到 70%. 这印证了单语句缺陷并不能很好地被通用缺陷检测系统所探查.

事实上, 语言间的泛化是一个同样重要且难度更大的研究点.

3.4 额外特征的挖掘

在缺陷检测的任务背景下, 除了代码段本身的特征, 还包含一些额外的特征, 例如代码的传统度量信息、缺陷的修复信息, 这些都可以为模型的特征挖掘提供指导.

缺陷代码和非缺陷代码的差异有时候很小, 有时甚至修改单个变量、符号甚至数值就可以改变代码的正确性. 而基于相似度的缺陷复用检测往往无法分辨该类细微的差异而产生误报. 因此, 重点关注缺陷代码及其修复是十分有意义的.

在机器学习的缺陷检测研究中, Xiao 等人[45]就将代码的修复信息作为重要特征来挖掘. 他们提出了 MVP 模型. 首先对缺陷代码及其对应的修复版本生成抽象语法树与代码属性图, 根据补丁的修改情况, 使用不同策略进行切片, 以分离出缺陷相关的语法与语义特征, 然后通过待测代码的特征与其比对, 若其匹配缺陷特征而不匹配修复特征, 则认为其有缺陷. MVP 在 10 个开源工程上的检测效果显著高于基于克隆(ReDeBug、VUUDY)、基于函数匹配(SourcererCC、CCAligner)、基于深度学习(VulDeePecker、Devign)的检测方法, 同时, 其性能高于商业分析工具(Coverity、Checkmarx), 拥有更低的误报率和漏报率. 不过, 该模型并未开源.

Wu 等人[84]同样利用了修复信息来降低误报率, 不同的是, 其只是简单地通过相似度来进行计算, 若待测代码和匹配缺陷的修复后版本相似度更高, 则认定该报出为误报.

在深度学习的缺陷检测研究中, 也应该重点关注如何更好地利用缺陷的修复信息.

使用深度学习进行缺陷检测, 并不一定要完全抛弃传统方法, 也可以进行一定的结合.

Clemente 等人[50]利用多层前馈神经网络来寻找适用于缺陷检测的代码度量组合. 其计算了代码的 3 个维度的度量值, 包括复杂度度量、组成数值度量、面向对象度量, 并输入多层前馈神经网络来寻找合适的组合. 实验结果显示: 使用深度网络来结合传统度量值是有效果的, 且其效果要好于使用机器学习(对比决策树、随机森林、支持向量机、朴素贝叶斯), 但提升效果有限.

传统方法通过代码的度量信息来进行缺陷预测, 但是相对于缺陷模式的多样性, 通过度量进行表征还不够充分. Zhang 等人[54]提出了 DefectLearner, 其在传统代码度量中加入了交叉熵(cross entropy)作为代码特征, 一同投入深度网络进行缺陷检测. DefectLearner 首先对代码进行序列化的向量表示, 使用 LSTM 构造语言模型来捕捉代码模式与隐层特征, 最后加入标签信息与交叉熵及度量特征来训练分类器. 该模型使用了支持向量机、随机森林、朴素贝叶斯与逻辑回归这 4 种分类器. 实验结果显示: 交叉熵的判别效果可以高于一半的传统度量值, 在加入交叉熵特征后, 能够全面提高模型的检测性能.

传统的缺陷检测方法在一定程度上是有效果的, 这些效果在实际的工业领域也已经得到了验证. 但是传统方法往往针对的是不同的使用场景, 在不同类型上的检测水平存在差异. 为此, Jabeen 等人[85]提出了一个

整合的缺陷检测模型, 以对传统检测方法加以综合考虑. 该模型选择了 4 个传统缺陷检测模型, 将其检测结果作为输入, 投入多层感知机(MLP)以获得整合后的缺陷预测结果. 其使用的是传统的 3 层感知机(输入层、隐层、输出层), 没有作特殊的修改. 在从 CVE 抽取出的真实用例作为数据集的实验结果上看, 使用 MLP 进行整合能够获得最低的错误率.

就当前缺陷检测商业软件的发展情况来看, 绝大多数商业软件还是基于传统的规则方法. 这是由于基于规则的方法往往能够提供完整的缺陷路径, 对于其后续的确认更为方便, 且当前基于规则的方法检测效果更为稳定, 不需要考虑跨域迁移等问题. 这种商业现象显示出了传统方法在当今的缺陷检测领域依然存在其价值. 因此, 如何将深度学习技术与传统方法相结合, 是一个值得探究的研究点.

3.5 当前深度学习模型存在的问题

为了便于对比, 我们对第 3 节中梳理的深度学习模型进行列表展示, 对于每一种代码表征形式挑选至多 3 项代表性工作, 详细列举其各项技术信息. 如表 3 所示, 表中的每一行都代表一项研究工作. 第 1 列代表了该研究使用的代码表征形式. 第 3 列为分析对象的编程语言. 第 4 列为分析对象的样本粒度. 第 5 列为分析对象表征时选择的特征, 包括抽象语法树(AST)、控制流图(CFG)、数据流图(DFG)、程序依赖图(PDG)、代码属性图(CPG)、离散傅里叶变换(DFT)、编译中间代码(LLVM)等. 第 6 列为该深度模型使用的模型架构, 包括前馈神经网络(FNN)、卷积神经网络(CNN)、双向长短期记忆(bi-LSTM)、双向门控循环单元(bi-GRU)等. 第 7 列为该模型所使用的评价指标, *F*1(以及多分类下的 *W*-*F*1)即指代包含误报率(FPR)、漏报率(FNR)、召回率(TPR/recall)、精确率(P)和 *F*1 值的传统评价体系, 其他指标还包括正确率(accuracy)等. 第 8 列标记其是否开源. 第 9 列表示该类表征的预处理难度.

表 3 深度学习模型统计信息小结

代码表征	文献	语言	样本粒度	特征选择	模型构架	评价指标	开源	预处理难度
序列	[37]	JavaScript	代码片段	序列	FNN	Accuracy/Recall	是	低
	[42]	C/C++	切片	序列	Bi-LSTM	*F*1	否	
	[43]	C/C++	切片	序列	Bi-LSTM	*W*-*F*1	否	
树	[34]	C/C++	函数	AST	Bi-GRU	*F*1	否	中
	[53]	Java	文件	AST	CNN	*F*1	否	
	[61]	Java	函数	AST/PDG/DFG	CNN	*F*1	是	
图	[31]	C/C++	函数	CPG	MLP	*F*1	否	高
	[35]	Java	文件	AST/CFG/PDG	GCN/GAT	*F*1	否	
	[65]	C/C++	函数	AST/CFG/DFG	GRU	Accuracy/*F*1	否	
其他	[32]	C/C++	函数	DFT	CNN/LSTM	*F*1	否	–
	[77]	C	中间代码段	LLVM	Bi-GRU	Accuracy/*F*1	是	

可以看到: 针对不同的语言, 研究者们已经尝试使用了主流的卷积、序列、图模型进行缺陷检测任务, 主要检测对象集中在主流编程语言. 但各模型针对的样本粒度差异较大, 且只有个别模型为开源状态, 使得模型间的比较较为困难.

从代码表征维度上看, 基于序列的表征较为直观, 且不需要对待测代码进行额外的分析处理, 在实现上较为简便. 但是代码语言具有较强的结构性与局部性, 其上下文的依赖关系复杂且距离较长, 因此, 单纯地使用序列作为代码表征会损失大量的代码结构特征. 基于树的表征可以更好地显式表征代码中的结构信息, 但事实上, 除了抽象语法树能够表示的结构信息以外, 源代码本身还具备数据流、控制流等多种不同维度的特征, 而这些特征无法显式地被抽象语法树表示. 基于图的表征能够显式表征更多形式与维度的代码特征, 当前被认为是更具前景的代码表征模式. 但基于图的表征也存在一些问题, 例如: 在基于图的表征模式下, 相同节点往往会被合并, 代码语言的自然顺序也会丢失; 其次, 大规模代码检测时的图构建需要耗费大量的计算资源与时间, 这些都是在使用基于图的表征时需要考虑的.

除了代码的表征, 在模型阶段对数据集的划分、使用与针对性设计是更加突出的问题.

3.5.1 多数据来源混用

在前述方法中, 许多研究[42−44]都使用了多来源的数据集, 作为训练和评估的基础. 多来源数据集往往是人工构造的数据集 Sard 与从真实项目中抽取的缺陷条目(NVD 抽取), 而由于真实项目缺陷条目获取成本较高, 这类混合数据集大多只包含少量的真实缺陷. 例如, VulDeePecker[42]的数据集中仅有 1.4% (840 个)来自于真实项目缺陷(NVD), 这种数据比例会导致基于大量人工构造缺陷数据训练的深度模型在真实缺陷上的检测能力并没有得到很好的评估.

事实上, 人工构造缺陷在代码长度与复杂度上都远低于真实抽取的缺陷, 人工构造缺陷上挖掘得到的特征可能并不适用于真实缺陷. 这需要更进一步的探究.

3.5.2 重复切片导致的数据泄露

表 4 为 VulDeePecker[42]构建的数据集中的两个样本, 其切片代码与标签完全一致. 由于在一个切片中的不同位置作为起点进行切片, 可能会得到一样的切片, 因此会导致数据重复. 而当前划分数据集时大多使用随机划分, 会使得这些重复的数据分别被划分到训练集与测试集, 这就导致了大量的泄露, 使得模型的检测效果虚高.

表 4 VulDeePecker[42]数据集中的两个样本

样本序号	样本来源	样本代码	是否为缺陷

20	CVE-2007-5849/cups_1.3.4_CVE-2007-5849_snmp.c	for (cache = (snmp_cache_t *)cupsArrayFirst(Devices);
		cache = (snmp_cache_t *)cupsArrayNext(Devices))
		free(cache->addrname);
			0

21	CVE-2007-5849/cups_1.3.4_CVE-2007-5849_snmp.c	for (cache = (snmp_cache_t *)cupsArrayFirst(Devices);
		cache = (snmp_cache_t *)cupsArrayNext(Devices))
		free(cache->addrname);
			0

事实上, 从不同位置作切片生成相同的切片符合实际应用的样本分布. 因此, 解决该问题不应从去重的角度, 而应从训练数据的划分角度来考虑. 需要在划分时避免泄露, 即避免相同的切片被分开, 即可规避重复切片带来的虚假测试值.

3.5.3 缺乏数据不平衡的针对性设计

由于代码中大部分代码均为正确代码, 因此在正常的构造流程下, 缺陷代码在其中的比例都较低. 例如在 VulDeePecker[42]中, 缺陷代码只占 28.8%. 因此, 在这种条件下, 需要针对不平衡问题进行针对性的设计.

已有部分研究考虑到该问题, 并通过例如 SMOTE 算法进行改进. 不过, 该方向依然存在较大改进的空间, 例如可以探究使用数据增强的方式改善不平稳问题.

3.5.4 评价指标不符合使用场景

当前, 绝大多数缺陷检测系统均使用基于 $F1$ 值的评价体系, 即分别计算准确率(accuracy)、精确率(precision)、召回率(recall), 最后通过 $F1$ 值综合评价模型性能. 一些研究也会使用真正例(TP)、假正例(FP)、假反例(FN)和真反例(TN)来评估. 这些均为较常用的评估指标, 这里不再赘述.

然而, $F1$ 值无法很好地评价真实使用场景. 在真实开发环境中, 人工审查往往精力有限, 不会审查全部报出缺陷, 而会根据系统预测的排序, 审查靠前的报出. 因此, 应当加入 TOP N 限制下的各项指标来评估, 其更加符合实际的应用场景.

4 未来研究展望

在对当前研究进展进行了归纳总结与问题阐述后, 我们对该任务未来的研究点进行展望.

4.1 构建通用多粒度数据集

当前, 缺陷检测领域没有一个统一通用的评价标准与评价流程, 这使得不同模型之间的比对较为不便.

如前文所述, 当前研究均使用自行构建的缺陷代码数据集, 这些数据集在数据来源、数据粒度与标注方式上均存在较大的差异.

从数据来源上看, 不同的数据来源意味着其样本复杂程度的不同. 缺陷库抽取数据(如 NVD)来源于真实的工业代码, 其复杂程度较高, 且模块化程度高, 各组件之间的依赖关系与调用关系复杂. 人工构造数据(如 Sard)来源于人工撰写的测试用例, 其复杂程度低, 且主要针对缺陷的表示, 而大多忽略其真实的功能, 因此为独立程序片段, 几乎没有组件之间的长程依赖与复杂的调用关系. 在这两类不同复杂度的数据来源中抽取的缺陷数据, 其对代码缺陷的表征能力差距较大, 因此不适合从数值(例如 $F1$ 等)上进行横向比较. 从实际应用的角度来看, 缺陷检测系统需要具备对真实工业代码的检测能力, 即理想的缺陷代码数据集应来源于 NVD 等工业代码而非人工撰写的测试用例. 但工业代码缺陷数据的获取难度与成本远大于测试用例, 需要更加完善的样本提取技术与大量不可避免的人工审查, 这可能需要相关领域的研究者共同努力.

从数据粒度上看, 不同的数据粒度之间难以进行统一评价. 当前, 缺陷代码数据集没有一个统一的数据粒度, 在文件级数据粒度过大成为共识后, 当前研究主要使用函数级数据与切片级数据. 然而如第 2.2.3 节所述, 即使是由同一个缺陷条目构造的函数级数据与切片级数据, 因其代码切割与分离方式的不同, 也无法进行横向的比较. 因此, 必须明确统一的数据粒度, 才能保证数值比较的公平性. 函数级数据在划分时成本极低, 且在检测过程中不需要对被测代码进行分析(切割为函数即可). 而切片级数据需要在构建数据集与检测被测代码时均对代码进行数据流、控制流等分析, 对资源的需求较大, 时间成本也较高. 然而, 函数级数据基于的朴素假设是: 缺陷发生在函数的范围内. 这在大规模工业代码中显然是难以支持的, 特别是在模块化的情况下, 数据的声明与使用往往不存在于单一函数中. 因此, 代码切片应该是更为合理的数据粒度, 可以更加完整地表征缺陷的流程.

从标注方式上看, 不同的标注方式使得标签的正确性存在差异. 安全相关人员的人工标注能够保证标签的质量, 但其获取成本较高. 为了解决这一问题, 许多研究者尝试使用启发式方法、自动化方法快速地进行标注. 然而就目前的研究情况看, 自动化与启发式方法均难以达到理想的标注效果. 因此在现阶段, 使用人工标注来构造通用数据集是无法避免的.

针对传统缺陷检测方法的统一评测, 有研究者进行了尝试. Zhang 等人[86]提出了 iTES, 其首先自动构建了一个缺陷代码库, 通过控制模块选择测试使用的条目供被测系统检测, 通过监控模块记录测试结果并通过评估模块对各项测试数据进行生成. iTES 最终会对检测系统的报出数目、误报率、召回率、资源使用、分析时间和关键程度进行评估.

该系统是针对静态与动态缺陷检测系统设计的, 其只依靠源代码即可进行检测. 而基于深度学习的缺陷检测系统在测试上涉及到不同的输入粒度(如各类属性图、切片等), 需要对源代码进行较多的预处理操作, 因此还需针对该类问题进行针对性的设计.

经过上述分析, 对于通用的缺陷代码数据集, 我们认为其应该来源于真实的工业软件代码, 使用切片对缺陷路径进行分离, 通过足量的人工审查进行标注, 并进行充分的多粒度预处理操作. 这些步骤都具有较高的成本, 需要该领域研究者的共同努力.

4.2 利用无标记数据

当前, 使用深度学习方法来进行缺陷检测, 绝大多数都遵循着传统的“learn-from-bugs”流程, 即先构造带标签的缺陷代码数据集, 再搭建深度神经网络来挖掘并学习数据集中的特征. 因此, 神经网络的缺陷检测能力极大程度上会受到数据集质量的影响. 这就意味着需要大量的标注数据来训练神经网络. 然而, 有别于其他深度学习任务(例如图像识别)可以较低成本地获得标注数据, 缺陷代码的数据标注需要较好的代码相关知识, 因此其必须由专业的从业人员完成, 导致其样本获得成本较高.

此外, 通过标注的缺陷代码数据集训练得到的深度学习模型, 只能针对数据集拥有的缺陷类型进行检测, 若要对检测缺陷类型进行扩展, 则必须增加大量的新类型标注数据, 扩展能力有限.

然而, 在日渐庞大的开源代码库中, 有海量无标签的代码资源. 因此, 如何利用海量未标记数据, 是一个重要的研究点.

Ahmadi 等人[87]使用非学习的方法, 在无标签的代码中通过 2 次聚类, 先找到功能相近的代码, 再从中找

到类内有差异的代码, 通过“大部分代码是无缺陷”的假设, 将聚类中有差异的代码标记为缺陷代码, 即不需要额外信息即可自动完成缺陷数据的标注.

在深度学习的框架下, 同样可以利用相似思路, 对无标签的数据加以表示与利用.

为了应对标注的缺陷数据较少的问题, Allamanis 等人[38]提出了一种自监督的缺陷检测模型. 其使用一个选择器对正确代码进行变异, 再通过一个检测器对变异的位置与类型进行判断, 通过对抗的方式使得检测器能够具备检测困难缺陷的能力. 由于其变异模式只有4种, 因此当检测器预测出缺陷模式后, 即可通过逆向变异完成对该缺陷的修复. 但有限的变异模式也导致了该方法在缺陷类型上的泛化能力有限. 在针对真实项目的检测中, 该方法报出了 19 个真实缺陷, 然而其误报率高达 98%, 在实用性上还有很大差距. 这同时也证实了使用人工变异构造的缺陷数据难以匹配真实缺陷的复杂度. 该模型已开源(https://github.com/microsoft/neurips21-self-supervised-bug-detection-and-repair).

除此之外, 语言模型近年来在自然语言处理领域展现出的强大能力, 也为无监督挖掘数据特征提供了新的思路. 当前已有一些研究者在探究代码语言模型的构建与应用, 如 CodeBER[71]与 CuBERT[88]. 此类代码语言模型在代码的简单缺陷(如变量误用)与错误检测上具有一定的效果, 但在复杂缺陷检测任务下能力有限[88]. 因此, 如何构建适用于缺陷检测任务的代码语言模型, 还需要进一步的探究.

当前已有较多的无监督、半监督方法被应用在自然语言处理领域并取得了不错的效果. 将无监督、半监督方法应用在缺陷检测领域, 可以极大地解决标注数据获取成本高的问题.

4.3 明确深度学习能力边界

代码缺陷的类型众多, 以CWE分类标准为例, 其将缺陷分为了10个大类, 其中的二级分类就有上百种之多, 这些不同的缺陷类型所拥有的特征和模式也是千差万别的. 传统的缺陷检测方法会对每一个缺陷类型有针对性地设计缺陷规则, 而当前的深度学习缺陷检测方法没有对其类型间差异的问题进行处理, 大多简单地将其按照二分类或多分类任务对待. 从缺陷原理上看, 其不同大类与小类在难易程度、复杂程度上存在较大的差异. 因此, 深度学习在不同缺陷类型上的效果, 即哪些缺陷类型适合使用深度学习方法, 是值得探究并明确的.

同时, 由于不同的缺陷类型在缺陷原理上的差异, 可能适合不同的深度网络机构对其进行挖掘. 因此, 对单一缺陷类型适合何种深度网络, 也是一个值得探究的研究点.

Yuan 等人[89]对基础的深度学习模型进行了初步的评估尝试, 他们在 3 个项目的数据下比较了 GRU、bi-GRU、DNN、LSTM 这 4 种模型, 并对其训练效率进行了比对. 从训练效率上看, DNN 显著优于其他几种序列模型. 但是该研究只在准确度(accuracy)上进行了测试, 由于数据集中正负样本不平衡问题较为严重, 该度量值并不能充分反映模型的判断能力. 对于模型的适用性还需要更加细致、全面的评估.

当前对缺陷检测的研究中, 从原始的完整源代码到最后抽取出的缺陷片段与判断结果, 中间的各个步骤都可以使用人工审查、传统规则与深度学习的方法进行. 例如, 当前有探究使用深度学习来辅助 SMT 求解的研究工作, 虽然使用少量数据训练来获得精准的 SMT 求解能力是非常困难的, 但是利用深度学习对中间步骤进行筛选和排序, 可以提高 SMT 求解的效率. 同理, 在缺陷检测的任务背景下, 应当从应用的角度出发, 即从大规模工程的检测角度出发, 综合地考虑与比对适合每个步骤使用的技术, 而不是仅仅将缺陷检测当作一个独立的分类任务.

4.4 大规模工程的检测

当前, 对深度学习在源代码缺陷检测领域的研究大多停留在学术探究阶段, 往往只将其当作一项独立的分类任务, 即对数据集中的条目进行分类. 事实上, 源代码缺陷检测作为软件开发过程中重要的一环, 具有较大的实际应用意义. 因此, 同样需要从工业应用的视角对待该项研究, 从工业视角的大规模工程的检测角度出发, 其要保证流程的完整性与方法的可用性.

- 流程的完整性, 即从原始的全部工程代码到抽取待测片段, 到模型生成缺陷判断结果, 中间的各个步

骤均可以使用人工审查、传统规则与深度学习的方法进行. 需要综合地考虑与对比适合每个步骤使用的技术.

- 而方法的可用性, 则对整套流程的效率提出了要求.

当前的相关研究往往忽略了以上两点. 例如: 使用图神经网络对经过预处理的代码属性融合图进行分类能够取得较好的数值效果, 但在实际的大规模工程检测任务中, 对百万行的工程代码进行代码属性融合图的构造会耗费极大的计算资源与时间. 因此在后续的研究中, 如何适配大规模工程的实际检测场景, 是一个需要重点关注的研究点.

5 总 结

基于深度学习进行源代码缺陷检测利用了深度神经网络自动挖掘深层特征的能力, 将安全开发者从繁重的规则编写与特征工程中解脱出来, 因此也受到越来越多的关注. 本文针对该领域近 5 年的论文发表情况进行了详细的分析, 从数据集构建与深度模型这两个方面对当前的技术进行了分类与总结, 并对面临的挑战与未来可能的研究重点进行了阐述.

在大量研究者的探究与实验下, 深度学习对于缺陷检测任务的能力已经得到了验证, 深度学习对挖掘代码中语义与结构信息的能力相对传统方法存在其优势. 但是需要看到的是: 深度学习在缺陷检测方向上的应用历史较短, 当前依然存在较多基础性问题亟待解决, 在数据集与模型层面都有其改进的方向. 特别是在通用数据集的建设上, 期待能有更多的研究人员共同努力, 为该领域的未来发展构建坚实的数据基础.

References:

[1] Planning S. The economic impacts of inadequate infrastructure for software testing. Technical Report, National Institute of Standards and Technology, 2002.

[2] LaToza TD, Venolia G, DeLine R. Maintaining mental models: A study of developer work habits. In: Proc. of the 28th Int'l Conf. on Software Engineering. 2006. 492−501.

[3] IEEE Standards Coordinating Committee. IEEE standard glossary of software engineering terminology (IEEE Std 610.12-1990). Los Alamitos: IEEE Computer Society, 1990, 169: 132.

[4] Adger WN. Vulnerability. Global Environmental Change, 2006, 16(3): 268−281.

[5] Coverity: Coverity scan static analysis. 2022. https://scan.coverity.com/

[6] KlocWork: Static code analysis for C, C++, C#, and Java. 2022. https://www.perforce.com/products/klocwork

[7] Gao Q, Ma S, Shao S, *et al*. CoBOT: Static C/C++ bug detection in the presence of incomplete code. In: Proc. of the 26th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). IEEE, 2018. 385−388.

[8] Cadar C, Dunbar D, Engler DR. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI, 2008, 8: 209−224.

[9] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, 2011, 46(3): 265−278.

[10] Cha SK, Avgerinos T, Rebert A, *et al*. Unleashing mayhem on binary code. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. IEEE, 2012. 380−394.

[11] LibFuzzer: A library for coverage-guided fuzz testing. 2022. http://llvm.org/docs/LibFuzzer.html

[12] Vimpari M. An evaluation of free fuzzing tools [MS. Thesis]. University of Oulu, 2015.

[13] AFL: American fuzzy lop. 2022. https://lcamtuf.coredump.cx/afl/

[14] Song CX, Wang X, Zhang WZ. Analysis and optimization of ANGR in dynamic software test application. Computer Engineering & Science, 2018, 40(S1): 167−172 (in Chinese with English abstract).

[15] Godefroid P, Levin MY, Molnar D. SAGE: Whitebox fuzzing for security testing. Communications of the ACM, 2012, 55(3): 40−44.

[16] Bochspwn. 2022. https://github.com/googleprojectzero/bochspwn

[17] Pan J, Yan G, Fan X. Digtool: A {virtualization-based} framework for detecting kernel vulnerabilities. In: Proc. of the 26th USENIX Security Symp. (USENIX Security 2017). 2017. 149−165.

[18] Syzkaller. 2022. https://github.com/google/syzkaller

[19] Rapidscan. 2022. https://github.com/skavngr/rapidscan

[20] Zhang X, Li ZJ. Survey of fuzz testing technology. Computer Science, 2016, 43(5): 1−8 (in Chinese with English abstract).

[21] Ye ZB, Yan B. Survey of symbolic execution. Computer Science, 2018, 45(s1): 28−35 (in Chinese with English abstract).

[22] Zou QC, Zhang T, Wu RP, Ma JX, Li MC, Chen C, Hou CY. From automation to intelligence: Survey of research on vulnerability discovery techniques. Journal of Tsinghua University (Science and Technology), 2018, 58(12): 45−60 (in Chinese with English abstract).

[23] Hindle A, Barr ET, Gabel M, *et al*. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122−131.

[24] Perl H, Dechand S, Smith M, *et al*. VCCfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. 2015. 426−437.

[25] Li Y, Huang CL, Wang ZF, *et al*. Survey of software vulnerability mining methods based on machine learning. Ruan Jian Xue Bao/ Journal of Software, 2020, 31(7): 2040−2061 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]

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

[27] Lin G, Wen S, Han QL, *et al*. Software vulnerability detection using deep neural networks: A survey. Proc. of the IEEE, 2020, 108(10): 1825−1848.

[28] SARD: Software assurance reference dataset. 2022. https://samate.nist.gov/SRD/index.php

[29] Owasp benchmark. 2022. https://owasp.org/www-project-benchmark/

[30] Xu A, Dai T, Chen H, *et al*. Vulnerability detection for source code using contextual LSTM. In: Proc. of the 5th Int'l Conf. on Systems and Informatics (ICSAI). IEEE, 2018. 1225−1230.

[31] Duan X, Wu J, Ji S, *et al*. VulSniper: Focus your attention to shoot fine-grained vulnerabilitie. In: Proc. of the IJCAI. 2019. 4665−4671.

[32] Cao D, Huang J, Zhang X, *et al*. FTCLNet: Convolutional LSTM with Fourier transform for vulnerability detection. In: Proc. of the 19th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020. 539−546.

[33] Saccente N, Dehlinger J, Deng L, *et al*. Project Achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering Workshop (ASEW). IEEE, 2019. 114−121.

[34] 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). IEEE, 2020. 722−727.

[35] Ghaffarian SM, Shahriari HR. Neural software vulnerability analysis using rich intermediate graph representations of programs. Information Sciences, 2021, 553: 189−207.

[36] Progex. 2022. https://github.com/ghaffarian/progex/

[37] Pradel M, Sen K. Deepbugs: A learning approach to name-based bug detection. Proc. of the ACM on Programming Languages, 2018, 2(OOPSLA): 1−25.

[38] Allamanis M, Jackson-Flux H, Brockschmidt M. Self-supervised bug detection and repair. In: Advances in Neural Information Processing Systems. 2021. 34.

[39] Choi MJ, Jeong S, Oh H, *et al*. End-to-end prediction of buffer overruns from raw source code via neural memory networks. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. 2017. 1546−1553.

[40] NVD. 2022. https://nvd.nist.gov/

[41] Lin G, Zhang J, Luo W, *et al*. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. on Industrial Informatics, 2018, 14(7): 3289−3297.

[42] Li Z, Zou D, Xu S, *et al*. VulDeePecker: A deep learning-based system for vulnerability detection. NDSS, 2018.

[43] 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, 2019.

[44] 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.

[45] Xiao Y, Chen B, Yu C, *et al*. {MVP}: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In: Proc. of the 29th {USENIX} Security Symp. ({USENIX} Security 2020). 2020. 1165−1182.

[46] Nikitopoulos G, Dritsa K, Louridas P, *et al*. CrossVul: A cross-language vulnerability dataset with commit data. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2021. 1565−1569.

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

[48] Liu S, Lin G, Qu L, *et al*. CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. IEEE Trans. on Dependable and Secure Computing, 2020.

[49] Jimenez M, Le Traon Y, Papadakis M. Enabling the continuous analysis of security vulnerabilities with VulData7. In: Proc. of the 18th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). IEEE, 2018. 56−61.

[50] Clemente CJ, Jaafar F, Malik Y. Is predicting software security bugs using deep learning better than the traditional machine learning algorithms? In: Proc. of the 2018 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2018. 95−102.

[51] Alexopoulos N, Egert R, Grube T, *et al*. Poster: Towards automated quantitative analysis and forecasting of vulnerability discoveries in debian GNU/Linux. In: Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security. 2019. 2677−2679.

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

[53] Li J, He P, Zhu J, *et al*. Software defect prediction via convolutional neural network. In: Proc. of the 2017 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2017. 318−328.

[54] Zhang X, Ben K, Zeng J. Cross-entropy: A new metric for software defect prediction. In: Proc. of the 2018 IEEE Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2018. 111−122.

[55] Ponta SE, Plate H, Sabetta A, *et al*. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2019. 383−387.

[56] Dong Y, Guo W, Chen Y, *et al*. Towards the detection of inconsistencies in public security vulnerability reports. In: Proc. of the 28th {USENIX} Security Symp. ({USENIX} Security 2019). 2019. 869−885.

[57] Rostami S, Kleszcz A, Dimanov D, *et al*. A machine learning approach to dataset imputation for software vulnerabilities. In: Proc. of the Int'l Conf. on Multimedia Communications, Services and Security. Cham: Springer, 2020. 25−36.

[58] Gonzalez D, Hastings H, Mirakhorli M. Automated characterization of software vulnerabilities. In: Proc. of the 2019 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2019. 135−139.

[59] Bhandari G, Naseer A, Moonen L. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In: Proc. of the 17th Int'l Conf. on Predictive Models and Data Analytics in Software Engineering. 2021. 30−39.

[60] Karampatsis RM, Sutton C. How often do single-statement bugs occur? The manysstubs4j dataset. In: Proc. of the 17th Int'l Conf. on Mining Software Repositories. 2020. 573−577.

[61] Li Y, Wang S, Nguyen TN, *et al*. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. of the ACM on Programming Languages, 2019, 3(OOPSLA): 1−30.

[62] Zhou Y, Sharma A. Automated identification of security issues from commit messages and bug reports. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. 2017. 914−919.

[63] Sabetta A, Bezzi M. A practical approach to the automatic classification of security-relevant commits. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2018. 579−582.

[64] 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.

[65] Zhou Y, Liu S, Siow J, *et al*. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems. 2019. 32.

[66] 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 (TOSEM), 2021, 30(3): 1−33.

[67] Russell R, Kim L, Hamilton L, *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). IEEE, 2018. 757−762.

[68] Dam HK, Pham T, Ng SW, *et al*. Lessons learned from using a deep tree-based model for software defect prediction in practice. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2019. 46−57.

[69] Garg A, Degiovanni R, Jimenez M, *et al*. Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach. arXiv:2012.11701, 2020.

[70] Devlin J, Chang MW, Lee K, *et al*. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv: 1810.04805, 2018.

[71] Feng Z, Guo D, Tang D, *et al*. CodeBERT: A pre-trained model for programming and natural languages. In: Proc. of the Findings of the Association for Computational Linguistics: EMNLP 2020. 2020. 1536−1547.

[72] Zhou X, Han DG, Lo D. Assessing generalizability of CodeBERT. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE Computer Society, 2021. 425−436.

[73] Ahmad W, Chakraborty S, Ray B, *et al*. Unified pre-training for program understanding and generation. In: Proc. of the 2021 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2021. 2655−2668.

[74] Lewis M, Liu Y, Goyal N, *et al*. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. 2020. 7871−7880.

[75] Feng Q, Feng C, Hong W. Graph neural network-based vulnerability predication. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2020. 800−801.

[76] Pechenkin A, Demidov R. Applying deep learning and vector representation for software vulnerabilities detection. In: Proc. of the 11th Int'l Conf. on Security of Information and Networks. 2018. 1−6.

[77] Li Z, Zou D, Xu S, *et al*. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. IEEE Trans. on Dependable and Secure Computing, 2021.

[78] An W, Chen L, Wang J, *et al*. AVDHRAM: Automated vulnerability detection based on hierarchical representation and attention mechanism. In: Proc. of the 2020 IEEE Int'l Conf. on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/ SustainCom). IEEE, 2020. 337−344.

[79] Zou D, Zhu Y, Xu S, *et al*. Interpreting deep learning-based vulnerability detector predictions based on heuristic searching. ACM Trans. on Software Engineering and Methodology (TOSEM), 2021, 30(2): 1−31.

[80] Mao Y, Li Y, Sun J, *et al*. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In: Proc. of the 2020 IEEE Int'l Conf. on Big Data (Big Data). IEEE, 2020. 4651−4656.

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

[82] Nguyen V, Le T, Le T, *et al*. Deep domain adaptation for vulnerable code function identification. In: Proc. of the 2019 Int'l Joint Conf. on Neural Networks (IJCNN). IEEE, 2019. 1−8.

[83] Hua J, Wang H. On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In: Proc. of the 18th IEEE/ ACM Int'l Conf. on Mining Software Repositories (MSR). IEEE, 2021. 530−534.

[84] Wu P, Yin L, Du X, *et al*. Graph-based vulnerability detection via extracting features from sliced code. In: Proc. of the 20th IEEE Int'l Conf. on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2020. 38−45.

[85] Jabeen G, Ping L, Akram J, *et al*. An integrated software vulnerability discovery model based on artificial neural network. In: Proc. of the SEKE. 2019. 349−458.

[86] Zhang C, Chen J, Cai S, *et al*. iTES: Integrated testing and evaluation system for software vulnerability detection methods. In: Proc. of the 19th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020. 1455−1460.

[87] Ahmadi M, Farkhani RM, Williams R, *et al*. Finding bugs using your own code: detecting functionally-similar yet inconsistent code. In: Proc. of the 30th {USENIX} Security Symp. 2021.

[88] Kanade A, Maniatis P, Balakrishnan G, *et al*. Learning and evaluating contextual embedding of source code. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2020. 5110−5121.

[89] Yuan X, Zeng P, Tai Y, *et al*. The efficiency of vulnerability detection based on deep learning. In: Proc. of the Advancements in Mechatronics and Intelligent Robotics. Singapore: Springer, 2021. 449−455.

附中文参考文献:

[14] 宋丛溪, 王辛, 张文喆. Angr 动态软件测试应用分析与优化. 计算机工程与科学, 2018, 40(S1): 167−172.

[20] 张雄, 李舟军. 模糊测试技术研究综述. 计算机科学, 2016, 43(5): 1−8.

[21] 叶志斌, 严波. 符号执行研究综述. 计算机科学, 2018, 45(s1): 28−35.

[22] 邹权臣, 张涛, 吴润浦, 马金鑫, 李美聪, 陈晨, 侯长玉. 从自动化到智能化: 软件漏洞挖掘技术进展. 清华大学学报(自然科学版), 2018, 58(12): 1079−1094.

[25] 李韵, 黄辰林, 王中锋, 袁露, 王晓川. 基于机器学习的软件漏洞挖掘方法综述. 软件学报, 2020, 31(7): 2040−2061. http://www.jos.org.cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]

邓枭(1995－), 男, 博士生, 主要研究领域为缺陷自动检测.

谢睿(1991－), 男, 博士, 助理研究员, 主要研究领域为程序语言理解, 缺陷自动检测.

叶蔚(1985－), 男, 博士, 副研究员, 主要研究领域为自然语言处理, 程序语言理解, 软件安全.

张世琨(1969－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为知识计算, 软件工程, 软件安全.