文章编号:1674-2974(2025)04-0103-11 DOI:10.16339/j.cnki.hdxbzkb.2025270

# 基于深度学习的混合语言源代码漏洞检测方法

张学军†,郭梅凤,张潇,张斌,黄海燕,蔡特立
(兰州交通大学 电子与信息工程学院,兰州 730070)

**摘 要**:现有基于深度学习的源代码漏洞检测方法主要针对单一编程语言进行特征学习,难以对混合编程语言软件项目因代码单元间的关联和调用产生漏洞进行有效检测.因此,本文提出了一种基于深度学习的混合语言源代码漏洞检测方法DL-HLVD.首先利用BERT层将代码文本转换为低维向量,并将其作为双向门控循环单元的输入来捕获上下文特征,同时使用条件随机场来捕获相邻标签间的依赖关系;然后对混合语言软件中不同类型编程语言的函数进行命名实体识别,并将其和程序切片结果进行重构来减少代码表征过程中的语法和语义信息的损失;最后设计双向长短期记忆网络模型提取漏洞代码特征,实现对混合语言软件漏洞检测.在SARD和CrossVul数据集上的实验结果表明,DL-HLVD在两类漏洞数据集上识别软件漏洞的综合召回率达到了95.0%,$F_1$值达到了93.6%,比最新的深度学习方法VulDeePecker、SySeVR、Project Achilles在各个指标上均有提升,说明DL-HLVD能够提高混合语言场景下源代码漏洞检测的综合性能.

**关键词**:漏洞检测;命名实体识别;程序切片;混合语言
**中图分类号**:TP311 **文献标志码**:A

# DL-HLVD:Deep Learning-based Hybrid Language Source Code Vulnerability Detection Method

ZHANG Xuejun†,GUO Meifeng,ZHANG Xiao,ZHANG Bin,HUANG Haiyan,CAI Teli
(School of Electronic and Information Engineering, Lanzhou Jiaotong University, Lanzhou 730070, China)

**Abstract**: The existing deep learning-based source code vulnerability detection methods mainly focus on the feature learning of a single programming language, and it is difficult to effectively detect the vulnerabilities caused by the association and invocation of code units in software projects of hybrid programming languages. To address this issue, a deep learning-based hybrid language vulnerability detection method DL-HLVD is proposed. Firstly, the BERT layer is used to convert the code text into low-dimensional vectors, which are then used as inputs to the

* **收稿日期**:2024-01-04
**基金项目**:国家自然科学基金资助项目(61762058), National Natural Science Foundation of China (61762058);甘肃省教育厅产业支撑项目(2022CYZC-38), Industrial Support Project of Gansu Provincial Department of Education (2022CYZC-38);国家电网科技项目(W32KJ2722010, 522722220013), State Grid Science and Technology Project (W32KJ2722010, 522722220013);甘肃省重点研发计划项目(25YEFA089), Key Research and Development Project of Gansu Province(25YEFA089)
**作者简介**:张学军(1977—),男,宁夏中宁人,兰州交通大学教授,博士生导师
† 通信联系人,E-mail: xuejunzhang@mail.lzjtu.cn

bidirectional gated loop unit to capture the contextual features, and the conditional random field is used to capture the dependency between adjacent labels. Secondly, functions from different types of programming languages are identified as named entity recognition in the hybrid software and reconstructed with the program slicing results to reduce the loss of syntactic and semantic information in the code characterization process. Finally, the bidirectional long short-term memory network model is designed to extract the vulnerability code features and realize the vulnerability detection of hybrid language software. The comprehensive experimental results on the SARD and CrossVul datasets show that the comprehensive recall rate of DL-HLVD on the two types of vulnerability datasets is 95.0%, and the F1 value reaches 93.6%, which is improved in all indicators compared with the VulDeePecker, SySeVR, and Project Achilles. It demonstrates that the DL-HLVD method can improve the comprehensive performance of source code vulnerability detection in hybrid language scenarios.

**Key words**: vulnerability detection; named entity recognition; program slicing; hybrid language

软件系统的复杂性和多样性导致软件漏洞成为影响软件安全的主要因素之一[1].大量未经审查的代码和库在多种平台间重用,带来了大量的漏洞,增加了软件安全风险.为了避免控制劫持攻击,攻击者可能会将恶意代码进行编程语言转换和技巧性操作,以发起对软件的成功攻击.另外,开发人员在检查语言安全时的失误可能会引入空间(如缓冲区溢出)和时间(如释放后使用)内存损坏漏洞,可能会被攻击者利用在两种不同编程语言间操作,从而在不违反任何一方安全检查的情况下有效规避安全检查[2].显然,跨语言编程所带来的漏洞一旦被恶意利用,可能会导致隐私泄露、数据丢失,甚至对企业声誉和经济造成重大损害.

物联网的普及以及物联网设备的多样性增加了物联网软件系统的复杂性和异构性,加剧了物联网软件系统的安全风险[3].为了提高物联网软件系统的适应能力和可扩展性,开发者更倾向于使用多种混合编程语言开发软件,例如TensorFlow、Microsoft Office和Mozilla Firefox都采用了包括Java、C/C++和Python在内的多种语言共同开发.然而,不同语言间的兼容性问题和相互调用,加上不安全编程语言本身的缺陷以及开发人员和团队组织管理等因素,使得多种编程语言开发的软件漏洞频发.因此,对多种编程语言开发的项目进行有效漏洞检测,并在新型语言出现时快速适配现有模型,已成为目前需要解决的一大问题.

深度学习在软件漏洞检测中的准确率高于传统机器学习方法,但在处理长文本等复杂任务时还存在记忆和理解能力不足的问题.为此,BERT模型[4]被提出,它具有强大文本表示和理解能力,能有效应对这些局限,为大型复杂软件源代码的漏洞检测提供了新的解决方案.当前的发展趋势是将BERT作为文本特征编码器,通过无监督预训练学习代码的语法和词的上下文信息来提取特征,并通过微调提高检测效率和准确性.然而,尽管使用单一编程语言的深度学习漏洞检测技术已取得很好的进展,但是针对混合编程语言的漏洞检测技术尚在起步阶段[4],仍存在许多亟待解决的挑战性问题:1)不同编程语言之间的结构和语法差异性大,正则表达式的支持和语法细节不同,因此,有效提取不同编程语言的代码表征难度较高;2)已有基于深度学习的源代码漏洞检测方法采用传统正则表达式实现程序代码关键实体的匹配,在面对不同编程语言模块时其适应能力和泛化能力有限;3)相关研究表明,常见的单实体识别神经网络一般只考虑样本输入,缺乏对输出关系的思考,且在预处理模块方面侧重于词与词之间的特征提取,往往忽略了词的上下文信息.

针对上述挑战,本文基于BERT、双向门控循环单元(bidirectional gated recurrent unit, BGRU)、条件随机场(conditional random field, CRF)和双向长短时记忆网络(bidirectional long short-term memory, BLSTM)技术,提出了一种基于深度学习的混合语言源代码漏洞检测(deep learning-based hybrid language source code vulnerability detection, DL-HLVD)方法.本文的主要贡献可总结如下:

1)DL-HLVD从SARD数据集[5]和CrossVul数据

集[6]中提取Java、C/C++的子数据集,并获取基于代码属性图(code property graph, CPG)的函数级程序切片.DL-HLVD首次利用前人字典和手工标注方法"BIO"自行构建了一套混合编程语言源代码特征领域数据集.该数据集标注了不同类型的实体,如变量(VAR)和函数(FUN).此外,代码切片涵盖了C/C++的底层细节并融入了Java的高级抽象特征,通过跨语言交互展现了多语言代码片段的混合特性.

2)提出一种基于BERT-BGRU-CRF的混合语言源代码命名实体识别模型,学习实体特征和依赖关系,并通过将命名实体识别学习结果与程序切片重构提高漏洞检测的泛化能力,以此解决正则表达式在开放域场景下的局限性.

3)为验证DL-HLVD对混合语言源代码漏洞检测的能力,基于SARD和CrossVul数据集进行了全面的实验验证与分析.结果表明,相比于VulDeePecker, SySeVR和Project Achilles等深度学习方法,DL-HLVD在混合编程语言漏洞数据集上表现出更好的性能.

## 1 相关工作

软件漏洞检测方法主要用于软件源代码漏洞的检测和分析,以保证软件系统的安全稳定运行.这些检测方法可分为基于规则的源代码漏洞检测方法和基于深度学习的源代码漏洞检测方法.

### 1.1 基于规则的源代码漏洞检测方法

通常,基于规则的源代码漏洞检测方法利用Flawfinder[7]、RATS[8]、ITS4[9]等开源工具通过特定语言漏洞规则进行静态分析.其中,Flawfinder对C/C++语言的缓冲区溢出、格式化字符串等漏洞规则进行存储,通过匹配内置漏洞缺陷数据库的形式对源代码进行安全检测.

商业工具Checkmarx[10]、Fortify[11]、Coverity[12]等虽能检测混合语言源代码漏洞,但它们依赖专家预定义特征,易受专家经验和主观性等因素影响,导致误报率和漏报率高.因此,基于规则的源代码漏洞检测方法难以满足复杂软件系统的检测需求.近年来,借助代码特征领域命名实体识别提高混合语言软件漏洞检测成为一种趋势.传统的命名实体识别方法主要包括基于字典和规则的方法[13-14],这些方法依赖领域字典和专家,而且手工特征选择不可避免地引入主观性和较高人工成本.机器学习技术的发展使得CRF、隐马尔可夫、支持向量机等模型在软件漏洞检测领域受到广泛关注.

### 1.2 基于深度学习的源代码漏洞检测方法

当前,深度学习技术在恶意代码检测、垃圾邮件过滤等领域取得了突破性进展,能很好地解决传统漏洞检测方法存在的不足.Anbiya等[15]利用抽象语法树(abstract syntax tree, AST)和PHP Token作为特征,通过TF-IDF进行特征向量化并结合深度学习进行源代码漏洞分类,但对源代码的结构和语义语法信息表征不够全面.杨宏宇等[16]研究表明,将源代码完全视为线性文本,难以充分表征源代码的特征.为此,一些研究工作[17-18]将AST和CFG融合为CPG的表征方式,并利用词向量和深度学习模型进行源代码漏洞检测,但检测粒度较粗且准确率不高.Li等[19-20]基于代码切片技术提出了VulDeePecker和SySeVR框架,并利用BLSTM和循环神经网络(recurrent neural network, RNN)训练漏洞检测模型,实现了细粒度的漏洞检测.然而,这两种方法在数据预处理时依赖正则表达式,对C/C++之外的数据集检测效果欠佳[21].而且,现有代码切片技术难以全面覆盖漏洞类间细微差异特征,张学军等[22]构建了适用于两级代码切片的深度学习漏洞检测融合模型,实现了对多类型源代码漏洞的准确检测;随后提出基于transformer的源代码切片级漏洞检测方法[23],通过采用CodeBERT提取源代码的语义信息,有效提升了对代码语句间远程上下文依赖的学习能力.

新近的研究工作[24]表明,在医疗和自然语言处理等领域的实体识别任务中,深度神经网络能自动提取单词特征,有效解决传统CRF模型在忽略上下文信息、模型效率低、易分词等方面的局限性.鉴于模型融合的思想,本文考虑在CRF中加入神经网络模型作为主要框架,以静态分析[25]为基础,提出一种基于深度学习的混合语言源代码漏洞检测方法DL-HLVD.通过考虑代码结构信息,DL-HLVD采用基于图表征的CPG和基于code gadgets[19]的Token表征对混合语言源代码特性统一表征,并设计了BERT-BGRU-CRF模型来学习与漏洞数据相关的源代码实体,同时构建了基于注意力机制的BLSTM模型.DL-HLVD主要涉及三大核心技术:数据处理及切片清

洗、命名实体识别模型、预处理代码与实体识别重构技术.首先,对源代码进行切片、分词和向量化;接着,将预处理后的源代码序列输入BERT-BGRU-CRF模型进行训练,以获得源代码序列的标注结果;随后,结合分类结果对源代码序列进行数据清洗,并根据预设规则覆盖实体标注中的关键词,以减少干扰;训练完成后,对程序切片进行命名实体识别,基于一对多的映射关系对预处理后的代码进行重构.实验证明,DL-HLVD能在混合语言代码环境中实现有效的漏洞检测.

## 2 DL-HLVD方法设计

### 2.1 背景知识

#### 2.1.1 程序静态分析相关概念

AST是源代码的树状表示,是构成其他代码表征图的基础,其节点表示源代码中的结构(如代码块、语句、声明、表达式等).控制流图(control flow graph, CFG)是一种有向图,描述了一个程序执行过程中可能遍历到的所有路径.CPG整合了AST和CFG,包含源代码的语义语法特征和行为信息.为了充分利用CPG中的信息并简化预处理,本文先将CPG转换成序列来表征函数,并通过神经网络模型抽取序列中潜在的编程模式.函数的CPG表征过程为:首先生成函数的CPG;然后提取CPG中的关键信息并转化成序列;接着将文本形式的序列映射成数值型的向量并根据设置的长度对向量进行填充和截断操作;最后对向量进行词嵌入处理,将向量中的每个元素替换成对应的词向量.

#### 2.1.2 词嵌入技术

在检测源代码漏洞时,通常需要采用词向量嵌入算法将经过数据预处理的信息看作纯文本输入漏洞检测模型.FastText[26]是基于skip-gram的改进模型,它将输入上下文中的每个单词用n-gram分解后和原单词相加来表示上下文的语义信息.

首先将源代码经过分词处理得到一个词序列,如$\text{code}=\{\text{word}_1,\text{word}_2,\cdots,\text{word}_n\}$,其中,code表示一段源代码,$\text{word}_n$表示经过分词处理的词,$n$代表词的序数.然后利用FastText将词序列的每个词转化为相应的带有语义相关性的词向量,这样原本由词组成的一维列表就转化为由向量组成的二维矩阵$\mathbb{R}^{n\times m}$,其中$n$为取词序列的前$n$个词进行向量转化,词个数不足则由全0组成的词向量补充.最后将得到的词向量矩阵输入BLSTM模型进行混合语言漏洞检测训练.

### 2.2 DL-HLVD方法架构

DL-HLVD的系统架构如图1所示,主要包括数据预处理及模型训练两个阶段.模型训练又分为命名实体识别模型训练和漏洞检测模型训练.

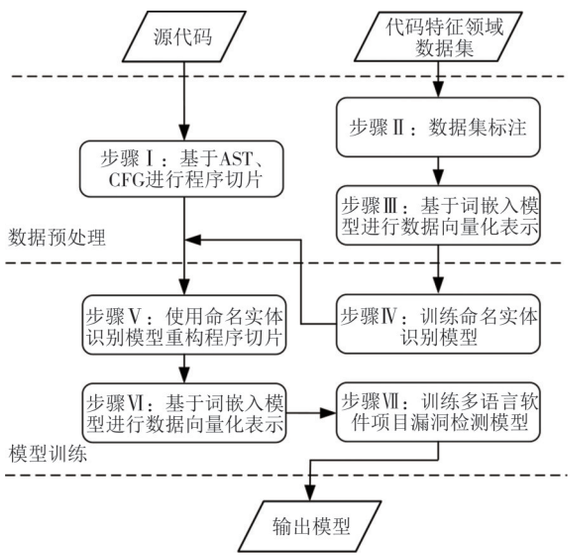



图1 DL-HLVD方法架构

Fig.1 DL-HLVD method architecture

数据预处理阶段主要对漏洞代码及代码的实体特征进行处理:对源代码按照步骤Ⅰ处理,通过CPG对源代码进行表征后输出程序切片;对代码特征领域数据集则按照步骤Ⅱ及步骤Ⅲ处理,将数据集通过标注后输入词嵌入模型并输出其向量化表示.在数据预处理后,将两类输出结果作为模型训练阶段的输入数据.在模型训练阶段,先将步骤Ⅲ得到的向量化表示作为基于BERT-BGRU-CRF的命名实体识别模型的输入进行模型训练;训练完成后利用步骤Ⅳ得到的模型对步骤Ⅰ得到的程序切片进行命名实体识别,而后根据识别结果对程序切片进行重构并作为步骤Ⅵ的输入.接着,用经过步骤Ⅵ得到的数据向量化表示作为输入训练多语言软件项目漏洞检测模型,并将重构结果分词编码后输入BLSTM网络中学习漏洞代码特征,最终得到多语言软件项目的源代码漏洞检测模型.

## 3 命名实体识别方法

### 3.1 相关定义

为了有效表征各类编程语言的语义、语法信息,以期在训练漏洞检测神经网络模型时获得较好的效

果,借助代码特征领域实体对程序语句进行程序切片重构.

**定义 1** 源程序:源程序 $P$ 是按某种编程语言规范编写的代码语句 $s_1, s_2, \cdots, s_n$ 组成的集合,源程序定义为 $P = \{s_1, s_2, \cdots, s_n\}$.

**定义 2** 代码令牌(Token):代码令牌由若干常量、关键字、标识符和运算符 $t_1, t_2, \cdots, t_n$ 等组成,代码令牌集合可以构成代码语句,表示为 $s_i = \{t_{i,1}, t_{i,2}, \cdots, t_{i,j}\}$.

**定义 3** 代码特征领域实体:对于一个源程序 $P = \{s_1, s_2, \cdots, s_n\}$,其代码语句表示为 $s_i = \{t_{i,1}, t_{i,2}, \cdots, t_{i,j}\}$,代码语句中一系列符合相应规则的代码令牌子集 $Y$ 为代码特征领域实体,表示为 $Y = \{y_{i,1}, y_{i,2}, \cdots, y_{i,j-w}\}$.

**定义 4** 程序切片:对于不同编程语言对应的一个源程序 $P = \{s_1, s_2, \cdots, s_n\}$,利用静态分析工具生成 CPG,并通过程序依赖图进行基于图可达性分析得到的程序切片,再根据相应的控制流、数据流、语义等切片准则删除无关代码后所剩下的部分 $S = \{s_1, s_2, \cdots, s_{n-k}\}$.程序切片与源程序在一定的切片准则下保持一致的语义.

**定义 5** 重构程序切片:先按照实体标注格式进行语句级的序列标注,然后通过获取的实体标注结果集对源代码集合 $S = \{s_1, s_2, \cdots, s_{n-k}\}$ 进行重构并输出重构代码集合.根据语义一致性准则、简化命名约定准则、优化代码结构准则替换切片中相关代码特征领域实体后产生的新程序切片 $S' = \{s'_1, s'_2, \cdots, s'_{n-k}\}$.

### 3.2　数据预处理流程

为提升模型泛化性能和训练精度,数据预处理环节至关重要,其目的是为检测模型和命名实体识别做好数据准备.其中,数据不平衡问题是影响模型性能的关键因素之一.为了应对数据不平衡对模型训练的影响,本文采用重采样技术(包括过采样和欠采样)来调整数据集中正负样本比例,对少数类进行过采样以增加其代表性,同时对多数类进行欠采样以减少其冗余信息,从而为模型的训练提供更优质的数据基础.数据处理过程如图 2 所示.

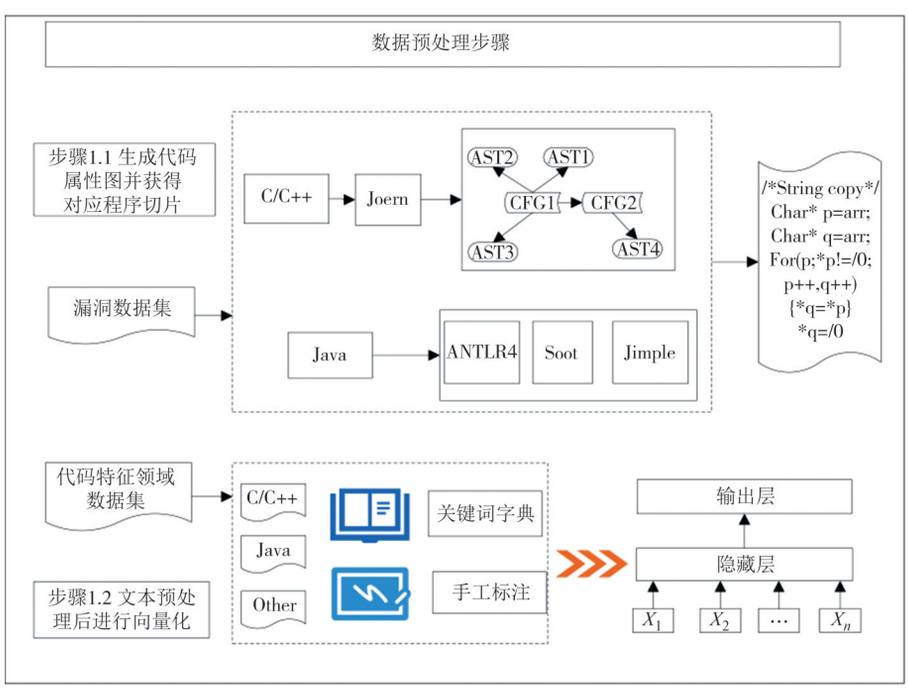



图 2　数据预处理流程

Fig.2　Data preprocessing process

1)针对漏洞数据集,通过 CPG 表征源代码输出程序切片并标注,以提取含漏洞信息的代码片段.对于 C/C++源代码,先使用 Joern[27] 生成 CPG 并存储在 Neo4j 图数据库中,然后利用程序依赖图进行基于图可达性分析得到程序切片;对于 Java 源代码,先基于 ANTLR4[28] 构建 AST,再基于 Soot[29] 生成 CFG,最后以 AST 和 CFG 构成的简化 CPG 为对象,用 Jimple 生成程序切片.之后为生成的中间表示进行标签标记,用“0”表示没有漏洞,用“1”表示包含漏洞.

2)针对代码特征领域数据集,通过词嵌入模型输出向量化表示后输入命名实体识别模型进行训练并对程序切片进行重构.具体地,先基于关键词字

典,采用手工标注“BIO”方法筛选源代码特征领域数据集,并为其打标签;然后使用FastText将实体数据转化为向量,输入BERT-BGRU-CRF模型训练后得到代码特征领域实体标注;最后据此标注结果对程序切片重构,在不依赖特定正则表达式的同时实现对多语言源代码关键词特征提取.

### 3.3 源代码漏洞代码特征实体及标注

源代码漏洞代码特征实体是指与各类语言漏洞相关的源代码命名实体的统称,目前命名实体识别常基于监督学习的序列标注方式.本文分析了源代码漏洞发生参数的类型、特征和相关性,确定了一组与漏洞代码数据表示相关的实体类型.

定义以下实体体系:A={fun_sys, fun_call, fun_user, var_return, var_num, var_str, var_specific, var_other},分别表示系统函数、调用函数、用户自定义函数、函数返回型变量、数字型变量、字符型变量、变量具体值和其他类型变量.用“B_”“I_”“O”分别代表某个实体的开头、中间字符以及其他非实体字符.标注格式如表1所示.

**表1 源代码漏洞代码特征领域实体标注格式**

**Tab.1 Source code vulnerability code feature domain entity annotation format**

| 标注符号 | 代表实体 |
|---|---|
| B_fun_sys | 系统函数实体开头字符 |
| I_fun_sys | 系统函数实体中间字符 |
| B_fun_call | 调用函数实体开头字符 |
| I_fun_call | 调用函数实体中间字符 |
| B_fun_user | 用户自定义函数实体开头字符 |
| I_fun_user | 用户自定义函数实体中间字符 |
| B_var_return | 函数返回型变量实体开头字符 |
| I_var_return | 函数返回型变量实体中间字符 |
| B_var_num | 数字型变量实体开头字符 |
| I_var_num | 数字型变量实体中间字符 |
| B_var_str | 字符型变量实体开头字符 |
| I_var_str | 字符型变量实体中间字符 |
| B_var_specific | 变量具体值实体开头字符 |
| I_var_specific | 变量具体值实体中间字符 |
| B_var_other | 其他类型变量实体开头字符 |
| I_var_other | 其他类型变量实体中间字符 |
| O | 其他非实体字符 |

在BIO标注过程中,本文采用人力标注与已有知识库相结合的方式,通过分析源代码的语法结构和语义内容,为每个词或符号手动分配BIO标签以确保实体标注准确性.此外,整合了经过长期积累的前人字典作为辅助工具来确保更高的标注精度,如VulDeePecker中的变量名、函数名等实体字典,将源代码文本分词并与字典匹配,转换为数值索引构建输入向量.通过这种方式,最终获得了高质量、高准确率的BIO标注数据,为模型训练和评估提供了可靠的数据支持.

### 3.4 基于BERT-BGRU-CRF的命名实体识别模型

BERT是一种基于transformer的预训练语言模型,主要包含12个transformer编码器层,每层由多头自注意力和前馈神经网络构成.其中,多头自注意力用来捕捉单词上下文关系,前馈神经网络用于特征提取和非线性变换.

在本文构建的BERT-BGRU-CRF模型中,由于BERT具有丰富的信息提取能力,直接将其放在最底层抽取上下文的文本信息,并对输入序列进行编码得到每个词语的上下文编码表示.在BERT之后添加BGRU层和Dropout层形成一个完整的网络,BGRU由两层GRU单元组成,每一层包含200个GRU单元.输入编码通过BGRU可捕捉到序列的上下文信息,但GRU没有考虑源代码自动标注后返回的结果中单词与标签之间的相关性,为此引入CRF层来构建BGRU-CRF联合模型,从少量标注中学习更多的词序信息,CRF层能够加入一些约束使最终预测结果有效.利用BERT-BGRU-CRF模型进行各类编程语言的特征学习后,再通过获取的实体标注结果集$Y$对程序切片集合$S$进行重构得到重构切片集合$S'$,具体过程如图3所示.在步骤1和步骤3所示代码块中,上面代码部分为C/C++代码,下面代码部分为Java代码.

本文将数据预处理后的源代码向量化,并输入BERT-BGRU-CRF模型训练,随后执行图3中的步骤1,先对C/C++和Java代码进行去注释、字符规范化等处理.然后进行命名实体识别,获取步骤2所示的代码段实体识别结果.在步骤3中,将步骤1获得的预处理代码与步骤2获得的命名实体识别结果进行结合重构,获得多语言重构代码片段.结合重构代码的过程为:创建一个映射表来存储程序切片和它们对应的命名实体识别结果,以程序切片为键,实体列表为值,形成一对多映射.其中,映射表中的每个实体由边界(B,I,O)和类型(fun_sys、var_specific、var_other、fun_user、var_str等)标识.接着,遍历程序切片,将实体添加到对应切片的列表.最后,用映射表重构代码切片,确保代码中的实体得到正确识别和处理.

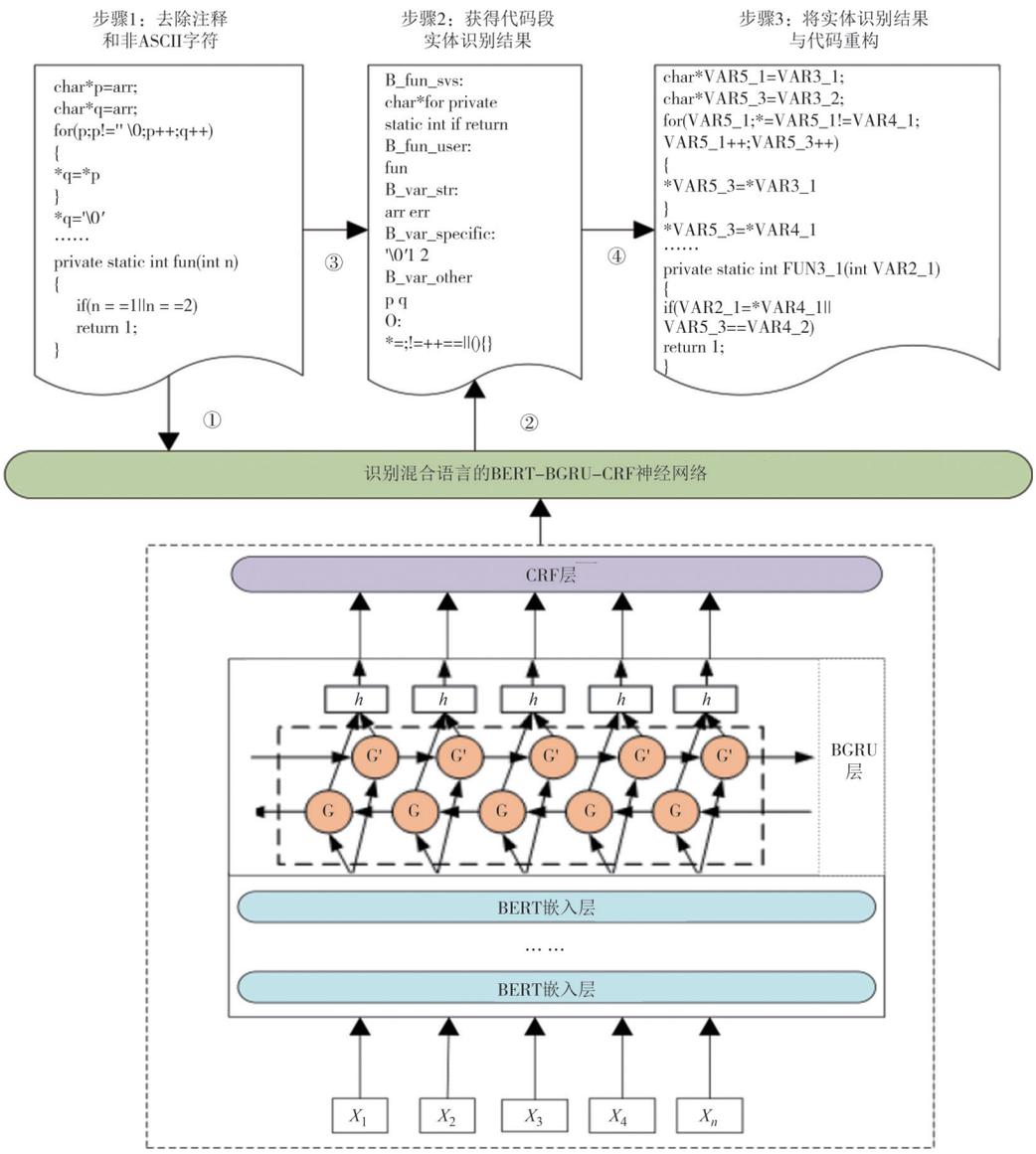



图 3　BERT-BGRU-CRF 模型识别过程

Fig.3　BERT-BGRU-CRF model recognition process

## 4　漏洞检测神经网络模型

如图 4 所示，DL-HLVD 的漏洞检测模型由 BLSTM 网络、Attention 层和 Softmax 层组成.具体来说，DL-HLVD 采用两层各含 300 个神经元的 BLSTM 结构，并通过 Attention 层增强关键特征的学习；Softmax 作为模型的输出层，将得分转换为概率分布，以执行漏洞检测的二分类任务.

## 5　实验结果与分析

### 5.1　数据集

为验证 DL-HLVD 的有效性，选取软件防护参考数据集 SARD[5] 和混合语言缺陷代码数据集 CrossVul[6] 进行实验验证.

SARD 数据集由美国国家标准与技术研究院提出，涵盖了 C/C++、Java、C# 和 PHP 等多种语言的漏洞样本.每个源文件有“bad”和“good”函数块，前者存在漏洞，后者修复其漏洞.实验在 SARD 中选取 C、C++ 和 Java 的漏洞子数据集，标注“good”为负样本(0)，“bad”为正样本(1)，最终以程序切片为基本单位构建漏洞数据集，其中 C/C++ 样本总数为 28 312，正样本 6 793 个，负样本 21 519 个，Java 样本总数为 26 184，正样本 8 836 个，负样本 17 348 个，总体包含 42 种漏洞类型.

CrossVul 数据集由 Nikitopoulos 等[6] 提出，涵盖了 40 多种编程语言编写的漏洞文件和相应的补丁

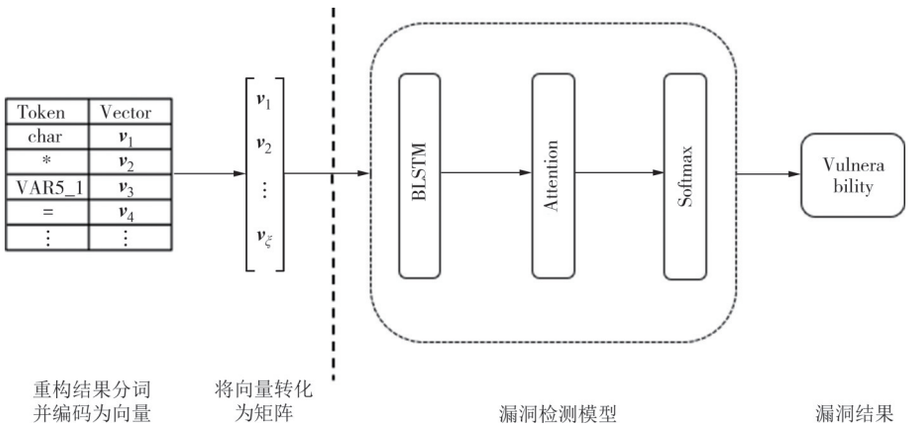



图4 BLSTM漏洞检测模型架构

Fig.4 BLSTM vulnerability detection model architecture

文件.该数据集收集了1 675个GitHub项目提供引用的NVD条目,将超链接和GitHub进行对应并过滤掉无效链接.本实验在CrossVul中选取C、C++和Java子集,并将其切片后进行细粒度标注,得到C/C++样本总数13 382个,正样本2 143个,负样本11 239个,Java样本总数26 303个,正样本2 078个,负样本24 225个.预处理后的数据集信息如表2所示.

**表2 预处理后的数据集信息**

**Tab.2 Preprocessed dataset information**

| 数据集 | 切片总数 | C/C++漏洞切片 | Java漏洞切片 |
|---|---|---|---|
| SARD | 45 660 | 6 793 | 8 836 |
| CrossVul | 26 303 | 2 143 | 2 078 |

### 5.2 评价指标

实验采用Li等[19]所提出的评价指标来评估DL-HLVD方法的性能,主要包括准确率(precision, $P$)、$F_1$分数($F_1$ score)、ACC分数(accuracy, ACC)、假阳性率(false positive rate, FPR)、假阴性率(false negative rate, FNR)、真阳性率(true positive rate, TPR)等,这些指标已被广泛认可并用于漏洞检测研究中.

### 5.3 实验结果与分析

#### 5.3.1 实验设置

为了评估DL-HLVD的有效性,实验设置了以下3个研究问题.

**RQ1**:BERT-BGRU-CRF中每层对漏洞检测起了什么关键作用?为什么要用命名实体识别方法,而不是对不同语言采用不同的正则表达式匹配关键实体,以及使用命名实体识别的好处是什么?

**RQ2**:本文方法与传统漏洞检测工具和现有的深度学习漏洞检测方法相比有什么优势?

**RQ3**:利用自己构建的数据集进行模型训练,训练完成的模型是否能够检测目前基于深度学习的漏洞检测的其他数据集?

#### 5.3.2 命名实体识别模型效果比较与分析

针对RQ1,为了深入探究BERT、BGRU以及CRF在多语言命名实体识别模型中对于漏洞检测准确率的提升效果,实验对BERT-BGRU-CRF模型中每层的关键作用进行详细比较,以明确各层对整体效果的影响和贡献.同时,为了全面评估传统正则表达式与命名实体识别在混合语言模块上的泛化能力与适应能力,进一步设置4组实验进行对比分析,实验结果如表3所示.

**表3 命名实体识别模型效果对比**

**Tab.3 Comparison of the effects of named entity recognition models**

| 组 | 模型 | $P$/% | $F_1$/% | ACC/% |
|---|---|---|---|---|
| 1 | BERT-CRF | 95.46 | 92.89 | 96.36 |
| 2 | BERT-BGRU | 93.35 | 90.83 | 95.13 |
| 3 | BGRU-CRF | 94.48 | 93.52 | 96.50 |
| 4 | BERT-BGRU-CRF | 95.51 | 94.68 | 97.61 |
| 5 | BGRU+正则 | 92.73 | 90.42 | 94.46 |
| 6 | BERT-BGRU+正则 | 90.56 | 89.49 | 95.92 |

由表3可见,BERT作为模型最底层时,整体性能表现最佳,这得益于其强大的深层结构和文本匹配能力,能有效捕捉代码上下文和Token间关联;从3,4组实验可知,BERT的加入在识别源代码中的变量名和函数名等关键实体方面表现优异,有助于理解函数语义和上下文;从1,4组实验发现,BERT基础上加入BGRU层对命名实体识别的提升有限,因为BERT具备强大的特征提取能力,而BGRU更多用于强化序列的语序关联;从2,4组实验观察发现,引

入CRF层通过添加约束规则提升了预测有效性,优化了命名实体识别结果,提高了关键实体识别的准确性,降低了误报.

3,4,5,6组实验表明,传统正则表达式在匹配关键实体上效果不佳,因为即使C和Java的正则表达式在某些方面相似,但在细节上仍存在显著差异.因此,使用正则表达式进行关键实体匹配时,需要针对每种编程语言单独设计,会耗费大量的时间和精力.更重要的是,正则表达式只能识别已知模式的命名实体,对未知模式的命名实体识别效果欠佳,这使得针对某种编程语言的正则表达式很难有效匹配其他编程语言的关键实体,无法适应混合编程环境的复杂性.而基于BERT-BGRU-CRF的命名实体识别模型在对源代码产生歧义、简写、组合时的良好泛化效果及特征提取能力能够有效辅助检测模型进行漏洞挖掘.因此,DL-HLVD适用于混合编程环境,能消除语言差异带来的分析难题,具备高度的灵活性,可以针对不同类型的实体进行训练来提高漏洞检测效率和准确性.

5.3.3　*各方法综合性能比较与分析*

针对RQ2,为评价各方法的综合性能,实验在SARD数据集上选取基于规则的传统漏洞检测工具Flawfinder[7],针对C/C++语言的漏洞检测方法VulDeePecker[19]、SySeVR[20]和针对Java语言的漏洞检测方法Project Achilles[30]与DL-HLVD进行对比分析,实验结果如表4~表6所示.

**表 4　各方法在SARD的Java子数据集上性能比较**

**Tab.4　Performance comparison of various methods on Java sub datasets of SARD**

| 方法 | FPR/% | FNR/% | TPR/% | *P*/% | $F_1$/% | ACC/% |
|---|---|---|---|---|---|---|
| **DL-HLVD** | **5.89** | **1.67** | **97.82** | **94.62** | **96.27** | **96.31** |
| VulDeePecker | 9.21 | 18.53 | 84.58 | 92.14 | 87.92 | 88.27 |
| SySeVR | 8.34 | 15.67 | 88.13 | 92.82 | 88.38 | 89.82 |
| Project Achilles | 7.52 | 14.82 | 90.85 | 93.49 | 92.93 | 92.52 |
| Flawfinder | 51.86 | 48.63 | 55.29 | 0.78 | 0.00 | 50.10 |

**表 5　各方法在SARD的C/C++子数据集上性能比较**

**Tab.5　Performance comparison of various methods on C/C++ sub datasets of SARD**

| 方法 | FPR/% | FNR/% | TPR/% | *P*/% | $F_1$/% | ACC/% |
|---|---|---|---|---|---|---|
| **DL-HLVD** | **5.85** | **5.66** | **95.86** | **92.78** | **94.13** | **95.17** |
| VulDeePecker | 6.52 | 5.98 | 93.89 | 90.38 | 92.19 | 93.51 |
| SySeVR | 6.85 | 6.82 | 94.91 | 91.63 | 92.31 | 94.42 |
| Project Achilles | 8.56 | 14.57 | 89.69 | 86.15 | 87.29 | 89.19 |
| Flawfinder | 30.28 | 25.36 | 79.23 | 71.94 | 74.91 | 75.86 |

**表 6　各方法在SARD的混合语言数据集上性能比较**

**Tab. 6　Performance comparison of various methods on Mixed language datasets of SARD**

| 方法 | FPR/% | FNR/% | TPR/% | *P*/% | $F_1$/% | ACC/% |
|---|---|---|---|---|---|---|
| **DL-HLVD** | **5.62** | **2.98** | **97.11** | **92.73** | **94.96** | **95.40** |
| VulDeePecker | 9.15 | 12.69 | 90.12 | 91.05 | 91.58 | 92.15 |
| SySeVR | 9.28 | 13.56 | 90.26 | 91.95 | 91.86 | 92.68 |
| Project Achilles | 9.71 | 13.95 | 89.39 | 90.21 | 90.91 | 91.58 |
| Flawfinder | 40.58 | 30.36 | 75.24 | 40.56 | 50.63 | 65.85 |

由表4和表5可见,VulDeePecker和SySeVR在C/C++数据集上表现良好,但C和Java的语法差异,致使这两种方法在Java漏洞数据集上的检测效果不佳.相反,Project Achilles使用RNN对Java源代码进行静态方法级别的漏洞检测,在Java数据集上表现良好,而在C/C++数据集上效果不佳.

由表4~表6可见,DL-HLVD与其他深度学习方法在多个数据集上的表现均优于Flawfinder,因为Flawfinder需依赖人工定义漏洞规则及构建相应漏洞库,但目前缺乏针对Java的漏洞规则库和语法规则,所以该方法综合性能表现低于其他深度学习方法,尤其在Java数据集上效果最不理想.

从表6可见,DL-HLVD在混合语言数据集上实现了5.62%的FPR、94.96%的$F_1$和95.40%的ACC,相比其他漏洞检测方法,在平均91.45%的$F_1$基础上提高了3.51个百分点.表明DL-HLVD中的BERT-BGRU-CRF模块优于原有依靠关键词字典的方式以及智能筛选优于人工打标的方式.具体来讲,BERT模型通过学习丰富的语言上下文信息,对于命名实体的识别具有优势,而其他深度学习方法在静态代码分析上受限于代码规则和特征工程.此外,基于深度学习的方法利用正则表达式清洗关键函数,但正则表达式模式固定,难以应对源代码的动态变化,且仅适用于特定代码块,无法全面覆盖漏洞类型和检测场景.相比之下,命名实体识别技术更灵活,分类效果更好.因此,DL-HLVD方法在各类数据集上表现更理想,泛化能力更强,能识别单一语言模型难以发现的漏洞特征.

5.3.4　*各方法在不同数据集上的比较与分析*

针对RQ3,选择SARD和CrossVul混合语言数据集进行验证,实验结果如图5所示.

由图5可见,DL-HLVD在SARD数据集上的准确率、精确率、召回率以及$F_1$指标分别为94.0%、91.5%、94.9%和93.3%,均优于其他方法,且在SARD数据集上的表现也均好于CrossVul数据集,这是因

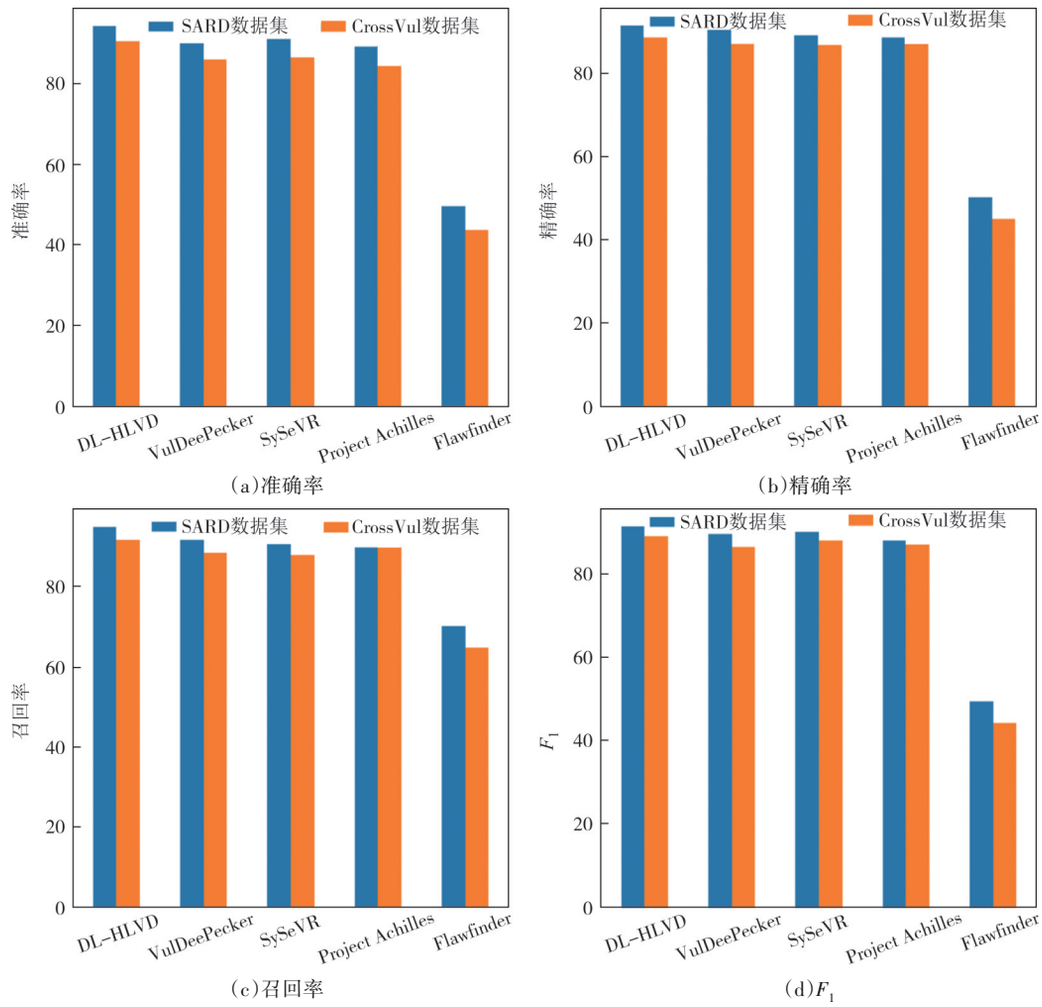



图5 各方法在SARD和CrossVul数据集的准确率、精确率、召回率、$F_1$对比

Fig.5 Comparison of precision、accuracy、recall、$F_1$ of various methods on SARD and CrossVul datasets

为CrossVul数据集来源于真实的开源软件项目,存在数据不平衡问题,漏洞样本的代码数量远小于良性代码的情况.DL-HLVD在CrossVul数据集的综合召回率达到了95.0%,表明DL-HLVD可以发现更多的潜在漏洞,而DL-HLVD的平均$F_1$达到了93.6%,表明DL-HLVD提高了混合语言源代码漏洞检测的综合性能.

## 6 结 论

针对多编程语言软件系统,为解决漏洞检测中代码表征不一致和正则表达式筛选泛化能力不足的问题,本文提出了一种基于深度学习的混合语言软件源代码漏洞检测方法DL-HLVD,利用CPG生成程序切片以表征代码,在自己构建的数据集上对BERT-BGRU-CRF模型进行训练,随后实现命名实体识别任务进行代码重构,并设计了基于BLSTM的漏洞检测模型.通过在SARD和CrossVul数据集上的实验验证与分析,结果证明了DL-HLVD在混合语言软件漏洞检测性能上的提升.然而,混合语言软件系统的复杂性使得DL-HLVD方法难以全面覆盖潜在的风险,漏洞模式的多样性也增加了准确识别的难度.未来工作中,我们将扩展DL-HLVD方法以覆盖更多编程语言,并提升其在实际软件项目中的漏洞检测精度和效率.

## 参考文献

[1] 邓泉, 叶蔚, 谢睿, 等. 基于深度学习的源代码缺陷检测研究综述[J]. 软件学报, 2023, 34(2): 625-654.
DENG X, YE W, XIE R, et al. Survey of source code bug detection based on deep learning[J]. Journal of Software, 2023, 34(2): 625-654.(in Chinese)

[2] MERGENDAHL S, BUROW N, OKHRAVI H. Cross-language attacks[C]//Proceedings 2022 Network and Distributed System Security Symposium. San Diego, CA, USA. Internet Society,

2022.

[3] LUO Z H, WANG P F, WANG B S, et al. VulHawk: cross-architecture vulnerability detection with entropy-based binary code search [C]//Proceedings 2023 Network and Distributed System Security Symposium. San Diego, CA, USA. Internet Society, 2023.

[4] PERL H, DECHAND S, SMITH M, et al. VCCFinder [C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, Colorado, USA. ACM, 2015: 426-437.

[5] OLIVEIRA C. Software assurance reference dataset (SARD) [OL]. [Accessed: December-2023]. https://sam ate. nist. gov/SARD/test-suites.

[6] NIKITOPOULOS G, DRITSA K, LOURIDAS P, et al. CrossVul: a cross-language vulnerability dataset with commit data [C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens, Greece. ACM, 2021: 1565-1569.

[7] DWHEELER A. Flawfinder software official website [EB/OL]. [2024-01-04]. https:// dwheeler.com/flawfinder/.

[8] HOSSAIN S, MOHAMMAD Z. Mitigating program security vulnerabilities: approaches and challenges [J]. ACM Computer Survey, 2012, 44(3): 1-46.

[9] VIEGA J, BLOCH J T, KOHNO Y, et al. ITS4: a static vulnera bility scanner for C and C code [C]//Proceedings 16th Annual Computer Security Applications Conference (ACSAC' 00). New Orleans, LA, USA. IEEE, 2002: 257-267.

[10] BASET A Z, DENNING T. IDE plugins for detecting input-validation vulnerabilities [C]//2017 IEEE Security and Privacy Workshops (SPW). San Jose, CA, USA. IEEE, 2017: 143-146.

[11] WALDEN J, DOYLE M. SAVI: static-analysis vulnerability indicator[J]. IEEE Security & Privacy, 2012, 10(3): 32-39.

[12] BALOGLU B. How to find and fix software vulnerabilities with coverity static analysis [C]//2016 IEEE Cybersecurity Development (SecDev). Boston, MA, USA. IEEE, 2016: 153.

[13] AFIFY H, MOHAMMED K, HASSANIEN A. Multi-images recognition of breast cancer histopathological via probabilistic neural network approach [J]. Journal of System and Management Sciences, 2020, 1(2): 53-68.

[14] EKBAL A, SAHA S. Simultaneous feature and parameter selection using multi-objective optimization: application to named entity recognition [J]. International Journal of Machine Learning and Cybernetics, 2020, 7(4): 597-611.

[15] ANBIYA D R, PURWARIANTI A, ASNAR Y. Vulnerability detection in PHP web application using lexical analysis approach with machine learning [C]//2018 5th International Conference on Data and Software Engineering (ICoDSE). Mataram, Indonesia. IEEE, 2018: 1-6.

[16] 杨宏宇, 应乐意, 张良. 基于结构化文本及代码度量的漏洞检测方法[J]. 湖南大学学报(自然科学版), 2022, 49(4): 58-68.
YANG H Y, YING L Y, ZHANG L. Vulnerability detection method based on structured text and code metrics [J]. Journal of Hunan University (Natural Sciences), 2022, 49(4): 58-68. (in Chinese)

[17] RUSSELL R, KIM L, HAMILTON L, et al. Automated vulnerability detection in source code using deep representation learning [C]//2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). Orlando, FL, USA. IEEE, 2018: 757-762.

[18] HARER J A, KIM L Y, RUSSELL R L, et al. Automated software vulnerability detection with machine learning [EB/OL]. 2018: 1803.04497. https://arxiv.org/abs/1803.04497v2.

[19] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection [C]//Proceedings 2018 Network and Distributed System Security Symposium. San Diego, CA. Internet Society, 2018.

[20] LI Z, ZOU D Q, XU S H, et al. SySeVR: a framework for using deep learning to detect software vulnerabilities [J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2244-2258.

[21] ZOU D Q, WANG S J, XU S H, et al. μVulDeePecker: a deep learning-based system for multiclass vulnerability detection [J]. IEEE Transactions on Dependable and Secure Computing, 2021, 18(5): 2224-2236.

[22] 张学军, 张奉鹤, 盖继扬, 等. mVul Sniffer: 一种多类型源代码漏洞检测方法[J]. 通信学报, 2023, 44(9): 149-160.
ZHANG X J, ZHANG F H, GAI J Y, et al. mVul Sniffer: a multi-type source code vulnerability sniffer method [J]. Journal on Communications, 2023, 44(9): 149-160. (in Chinese)

[23] ZHANG X J, ZHANG F H, ZHAO B, et al. VulD-transformer: source code vulnerability detection via transformer [C]// Proceedings of the 14th Asia-Pacific Symposium on Internetware. Hangzhou, China. ACM, 2023: 185-193.

[24] QIN L, YU N, ZHAO D. Applying the convolutional neural network deep learning technology to behavioural recognition in intelligent video[J]. Tehnicki Vjesnik-Techanical Gazette, 2018, 25(2): 528-535.

[25] LI J, HE P J, ZHU J M, et al. Software defect prediction via convolutional neural network [C]//2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). Prague, Czech Republic. IEEE, 2017: 318-328.

[26] AIT HAMMOU B, AIT LAHCEN A, MOULINE S. Towards a real-time processing framework based on improved distributed recurrent neural network variants with fastText for social big data analytics [J]. Information Processing & Management, 2020, 57(1): 102122.

[27] LEUTHÄUSER M. Joern [OL]. [Accessed: May-2023]. https://joern.readthedocs.io/en/latest/.

[28] PARR T, HARWELL S, VERGNAUD E, et al. ANTLR4 [OL]. [Accessed: February-2023] https://github.com/antlr/antlr4.

[29] ARZT S, OLHOTAK, MBENZ89, et al. Soot [OL]. [Accessed: April-2024]. https://github.com/Sable/soot.

[30] 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 [C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). San Diego, CA, USA. IEEE, 2019: 114-121.