文章编号：1674-2974（2022）04-0058-11 DOI：10.16339/j.cnki.hdxbzkb.2022270

基于结构化文本及代码度量的漏洞检测方法

杨宏宇[1,2†]，应乐意[2]，张良[3]
（1. 中国民航大学 安全科学与工程学院，天津 300300；
2. 中国民航大学 计算机科学与技术学院，天津 300300；
3. 亚利桑那大学 信息学院，图森 美国 AZ 85721）

摘 要：目前的源代码漏洞检测方法大多仅依靠单一特征进行检测，表征的维度单一导致方法效率低.针对上述问题提出一种基于结构化文本及代码度量的漏洞检测方法，在函数级粒度进行漏洞检测.利用源代码结构化文本信息及代码度量结果作为特征，通过构造基于自注意力机制的神经网络捕获结构化文本信息中的长期依赖关系，以拟合结构化文本和漏洞存在之间的联系并转化为漏洞存在的概率.采用深度神经网络对代码度量的结果进行特征学习以拟合代码度量值与漏洞存在的关系，并将其拟合的结果转化为漏洞存在的概率.采用支持向量机对由上述两种表征方式获得的漏洞存在概率做进一步的决策分类并获得漏洞检测的最终结果.为验证该方法的漏洞检测性能，针对存在不同类型漏洞的11种源代码样本进行漏洞检测实验，该方法对每种漏洞的平均检测准确率为97.96%，与现有基于单一表征的漏洞检测方法相比，该方法的检测准确率提高了4.89%~12.21%，同时，该方法的漏报率和误报率均保持在10%以内.

关键词：漏洞检测；结构化表征；抽象语法树；代码度量；深度神经网络
中图分类号：TP393 **文献标志码**：A

Vulnerability Detection Method Based on Structured Text and Code Metrics

YANG Hongyu[1,2†], YING Leyi[2], ZHANG Liang[3]
（1. College of Safety Science and Engineering, Civil Aviation University of China, Tianjin 300300, China;
2. College of Computer Science and Technology, Civil Aviation University of China, Tianjin 300300, China;
3. College of Information, University of Arizona, AZ 85721, USA）

Abstract: Most of the current source code vulnerability detection methods only rely on a single feature, and the single dimension of characterization results in inefficient methods. To address the above issues, a vulnerability detection method based on structured text and code metrics is proposed to detect vulnerabilities at the function-level granularity. Using source code structured text information and code metrics as features, long-term dependencies in structured text information are captured by constructing a self-attention based neural network to fit the relationship between structured text and the existence of vulnerabilities and translate them into the probability of vulnerabilities. The deep neural network is used to learn the characteristics of the results of code metrics to fit the relationship be-

* **收稿日期**：2021-08-01
基金项目：国家自然科学基金民航联合研究基金项目（U1833107），Civil Aviation Joint Research Fund Project of National Natural Science（U1833107）
作者简介：杨宏宇（1969—），男，吉林长春人，中国民航大学教授，博士
† 通信联系人，E-mail：yhyxlx@hotmail.com

tween code metrics and the existence of vulnerabilities, and the fitted results are transformed into the probability of vulnerabilities. Support Vector Machine (SVM) is used to further classify the probabilities of vulnerabilities obtained by the above two representations and obtain the final results of vulnerability detection. To verify the vulnerability detection performance of this method, 11 source code samples with different types of vulnerabilities are tested. The average detection accuracy of this method for each vulnerability is 97.96%. Compared with the existing vulnerability detection methods based on a single representation, this method improves the detection accuracy by 4.89%~12.21%, and at the same time. the false positive and false negative rates of this method are kept within 10%.

Key words: vulnerability detection; structured representation; abstract syntax tree; code metrics; deep neural network

计算机软件在各个领域的广泛应用,使得软件漏洞问题也日益严重.面对多样化的软件漏洞类型,如何高效地进行漏洞检测成为当前研究的热点问题.对源代码进行漏洞检测是保障软件安全的有效手段之一.目前,基于代码度量和基于深度学习的方法是较为常见的源代码漏洞检测方法[1].

代码度量[2]被用于描述软件代码特性,以相关定义的数值来描述代码的基本状况.代码度量虽然是一种粗粒度的源代码表征方式,但是在一定程度上可以表征代码的基本状况.基于代码度量的漏洞检测方法通过源代码度量工具对目标代码进行代码度量获取对应指标的数值,利用机器学习算法,经过训练生成漏洞检测器. Ferenc 等[3]基于代码度量应用机器学习算法和网格搜索算法构建漏洞检测模型并采用重采样策略解决训练数据不平衡问题.Sultana[4]利用机器学习和统计学方法追踪代码度量、代码模式和漏洞之间的联系,提出一种漏洞检测方法,并利用该方法对开源软件进行漏洞检测.基于代码度量的漏洞检测方法的主要不足是:①检测粒度粗且可解释性差;②精确率低且误报率高.

随着深度学习在自然语言处理领域的应用,研究人员尝试将深度学习技术应用于编程语言.当前深度学习技术已经普遍应用于信息安全领域[5],基于深度学习的漏洞检测方法能够自主学习代码文本信息与漏洞之间的关联性以此建立漏洞检测模型.Li 等[6]首次将深度学习技术引入漏洞检测领域,提出一种 VulDeePecker 自动化漏洞检测系统,能够对 C/C++语言编写的源代码进行漏洞检测. Saccente 等[7]提出 Achilles 漏洞检测方法,在 Java 源代码上进行测试并取得不错的效果,表明基于深度学习的源代码漏洞检测方法能够应用于多种编程语言.上述两种方法将源代码完全视为线性文本,无法充分表征源代码特征.为更加充分表征编程语言的语法和语义,结构化表征方式被应用于源代码的表征.陈肇炫等[8]提出一种基于结构化表征的智能化漏洞检测系统 Astor,在复杂且语法丰富的数据集中,检测效果优于线性表征方法.基于深度学习的漏洞检测方法能够完全脱离人工干预进行漏洞检测,但仍存在不足.基于深度学习的漏洞检测方法的主要不足有:①需要依赖大量数据进行训练;②对不同类型漏洞的检测结果波动性较大;③精确率和召回率有待提升.

针对计算机软件二进制代码的漏洞检测技术是一种底层的漏洞检测技术.当无法获取软件高级语言源代码时,文献[9]将反编译软件二进制代码作为特征,应用深度学习技术进行特征学习并构造漏洞检测模型,得到了较好的检测性能.文献[10]通过计算二进制函数和漏洞二进制函数特征库的相似度进行漏洞检测,该方法通过大量的训练后的模型准确率得到提升.文献[11]在经典代码切片技术的基础上改善二进制代码过程间切片方式及切片粒度,使得检测精度和效率有所提升.在无法获取源代码的情况下,利用二进制代码依旧可以进行漏洞检测并且有较好的检测能力.但是基于二进制代码的检测方法并不直观且表征方式单一.

上述工作在源代码表征方式上,均采用了单一的表征方式,无法全面表征源代码,因此导致检测效果不佳.针对上述检测方法表征方式单一导致检测效果不佳的问题,为进一步提升漏洞检测效果,本文提出一种基于结构化文本及代码度量的漏洞检测方法,在现有研究的基础上从表征方法和特征拟合两个方面做出改进,在函数级粒度上对源代码进行漏洞检测.首先,深度优先遍历源代码抽象语法树得到

结构化文本信息,使用源代码静态解析工具获取代码度量值;其次,通过构造基于自注意力机制的神经网络捕获结构化文本信息中的长期依赖关系,以拟合结构化文本和漏洞存在之间的联系并转化为漏洞存在的概率,采用深度神经网络对代码度量的结果进行特征学习以拟合代码度量值与漏洞存在的关系,并将其拟合的结果转化为漏洞存在的概率;最后,采用支持向量机对由上述两种表征方式获得的漏洞存在概率做进一步的决策分类,并获得漏洞检测的最终结果.在对比实验中,针对存在不同类型漏洞的11种源代码样本进行漏洞检测实验,本文方法对每种漏洞的平均检测准确率为97.96%.与现有基于单一表征的漏洞检测方法相比,本文方法的检测准确率提高了4.89%~12.21%,同时,本文方法的漏报率和误报率均保持在10%以内.

1 源代码漏洞检测方法设计

1.1 方法设计思路

本文方法从结构化文本信息和代码度量两个维度对源代码进行表征,利用表征结果和预设的标签对构造的神经网络进行训练.训练完成的神经网络模型即为漏洞检测模型,应用漏洞检测模型对待检测源代码进行漏洞检测得到检测结果.本文方法中4个部分的设计思路如下.

1)数据预处理.为生成符合本文漏洞检测粒度的训练集和测试集,在本阶段对原始数据集以函数级粒度进行切片并设置监督学习标签.预处理阶段的输出为代码的函数切片以及对应的监督学习标签.

2)数据表征.为充分表现源代码特征,从结构化文本信息和代码度量两个维度对预处理后的数据进行表征.为从源代码结构化文本信息角度表征源代码,利用AST作为中间载体,采用深度优先遍历机制收集源代码文本特征并转化为向量形式;为从代码度量角度表征源代码,需要定义代码度量指标,通过源代码静态解析工具获取对应的度量值.本阶段的输出为向量形式的结构化文本信息以及代码度量值序列.

3)模型构建及训练.为拟合数据表征的结果和漏洞存在之间的关系,构建合适的神经网络模型对表征结果进行特征学习.构建的神经网络模型能对结构化文本信息和代码度量值进行特征学习,综合两种特征给出漏洞检测结果.最后,采用表征结果和预设的标签对模型进行训练,本阶段的输出为训练完成的漏洞检测模型.

4)源代码漏洞检测.为在本阶段应用训练完成的模型进行漏洞检测,对待检测源代码进行特征提取,提取特征的方式与表征方式相同.将提取到的特征输入训练完成的模型中,输出漏洞检测的结果.

1.2 方法架构设计

本文提出的源代码漏洞检测模型由数据预处理、数据表征、模型搭建及训练、源代码漏洞检测4个部分组成,该方法的核心框架如图1所示.该漏洞检测模型的4个部分的主要处理过程为

1)数据预处理.数据预处理阶段包括代码切片和设置监督学习标签两个部分.本文方法在函数级粒度进行漏洞检测,因此需将源代码数据切分为函数片段并根据函数片段是否存在漏洞设置标签.

2)数据表征.为充分表征函数片段的信息,分别从结构化文本信息和代码度量两种维度对预处理后

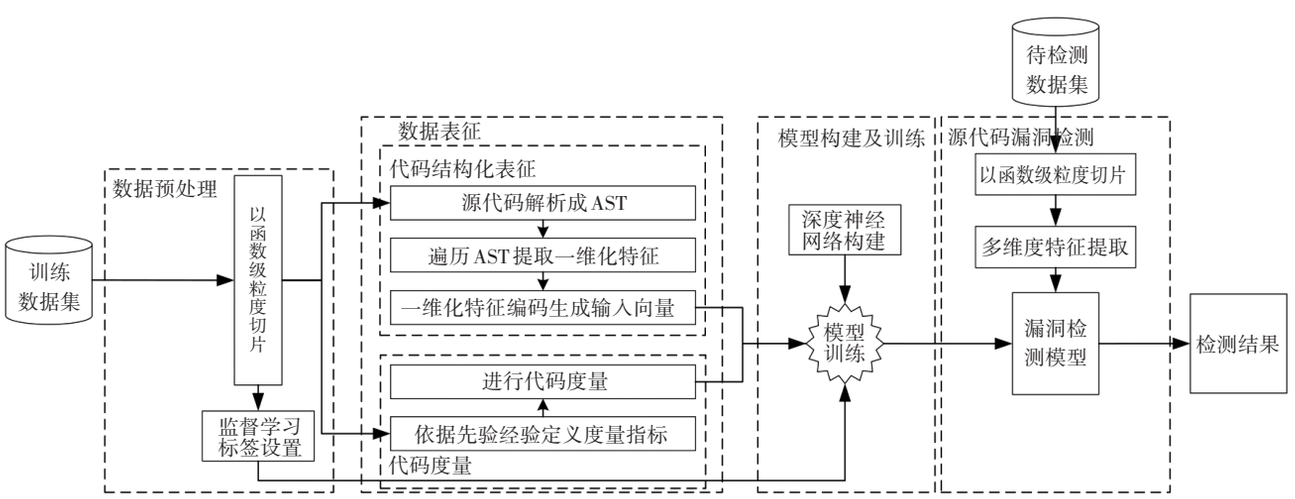


图1 本文漏洞检测方法框架

Fig.1 Framework for vulnerability detection in this paper

的数据进行表征.利用抽象语法树(Abstract Syntax Tree, AST)表征函数片段的文本信息.定义代码度量指标对函数片段进行代码度量.

3)模型构建及训练.构建一种神经网络,由该神经网络针对两种维度表征结果的数据类型进行特征学习.利用两种表征结果以及预设的标签对神经网络进行训练以构造漏洞检测模型.

4)源代码漏洞检测.利用训练完成的漏洞检测模型对待检测源代码进行漏洞检测.待检测源代码的预处理和表征方式与训练数据相同,将表征结果输入训练完成的漏洞检测模型得到检测结果.

2　数据处理及表征

2.1　数据预处理

本文采用的数据集为美国国家标准与技术研究院的Juliet Test Suite数据集[12],该数据集包含118种CWE[13]类型的28 881个Java文件.由于基于深度学习的方法对于数据量的需求较大,所以在研究中选取测试用例超过1 000例的漏洞类型.虽然在本文研究中以Java语言源代码作为研究对象,但本文方法并不受编程语言类型限制,只要被检测程序的源代码能够进行结构化表征和代码度量,本文方法依然适用.

为生成符合本文检测粒度的训练集和测试集,需要对收集的数据进行预处理.数据预处理阶段包括代码切片和监督学习标签设置两部分.

2.1.1　代码切片

本文方法的检测粒度是函数级别,所以对需要表征的源代码按函数进行切片.漏洞源代码用例如表1所示.

代码切片可以从Java文件中分离出不含空行和注释的函数代码,代码切片的具体过程如下.

1)源代码清洗.为提升源代码的信息密度,防止无用信息被表征,以字符串匹配的方式消除代码中的空行和注释.

2)函数切片.利用Java静态解析工具Javalang[14]解析Java源文件获得类中包含的所有函数并存储在列表中.

2.1.2　监督学习标签设置

判断源代码函数是否存在漏洞是一个典型的二分类问题.本文针对漏洞检测设计的神经网络是一个二分类监督学习模型,因此需要对训练数据设置标签.

Juliet Test Suite数据集中,已经在函数名称上标注了标记“good”(无漏洞)或“bad”(有漏洞).采用字符匹配的方法匹配函数名称中的标记,标记为“good”的函数片段设置标签为“0”,标记为“bad”的函数片段设置标签为“1”.由于函数名称也会作为文本信息被表征,为了不使上述标记影响模型的训练效果,依据标记添加标签后,将其用随机字符替代.

表1　漏洞源代码用例

Tab.1　Vulnerability source code case

CWE# 类型	名称	测试用例 文件数量	函数切片 数量
CWE80	基础跨站脚本攻击	1 080	3 168
CWE89	SQL注入漏洞	3 660	15 000
CWE113	HTTP响应拆分攻击	2 196	9 000
CWE129	数组索引验证不当	4 392	11 103
CWE134	被污染的格式化字符串	1 098	4 500
CWE190	整数上溢	7 015	28 750
CWE191	整数下溢	5 612	23 000
CWE197	数值截断错误	1 980	5 808
CWE369	除数为零	3 050	12 500
CWE400	资源耗尽	2 396	9 814
CWE789	内存分配失控	2 537	7 456

2.2　数据表征

为充分表现源代码特征,从代码结构化表征和代码度量两个不同维度对源代码进行表征.代码结构化表征可以获得代码结构化的文本信息,代码度量能够表征代码的基本状况.

2.2.1　代码结构化表征

编程语言是一种结构化的语言,源代码中的信息有明确的结构关系.因此表征自然语言的方法并不能充分表征源代码中的语法和语义.为了得到更贴合实际的源代码特征,采用结构化表征方法对源代码进行表征.结构化表征方法包括以下三个步骤.

步骤1:利用Java源代码解析工具javalang解析代码,得到抽象语法树节点和边的信息,根据节点和边的信息生成抽象语法树.

步骤2:深度优先遍历抽象语法树,依次收集节点信息.深度优先遍历抽象语法树的结果使得树形数据转化为一维文本数据.

步骤3:将一维文本数据转化为神经网络的输入.由于神经网络的输入是向量形式的数据,因此需要进一步处理一维的文本数据.首先对文本数据作

分词处理,然后通过统计方法生成词典,根据词典将文本表示为向量.

2.2.2 代码度量

本文方法旨在对函数级别的源代码进行漏洞检测,因此需要在代码函数级别上进行度量.为使完全依赖数据的深度学习方法能与安全专家的先验知识进行有效交互,并使检测模型的自适应性更强,在代码度量中需要人工参与定义代码度量指标.本文方法中的代码度量处理过程包括2个步骤.

步骤1:度量指标定义.对代码度量的指标进行定义,在代码度量阶段使用的主要度量指标是Chidamber&Kemerer指标[15],与传统的McCabe指标和Halstead metrics指标相比,Chidamber&Kemerer指标是专门针对面向对象程序语言提出的,故对Java语言的适应性更强.具体的度量指标如表2所示,其中包含函数和函数所在类的相关信息.

步骤2:代码度量.使用代码度量工具[16]进行代码度量可得到表2所示指标的具体量化数值.

表2 代码度量指标

Tab.2 Code metrics

指标名	备注
CBO(Coupling between objects)	类耦合
Wmc(Weight Method Class)	类方法加权
Rfc(Response for a Class)	类响应
Quantity of returns	返回值
variablesQty	声明变量
parametersQty	输入参数
methodsInvokedQty	函数被调用次数
methodsInvokedLocalQty	调用函数次数
loopQty	循环数
comparisonsQty	分支判别
tryCatchQty	异常捕获
parenthesizedExpsQty	括号表达式
stringLiteralsQty	字符串文本变量
numbersQty	数值型变量
mathOperationsQty	操作符
maxNestedBlocksQty	块嵌套
anonymousClassesQty	匿名内部类
innerClassesQty	内部类
lambdasQty	Lambda表达式
max_cc	最大圈复杂度
avg_cc	平均圈复杂度

3 深度神经网络的构建与训练

3.1 神经网络总体框架

对源代码的表征结果分别为结构化的文本信息和代码度量产生的数字序列.因此需要设计神经网络对结构化文本信息和数字序列进行特征学习,并综合二者判断结果给出最终的漏洞检测结果.

在本文方法中构建的神经网络模型有三个部分:①基于自注意力(Self-Attention,SA)机制[17]的神经网络;②深度神经网络(Deep Neural Networks,DNN);③支持向量机(Support Vector Machine,SVM).该神经网络的主要结构如图2所示.

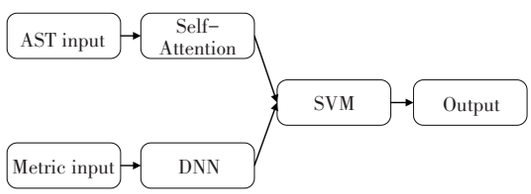


图2 神经网络总体框架

Fig.2 Framework of neural network

其中,基于SA的神经网络模型用于文本序列的特征学习,DNN模型用于代码度量结果的特征学习,SVM模型用于对上述两个模型的输出结果进行处理和分类并得到最终的漏洞检测结果.

3.2 基于SA机制的神经网络模型构建

分析文本数据最重要的目的是捕获其中的长期依赖关系,这种依赖关系在编程语言中尤为关键.受自然语言语法和人类文字编辑习惯的影响,自然语言中的依赖关系在时间跨度上是有限的.但是这种依赖关系在编程语言中的时间跨度是不受控制的,例如定义的变量或函数,在代码中的任意位置都可能被调用.因此在对源代码结构化文本漏洞检测时,通过SA机制解决依赖问题.

SA结构如图3所示,对于每一个输入的词向量$\boldsymbol{x}_i$,SA将其表示为向量$\boldsymbol{q}_i$、$\boldsymbol{k}_i$、$\boldsymbol{v}_i$.为获取这3个向量,分别定义3个不同的权值矩阵$\boldsymbol{W}^Q$、$\boldsymbol{W}^K$、$\boldsymbol{W}^V$,这3个矩阵在训练阶段通过反向传播算法不断更新优化.将权值矩阵与输入矩阵$\boldsymbol{X}=[\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{x}_3,\cdots,\boldsymbol{x}_n]$相乘来获得对应的向量集和,计算方式如公式(1)~公式(3)所示.

$$\boldsymbol{Q}[\boldsymbol{q}_1,\boldsymbol{q}_2,\boldsymbol{q}_3,\cdots,\boldsymbol{q}_i]=\boldsymbol{X}\otimes\boldsymbol{W}^Q \tag{1}$$

式中:$\boldsymbol{X}$为输入词向量$\boldsymbol{x}_i$组成的矩阵,$\boldsymbol{W}^Q$为对应的权值矩阵,$\boldsymbol{Q}$是由向量$\boldsymbol{q}_i$组成的矩阵.

$$\boldsymbol{K}[\boldsymbol{k}_1,\boldsymbol{k}_2,\boldsymbol{k}_3,\cdots,\boldsymbol{k}_i]=\boldsymbol{X}\otimes\boldsymbol{W}^K \tag{2}$$

式中:$\boldsymbol{X}$为输入词向量$\boldsymbol{x}_i$组成的矩阵,$\boldsymbol{W}^K$为对应的权值矩阵,$\boldsymbol{K}$是由向量$\boldsymbol{k}_i$组成的矩阵.

$$\boldsymbol{V}[\boldsymbol{v}_1,\boldsymbol{v}_2,\boldsymbol{v}_3,\cdots,\boldsymbol{v}_i]=\boldsymbol{X}\otimes\boldsymbol{W}^V \tag{3}$$

式中:$\boldsymbol{X}$为输入词向量$\boldsymbol{x}_i$组成的矩阵,$\boldsymbol{W}^V$为对应的权值矩阵,$\boldsymbol{V}$是由向量$\boldsymbol{v}_i$组成的矩阵.

SA的计算结果为:

$$\boldsymbol{Z}=\text{soft max}\left(\frac{\boldsymbol{Q}\times\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{4}$$

式中:d_k为尺度标度,与向量$\boldsymbol{q}_i$的维度相等;$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$分别为公式(1)~公式(3)的计算结果,是输入矩阵$\boldsymbol{X}$的三种不同的表示形式.

在公式(4)中,$\boldsymbol{Q}$和$\boldsymbol{K}$相乘的结果用于反映每个词与其他词的相关程度,但是这个结果会随着词向量维度的增加而不断增大.如果$\boldsymbol{Q}$和$\boldsymbol{K}$相乘的结果非常大,会造成softmax结果无限接近1,会使得梯度较小,从而影响参数的更新.因此需要利用d_k约束计算结果的大小.softmax能够计算词与词的关联程度在句子中的比重,softmax的结果再与$\boldsymbol{V}$相乘,相当于一个加权求和的结果,这个结果可以反映每个词对于句子的贡献程度.在本文研究中,这个贡献程度表示这个词与漏洞存在的关联程度.

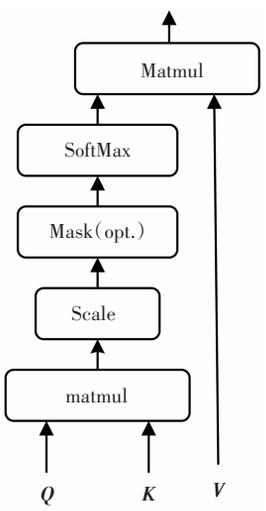


图3　SA结构

Fig.3　SA structure

由上述计算过程可知,SA机制能够反映文档内部每个词和其他所有词的直接交互情况,对比循环神经网络(Recurrent Neural Network, RNN)需要按照序列逐步累计计算来获得文本信息中的长距离相互依赖关系,SA机制能更好地捕捉文本信息长期依赖关系.所以相较于传统RNN,SA更适合进行结构化文本特征学习任务.

本文构建的基于SA的神经网络由输入层、SA层、全连接层、输出层构成,其中全连接层由128个神经元组成.由于SA中的计算都是线性计算,加入全连接层以拟合非线性特征.为通过文本特征得到漏洞存在的概率,输出层以Sigmoid作为激活函数.Sigmoid函数如公式(5)所示.

$$\text{Sigmoid}(x)=\frac{1}{1+\mathrm{e}^{-x}} \tag{5}$$

Sigmoid函数能将神经网络的输出映射到[0,1]之间,能将学习到的文本特征转化为漏洞存在的概率.通过源代码结构化表征结果和预设的标签对基于SA的神经网络进行训练,在训练完成的神经网络中输入源代码结构化文本信息,即可输出对应源代码存在漏洞的概率.

3.3　DNN模型构建

代码度量的结果是一段数字序列,序列中的每个元素表示对应度量指标的具体数值,并且度量结果各个元素之间不存在相互依赖关系.基于上述应用场景,DNN相较于传统机器学习算法能够在较短的时间内学习到序列特征.因此应用DNN进行代码度量的特征学习,其结构如图4所示,隐藏层的神经元个数均为64.

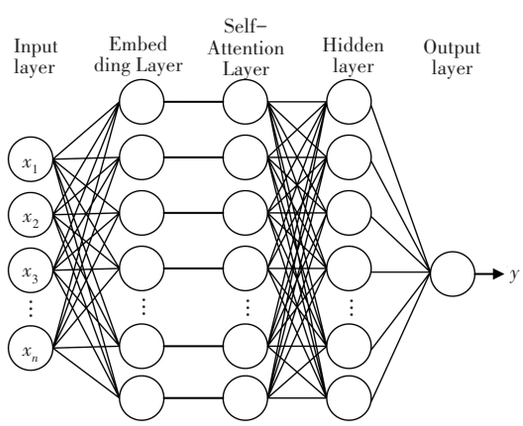


图4　DNN结构

Fig.4　DNN structure

对输入的代码度量结果,在经过两层隐藏层拟合代码度量特征后,利用Sigmoid函数作为激活函数将输出结果映射到[0,1]之间.利用代码度量结果和预设的标签对DNN进行训练,在训练完成的DNN模型中输入代码度量结果即可输出对应源代码存在漏洞的概率.

3.4　SVM模型构建

本文漏洞检测模型以学习文本序列特征的基于SA的神经网络和学习代码度量特征的DNN为基础

构建.在完成上述模型的训练后,能够应用这两种模型分别从文本信息和代码度量两个维度判断代码是否存在漏洞.为得到更加精确的漏洞检测结果,需要综合上述两种模型的输出结果.因此本文以上述两种模型的输出作为特征,应用SVM作进一步分类,判断代码是否存在漏洞.

在获取两种模型输出的漏洞存在概率后,需要应用分类算法对输出进行分类以尽可能消除两种表征方式判断出现分歧的部分.因此,在应用分类算法后,检测效果会得到明显提升.

选择SVM作为这一阶段的分类器主要有以下两个原因:①SVM在分类任务中效果好,并且分类思想简单直观,能够准确绘制其决策边界,以直观表现本文方法的可行性;②分类方式灵活,可以通过调整其核函数进行线性分类和非线性分类.由于无法提前判断两种模型的输出是否是线性可分的,SVM的分类方式相较于其他分类算法更加适合作为本阶段的分类器.

常规的SVM通过绘制最大间隔的超平面进行分类,但这种方法无法进行非线性分类.由于基于SA的神经网络和DNN的输出结果可能出现线性不可分的情况,设置SVM的核函数以进行非线性分类.本方法构建的SVM模型利用线性核(linear)、多项式核(poly)和高斯核(rbf)对基于SA的神经网络和DNN的输出结果进行分类.在训练完成的SVM模型中,输入基于SA的神经网络和DNN输出的漏洞存在概率,输出漏洞检测的最终结果.

4 实验设计及结果分析

4.1 评价指标

本文采用准确率、精确率、召回率、*F*1-Score、误报率、漏报率6个指标对提出的漏洞检测模型进行评价.为计算上述6个评价指标,需要在实验中收集以下4种数据:①真正类(True Positive, TP)即被正确分类的有漏洞样本数量;②假正类(False Positive, FP)即不含漏洞样本被误报的数量;③假负类(False Negative, FN)即未被成功检测的漏洞样本数量;④真负类(True Negative, TN)即不存在漏洞的样本被准确判断的数量.6个评价指标的定义如下.

1)准确率*A*:准确分类的样本占总样本的比例.

$$A = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FP} + \mathrm{TN} + \mathrm{FN}} \tag{6}$$

2)精确率*P*:在所有被判断为存在漏洞的样本中,判断正确的样本比例.

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{7}$$

3)召回率*R*:被成功检测出的漏洞样本占所有漏洞样本的比例.

$$R = \frac{\mathrm{TP}}{\mathrm{FN} + \mathrm{TP}} \tag{8}$$

4)*F*1-Score:精确率和召回率的调和平均值,反映模型整体表现情况.

$$F1 - \mathrm{Score} = \frac{2 \times P \times R}{P + R} \tag{9}$$

5)误报率FPR:无漏洞样本被误报的比例.

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{10}$$

6)漏报率FNR:漏洞样本中未被检测出的样本所占比例,FNR=1-*R*.

4.2 实验与结果分析

为验证本文方法的性能,将本文方法与基于源代码文本结构化表征的漏洞检测方法[8]、基于文本线性表征的漏洞检测方法[7]和基于代码度量的漏洞检测方法[3]进行对比实验.具体实验环境配置如表3所示.

表3 实验环境配置

Tab.3 Experimental environment configuration

环境配置	参数
CPU	Intel(R) Xeon(R) Silver 4110 CPU 2.10 GHz
GPU	NVIDIA Quadro P2000
磁盘	2TB
内存	32GB
软件	Keras2.2.2、Tensorflow1.10.0、Python3.5、Scikit-learn0.20.0
操作系统	Windows10 18363.1440

4.2.1 检测模型构建及性能评估

本文提出的基于结构化文本及代码度量的漏洞检测方法,综合了源代码文本的结构化表征和代码度量两种表征方式.因此模型的构造需要分三步进行:①基于SA的神经网络训练及测试;②DNN模型训练及测试;③SVM模型训练及测试,其中SVM模型的输出结果是本文检测方法的最终结果.训练及测试所需数据集如表1所示,将表1中收集的测试用例进行分割,得到训练集和测试集.

1)基于SA的神经网络训练及测试.本文利用基于SA的神经网络进行代码结构化文本特征的学习.为验证SA机制在捕获代码结构化文本长期依赖能

力优于其他神经网络,与其他4种神经网络进行对比实验,分别为:CNN、LSTM、BLSTM、GRU.对上述模型进行训练,经过测试分别得到5种模型的性能指标,实验结果如图5~图10所示.

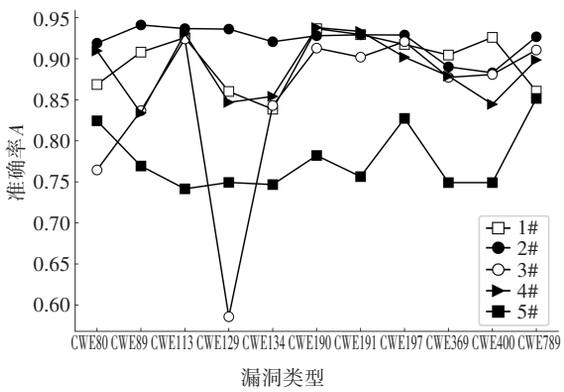


图5　五种神经网络准确率A对比结果

Fig.5　Accuracy comparison of five neural networks

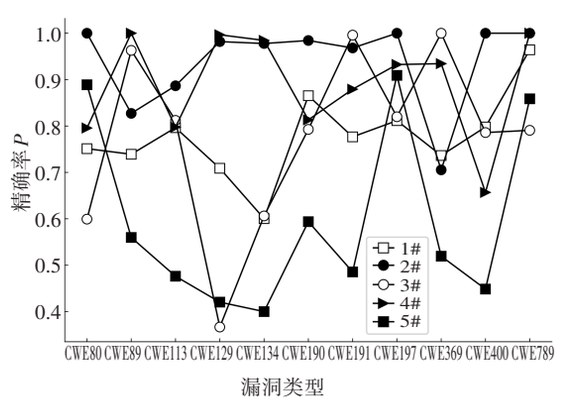


图6　五种神经网络精确率P对比结果

Fig.6　Precision comparison of five neural networks

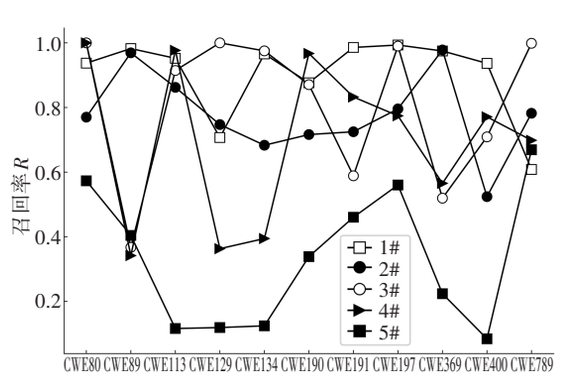


图7　五种神经网络召回率R对比结果

Fig.7　Recall comparison of five neural networks

由图5~图10可见,基于SA的神经网络在测试集上的准确率和精确率较高且误报率较低,这说明与采用其他4种神经网络模型相比,基于SA的神经网络模型对源代码的结构化文本特征拟合效果更好.并且基于SA的神经网络$F1$-Score保持在较高的水平,这说明该神经网络在利用结构化文本特征进行漏洞检测时的整体表现优于其他模型.由图6~图11的曲线趋势可见,基于SA的神经网络在面对不同漏洞类型的表现也较为稳定.综上可知,基于SA的神经网络能够充分拟合源代码结构化文本和漏洞

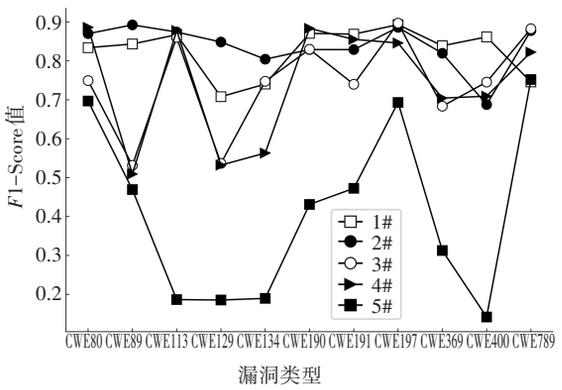


图8　五种神经网络$F1$-Score对比结果

Fig.8　$F1$-Score comparison of five neural networks

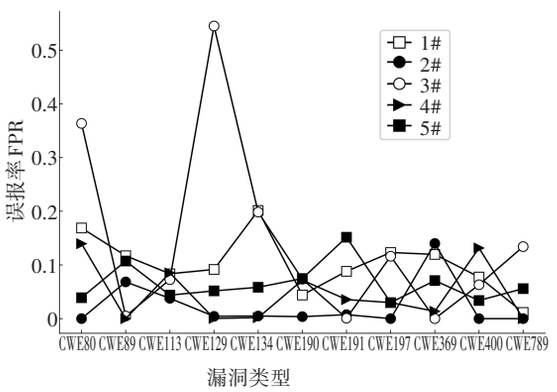


图9　五种神经网络误报率FPR对比结果

Fig.9　FPR comparison of five neural networks

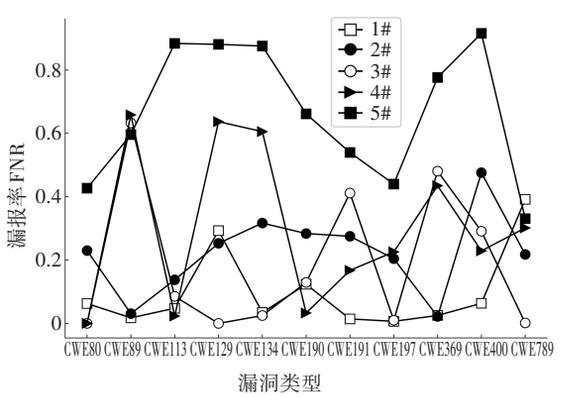


图10　五种神经网络漏报率FNR对比结果

Fig.10　FNR comparison of five neural networks

存在之间的联系,比其他神经网络更加适合基于文本结构化表征的漏洞检测任务.

2)DNN 测试及训练.针对代码度量特征,采用DNN 构建一个漏洞检测模型,其在测试数据上的实验结果如表4所示.

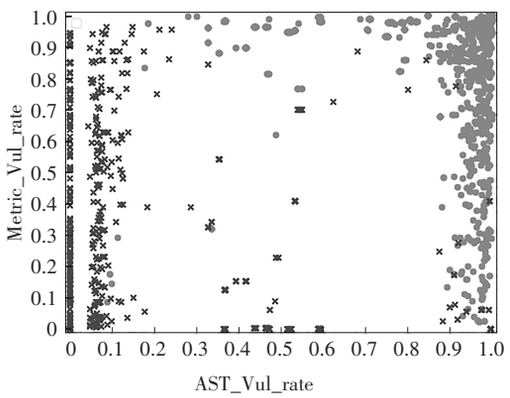
表4 基于代码度量方法的实验结果

Tab.4 Experimental results based on code metrics

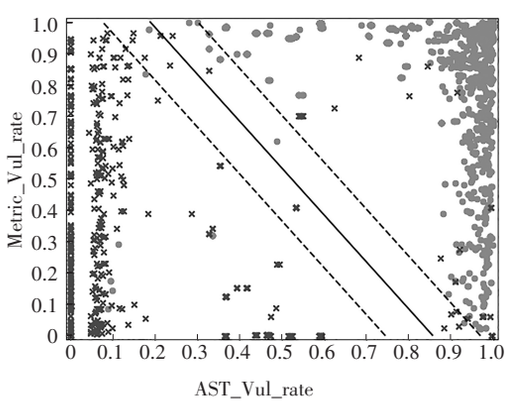
CWE 类型	*A*	*P*	*R*	*F*1-Score	FPR	FNR
CWE80	0.930 6	0.952 9	0.844 8	0.895 6	0.022 7	0.155 2
CWE89	0.786 2	0.645 5	0.339 8	0.445 2	0.063 0	0.660 2
CWE113	0.827 0	0.749 4	0.485 5	0.589 3	0.055 7	0.514 5
CWE129	0.819 3	0.831 6	0.309 1	0.450 7	0.019 7	0.690 9
CWE134	0.842 2	0.674 1	0.655 3	0.664 6	0.099 2	0.344 7
CWE190	0.910 3	0.918 2	0.693 9	0.790 5	0.019 9	0.306 1
CWE191	0.917 8	0.928 6	0.707 8	0.803 3	0.016 91	0.292 2
CWE197	0.861 2	0.948 4	0.635 9	0.761 3	0.018 5	0.364 1
CWE369	0.876 8	0.941 2	0.551 7	0.695 7	0.011 8	0.448 3
CWE400	0.882 2	0.852 6	0.630 3	0.724 8	0.035 6	0.369 7
CWE789	0.778 7	0.988 6	0.346 6	0.513 3	0.002 0	0.653 4

实验结果表明,基于代码度量的方式虽然准确率较高,但漏报率极高.例如,在对CWE129的检测准确率达到81.93%的前提下,漏报率达到69.09%.表明代码度量表征方法对存在漏洞的代码表征效果不好,导致检测结果出现偏差.可见,采用这种粗粒度的表征方式只适用于粗略判断源代码是否存在漏洞,不能准确检测源代码的漏洞.因此代码度量在一定程度上能够判断代码的健康状况,但仅依靠代码度量不能充分表示漏洞代码的特性.

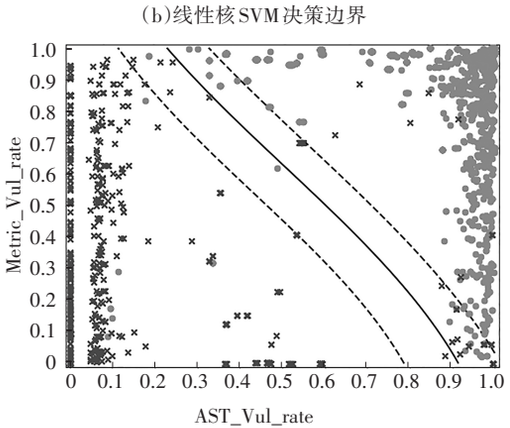
3)SVM模型训练及测试.在前序实验中,通过对结构化文本特征和代码度量特征训练,得到2种不同维度的检测模型.在本实验中,将测试数据输入2种检测模型分别得出源代码存在漏洞的概率,以2个神经网络检测模型的输出作为新的特征,采用SVM进行决策分类,得到最终检测结果,即判断漏洞是否存在.通过调整核函数利用SVM进行线性分类和非线性分类.本文分别使用线性核(linear)、多项式核(poly)和高斯核(rbf)对上述两种检测模型的输出作进一步分类.以CWE113漏洞为例说明分类过程,不同核函数的SVM决策边界如图11所示.



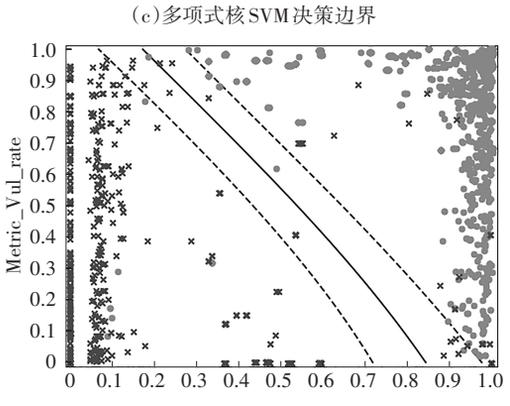
(a)SA和DNN检测模型输出散点图



(b)线性核SVM决策边界



(c)多项式核SVM决策边界



(d)高斯核SVM决策边界

图11 CWE113测试数据的决策边界

Fig.11 Decision boundary for CWE113 test data

图 11(a)是基于SA的检测模型和DNN检测模型的输出散点图,图 11(a)~图 11(d)中的每一个点表示测试数据集中的一个函数片段,圆点表示该函数真实存在漏洞,“×”点表示该函数不存在漏洞.图中横坐标表示基于结构化文本特征的检测模型输出的漏洞存在概率,纵坐标表示基于代码度量的检测模型输出的漏洞存在概率.例如靠近右上角的点表示基于SA的检测模型和DNN检测模型都判断该函数有很大概率存在漏洞.图 11(b)~图 11(d)分别表示SVM中3种不同核函数的决策边界.SVM模型的具体评估结果如表5~表7所示.

表5　线性核SVM分类结果

Tab.5　Linear kernel SVM classification results

CWE	*A*	*P*	*R*	*F*1-Score	FPR	FNR
CWE80	0.986 3	0.990 9	0.970 1	0.980 4	0.004 9	0.029 9
CWE89	0.977 1	0.933 7	0.978 9	0.955 7	0.023 5	0.021 1
CWE113	0.977 8	0.981 7	0.930 4	0.955 4	0.006 0	0.069 6
CWE129	0.979 0	0.956 2	0.956 2	0.956 2	0.013 8	0.043 8
CWE134	0.937 0	0.950 6	0.776 4	0.854 7	0.012 6	0.223 6
CWE190	0.990 1	0.987 0	0.972 4	0.979 7	0.004 1	0.027 6
CWE191	0.990 9	0.982 8	0.978 6	0.980 7	0.005 3	0.021 4
CWE197	0.990 8	0.983 6	0.990 1	0.986 9	0.008 8	0.009 9
CWE369	0.978 9	0.968 0	0.948 8	0.958 3	0.010 7	0.051 2
CWE400	0.967 7	0.966 0	0.900 7	0.932 2	0.010 4	0.099 3
CWE789	0.992 4	0.989 4	0.988 0	0.988 7	0.005 4	0.012 0

表6　多项式核SVM分类结果

Tab.6　Polynomial kernel SVM classification results

CWE#	*A*	*P*	*R*	*F*1-Score	FPR	FNR
CWE80	0.988 4	0.988 0	0.979 1	0.983 5	0.006 5	0.020 9
CWE89	0.976 9	0.921 6	0.993 0	0.955 9	0.028 5	0.007 0
CWE113	0.978 9	0.980 3	0.936 2	0.957 7	0.006 5	0.063 8
CWE129	0.979 9	0.944 2	0.973 7	0.958 7	0.018 2	0.026 3
CWE134	0.940 0	0.944 6	0.795 0	0.863 4	0.014 6	0.205 0
CWE190	0.991 5	0.986 6	0.978 6	0.982 6	0.004 3	0.021 4
CWE191	0.991 0	0.979 3	0.982 9	0.981 1	0.006 5	0.017 1
CWE197	0.993 1	0.980 6	1.000 0	0.990 2	0.010 6	0.000 0
CWE369	0.975 5	0.964 6	0.938 3	0.951 3	0.011 8	0.061 7
CWE400	0.968 8	0.948 9	0.922 8	0.935 7	0.016 2	0.077 2
CWE789	0.992 4	0.989 4	0.988 0	0.988 7	0.005 4	0.012 0

表7　高斯核SVM分类结果

Tab.7　Gauss kernel SVM classification results

CWE#	*A*	*P*	*R*	*F*1-Score	FPR	FNR
CWE80	0.986 3	0.990 9	0.970 1	0.980 4	0.004 9	0.029 9
CWE89	0.980 0	0.945 5	0.977 1	0.961 0	0.019 0	0.022 9
CWE113	0.977 4	0.981 6	0.929 0	0.954 6	0.006 0	0.071 0
CWE129	0.981 4	0.956 6	0.966 2	0.961 4	0.013 8	0.033 8
CWE134	0.937 0	0.957 5	0.770 2	0.853 7	0.010 7	0.229 8
CWE190	0.989 9	0.982 3	0.976 2	0.979 3	0.005 7	0.023 8
CWE191	0.991 0	0.982 8	0.979 2	0.981 0	0.005 3	0.020 8
CWE197	0.990 8	0.983 6	0.990 1	0.986 9	0.008 8	0.009 9
CWE369	0.980 3	0.971 2	0.950 9	0.960 9	0.009 7	0.049 1
CWE400	0.968 8	0.958 0	0.913 1	0.935 0	0.013 1	0.086 9
CWE789	0.992 8	0.990 7	0.988 0	0.989 4	0.004 7	0.012 0

由表5~表7可见,经过SVM的进一步分类决策,漏洞检测的各项指标均有大幅提升,但是对于不同核函数的SVM分类结果相差不大.出现这种现象的原因是,在CWE113测试数据中,基于SA的神经网络和DNN的输出是线性可分的.但是本文方法在应用过程中,由于漏洞类型的多样性和源代码的复杂性,无法保证基于SA的神经网络和DNN的输出都是线性可分的,因此本文方法采用三种不同核函数进行分类.对比表4~表7以及图5~图10中的数据可见,本文方法在结构化表征方法和代码度量方法的基础上,提高了准确率、精确率和召回率.

4.2.2　对比检测实验

为验证本文提出方法的优越性,将本文方法和基于文本结构化表征的漏洞检测方法[8]、基于代码度量的漏洞检测方法[3]、基于线性文本表征的漏洞检测方法Achilles[7]进行对比实验.其中基于结构化文本的方法采用的是基于SA机制的神经网络模型、基于代码度量的方法采用DNN模型、基于线性文本的方法采用LSTM模型.分别搭建上述4种检测模型,在相同数据集下进行训练和测试,4种模型的漏洞检测准确率对比结果如图12所示.

从图12可见,基于代码度量的方法、基于结构化表征的方法和Achilles对不同漏洞的检测平均准确率分别为85.75%、93.07%和92.18%,本文方法对不同漏洞的检测平均准确率为97.96%,均高于其他3种方法.本文方法能够取得较好的漏洞检测效果,有以下两个原因:①本文方法从源代码结构文本信息以及代码度量两个维度对源代码进行表征,相比

于单一表征方法,本文的表征方法更加全面;②文本信息特征是漏洞检测过程中较为重要的特征,本文所构建的基于SA的神经网络,能够较好地捕捉文本信息中的长期依赖关系.

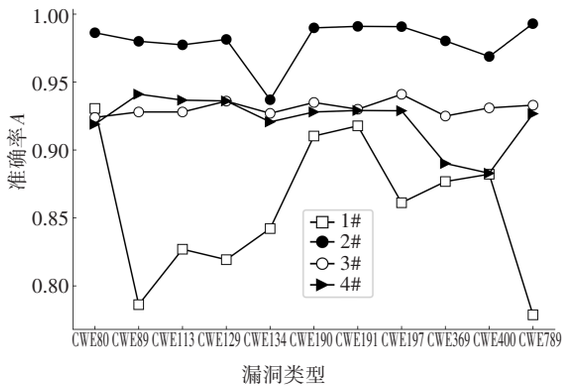


图12 4种模型的漏洞检测准确率对比结果

Fig.12 Accuracy comparison results of four vulnerability detection methods

5 结束语

为进一步提高源代码漏洞检测准确率,降低误报率,本文提出一种基于结构化文本及代码度量的漏洞检测方法.通过代码度量和结构化文本两种表征方法对源代码进行表征,利用神经网络模型进行特征学习以构造漏洞检测模型,进行漏洞检测.实验结果表明本文提出的方法有较好的检测效果.

本文方法仅从两个维度对源代码进行表征,考虑的表征维度仍不够全面.未来的工作重点是发掘更多适合漏洞检测的源代码表征方式,改进表征方式以获得更优的检测性能.

参考文献

[1] LI Z J, SHAO Y. A survey of feature selection for vulnerability prediction using feature-based machine learning[C]//Proceedings of the 2019 11th International Conference on Machine Learning and Computing-ICMLC'19. New York:ACM Press,2019:36-42.

[2] HORCH J W. Metrics and models in software quality engineering [J]. Control Engineering Practice,1996,4(9):1333-1334.

[3] FERENC R, HEGEDŰS P, GYIMESI P, *et al*. Challenging machine learning algorithms in predicting vulnerable JavaScript functions[C]//2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE). Montreal,QC,Canada: IEEE,2019:8-14.

[4] SULTANA K Z. Towards a software vulnerability prediction model using traceable code patterns and software metrics[C]//2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Urbana,IL,USA: IEEE,2017:1022-1025.

[5] YANG H Y,ZENG R Y,XU G Q,*et al*. A network security situation assessment method based on adversarial deep learning[J]. Applied Soft Computing,2021,102:107096.

[6] LI Z,ZOU D Q,XU S H,*et al*. VulPecker: an automated vulnerability detection system based on code similarity analysis[C]//Proceedings of the 32nd Annual Conference on Computer Security Applications. New York:ACM,2016:201-213.

[7] 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.

[8] 陈肇炫,邹德清,李珍,等. 基于抽象语法树的智能化漏洞检测系统[J]. 信息安全学报,2020,5(4):1-13.
CHEN Z X,ZOU D Q,LI Z,*et al*. Intelligent vulnerability detection system based on abstract syntax tree[J]. Journal of Cyber Security,2020,5(4):1-13.(In Chinese)

[9] ZHENG J Y,PANG J M,ZHANG X C,*et al*. Recurrent neural network based binary code vulnerability detection[C]//Proceedings of the 2019 2nd International Conference on Algorithms,Computing and Artificial Intelligence. New York: ACM, 2019: 160-165.

[10] 杨鑫. 基于语义学习的二进制漏洞代码克隆检测[D]. 北京:清华大学,2019.
YANG X. Semantic learning based binary vulnerability code clone detection[D]. Beijing:Tsinghua University,2019.(In Chinese)

[11] 梅瑞,严寒冰,沈元,等. 二进制代码切片技术在恶意代码检测中的应用研究[J]. 信息安全学报,2021,6(3):125-140.
MEI R,YAN H B,SHEN Y,*et al*. Application research of slicing technology of binary executables in malware detection[J]. Journal of Cyber Security,2021,6(3):125-140.(In Chinese)

[12] BOLAND JR F E, BLACK P E. Juliet 1.1 C/C++ and Java test suite [OL]. [Accessed: May 2021]. https://www.nist.gov/publications/juliet-11-cc-and-java-test-suite.

[13] MITRE. MITRE Common Weakness Enumeration (CWE) [OL]. [Accessed: May 2019]. https://cwe.mitre.org/.

[14] THUNES C. Javalang 0.13.0 [OL]. [Accessed: May-2020] https://github.com/c2nes/javalang.

[15] 刘炜. 基于机器学习的代码漏洞检测机制研究与应用[D]. 成都:电子科技大学,2018.
LIU W. Research and application of machine learning based code vulnerability detection mechanism[D]. Chengdu: University of Electronic Science and Technology of China,2018.(In Chinese)

[16] MAURICIO A. CK [OL]. [Accessed: May 2021]. https://github.com/mauricioaniche/ck.

[17] VASWANI A, SHAZEER N, PARMAR N, *et al*. Attention is all you need [C]// Advances in Neural Information Processing Systems. 2017: 5998-6008.