基于词法分析的代码漏洞模式识别

何峙 21215122 大数据与人工智能

摘要

软件技术与社会生活的方方面面越来越紧密，软件开发又不可避免的产生各种漏洞，而漏洞识别与定位十分耗费人力物力。如何快速识别并定位漏洞以提升软件运行的稳定与安全成为越来越严峻的问题。随着深度学习的发展，出现了一些可快速自动化的识别软件漏洞的方法，譬如基于代码抽象语法树（以下简称 AST）或程序数据流图(以下简称 PDG)的漏洞识别方法，从 AST 或 PDG 中提取代码特征进行漏洞模式识别，但大部分代码组织结构譬如通过 AST 进行抽象后会消失，导致代码元素间相互依赖的语义难以捕捉，不利于漏洞的识别。本文提出一种基于词法分析的代码漏洞特征提取方法，以利用更广的语义依赖进行漏洞识别。

关键字：漏洞识别，词法分析，软件安全防护，深度学习

1 引言

软件漏洞与软件开发相伴相生。由于软件规模的不断扩大，导致软件复杂度也不断变高，兼之开发人员对软件开发技术熟练程度、对开发逻辑理解等存在差异，不可避免的在软件开发过程中不自觉的引入各种漏洞（业界也称为“Bug”），而且随着业务的迭代，老旧的程序代码可能不适应新业务的需求发展，也导致出现各种漏洞。

漏洞是潜伏在软件系统中的，发现这些各种各样的漏洞通常是一种被动的行为，一般在软件运行期时系统出现问题了，漏洞才会被开发人员识别并修复，这使得维护软件运行的稳定性相对滞后。而且如果软件系统复杂性很高，开发人员识别漏洞的时间也可能相对变长，对开发人员的技术水平要求也可能相对提高，从而使得漏洞识别是一件十分耗费人力的事。

其次，已经有一些静态的漏洞分析工具可以帮助开发人员较快的识别漏洞。如 Clang Static Analyzer[1]，这些工具一般是通过人工预设的漏洞语义判断逻辑，譬如：如果一个对象分配方法与其释放方法要成对调用，否则就判断代码片段有内存泄漏漏洞。这要求静态分析工具要随漏洞语义的变化而不断更新其的判断逻辑，十分耗费物力财力。而且有些漏洞要在软件运行期才有可能表现出来，静态分析工具无法动态识别漏洞，覆盖率低，导致很容易出现漏报或误报的情况。

2016 年，Zhen Li 等提出了称为“VulPecker”[1]的自动识别系统漏洞的方法，基于代码克隆技术，利用代码相似性从源代码中检测漏洞。但这些方法本质还是通过静态分析方法预先抽取已知的漏洞特征，通过相似性比较来检测代码是否有漏洞，但不能自动提抽取代码特征来识别新的漏洞。2018 年，Zhen Li 等又先后提出了 VulDeePecker[2]和 SySeVR[3]两个利用机器学习可自动识别漏洞的方案，其系统的主要思想是先通过 AST 找出可以可疑漏洞点（变量或函数方法），然后找出与漏洞点语义相关的语句进行训练学习，其模型的效果有不错的漏洞低误报率和低漏报率，相比现有的静态分析工具有漏洞识别成功率有很大提升。但其可疑漏洞点的假设仍然离不开静态的判断规则，人工假设性过强。Aram Hovsepyan[4]等基于词袋模型（bag-of-word）将其 Java 代码进行编码，然后用支持向量机自动进行代码段的漏洞识别，但其训练与测评方法只局限于单个数据集，算法通用性与泛化能力有待考量。Yulei Pang[5]等在此基础上，使用 N-gram 模型对代码片段进行编码，然后也使用支持向量机自动进行代码漏洞识别，获得不错的性能。Jón Arnar Briem[6]等将 AST 的结点编码为向量，然后输入到神经网络进行漏洞识别学习，得益于 AST 能比较好的抽象出程序代码元素间的相互依赖关系，此模型对漏洞的识别率比较高，但其只提供一种验证方案，无法考虑其算法通用性。Hantao Feng[7]等也对基于 AST 的漏洞识别方法进行研究，使用 BiGRU 作为主要的模型架构。还有张启航[8] 同样也是提出了基于 AST 的漏洞识别方案，其主要区别是没使用 AST 的全部结点对代码进行编码，而只选取了变量、方法名和程序控制逻辑结点，一定程序度减轻了这类基于 AST 编码方案的复杂度，且使用 GPT 和 BiGRU 作为漏洞特征的抽取模型架构，增加了代码内部各元素对于其漏洞语义的注意力因素，对漏洞识别率的提升也有很好的帮助。

本文的主要工作如下：

（1）基于对源代码进行词法分析方法，使用词嵌入技术将源代码编码为特征向量。

（2）使用深度学习方法提取漏洞特征，从更广的语义依赖维度进行漏洞模式识别，力求提高漏洞识别的准确率。

---

[1] https://clang-analyzer.llvm.org/

2 研究方法

2.1 代码编码方法

漏洞识别的核心问题就是如何将代码编码为语义相关的向量。

首要问题是如何选择研究对象的粒度。具体到代码研究，考虑到以代码文件为单位的话，每个样本的代码元素将可能十分巨大。而如果以字符级别元素为单位，可能出现相同字符串表示不同语义的问题。本文认为函数粒度为单位比较适合，即一个函数为一个样本，理由为：（1）函数通常是只完成一个功能，是一群语义相近的代码语句集合，可以形成一个特定语义的闭包。（2）函数包含的代码元素一般不会太多，编码效率会比较高。

第二个问题是抽取代码元素间的依赖关系。当前漏洞识别方法也多使用以 AST 节点作为代码元素，甚至仅选取变量类型和方法类型的节点作为代码元素。虽然 AST 可以一定程度的反映代码元素间的依赖关系， 但 AST 会把代码元素的组织架构去掉，而有些漏洞在代码原有结构中才比较容易看出来，譬如 offset by one error、递归中忘记终结条件，等。本文将使用的全量代码元素对代码进行编码，以解决这种将代码抽象后结构缺失后的问题。编码过程如下：

（1）因为用户自定义的标识符一般出现的频率较少，而且对于不同软件项目，标识符的命名方式、代码的编写风格不统一， 如果都分别编码，将可能引起过多权重很小的参数，影响模型识别效果。为此，本文将用户自定义的变量名、非指针型的变量类型、指针型的变量类型和函数名分别替换为“hoho_var”、“hoho_var_type”、“hoho_pt_type”和“hoho_func”4 种固定的通配符，如图 1 操作。

（2）使用编译器对代码样本进行词法分析，即可将连续的代码语句切分为一个个词素，形容词素列表。本文使用的数据集是 C 语言代码数据集，选用 Clang[2]作为词法分析器，譬如对图 2 代码段进行切分，可得到图 3 切分结果。

（3）使用当前比较流行 word2vec[9]词嵌入方法，它是基于分布式表示的思想，可以反映不同代码元素的在特征空间的分布，从而表示出它们的相关程度。用此方法将每个划分出来的词素编码为数值，形成数值型向量表示。

```
copyFile(void *file, char * path);
```

图 2 代码切分前

```
"copyFile", "(", "void", "*", "file", ",", "char", "*", "path", ")", ";"
```

图 3 代码切分后

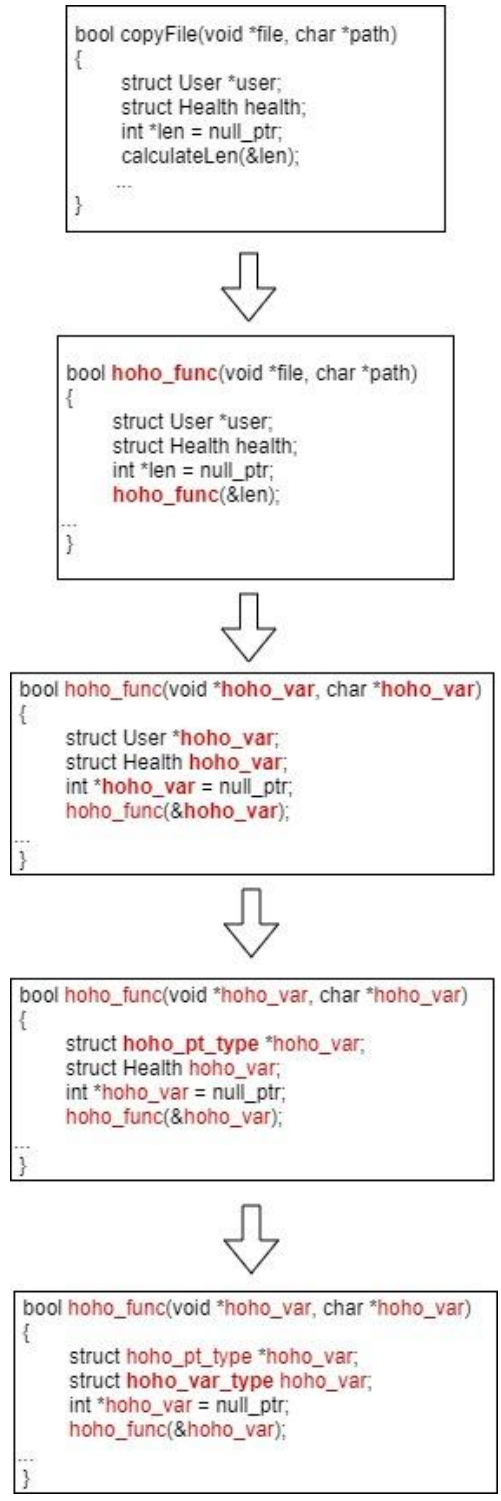

图 1 用户自定义的代码标识符替换为通配符

2.2 模型基本架构

本文采用 TextCNN[10]网络架构进行模型的训练。它借鉴了 CNN[11]技术在图像领域中的成功经验，譬如可以

[2] https://clang.llvm.org/

共享网络权重，可以并行进行卷积运算，等，将 CNN 运用在语言处理中，区别是不像图像处理中的 CNN 是横向纵向两个维度做对数据点进行卷积运算，TextCNN 只在一个维（纵向）做卷积运算，这要求卷积核的宽度必须等于经过 word2vec 之后词嵌入的维度。如图 4 所示。

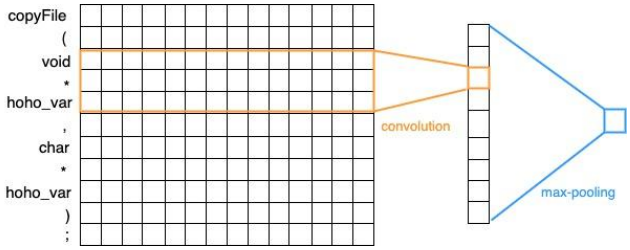

图 4 TextCNN 网络基本结构

具体到漏洞识别来说，将词向量纵向堆叠后，可分别与不同尺寸的卷积核进行卷积运算，再将运算结果进行 max-pooling 操作，将结果堆叠拉直为 1 维向量，再输入到全连接网络进行分类识别。

表 1 对不定长向量进行定长处理

for	(	int	index	=	0	;
if	(	hoho_var	==	9	)	**<pad>**
float	*	hoho_var	=	NULL	**<pad>**	**<pad>**
delete	(	)	**<pad>**	**<pad>**	**<pad>**	**<pad>**

3 实验

3.1 数据预处理

本文使用的漏洞识别数据集来自于 microsoft 的 CodeXGLUE[3]其中的“code-code”漏洞数据集，每条样本包含：所来自的软件项目名、版本控制提交的哈希值、代码标注和函数定义代码段，一共 27296 条样本数据，每条样本分别有 0（有漏洞）和 1（无漏洞）标注。本文将其中 80%作为训练集，剩余 20%作为测试集。每条样本的词素列表长度分布如图 5 所示。可见大部分样本包含代码词素的数量为 2500 以下，故本文以 2000 作为样本最大的词素数量。对数据进行预处理时，若每条样本代码词素数量超过 2000，则进行截断至 2000 大小，否则，则对样本进 行 padding 处理，如表 1，用特殊标识符“<pad>”填充至 2000 长度。

3.2 模型细节

漏洞识别分类的整个模型架构如图 6 所示。

（1）输入层，代码词素列表经过 embedding 层后，编码成 200 维代码向量，再将每一个词素纵向堆叠，使得每个向量表示为 2000x200 的词嵌入矩阵。

（2）卷积层，为了尽量抽取更多的词素依赖关系，本文选用 4 种尺寸大小卷积核，设置长度为 3、5、7、11，分别与词嵌入矩阵进行卷积运算。将各个卷积核输出结果进行堆叠，输出形成 1x400 的特征向量。这里还会使用 dropout[10]方法，设置 dropout 率为 50%，以防止模型过拟合。

（3）全连接层。这里设置两个隐藏层，第一个隐藏层大小即为以上卷积层输出的特征向量大小（400），后再接一个大小为 64 的全连接层。

（4） 输出层大小为 2，分别对应两个类别（有漏洞或无漏洞），最后使用 softmax 函数计算两个类别的概率。

本模型采用预测分类与输入分类标记的交叉熵作为损失函数，并使用 Adam[11]算法作为模型的优化算法。

3.3 实验结果

在模型训练完成后，会自动得到各代码元素与其对应的词嵌入向量，其表征了代码元素在几何空间的位置。语义相近的代码向量在几何空间中具有更近的距离。因此可以通过测试某些代码元素对应的向量的距离大小，并对比这些代码元素的现实含义，来验证这种代码编码方法的有效性。以下随机选出若干代码元素，如数据类型标识符、函数名、控制流程关键字等，查看与其距离最近有哪些代码元素。这里衡量距离大小使用余弦相似度计算。如表 2 所示，左边栏是待评估的代码元素，右边栏列出了与其距离最近的前 5 个代码元素，结果也基本符合预期，即具有相似语义的代码，其在几何空间的距离也更接近。

表 2 指定代码元素与其最相近的 5 个其他代码元素

代码元素	最相近的前 5 个代码元素
“for”	“int”, “(“, “*”, “VTIME”, “;”
“hoho_func”	“(”, “hoho_pt_type”, “)”, “,”, “hoho_var”
“int”	“*”, “hoho_var”, “reorder_pts”, “bps_rd”, “,”

[3] https://github.com/microsoft/CodeXGLUE/

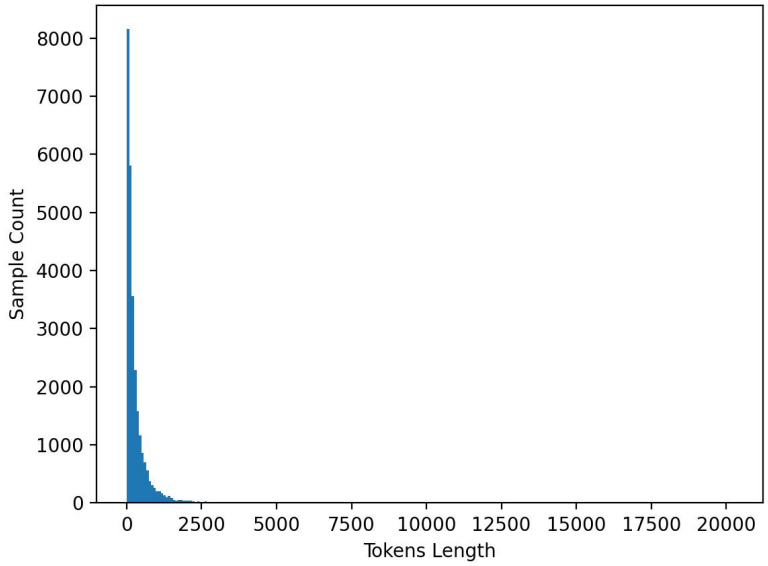

图 5 数据集样本词素数量分布

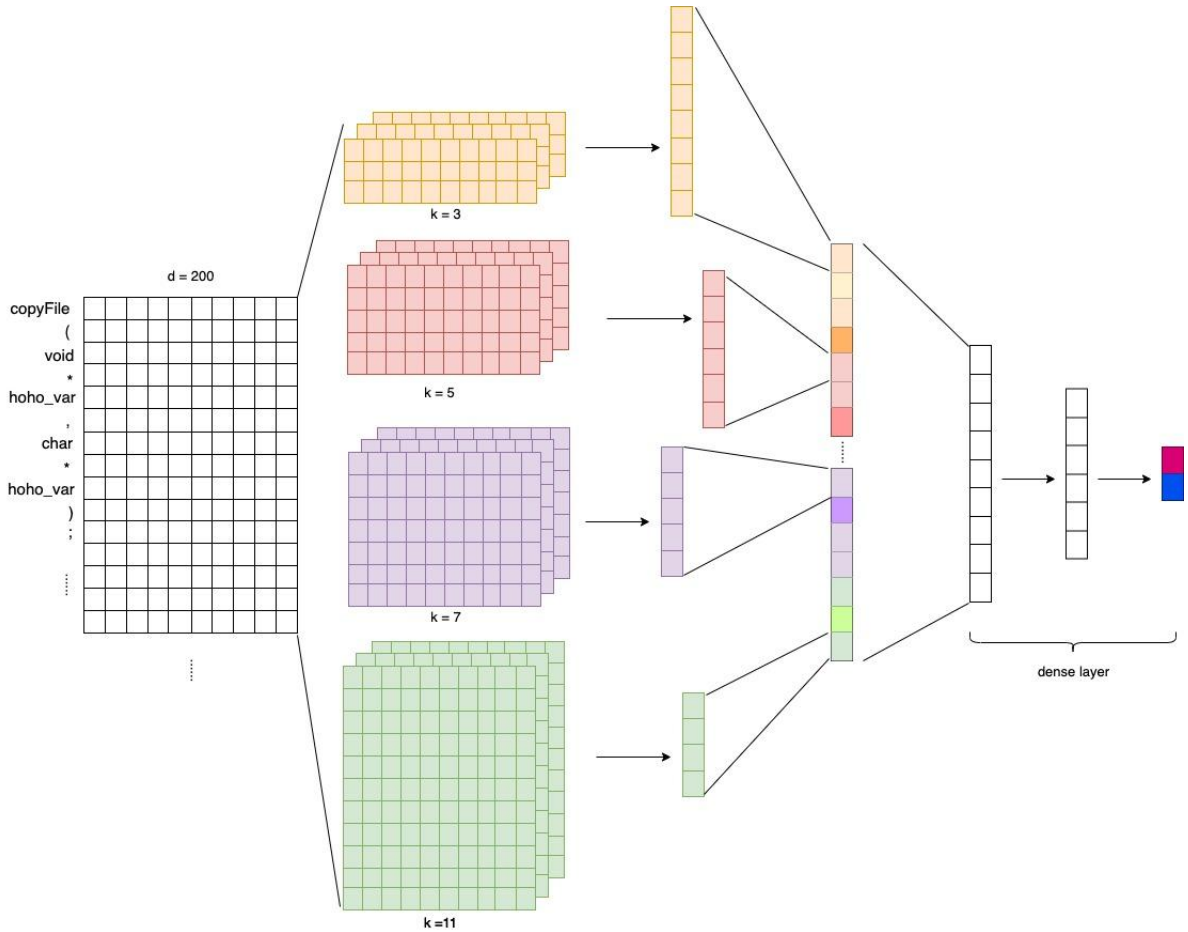

图 6 漏洞识别流程网络架构

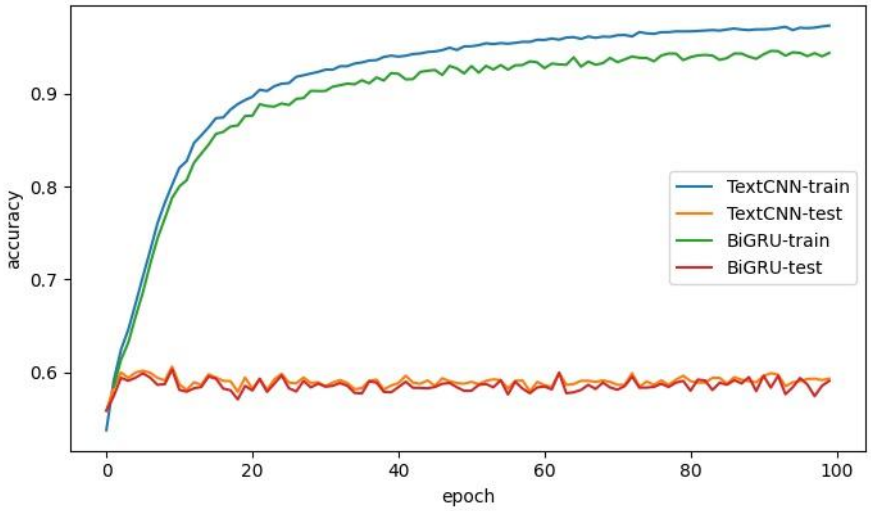

图 7 TextCNN 与 BiGRU 训练与测试准确率对比

为了验证本文的漏洞识别模型效果，本文将 Hantao Feng[7]等的基于 AST 并使用 BiGRU 提取漏洞特征的方法做比较。GRU 是 RNN（Recurrent Neural Network，循环神经网络）—种，也称为门控循环神经网络（Gated Recurrent Unit）[12]，相比于传统的 RNN，GRU 增加了一些权重门阀来决定输入的信息应该保留什么和应该删除什么，所以它也称为门控循环神经单元，一定程度上解决了传统 RNN 因信息的长期依赖问题导致梯度消失的问题。而 BiGRU 即双向门控循环神经网络，是 BiRNN（Bidirection RNN）[13]的一种。由于 Hantao Feng 等其论文没有提供相关代码，本文按照其研究方法编写了对应的模型训练代码[4]。图 7 是使用全量代码元素和 TextCNN 与使用 AST 和 BiGRU 的模型训练与测试结果，两者都用相同的数集据和相同划分的训练集与测试集进行训练与测试。从图中可见，两者在训练集上的准确率 TextCNN 要优于 BiGRU，而在测试集上的准确率差别不大。两种模型的测试准确率都要明显低于训练准确率，模型都过拟合，可能是由于模型结构比较复杂，权重参数过多导致，还需要加强正则化力度。表 3 也列出了两种模型最终的准确度数据。

表 3 TextCNN 与 BiGRU 训练与测试具体准确率

模型	Train	Test
TextCNN	**97.31%**	**59.34%**
BiGRU	93.37%	59.09%

本文也使用广泛被采用的精确率、召回率、F1 值和 MCC 值衡量模型的性能，这些衡量标准定义如下：

(1) 精确率（Precision Rate）表示模型预测为正例的样本中有多少实际为正例的：

$$PrecisionRate = \frac{TP}{TP + FP}$$

(2) 召回率（Recall Rate）表示实际为正例的样本中有多少是被模型正确预测到的：

$$RecallRate = \frac{TP}{TP + FN}$$

(3) F1 即为精确率和召回率的平均值，计算方式为：

$$F1 = \frac{2 \times PrecisionRate \times RecallRate}{PrecisionRate + RecallRate}$$

(4) MMC 描述了实际分类与预测分类间的相关系数，取值范围为[-1, 1]，1 表示完美预测，-1 表示实际与预测完全不一致，0 表示预测结果还不如随机，计算方式为：

$$MMC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

其中，TP 表示预测为正例实际也为正例的数量，FP 表示预测为正例但实际为负例，FN 为预测为负例但实际为正例，TN 则为预测与实际都为负例，这 4 个数值一定程度反映漏洞识别系统的性能状况，譬如系统误报情况可用 FP 表现，而漏报情况可用 FN 表现。表 4 展示了 TextCNN 和 BiGRU 两种模型在进行漏洞识别时以上各种评价标准的表现情况。

表 4 TextCNN 与 BiGRU 各项性能指标对比

模型	精确率	召回率	F1	MMC
TextCNN	**56.19%**	**54.23%**	**55.12%**	**0.176**
BiGRU	48.42%	53.26%	50.72%	0.096

[4] https://github.com/kevinva/CodeDetector

综合来看，使用全局代码元素与 TextCNN 的表现要略优于使用 AST 与 BiGRU，分析原因为：（1）使用全局代码元素比仅使用 AST 某些节点进行漏洞模式识时更能捕获代码间的语义。（2）TextCNN 模型参数比 BiGRU 更少，加上 max-pooling 方法的使用，TextCNN 可使代码向量表示保留主要特征，模型结构相对不那么复杂，训练速度更快，算法收敛得更快，所以使用 TextCNN 的效果相对较好。

4 结论

本文提出一种基于全量代码元素的软件漏洞模式识别技术，将程序代码进行词法分析，形成代码元素列表，利用 word2vec 方法编码成代码向量，最后输入到 TextCNN 网络进行漏洞特征提取，进而完成漏洞模式识别。相比于基于抽象语法树和 BiGRU 方法，本文使用的方法其模型性能和预测准确程度有一定的提升。但此方法设计还存在若干不足，有待改进和深入研究的地方：

（1）本文的模型只能识别出代码是否有漏洞，而没有进一步定位到漏洞出现的具体位置。可尝试考虑将样本划分为更细的粒度，如以代码行为研究单位，模型最后输出函数中每个代码行的概率分布，以此来判定漏洞出现位置的概率。

（2）在数据预处理上还需要进一步提升代码元素间的语义依赖关系，尤其在实现第（1）点漏洞精确定位上更要如此，漏洞出现在某行代码，常常是跟其他行代码的数据和控制流相关联的。这时可考虑将相关代码行提取的特征进行合并，还可以考虑将词法分析、抽象语法树和 n-gram 模型提取特征进行多维度融合，以此来获取代码元素间更紧密的关系，进一步提升漏洞识别效果。

（3）有预测结果可见，TextCNN 模型参数过多，导致结构比较复杂，出现过拟合现象。可参照 NLP 中广泛使用的 Transformer[14]模型结构，增加注意力机制，一方面可以在第（2）点的基础上挖掘更深层的代码元素间语义依赖关系，另一方面也可以使得模型更加健壮。

（4）本文只分析了关于 C 语言代码的漏洞识别，难以反映真实软件工程中漏洞检测技术的通用性。针对其他代码语言漏洞识别方法相关的数据集建设、泛化能力提升，还有待深入研究。

参考文献

[1] Zhen Li,Deqing Zou,Shouhuai Xu, et al. Vulpecker: an automated vulnerability detection system based on code similarity analysis[Z], 2016.

[2] Zhen Li,Deqing Zou,Shouhuai Xu, et al. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection[Z], 2018.

[3] Zhen Li,Deqing Zou,Shouhuai Xu, et al. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities[Z], 2018.

[4] Aram Hovsepyan,Riccardo Scandariato,Wouter Joosen, et al. Software vulnerability prediction using text analysis techniques[Z], 2012.

[5] Yulei Pang,Xiaozhen Xue,Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection[Z], 2015.

[6] Jón Arnar Briem,Jordi Smit,Hendrig Sellik, et al. Using Distributed Representation of Code for Bug Detection[Z], 2019.

[7] Hantao Feng,Xiaotong Fu,Hongyu Sun, et al. Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning[Z], 2020.

[8] 张启航. 基于抽象语法树的代码缺陷检测技术设计与实现[D], 2020.

[9] Tomas Mikolov,Kai Chen,Greg Corrado, et al. Efficient Estimation of Word Representations in Vector Space[Z], 2013.

[10] Nitish Srivastava,Geoffrey Hinton,Alex Krizhevsky, et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting[Z], 2014.

[11] Diederik P. Kingma,Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION[Z], 2017.

[12] Sepp Hochreiter,J urgen Schmidhuber. LONG SHORT-TERM MEMORY[Z], 1997.

[13] M. Schuster,K.K. Paliwal. Bidirectional recurrent neural networks[Z], 1997.

[14] Ashish Vaswani,Noam Shazeer,Niki Parmar, et al. Attention Is All You Need[Z], 2017.