doi :10.3969/j.issn.1671-1122.2022.10.009

基于图神经网络和通用漏洞分析框架的 C 类语言漏洞检测方法

朱丽娜[1]，马铭芮[2,3,4]，朱东昭[5]

（1. 广东警官学院网络信息安全系，广州 510442；2. 华中科技大学网络空间安全学院，武汉 430074；3. 分布式系统安全湖北省重点实验室，武汉 430074；4. 湖北省大数据安全工程技术研究中心，武汉 430074；5. 中国移动信息技术有限公司黑龙江分公司，哈尔滨 150001）

摘　要：现有的自动化漏洞挖掘工具大多泛化能力较差，具有高误报率与漏报率。文章提出一种针对 C 类语言的多分类漏洞静态检测模型 CSVDM。CSVDM 运用代码相似性比对模块与通用漏洞分析框架模块从源码层面进行漏洞挖掘，代码相似性比对模块运用最长公共子序列（Longest Common Subsequence，LCS）算法与图神经网络对待检测源码与漏洞模板实施代码克隆与同源性检测，根据预设阈值生成漏洞相似度列表。通用漏洞分析框架模块对待检测源码进行上下文依赖的数据流与控制流分析，弥补了代码相似性比对模块在检测不是由代码克隆引起的漏洞时高假阴性的缺陷，生成漏洞分析列表。CSVDM 综合漏洞相似度列表与漏洞分析列表，生成最终的漏洞检测报告。实验结果表明，CSVDM 相较于 Checkmarx 等漏洞挖掘工具在评价指标方面有较大幅度提升。

关键词：通用漏洞分析框架；LCS 算法；Skip-Gram 模型；图神经网络；图注意力机制

中图分类号：TP309　**文献标志码：**A　**文章编号：**1671-1122（2022）10-0059-10

中文引用格式：朱丽娜，马铭芮，朱东昭 . 基于图神经网络和通用漏洞分析框架的 C 类语言漏洞检测方法 [J]. 信息网络安全，2022，22（10）：59-68.

英文引用格式：ZHU Lina, MA Mingrui, ZHU Dongzhao. Detection Method for C Language Family Based on Graph Neural Network and Generic Vulnerability Analysis Framework[J]. Netinfo Security, 2022, 22(10): 59-68.

Detection Method for C Language Family Based on Graph Neural Network and Generic Vulnerability Analysis Framework

ZHU Lina[1], MA Mingrui[2,3,4], ZHU Dongzhao[5]

(1. *Department of Network Information Security, Guangdong Police College, Guangzhou* 510442, *China*; 2. *School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan* 430074, *China*; 3. *Hubei Key Laboratory of Distributed System Security, Wuhan* 430074, *China*; 4. *Hubei Engineering Research Center on Big Data Security, Wuhan* 430074, *China*; 5. *Heilongjiang Branch of China Mobile Information Technology Co., Ltd., Harbin* 150001, *China*)

收稿日期：2022-07-01

基金项目：国家自然科学基金 [6217071437，62072200，62127808]；广东省自然科学基金 [2020A1515011096，2019A1515011841]；广东警官学院院级科研项目 [2022SY02]

作者简介：朱丽娜（1974—），女，山东，讲师，硕士，主要研究方向为网络信息安全；马铭芮（2000—），男，黑龙江，硕士研究生，主要研究方向为神经网络、深度学习和网络信息安全；朱东昭（1977—），男，山东，高级工程师，硕士，主要研究方向为大数据和网络信息安全。

通信作者：马铭芮 jkpathfinder@126.com

Abstract: Most of the existing automated vulnerability mining tools have poor generalization ability and high false positive and false negative rale. In this paper, a static detection model called CSVDM was proposed for multi-class vulnerabilities in C language family. CSVDM used code similarity detection and generic vulnerability analysis framework module to perform vulnerability mining at the source code level. The similarity detection module integrated longest common subsequence(LCS) algorithm and graph neural network to implement code cloning and homology detection, generating the vulnerability similarity list according to a preset threshold. The generic vulnerability analysis framework module performed context-dependent data flow and controled flow analysis of the source code to be tested to compensate for the the similarity detection module's high false negatives in detecting vulnerabilities not caused by code cloning, and generated the vulnerability analysis list. CSVDM combined the vulnerability similarity list and the vulnerability analysis list to generate the final vulnerability detection report. The experimental results show that CSVDM has a substantial improvement in evaluation metrics compared to other vulnerability mining tools such as checkmarx.

Key words: generic vulnerability analysis framework; LCS algorithm; Skip-Gram model; graph neural network; graph attention mechanism

0 引言

通过软件漏洞[1]进行攻击是一种常用的网络攻击形式。目前，主流的漏洞检测工具分为静态漏洞分析[2]与动态模糊测试[3]两大类。前者已经有许多成熟的工具软件，如Checkmarx和Flawfinder等，这类工具大多依赖特征库匹配或数据流分析等固定模式进行漏洞分析，具有较高的误报率与漏报率。后者耗费计算资源较多，且通常需要与前者结合才能达到较好的漏洞分析效果。

静态漏洞分析工具可以进一步分为基于代码相似度的检测器[4,5]和基于行为模式[6,7]的检测器。前者仅能有效检测由代码复制产生的漏洞，后者又可以被分为基于特定规则的检测器与基于人工智能技术的检测器。Checkmarx和Flawfinder属于基于特定规则的检测器，它们可以识别特征模式固定的漏洞源码，但难以泛化并且需要人为定义漏洞检测规则。

基于人工智能技术的漏洞检测器是当前该领域的一个研究热点。GRIECO[8]等人、YAMAGUCHI[9]等人使用机器学习技术进行检测，依据漏洞程序的表征进行学习，但这类方法不具备高精度漏洞定位能力。一些研究人员使用深度学习技术进行检测，例如，LI[10]等人使用BiLSTM神经网络进行漏洞检测，通过将源代码转换为代码小部件提高检测细粒度。LIN[11]等人借助迁移表征学习，解决了人为定义规则的问题。ZHOU[12]等人使用图神经网络进行漏洞识别，但该方向的研究仍处于初级阶段。

本文提出一种集成代码相似性比对模块与通用漏洞分析框架模型，面向C类语言的多分类漏洞检测模型CSVDM，该方案的主要创新点在于CSVDM能够同时实施包括缓冲区溢出、格式化字符串的多分类漏洞检测任务，解决了以往检测漏洞种类固定、形态单一的缺陷。同时在源代码与中间代码层面分析，能够进行精准的漏洞定位，提高了检测细粒度并结合通用漏洞分析框架、LCS算法和图神经网络等多个不同模块，具备较高的检测精度、更低的误报率与漏报率。本文中的C类语言包括C语言、C++和C#等。

1 CSVDM 整体结构与工作流程设计

CSVDM总体运行流程如图1所示。CSVDM包括代码相似度计算与通用漏洞分析框架两个模块。在代码相似度计算模块，漏洞模板源码是一个map映射，每个源码都有一个与之对应的Token ID列表，用于指示该源码包含哪些类型漏洞及具体的位置信息，若Token ID为0则指示代码的非脆弱性。为进一步放大漏洞特征，模板源码也包含部分健壮程序。待检测源码与样本源码都需要进行预处理，目的是统一源码的格式与规范，避免不同的编码习惯影响模型的检测能力。

随后将经过预处理的源码分别送入代码相似性比对模块与通用漏洞分析框架模块处理，生成各自的分析列表，并通过合并列表生成最终的漏洞检测报告。被核实查验的待检测源码会作为反馈数据添加在漏洞模板源码数据库中，以进一步扩充其容量。

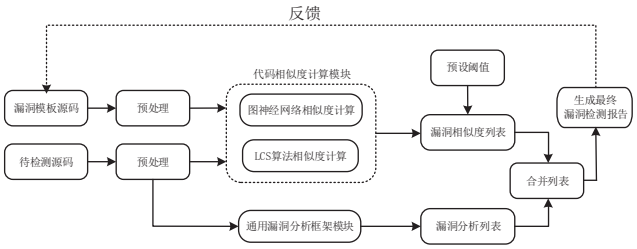


图 1 CSVDM 总体运行流程

2 预处理

2.1 数据清洁

在数据清洁阶段，对待检测源码进行形式化简、格式统一化，具体工作包括代码对齐、去除注释、去除文件头、将单行语句拆分为多行语句、使用括号匹配算法以代码段的形式存储源码和重新排序等。

2.2 词法分析与 Token 转换

与编译器常用的语法分析器不同，CSVDM 提出 VUL-LEX 的词法分析方法。VUL-LEX 方法先构造出 NFA，再通过具体规则转化为 DFA，并对 DFA 进行化简。以 VUL-DFA 中的标识符 DFA 图为例，C/C++ 标识符的正规式如公式（1）所示。

$$R=(L|U)(L|D|U)^{*} \quad (1)$$

其中，R 代表正则式结果，L 代表字母，D 代表数字，U 代表下划线，其对应的标识符 NFA 示例如图 2 所示。

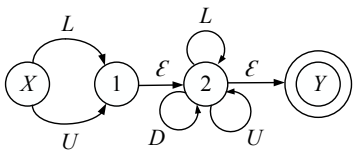


图 2 标识符 NFA 示例

通过提取 NFA 相应的状态转换关系，构造相应的状态矩阵如表 1 所示。通过该状态矩阵可以得到确定化 DFA，如图 3 所示。

表 1 状态矩阵

状态 / 字符	Q'	L	D	U
0	$\{X\}$	$\{1,2,Y\}$	φ	$\{1,2,Y\}$
1	$\{1,2,Y\}$	$\{2,Y\}$	$\{2,Y\}$	$\{2,Y\}$
2	$\{2,Y\}$	$\{2,Y\}$	$\{2,Y\}$	$\{2,Y\}$

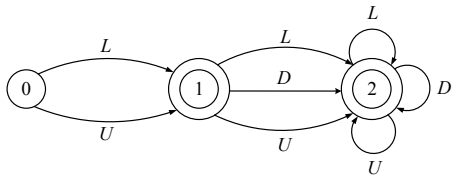


图 3 确定化 DFA

随后对该 DFA 进行化简，因为节点 1 与节点 2 的状态转移关系一致，所以将两个节点合并为一个节点，得到最简形式的 DFA，如图 4 所示。

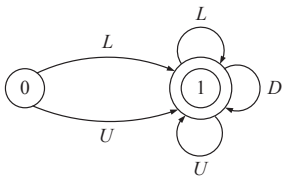


图 4 最简形式的 DFA

VUL-DFA 中的每个 DFA 图均采取上述方式生成，再将源码通过 VUL-DFA 集合处理转换为 Token 格式，并统一为 LLVM IR 中间代码格式，便于后续控制流图的生成。图 5 展示了一个预处理流程的简单示例。

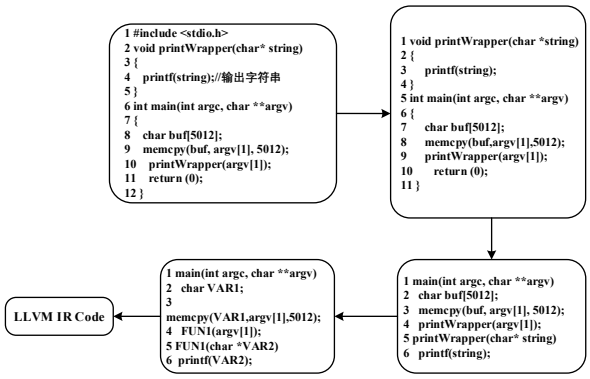


图 5 预处理流程简单示例

3 代码相似性比对

3.1 LCS 算法计算代码相似度

LCS 算法从源码段级别的细粒度进行相似度计算，用于检测是否存在代码复制和代码重用等行为，相应

伪代码如下。

```
LCS-LENGTH(X,Y)
1 m=X.length
2 n=Y.length
3 let b[1..m,1..n] and c[0..m,0..n] be new tables
4 for i = 1 to m
5    c[i,0] = 0
6 for j=0 to n
7    c[0, j] = 0
8 for i=1 to m
9    for j=1 to n
10     if xi= =yj
11       c[i, j]=c[i−1, j−1]+1
12       b[i, j]=”left up”
13     else if c[i−1, j]>=c[i, j−1]
14       c[i, j]=c[i−1, j]
15       b[i, j]=”up”
16     else c[i, j]=c[i, j−1]
17       b[i, j]=”left”
18 return c and b
```

依据上述伪代码得到LCS的状态转换矩阵，如表2所示。

表 2 LCS 的状态转换矩阵

状态	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
2	0	0	0	1	1	1	1
3	0	0	0	1	1	2	2
4	0	0	0	1	2	2	2
5	0	0	0	1	2	2	2
6	0	0	0	1	2	2	2

从矩阵右下角开始，依次向左上方查找状态数组C中元素减一的位置，直至$C[m, n]$=1为止，得到比对样本的LCS序列。LCS算法相似度计算过程如公式（2）所示。

$$LCS_{similarity} = \frac{LCS_{length}}{INPUTCODE_{length}} \quad (2)$$

其中，LCS_{length}是比对样本的LCS长度，$INPUTCODE_{length}$是输入样本的长度。

3.2 图神经网络计算代码相似度

单纯使用LCS算法存在许多弊端，例如，在总行数以万计的完整程序中，关键行大概率只占极小比例，容易导致误判，并且难以检测特征复杂的漏洞。因此，本文提出基于图神经网络的代码相似度计算方法，其运行过程如图6所示。

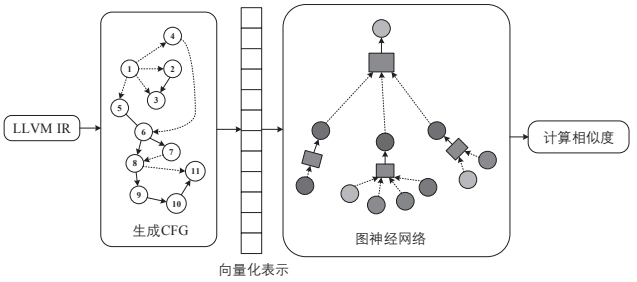


图 6 图神经网络计算代码相似度运行过程

CSVDM借助LLVM工具生成漏洞样本源码控制流图（Control Flow Graph，CFG），本文在Token生成后的LLVM IR代码格式基础上，使用LLVM opt工具自动化生成CFG。具体而言，以LLVM IR为输入，通过运行PASS模块实现指定优化或分析，输出优化后的文件分析结果并生成dot格式的CFG。

控制流图中的每个节点都是一个独立的代码段，被称为基本块。在得到CFG后，源代码中所有跳转指令都用CFG中两点之间边的关系表示，在每个基本块的代码段中已经不存在跳转指令，因此基本块是一种完全顺序结构。一个基本块中的所有代码语句被视为自然语言中完整的一句话。CSVDM使用Skip-Gram模型[13]进行向量编码，其基本结构如图7所示。

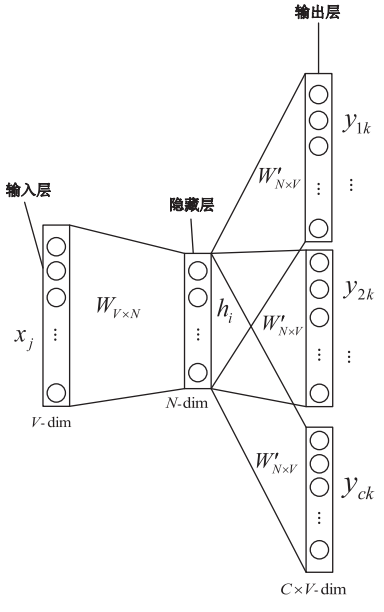


图 7 Skip–Gram 模型基本结构

本文应用多元逻辑回归方法求解中心词概率分布以减少分母的运算量，提高程序的运行效率。优化计算过程如公式（3）所示。

$$J_t(\theta)=\log\sigma(\boldsymbol{u}_o^{\mathrm{T}}v_c)+\sum_{i=1}^{k}E_{j\sim P(w)}[\log\sigma(-\boldsymbol{u}_j^{\mathrm{T}}v_c)] \quad (3)$$

进一步将其优化为损失函数，如公式（4）所示。

$$J_{neg-sample}(o,v_c,U)=-\log(\sigma(\boldsymbol{u}_o^{\mathrm{T}}v_c))-\sum_{k=1}^{K}\log(\sigma(-\boldsymbol{u}_k^{\mathrm{T}}v_c)) \quad (4)$$

第一个对数项是为了保证中心词的点积经过Sigmoid函数运算后取得最大值，负数运算将其转换为取最小值。第二项是选取一些与当前中心词无关的k个样本，使负样本出现的概率值尽可能小。U的计算方法如公式（5）所示。

$$P(w)=U(w)^{3/4}/Z \quad (5)$$

Z是对语料库中所有单词执行$U(w)^{3/4}$运算后求和得到的结果，w与3/4是超参数，P为对应单词的概率分布。为了进一步提高词向量矩阵的表示能力，CSVDM增加了共生矩阵，共生矩阵示例如表3所示。

表 3 共生矩阵示例

数量	I	like	enjoy	deep	Learning	NLP	Flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
Learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
Flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

CSVDM使用改进的struct2vec方法进行图相似性计算，struct2vec方法结构如图8所示。

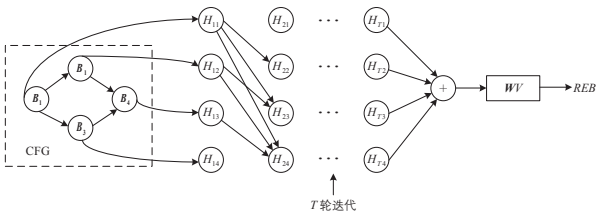


图 8 struct2vec 方法结构

图8中的输入是CFG，每个节点i都具有节点特征的向量表示形式B_i。struct2vec通过T轮迭代进行学习，输出一个相同结构的图并称之为图REB，每个节点特征被重新表示为REB_i。

从B_i到REB_i采用变分推断的思想对拓扑图进行迭代计算。节点i的新特征表示REB_i既要考虑一个网络的拓扑结构也要考虑一定范围内的邻居节点特征对当前节点特征的影响，灵感来自LINE算法[14]。每个节点的单次迭代过程如公式（6）所示。

$$REB_i^{t+1}=F(B_i,\sum_{j\in N_i}REB_j^{(t)}) \quad (6)$$

其中N_i是节点i的直接邻居，$i\in V$。通过T轮迭代后，每个节点的新特征U_i包含距离为i、跳数为T的全部节点相关信息。F是一个非线性激活函数，通过该函数选择相关信息进行迭代更新，本文选择的激活函数如公式（7）所示。

$$F(B_v,\sum_{j\in N_i}REB_j)=\tanh(\boldsymbol{W}_1B_v+\sigma(\sum_{j\in N_i}REB_j)) \quad (7)$$

其中，σ是n维全连接神经网络，如公式（8）所示。

$$\sigma(l)=P_1\times ReLU(P_2\times\cdots\times ReLU(P_nl)) \quad (8)$$

本文将struct2vec结构中迭代层的每个节点都改为注意力机制学习权重，不再同等地对待每个邻居节点对当前节点的影响，而是对重要的邻居赋予较高的权重。迭代层的注意力层输入为当前层全部节点的特征，如公式（9）所示。

$$H=\{H_{i1},H_{i2},\cdots,H_{iN}\} \quad (9)$$

其中，N为节点数量，sp为相应节点的特征维度，$H_{ik}\in R^{sp}$。相应注意力层的输出是相应所有节点新的特征，如公式（10）所示。

$$H'=\{H'_{i1},H'_{i2},\cdots,H'_{iN}\} \quad (10)$$

其中，sp'是每个节点新的特征维度，$H'_{ik}\in R^{sp'}$。因为注意力层需要将输入特征转换为更高级别的特征从而获得更充分的表达能力，所以需要一个可以学习的线性变换。因此，注意力层通过权重矩阵$\boldsymbol{W}$构建共享的线性转换过程，将该转换过程应用于图中的每个节点，如公式（11）所示。

$$R^{sp'}\times R^{sp}\rightarrow R \quad (11)$$

第x个节点对第v个节点的权重系数为$\boldsymbol{WE}_{xv}=$

$a_{xv}(\boldsymbol{WH}_{ix},\boldsymbol{WH}_{iv})$，用于表示节点 x 对节点 v 的重要程度。为保证所有节点间的权重单位一致，对 a_{xv} 进行归一化操作，如公式（12）所示。

$$a_{xv}=softmax_v(\boldsymbol{WE}_{xv})=\frac{\exp(\boldsymbol{WE}_{xv})}{\sum_{k\in N_i}\exp(\boldsymbol{WE}_{ik})} \quad (12)$$

公式（12）通过概率分布计算得到每个节点的权重系数在所有节点权重系数构成的概率空间中的比例，节点相对权重信息计算结构如图 9 所示。

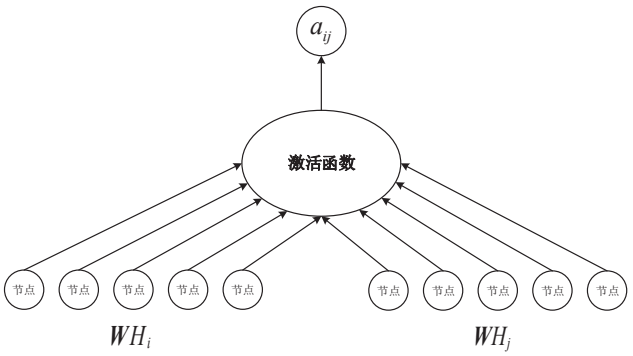


图 9 节点相对权重信息计算结构

为了避免 *ReLu* 负值区域(x<0)梯度消失，得到负值输出，帮助网络向正确的方向推动权重和偏置的变化，CSVDM 选择 *ELU* 非线性激活函数，如公式（13）所示。

$$f(x)=\begin{cases}x, x\geqslant 0\\ \alpha(\mathrm{e}^x-1), x<0\end{cases} \quad (13)$$

在计算出归一化权重系数 α 后，得到最终的节点新特征生成式如公式（14）所示。

$$H'_i=\sigma(\sum_{j\in N_i}a_{ij}WH_j) \quad (14)$$

CSVDM 定义代码的相似度计算如公式（15）所示。

$$Graph_{similarity(Graph_1,Graph_2)}=\frac{\sum_{i=1}^{p}Graph_1[i]\times Graph_2[i]}{\sqrt{\sum_{i=1}^{p}Graph_1[i]}\times\sqrt{\sum_{i=1}^{p}Graph_2[i]}} \quad (15)$$

其中，$Graph_k[i]$ 代表第 k(k=1, 2) 个 CFG 中某个基本块的特征向量的第 i 个分量。通过均方误差得到本文的目标损失函数，如公式（16）所示。

$$J=\sum_{i=1}^{B}(Graph_{similarity(Graph_1,Graph_2)}-y)^2 \quad (16)$$

3.3 生成漏洞相似度列表

在得到 $LCS_{similarity}$ 与 $Graph_{similarity(Graph_1,Graph_2)}$ 后，引入超参数 δ 对两者的计算结果取加权平均，得到最终的相似度计算结果 $Code_{similarity}=\delta\times LCS_{similarity}+(1-\delta)\times Graph_{similarity(Graph_1,Graph_2)}$。如果 $Code_{similarity}>\theta$（θ 是相似度阈值），则将模板源码 TokenID 列表的相关信息追加到待检测源码的 SimilarityList 列表中，完成漏洞相似度列表的生成。

4 通用漏洞分析框架

通用漏洞分析框架依据数据流和控制流对预处理生成的 IRCode 进行模块化分割，生成相应的 IRCode Block，并依据 IRCode Block 模块内与模块间的依赖关系生成 Token 化的变量列表、函数列表与参数列表。通用漏洞分析框架运行在中间代码层次上，因此其源码层面分析的细粒度更高，并能够识别出部分由宏定义引发的漏洞。这些是源码层面分析工具无法实现的，原因是基于中间代码的表示采用静态单一赋值形式，可确保基本块中每个被定义的变量都会被使用，且只被赋值一次。

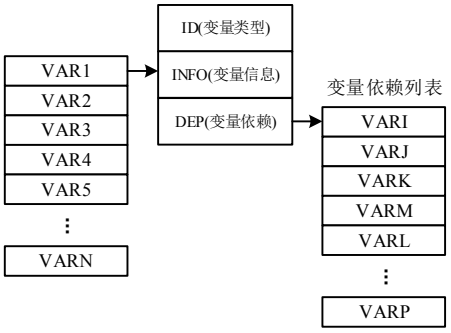
分析框架基于变量列表、函数列表及参数依赖关系，根据漏洞类型特征追踪 IRCode 中控制与数据逻辑的转移方向，定位漏洞所在的 IRCodeBlock 块。每个 IRCodeBlock 往往仅由几行中间代码构成，因此可以根据其特征标签快速进行漏洞分类与漏洞定位。

4.1 通用漏洞分析框架 API 设计与检测

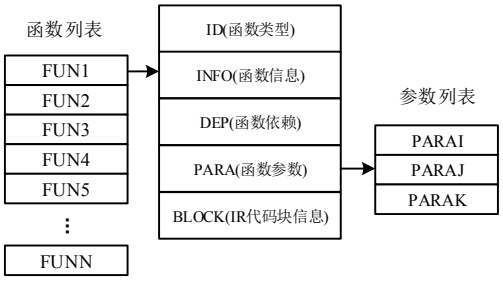
通用漏洞分析模块预留 API 如图 10 所示。图 11 以图形化形式展示通用漏洞分析框架的工作流程，不同 Code Block 之间的虚线箭头表示数据流依赖，实线箭头表示控制流依赖。

4.2 生成漏洞检测报告

在分别得到漏洞相似度列表与漏洞分析列表后，通过合并列表生成最终的漏洞检测报告，其流程如图 12 所示。



a）通用漏洞分析模块变量列表 API



b）通用漏洞分析模块函数列表与参数列表 API

图 10 通用漏洞分析模块预留 API

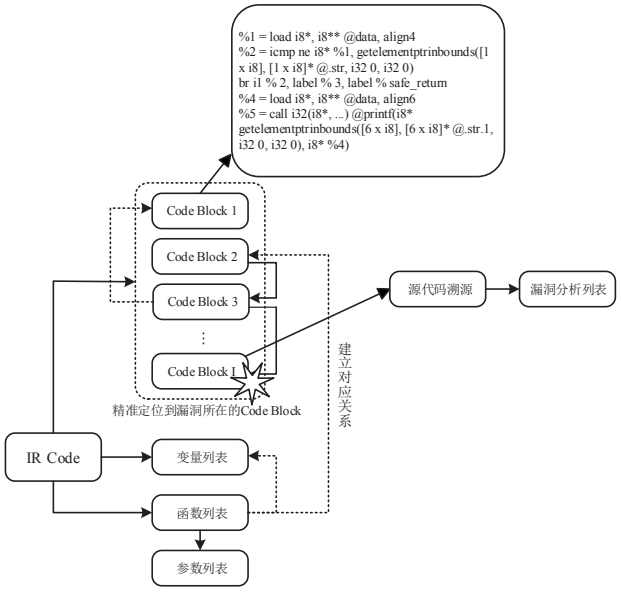


图 11 通用漏洞分析框架工作流程

本节分别对漏洞相似度列表和漏洞分析列表取交集与对称差集运算，交集运算包含同时出现在两个列表中且相关信息一致的漏洞，对称差集运算是两个列表中仅出现在自己列表的漏洞。本文将取交集运算和对称差集运算的结果赋予不同等级，并整合成最终的漏洞检测报告。

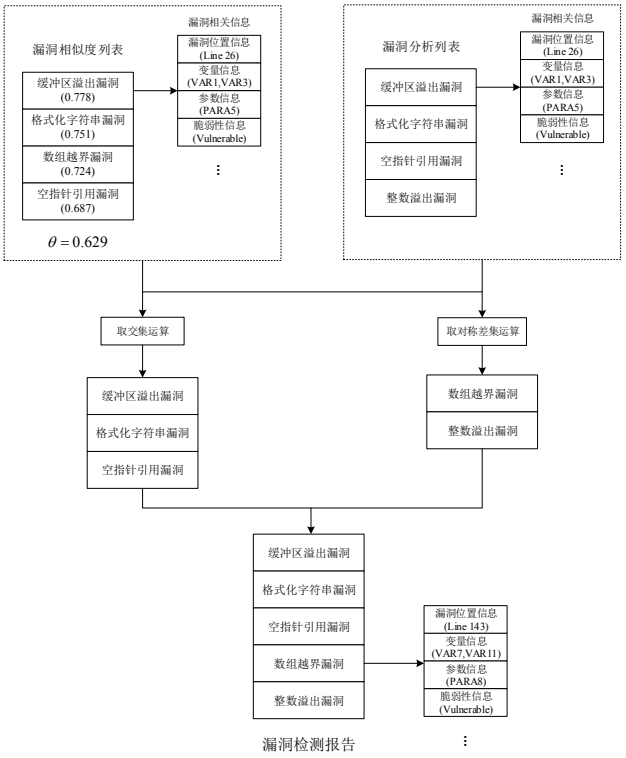


图 12 生成漏洞检测报告

5 实验

5.1 实验数据来源

本文在中国国家信息安全漏洞库（China National Vulnerability Database of Information Security，CNNVD）、美国国家计算机通用漏洞数据库（National Vulnerability Database，NVD）[15]、赛门铁克的漏洞库与SARD项目中进行采集，并筛选出32760个用C/C++语言编写的源程序。其中，70%样例用于构建初始漏洞模板源码库，并用这些样例训练与微调图神经网络，这些样例在CSVDM中被视为先验知识，剩余的30%样例用于测试阶段。

5.2 评价指标

CSVDM属于多类漏洞检测器，因此本文修改了*TPR*、*FPR*、*FNR*、*F*1等评价指标，使之更好地适应多类漏洞检测任务。设K为漏洞类型数量，本文实验K=6。设TP_i、FP_i、TN_i、FN_i、Pre_i和Rec_i分别为第i

类漏洞的真阳性、假阳性、真阴性、假阴性等的样本数量以及准确率与召回率。其中 $1 \leqslant i \leqslant K$。本文提出 $W_{Avg}FPR$ 、$W_{Avg}TPR$ 和 $W_{Avg}F1$ 评价指标，分别如公式（17）~公式（19）所示。

$$W_{Avg}FPR = \frac{1}{Total_{Num}} \times \sum_{i \in K} N_i \times \frac{FP_i}{FP_i + TN_i} \quad (17)$$

$$W_{Avg}TPR = \frac{1}{Total_{Num}} \times \sum_{i \in K} N_i \times \frac{TP_i}{TP_i + FN_i} \quad (18)$$

$$W_{Avg}F1 = \frac{1}{Total_{Num}} \times \sum_{i \in K} N_i \times \frac{2Pre_i \times Rec_i}{Pre_i + Rec_i} \quad (19)$$

其中，$Total_{Num}$ 表示全部测试样本数量总和，N_i 表示第 i 类测试样本的数量和。本文采用加权平均方式，主要原因是考虑包含不同类型的漏洞样本数量差异很大，如果仅对所有结果取平均值，会放大样本数量较多的漏洞类型的统计学优势，不能客观衡量一个模型的多类漏洞检测能力。

5.3 CSVDM 模型综合性能评估

本节将CSVDM与Checkmarx、Flawfinder等自动化漏洞挖掘工具及同样使用神经网络与深度学习技术的代码相似性模型[5]、Vuldeepecker进行对比测试，测试结果如表4所示。

表 4 CSVDM 与对比方法的性能测试

评价指标模型	$W_{Avg}FPR$	$W_{Avg}TPR$	$W_{Avg}F1$
CSVDM	1.03%	92.88%	91.37%
Vuldeepecker	19.84%	63.87%	65.32%
Checkmarx	23.81%	46.52%	43.28%
Flawfinder	39.64%	32.38%	31.73%
相似性模型	10.33%	69.84%	69.24%

CSVDM相较于4种对比模型具有明显的性能优势。其中，Checkmarx和Flawfinder两种自动化漏洞挖掘工具相比于其余3种融合神经网络与深度学习技术方法性能劣势非常明显，过高的误报率与过低的准确性使得这3种方法在实际场景中的应用效果不好。Vuldeepecker能够检测的漏洞类型有限，在多分类漏洞评价指标体系下出现明显的性能下滑，但Vuldeepecker在仅面向缓冲区溢出类型的漏洞检测中有优异的检测性能。而同样采用BiLSTM网络的相似性模型，尽管在缓冲区溢出类型的漏洞检测能力不及Vuldeepecker，但凭借更强的多分类漏洞综合检测能力在整体评价指标上取得性能领先。但BiLSTM网络更擅长建模时间序列数据而非抽象图结构，因此CSVDM采用的图神经网络比BiLSTM更好地完成抽象图结构特征提取与建模任务，取得性能的领先优势。图13形象地展示了CSVDM与其他方法的性能对比结果。

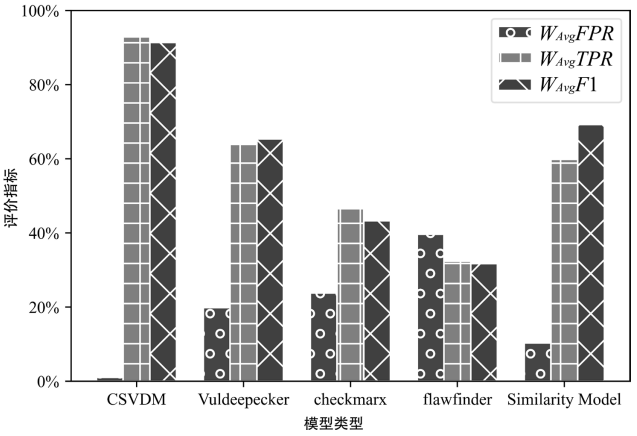


图 13 CSVDM 与其他方法性能对比

5.4 消融分析及敏感度分析

为了考察CSVDM模型各个模块的重要性，本节使用去除通用漏洞分析框架（Model1）、去除完整代码相似度比对模块（Model2）、去除图神经网络的相似度计算方法（Model3）、去除LCS算法的相似度计算方法（Model4）与完整的CSVDM模型（Model5）进行性能对比，消融分析结果如表5所示。

表 5 消融分析结果

评价指标模型	$W_{Avg}FPR$	$W_{Avg}TPR$	$W_{Avg}F1$
Model1	5.84%	82.17%	80.65%
Model2	8.89%	77.69%	75.64%
Model3	8.17%	78.17%	76.21%
Model4	1.96%	89.69%	87.24%
Model5	1.03%	92.88%	91.37%

从表5可以看出，去除CSVDM的各个模块会导致不同程度的性能下降。但去除图神经网络相似度计算方法与通用漏洞分析框架导致 $W_{Avg}TPR$ 与 $W_{Avg}F1$ 评价指标分别下降15.84%、16.59%与11.53%、11.73%，说明相关模块是CSVDM的重要组成部分。结合Model2与

Model3、Model4与Model5的对比结果可以发现，去除LCS算法只会导致微弱的性能下降，说明LCS算法效果有限。但考虑LCS算法的资源开销远小于图神经网络，因此本文保留LCS算法作为代码相似度比对模块的组成部分。CSVDM消融分析如图14所示。

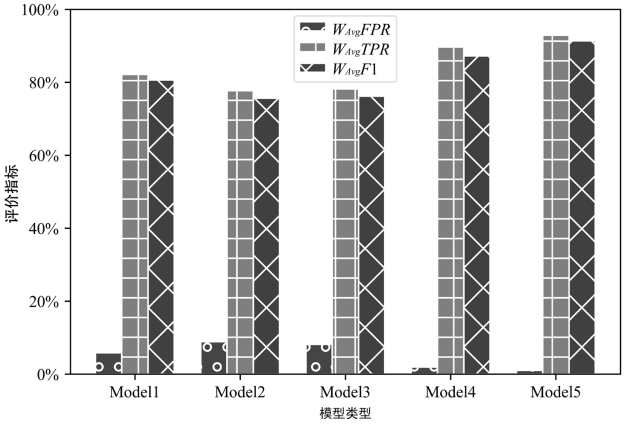


图 14 CSVDM 消融分析

本文考察 θ对CSVDM模型的性能影响，实验结果如表6所示。从表6可以看出，θ过小导致过多的待检测源码被打上漏洞标签，提高了误报率；θ过大导致仅能正确识别出形态特征与模板源码高度相似的待检测源码，提高了漏报率。因此，当 θ的值为0.6左右时，漏洞检测性能达到最佳。本文使用网格搜索方法确定 $\theta=0.6076$，θ值对CSVDM模型性能的影响如图15所示。

表 6 超参数 θ对 CSVDM 模型的性能影响

评价指标 θ	$W_{Avg}FPR$	$W_{Avg}TPR$	$W_{Avg}F1$
0.2	7.83%	82.37%	79.66%
0.3	6.25%	84.36%	81.74%
0.4	4.89%	87.87%	85.23%
0.5	2.52%	89.97%	88.35%
0.6	1.04%	92.78%	91.29%
0.7	1.74%	90.46%	89.33%
0.8	3.84%	88.71%	84.99%

6 结束语

本文结合LCS算法、图神经网络的代码相似度比对模块与通用漏洞分析框架模块，提出一种多分类漏洞静态检测模型CSVDM。CSVDM解决了以往方法面

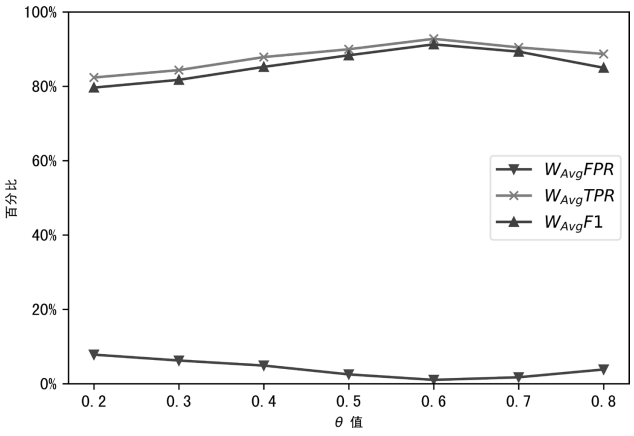


图 15 θ值对 CSVDM 模型性能的影响

临的两大技术挑战，即分析包含多种类型的漏洞源码、进行高细粒度的漏洞检测。实验结果表明，与其他模型相比，CSVDM能取得更高的检测精度，并进一步降低了误报率与漏报率。下一步希望寻找更高性能的代码相似度算法取代LCS算法，并尽快将CSVDM集成在实际软件开发环境中。

参考文献：

[1] LIU Jian, SU Purui, YANG Min, et al. Software and Cyber Security-A Survey[J]. Journal of Software, 2018, 29(1): 42-68.
刘剑，苏璞睿，杨珉，等 . 软件与网络安全研究综述 [J]. 软件学报，2018，29（1）: 42-68.
[2]KULENOVIC M, DONKO D. A Survey of Static Code Analysis Methods for Security Vulnerabilities Detection[C]//IEEE. 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). New York: IEEE, 2014: 1381-1386.
[3]MA Mingrui, HAN Lansheng, QIAN Yekui. CVDF DYNAMIC-A Dynamic Fuzzy Testing Sample Generation Framework Based on Bi-LSTM and Genetic Algorithm[J]. Sensors, 2022, 22(3): 12-25.
[4]LI Zhen, ZOU Deqing, XU Shouhuai, et al. Vulpecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis[C]// ACM. Proceedings of the 32nd Annual Conference on Computer Security Applications. New York: ACM, 2016: 201-213.
[5] XIA Zhiyang, YI Ping, YANG Tao. Static Vulnerability Detection Based on Neural Network and Code Similarity[J]. Computer Engineering, 2019, 45(12): 141-146.
夏之阳，易平，杨涛 . 基于神经网络与代码相似性的静态漏洞检测 [J]. 计算机工程，2019，45（12）: 141-146.
[6]LIANG Hongliang, WANG Lei, WU Dongyang, et al. MLSA: A Static Bugs Analysis Tool Based on LLVM IR[C]//IEEE. 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence,

Networking and Parallel/Distributed Computing (SNPD). New York: IEEE, 2016: 407−412.
[7]FANG Zhejun, LIU Qixu, ZHANG Yuqing, et al. A Static Technique for Detecting Input Validation Vulnerabilities in Android Apps[J]. Science China Information Sciences, 2017, 60(5): 1−16.
[8] GRIECO G, GRINBLAT G L, UZAL L, et al. Toward Large−Scale Vulnerability Discovery Using Machine Learning[C]//ACM. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. New York: ACM, 2016: 85−96.
[9] YAMAGUCHI F, MAIER A, GASCON H, et al. Automatic Inference of Search Patterns for Taint−Style Vulnerabilities[C]//IEEE. 2015 IEEE Symposium on Security and Privacy. New York: IEEE, 2015: 797−812.
[10] LI Zhen, ZOU Deqing, XU Shouhuai, et al. Vuldeepecker: A Deep Learning−Based System for Vulnerability Detection[J]. (2018−01−05)[2022−06−22]. https://arxiv.org/abs/1801.01681v1.
[11] LIN Guanjun, ZHANG Jun, LUO Wei, et al. Cross−Project Transfer Representation Learning for Vulnerable Function Discovery[J]. IEEE Transactions on Industrial Informatics, 2018, 14(7): 3289−3297.
[12] ZHOU Yaqin, LIU Shangqing, SIOW J, et al. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks[J]. Advances in Neural Information Processing Systems, 2019(32): 12−18.
[13] MIKOLOV T, CHEN Kai, CORRADO G, et al. Efficient Estimation of Word Representations in Vector Space[EB/OL]. (2012−09−07)[2022−06−22]. https://arxiv.org/abs/1301.3781.
[14] TANG Jian, QU Meng, WANG Mingzhe, et al. LINE: Large−Scale Information Network Embedding[EB/OL]. (2015−03−12)[2022−06−22]. https://arxiv.org/abs/1503.03578.
[15] NIST. NVD[EB/OL]. [2022−06−29]. https://nvd.nist.gov/.