# 基于特征依赖图的源代码漏洞检测方法

杨宏宇[1,2]，杨海云[2]，张良[3]，成翔[4,5]

（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 AZ85721；
4. 扬州大学信息工程学院，江苏 扬州 225127；
5.江苏省知识管理与智能服务工程研究中心，江苏 扬州 225127）

**摘 要**：针对现有源代码漏洞检测方法未显式维护源代码中与漏洞相关的语义信息，导致漏洞语句特征提取困难和漏洞检测误报率高的问题，提出一种基于特征依赖图的源代码漏洞检测方法。首先，提取函数片中的候选漏洞语句，通过分析候选漏洞语句的控制依赖链和数据依赖链，生成特征依赖图。其次，使用词向量模型生成特征依赖图的节点初始表示向量。最后，构建一种面向特征依赖图的漏洞检测神经网络，由图学习网络学习特征依赖图的异构邻居节点信息，由检测网络提取全局特征并进行漏洞检测。实验结果表明，所提方法的召回率、F1 分数分别提高 1.50%～22.32%、1.86%～16.69%，优于现有方法。

**关键词**：源代码；漏洞检测；语义信息；依赖图；神经网络

**中图分类号**：TP393

**文献标志码**：A

**DOI:** 10.11959/j.issn.1000−436x.2023018

# Feature dependence graph based source code loophole detection method

YANG Hongyu[1,2], YANG Haiyun[2], ZHANG Liang[3], CHENG Xiang[4,5]

1. School of Safety Science and Engineering, Civil Aviation University of China, Tianjin 300300, China
2. School of Computer Science and Technology, Civil Aviation University of China, Tianjin 300300, China
3. School of Information, University of Arizona, Tucson AZ85721, USA
4. School of Information Engineering, Yangzhou University, Yangzhou 225127, China
5. Jiangsu Engineering Research Center for Knowledge Management and Intelligent Service, Yangzhou 225127, China

**Abstract:** Given the problem that the existing source code loophole detection methods did not explicitly maintain the semantic information related to the loophole in the source code, which led to the difficulty of feature extraction of loophole statements and the high false positive rate of loophole detection, a source code loophole detection method based on feature dependency graph was proposed. First, extracted the candidate loophole statements in the function slice, and generated the feature dependency graph by analyzing the control dependency chain and data dependency chain of the candidate loophole statements. Secondly, the word vector model was used to generate the initial node representation vector of the feature dependency graph. Finally, a loophole detection neural network oriented to feature dependence graph was constructed, in which the graph learning network learned the heterogeneous neighbor node information of the feature dependency graph and the detection network extracted global features and performed loophole detection. The experimental results show that the recall rate and F1 score of the proposed method are improved by 1.50%～22.32% and 1.86%～16.69% respectively, which is superior to the existing method.

**Keywords:** source code, loophole detection, semantic information, dependence graph, neural network

**收稿日期**：2022−10−01；**修回日期**：2022−12−03

**通信作者**：成翔，huozhai9527@126.com

**基金项目**：国家自然科学基金资助项目（No. U1833107）

**Foundation Item:** The National Natural Science Foundation of China (No. U1833107)

## 0 引言

计算机系统是构建数字社会的基石，然而由软件安全漏洞引发的数据泄露和安全攻击时刻威胁着计算机系统的运行安全[1-2]。为避免安全威胁事件的发生，大量源代码漏洞检测方法被提出，但大多数方法仅根据安全专家手动定义的代码漏洞特征或模板规则进行漏洞检测，这些方法不仅误报率高、漏洞覆盖率低，而且模板规则需要伴随新漏洞模式的出现动态更新[3]。

随着深度学习在图像识别和自然语言处理等领域取得显著进展，研究者尝试采用深度学习方法自动提取源代码漏洞的隐式特征以进行源代码漏洞检测[4-5]。基于深度学习的源代码漏洞检测方法从大量训练样本中学习易受攻击的代码模式，当检测新的源代码样本时，将其输入预训练的神经网络，判断是否包含易受攻击的代码模式[6]，以确定代码是否存在漏洞。最初，研究者采用卷积神经网络（CNN, convolutional neural network）和循环神经网络（RNN, recurrent neural network）等传统神经网络进行源代码漏洞检测[7]，但传统神经网络较难学习到长序列的代码标记特征。近年来，部分研究者尝试将源代码建模为图结构，采用图神经网络进行源代码漏洞检测。

基于深度学习的源代码漏洞检测方法包括基于标记的方法和基于图结构的方法[8]。其中，基于标记的方法将源代码视为标记序列。文献[9]通过 CNN 提取源代码的特征向量并使用随机森林算法进行源代码漏洞检测分类，但是该方法对较长的标记序列特征提取困难，且未考虑源代码中丰富的语义信息。文献[10]提出的 VulDeePecker 方法使用容易引发安全漏洞的应用程序接口（API）对源代码进行切片，选择双向长短期记忆神经网络提取源代码漏洞特征，但是该方法仅使用与数据依赖相关的语义信息，缺失源代码中与控制依赖相关的语义信息，导致误报率较高。文献[3]提出的 SySeVR 方法使用 4 种漏洞语法特征获得候选漏洞语句，并根据候选漏洞语句的数据依赖和控制依赖信息对源代码进行切片。同时，SySeVR 选用双向门控循环单元等神经网络进行源代码漏洞检测。但是该方法未在源代码切片中显式维护与候选漏洞语句相关的依赖关系，导致神经网络难以对源代码语义信息学习和推理。文献[11]提出的 μVulDeepecker 方法采用代码注意力协助进行漏洞检测，并使用改进的长短期记忆神经网络实现了多分类漏洞检测系统，但该模型只针对与函数调用相关的漏洞。文献[12]提出的 VulDeeLocator 通过定义–引用关系关联一个项目下的多个文件，使用底层虚拟机（LLVM, low level virtual machine）将源代码转换为中间代码表示，并通过改进的双向循环神经网络将漏洞出现位置限制在若干行之内。但该方法需要将源代码编译为中间代码，对无法编译的源代码较难适用。

基于图结构的方法使用不同类型的源代码表示图表征源代码，采用图神经网络进行源代码漏洞检测。文献[13]采用由抽象语法树（AST, abstract syntax tree）、数据流图（DFG, data flow graph）、控制流图（CFG, control flow graph）和代码执行顺序组成的联合图表征源代码，并使用门控图神经网络（GGNN, gated graph neural network）[14]学习并感知联合图中的语法语义信息，然后完成漏洞检测。文献[7]提出的 Reveal 方法使用代码属性图（CPG, code property graph）表示源代码，并将代码属性图的节点类型编码在节点的表示向量中，该方法使用 GGNN 提取代码属性图的特征以进行源代码漏洞检测。上述基于图结构的方法仅简单使用源代码的一种或多种表示图表征源代码，未深度分析表示图中与漏洞语句相关的节点和边，导致神经网络无法全面学习与推理漏洞语句的语义信息。

针对上述方法的不足，本文提出一种基于特征依赖图的源代码漏洞检测方法（FBLD），该方法包括 3 个部分，分别是特征依赖图（FDG, feature dependence graph）、FDG 生成算法和面向 FDG 的检测网络（FDN, FDG detection network）。具体贡献如下。

1) 提出一种针对函数片的源代码表征结构——特征依赖图。与现有漏洞检测方法采用的源代码表征结构不同，FDG 不仅提取源代码的全部控制依赖和数据依赖语义信息，而且深度分析了与潜在风险语句相关的控制依赖和数据依赖，并在图中显式维护候选漏洞语句的两类语义信息，有助于漏洞检测神经网络获取源代码包含的潜在漏洞信息，做出更准确的判断。

2) 提出一种面向函数片的特征依赖图生成算法。首先，根据漏洞语法特征获得函数中的若干条

候选漏洞语句。其次，分别分析控制依赖图（CDG, control dependence graph）和数据依赖图（DDG, data dependence graph）中候选漏洞语句的控制依赖链和数据依赖链。然后，在控制依赖图和数据依赖图上，分别标注控制依赖链和数据依赖链，得到特征控制依赖图（FCDG, feature control dependence graph）和特征数据依赖图（FDDG, feature data dependence graph)。最后，将 FCDG 和 FDDG 融合为 FDG。

3) 提出一种面向 FDG 的检测网络 FDN，将对源代码函数片的漏洞检测工作转化为函数片对应的 FDG 检测。FDN 通过图学习网络获取 FDG 中的语义信息，可以动态调整 FDG 中不同边类型对于漏洞检测目标的重要性。同时，为了便于高效实施漏洞检测，FDN 通过检测网络获取 FDG 的全局表示向量并提取全局特征。

## 1　源代码漏洞检测方法设计

基于特征依赖图的源代码漏洞检测方法框架如图 1 所示，该方法由 FDG 生成、节点嵌入和源代码漏洞检测 3 个部分组成，各部分的主要操作设计如下。

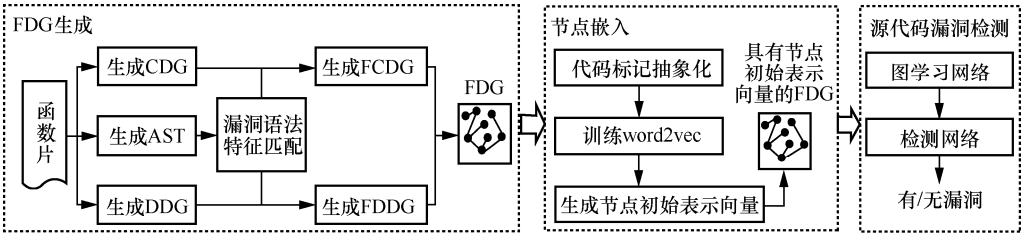


图 1　基于特征依赖图的源代码漏洞检测方法框架

1) FDG 生成。首先，使用代码分析工具 Joern 生成函数片对应的 AST、CDG 和 DDG。其次，在 AST 上匹配漏洞语法特征获取候选漏洞语句。然后，分析候选漏洞语句的控制依赖链和数据依赖链，修改控制依赖图和数据依赖图的节点属性和边属性，分别得到 FCDG 和 FDDG。最后，将 FCDG 和 FDDG 融合为 FDG。

2) 节点嵌入。首先，重命名函数片中的变量名和函数名，完成代码标记抽象化。然后，使用数据集中全部函数片的代码标记训练词向量模型 word2vec。最后，使用预训练的 word2vec 模型生成 FDG 的节点初始表示向量，得到具有节点初始表示向量的 FDG。

3) 源代码漏洞检测。首先将 FDG 输入预训练的源代码漏洞检测网络 FDN，由图学习网络部分学习并感知 FDG 中包含的语义信息，并在学习感知过程中更新节点表示向量。由检测网络进行全局池化并获取 FDG 的全局表示向量，然后对 FDG 的全局向量进行降维操作得到 FDG 的低维表示向量，最后采用线性层检测 FDG 的低维向量并得到漏洞检测结果。

## 2　特征依赖图

### 2.1　方法分析

基于序列的方法将源代码视为标记序列，这种表征方法容易忽略源代码中包含的数据信息和控制信息，从而导致误报的发生。以图 2 所示的函数片为例进行分析，基于序列的方法认为“free(data)”出现在“data”的使用之前，将忽略变量“flag1”的值为−1、“flag2”的值为 1 以及存在控制语句“if”的情况，从而误报该函数片存在内存释放重用漏洞，即“use_after_free”漏洞。实际上，因为“flag2”值大于 0，不会执行“free”函数，后续的输出操作是合法的堆内存访问，但是序列方法难以感知此类复杂的控制信息和数据信息。基于序列的方法 VulDeePecker 和 SySeVR 根据候选漏洞语句的数据依赖信息和控制依赖信息对源代码进行切片，尽管考虑了源代码的控制依赖和数据依赖相关的语义信息，但是未将这些信息与候选漏洞语句直接关联，使神经网络很难学习并感知这些关键信息，从而使神经网络无法做出准确判断。

基于图结构的源代码漏洞检测方法使用边连接代码语句和其相关的数据依赖和控制依赖节点，与基于序列的方法相比，该类方法更容易获取候选漏洞语句相关的语义信息。在现有基于图结构的方法中，一部分方法仅采用单一代码表示图来表示源代码，一部分方法则组合多种代码表示图以表征源代码。这些方法采用的代码表示图均未显式维护与候选漏洞语句相关的边或节点。

```
1)  void example_uaf(char * content)
2)  {
3)  int flag1 = –1;
4)  int flag2 = 1;
5)  char * data;
6)  int len = strlen(content)
7)  data = (char *)malloc(len);
8)  strncpy(data, content, len);
9)  if (flag1<0)
10) {
11) if(flag2<0)
12) {
13) free(data);
14) printf("%d\n",len);
15) }
16) }
17) printf("%s\n",data);
18) }
```

图 2 函数片示例

为此，本文使用 FDG 表征源代码。首先，FDG 作为一种基于图的代码表征结构，语句间的语义关系通过边结构维护，使神经网络更容易捕获语义信息。其次，FDG 通过漏洞语法特征获取候选漏洞语句，在图中显式维护与候选漏洞语句相关的控制信息和数据信息，使神经网络可以充分获取与候选漏洞语句相关的语义信息。最后，FDG 保留与候选漏洞语句的语义信息无关的边和节点，因为这些边和节点共同构成候选漏洞语句的上下文环境，从另一个角度反映了现实软件源代码中漏洞发生场景的复杂性，保留这些信息可以检验源代码漏洞检测方法的稳健性。

### 2.2 相关定义

FDG 是一种图类型的源代码表征结构，对于函数片 $f$ 生成的 FDG，可以形式化记为 $f_{\text{fdg}}=(V,E,X,D)$，其中，$V$ 表示节点集；$E$ 表示有向边集合；$X$ 表示节点属性集，用于保存节点的类型属性，节点类型分为 2 种，分别为候选漏洞语句对应的节点和其他语句对应的节点；$D$ 保存边的类型属性，FDG 中边类型分为 4 种，分别为控制依赖边（CDE, control dependence edge）、特征控制依赖边（FCDE, feature control dependence edge）、特征数据依赖边（FDDE, feature data dependence edge)、数据依赖边（DDE, data dependence edge)。

FCDG 是一种维护候选漏洞语句和其他语句控制依赖信息的图结构，对于函数片 $f$ 生成的 FCDG，可以形式化记为 $f_{\text{fcdg}}=(V,E,X,D)$，它的边类型分为 2 种，与候选漏洞语句相关的控制依赖边记为 FCDE，其他语句对应的控制依赖边记为 CDE。

FDDG 是一种维护候选漏洞语句和其他语句数据依赖信息的图结构，对于函数片 $f$ 生成的 FDDG，可以形式化记为 $f_{\text{fddg}}=(V,E,X,D)$，它的边类型分为 2 种，与候选漏洞语句相关的数据依赖边记为 FDDE，其他语句对应的数据依赖边记为 DDE。

本文提出控制依赖链反映代码语句的多级控制依赖关系。对于代码语句而言，其直接控制依赖决定它的执行流方向，其直接控制依赖的控制依赖也影响它的执行。如图 3 所示，代码语句 b 是 a 的直接控制依赖，c 是 b 的直接控制依赖，因此代码语句 c 的执行效果也影响 a 的执行。对于源代码漏洞检测而言，使神经网络获取候选漏洞语句的控制依赖链信息，有助于神经网络做出准确判断。

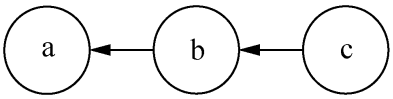


图 3 控制依赖链示例

与控制依赖链同理，本文提出数据依赖链反映变量的初始化和更新过程。控制依赖链可以反映候选漏洞语句所涉及变量的处理过程，使神经网络捕获更全面的数据信息，从而提高漏洞检测准确率。

### 2.3 特征依赖图生成过程

本文提出一种函数片 $f$ 到 FDG 的生成算法——FDG 生成算法（如算法 1 所示）。该算法包括获取候选漏洞语句、生成特征控制依赖图、生成特征数据依赖图和生成特征依赖图 4 个步骤。

**算法 1** FDG 生成算法

**输入** 函数片 $f$

**输出** 一个以 FDG 为元素的集合 $S$

1) 生成函数片 $f$ 对应的抽象语法树 $f_{\text{ast}}$
2) 在 $f_{\text{ast}}$ 上进行漏洞语法特征匹配，生成 $f_{\text{ast}}$ 的候选漏洞语句集合 $S_{\text{cvs}}$
3) $S \leftarrow \varnothing$
4) 生成函数片 $f$ 对应的 $f_{\text{cdg}}$ 和 $f_{\text{ddg}}$
5) for each cvs $\in S_{\text{cvs}}$ do
6) $f_{\text{fcdg}} \leftarrow f_{\text{cdg}}$
7) 令 cvs 的控制依赖节点集合 $S_c \leftarrow \varnothing$
8) $S_c \leftarrow S_c \cup$ cvs
9) while $S_c$ do
10) 从 $S_c$ 中移出元素 cvs
11) for each edge$_{\text{fcdg}} \in f_{\text{fcdg}}$ do

```
12)        if edge_fcdg.dst==cvs and edge_fcdg.label≠
             “fcde” then
13)          S_c ← S_c ∪ edge_fcdg.src
14)          edge_fcdg.dst.type ← “bad”
15)          edge_fcdg.label ← “fcde”
16)        else if edge_fcdg.dst ≠ cvs and edge_fcdg.
             label ≠ “fcde” then
17)          edge_fcdg.dst.type ← “good”
18)          edge_fcdg.label ← “cde”
19)        end if
20)        edge_fcdg.src.type ← “good”
21)      end for
22)    end while
23)    f_fddg ← f_ddg
24)    令 cvs 的数据依赖节点集合 S_d ← ∅
25)    S_d ← S_d ∪ cvs
26)    从 S_d 中移出元素 cvs
27)      for each edge_fddg ∈ f_fddg do
28)        if edge_fddg.dst==cvsand edge_fddg.label ≠
             “fdde” then
29)          S_d ← S_d ∪ edge_fddg.src
30)          edge_fddg.label ← “fdde”
31)        else if edge_fddg.dst == cvs and
             edge_fddg. label ≠ “fdde” then
32)          edge_fddg.label ← “dde”
33)        end if
34)      end for
35)      f_fdg ← f_fcdg ∪ f_fddg
36)      S ← S ∪ f_fdg
37) end for
```

1) 获取候选漏洞语句

候选漏洞语句指具有漏洞语法特征的潜在漏洞语句。本步骤采用的漏洞语法特征由 SySeVR[3]提出，包括函数调用、数组用法、指针用法和算法表达式 4 种，其中，函数调用类型的漏洞语法特征包括 811 种 C/C++的敏感 API；SySeVR 提出的 4 种漏洞语法特征对应 126 种 CWE 类型，覆盖了常见漏洞类型。本文方法旨在利用 4 种漏洞语法特征更大程度地获取候选漏洞语句相关的语义信息，从而提高漏洞检测准确率，降低误报率。

算法 1 的步骤 1)～步骤 2)获取函数片 $f$ 的候选漏洞语句。首先使用代码分析工具 Joern 生成函数片 $f$ 对应的抽象语法树 $f_{ast}$。然后在 $f_{ast}$ 上匹配 4 种漏洞语法特征，得到候选漏洞语句。具体而言，在遍历 $f_{ast}$ 节点过程中，若节点包含的代码语句为函数调用且属于 811 种敏感 API 之一，则该代码语句为敏感函数调用；若节点包含的代码语句包含"[]"字符，则为数组用法；同理，代码语句包含"*"字符，视为指针用法，若包含"+–*/^"等运算符，则将该代码语句视为算法表达式。代码语句符合以上 4 种规则时，将该语句视为候选漏洞语句，并将其加入 $S_{cvs}$。当遍历完 $f_{ast}$ 的全部节点时，得到函数片 $f$ 对应的候选漏洞语句集合 $S_{cvs}$。

图 2 所示的函数片示例对应的部分 AST 如图 4 所示，其中“free(data)”是库函数调用，同“free”函数属于 811 种敏感 API 之一，因此“free(data)”是一条候选漏洞语句。

2) 生成特征控制依赖图

生成特征控制依赖图的本质是在控制依赖图

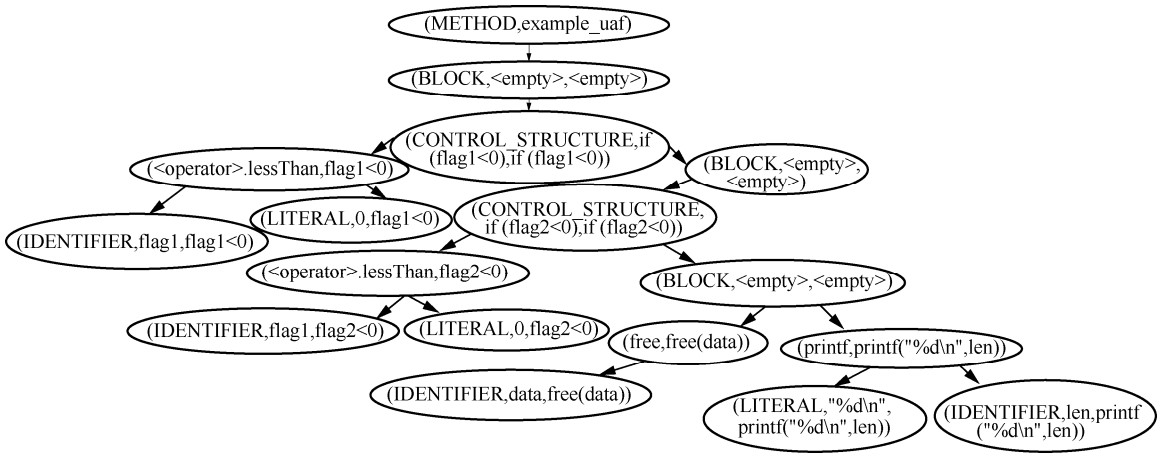


图 4　函数片示例对应的部分 AST

上标注候选漏洞语句对应的控制依赖链。在 FCDG 中，候选漏洞语句的控制依赖链的每条边被标注为 FCDE，与候选漏洞语句无关的控制依赖边被标注为 CDE。算法 1 的步骤 6)～步骤 22)为生成 FCDG 的过程。

函数片 $f$ 对应的 FCDG 记为 $f_{\text{fcdg}}$，$f_{\text{fcdg}}$ 被初始化为函数片 $f$ 对应的控制依赖图，然后在计算候选漏洞语句的控制依赖链过程中，修改原始控制依赖图得到 $f_{\text{fcdg}}$，即特征控制依赖图。在候选漏洞语句对应的控制依赖节点集 $S_c$ 初始化时，$S_c$ 仅包含候选漏洞语句一个元素，算法 1 从 $S_c$ 中取出候选漏洞语句并开始查找它的控制依赖节点。在遍历 $f_{\text{fcdg}}$ 各边的过程中，当某条边的目的节点是候选漏洞语句时，将该边的目的节点属性设置为“bad”，表示此节点对应候选漏洞语句；同时，将该边的属性设置为特征控制依赖边 FCDE，表示该边为候选漏洞语句对应控制依赖链的一部分。此外，将该边对应的源节点加入 $S_c$ 中继续计算其控制依赖节点。如果某条边的目的节点不是候选漏洞语句，则将该边的属性设置为控制依赖边 CDE。除候选漏洞语句对应的节点外，其余节点属性设置为“good”，表示不包含候选漏洞语句。当 $S_c$ 为空时，控制依赖链标注完毕，得到当前处理的候选漏洞语句的 $f_{\text{fcdg}}$。

以图 4 中的候选漏洞语句“free(data)”为例，其对应的 FCDG 如图 5 所示。候选漏洞语句“free(data)”的控制依赖链由 2 条 FCDE 组成。

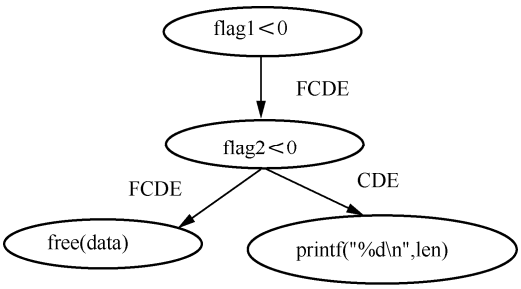


图 5 候选漏洞语句“free(data)”对应的 FCDG

3) 生成特征数据依赖图

生成特征数据依赖图的本质是在数据依赖图上标注候选漏洞语句对应的数据依赖链。在 FDDG 中，候选漏洞语句对应数据依赖链的每条边被标注为 FDDE，与候选漏洞语句无关的边被标注为 DDE。FDDG 的生成过程如算法 1 的步骤 24)～步骤 34)所示。

与生成 FCDG 的过程类似，函数片 $f$ 对应的 FDDG 记为 $f_{\text{fddg}}$，它被初始化为函数片 $f$ 对应的数据依赖图。在遍历 $f_{\text{fddg}}$ 的边集时，如果某条边的目的节点是候选漏洞语句，则将该边的属性标注为 FDDE，表示源节点是候选漏洞语句的数据依赖节点。同时将源节点加入候选漏洞语句的数据依赖节点集合 $S_d$。在下一轮循环时，取出上一轮保存的源节点，继续查找其控制依赖节点。通过上面的递归过程，可以查找候选漏洞语句的数据依赖链，并修改边属性为 FDDE，标注候选漏洞语句的数据依赖链。当 $S_d$ 为空时，候选漏洞语句的数据依赖链标注完毕，得到特征数据依赖图 $f_{\text{fddg}}$。

以图 4 中的候选漏洞语句“free(data)”为例，其对应的部分 FDDG 如图 6 所示。候选漏洞语句“free(data)”的数据依赖链由 2 条 FDDE 组成。

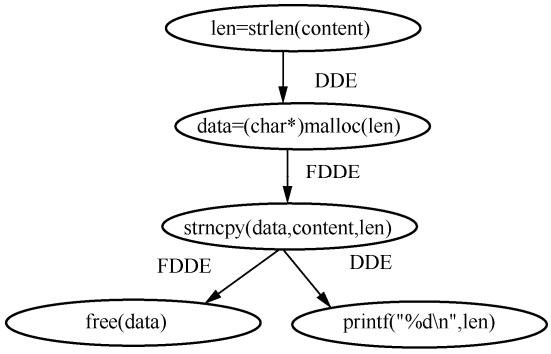


图 6 候选漏洞语句“free(data)”对应的 FDDG

4) 生成特征依赖图

特征依赖图由特征控制依赖图和特征数据依赖图融合而来，融合过程如算法 1 的步骤 35)所示。在融合过程中，首先将 FDG 的节点集设置为 FCDG 节点集和 FDDG 节点集的并集，然后将 FDG 的边集设置为 FCDG 边集和 FDDG 边集的并集，最后将节点属性和边属性合并。

此外，每条候选漏洞语句对应特有的 FDG，因此一条候选漏洞语句处理完成时，将生成的 FDG 加入集合 $S$。当处理完毕函数片 $f$ 的全部候选漏洞语句，得到集合 $S$，$S$ 包含函数 $f$ 对应的全部 FDG。

以图 4 中的候选漏洞语句“free(data)”为例，经过 FCDG 和 FDDG 融合过程，生成的部分 FDG 如图 7 所示。

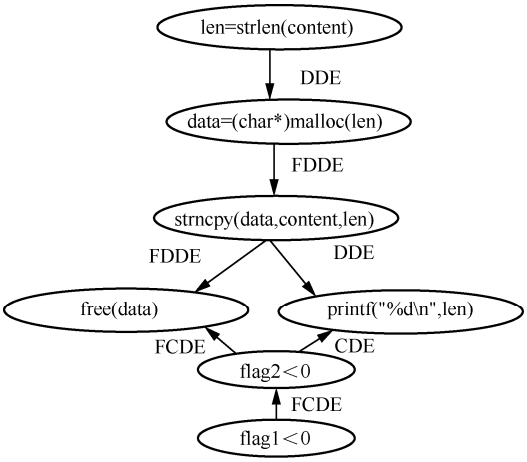


图 7　候选漏洞语句“free(data)”对应的 FDG

### 2.4　算法复杂度分析

对于每条候选漏洞语句，算法 1 生成其对应的 FDG，算法 1 处理控制依赖图得到特征控制依赖图，设控制依赖图平均节点数量为 $n$，$n$ 个节点最差情况下有 $n(n+1)$条有向边。在查找候选漏洞语句的控制依赖链时，需要遍历全部边以计算下一个控制依赖节点，最坏情况下，对全部边进行 $n$ 次遍历，因此时间复杂度为 $O(n^3)$。特征数据依赖图生成的时间复杂度同理，因此算法的时间复杂度为 $O(n^3)$。算法执行过程中，保存的主要数据是候选漏洞语句的控制依赖或数据依赖节点集，最坏情况下保存全部 $n$ 个节点，因此空间复杂度为 $O(n)$。

在实际的 FDG 生成过程中，因为控制依赖图或数据依赖图为函数片的代码表示结构，源代码中控制依赖关系和数据依赖关系有限，因此大部分语句节点间均无连接边。此外，在真实函数片中，控制语句的嵌套层数和数据依赖的层数有限，因此生成 FDG 的实际执行时间适中，具备可行性，5.5 节的实验数据证明了这一结论。

## 3　节点嵌入

节点嵌入指设置 FDG 的节点初始表示向量。图神经网络的本质是通过节点间的信息交换和聚合邻居信息以获得新的节点表示向量[15-16]，因此需要对输入图神经网络的 FDG 节点设置初始表示向量。节点嵌入阶段的输入是 FDG，输出是具有节点初始表示向量的 FDG。节点嵌入的处理过程包括以下 3 个步骤。

1) 代码标记抽象化。首先将函数片 $f$ 拆分为单个代码标记的组合，记为 $f_1=\{t_1, t_2,\cdots,t_n\}$，其中 $t_i$ 为单个代码标记。然后依次遍历 $t_i(1\leqslant i\leqslant n)$，若 $t_i$ 为变量名，则将 $t_i$ 及其相同的变量标记抽象化为 $\text{var}_i$，若 $t_i$ 为函数名，则将 $t_i$ 及其相同的函数标记抽象化为 $\text{fun}_i$，得到抽象化后的代码标记组合为 $f_T=\{t'_1,t'_2,\cdots,t'_n\}$。最后，遍历 FDG 的节点集，修改经过更新的变量名和函数名，得到不含具体变量名和函数名的 FDG。经过代码标记抽象化后，具体的变量名和函数名被抽象名称替代，使神经网络更易感知 FDG 节点间语义关系的学习，减小具体的变量名和函数名对神经网络感知过程的影响。

2) word2vec 训练。此步骤仅在首次节点嵌入时执行。word2vec 是一种词嵌入模型，它的主要思想是构造一个向量空间，使具有相似上下文的词在向量空间中彼此接近，不同上下文的词在向量空间中距离较大[17]。本文使用 word2vec 模型生成单个代码标记的表示向量，首先将经过标记抽象化的函数片拆分为单个代码标记的组合，然后将全部代码标记组织为一个词库文件，最后以词库文件为 word2vec 模型的输入，训练 word2vec 模型，获得最佳模型参数。

3) 节点初始表示向量生成。遍历 FDG 的每个节点，将节点对应的代码语句拆分为单个代码标记的组合，对于每个代码标记，使用预训练的 word2vec 生成其表示向量，将全部代码标记的表示向量相加得到节点初始表示向量。此外，为使初始表示向量包含节点的类型信息，在节点初始表示向量起始处增加一个维度，候选漏洞语句对应的节点此维度的值设置为 1，其他节点此维度的值设置为 0。因此节点 $n$ 的初始表示向量可以记为 $\boldsymbol{V}_n^{\text{init}}=\{p_n,a_1,\cdots,a_m\}$，其中，$p_n$ 表示节点 $n$ 的类型，即该节点是否为候选漏洞语句；$a_i(1\leqslant i\leqslant m)$为表示向量的维度，本文设置 $m$=128。

## 4　源代码漏洞检测

### 4.1　漏洞检测网络架构

本文所提的源代码表征结构 FDG 是一种异构图，包含 2 种不同的节点类型和 4 种不同的边类型，FDG 使漏洞检测网络更容易学习并感知源代码中与候选漏洞语句相关的控制和依赖信息。对于源代码漏洞检测而言，FDG 中候选漏洞语句对应的节点相比其他节点而言更重要，其对应的控制依赖和数据依赖也比其他节点的语义信息更重要。现有基于图结构的源代码漏洞检测方法基于 GGNN 构建漏洞检测网络，GGNN 适合同构图的学习和感知，并未考虑不同类型的节点和边在图表示方面的差异性。

为克服以上问题，本文提出一种针对 FDG 的漏洞检测网络 FDN。FDN 由图学习网络和检测网络组成，其网络结构如图 8 所示。FDN 使用函数片训练集最小化损失函数，通过梯度下降算法获得最优网络参数，得到预训练的 FDN。当检测新的函数样本时，首先生成函数片对应的 FDG，然后将 FDG 输入预训练的 FDN，得到源代码漏洞检测结果。

### 4.2 图学习网络

图学习网络通过学习感知 FDG 的邻居节点信息，更新节点表示向量。图学习网络由异构图转换器（HGT, heterogeneous graph transformer）[18]和线性整流函数——修正线性单元（ReLU, rectified linear unit）组成。HGT 通过异构注意力机制学习并感知 FDG 节点间的信息。ReLU 是一种高效的激活函数，可以有效避免梯度消失和梯度爆炸问题。对于 FDG，HGT 的学习感知过程包括异构注意力计算、邻居信息传递和目标节点聚合 3 个步骤。

1) 异构注意力计算

此步骤的目的是计算 FDG 中每个节点 $t$ 的全部邻居节点相对于节点 $t$ 的注意力头分数。$s$ 的全部邻居节点集合为 $N(t)$，注意力头数为 $h$，$t$ 和 $s$ 之间连接边为 $e$，层数为 $l$，节点表示向量的维度为 $d$。此外，定义函数 $\tau(n)$，$\tau$ 用于获取节点 $n$ 的类型，值域为{‘good’, ‘bad’}。定义函数 $\phi(e)$，$\phi$ 用于获取边 $e$ 的类型，值域为{‘cde’, ‘fcde’, ‘dde’, ‘fdde’}。对于第 $i$ 个注意力头分数的计算，首先使用式(1)计算源节点 $s$ 的键向量 $\boldsymbol{K}$。

$$\boldsymbol{K}^i(s)=\boldsymbol{K}\_\text{Linear}^i_{\tau(s)}\left(\boldsymbol{H}^{(l-1)}[s]\right) \tag{1}$$

其中，$\boldsymbol{K}^i(s)$为第 $i$ 个注意力头上节点 $s$ 的键向量，$\boldsymbol{K}$_Linear 为第 $i$ 个注意力头上和节点类型相关的线性映射，$\boldsymbol{H}$ 为节点的表示向量。对于不同类型的节点使用不同的线性映射，这使候选漏洞语句节点和其他节点在计算键向量 $\boldsymbol{K}$ 时具有不同的特征表示空间，并且这些线性映射是可学习的，从而可以优化不同类型的节点表示。然后，使用式(2)计算查询向量 $\boldsymbol{Q}$。

$$\boldsymbol{Q}^i(t)=\boldsymbol{Q}\_\text{Linear}^i_{\tau(t)}\left(\boldsymbol{H}^{(l-1)}[t]\right) \tag{2}$$

其中，$\boldsymbol{Q}^i(t)$为第 $i$ 个注意力头上节点 $t$ 的查询向量，$\boldsymbol{Q}$_Linear 为第 $i$ 个注意力头上和节点类型相关的线性映射。使用式(3)计算目标节点 $t$ 与邻居 $s$ 注意力分数。

$$\text{ATT\_head}(s,e,t)^i=\left(\boldsymbol{K}^i(s)\boldsymbol{W}^{\text{ATT}}_{\phi(e)}\boldsymbol{Q}^i(t)^{\text{T}}\right)\frac{\mu_{<\tau(s),\phi(e),\tau(t)>}}{\sqrt{d}} \tag{3}$$

其中，$\mu$ 为与源节点类型、连接边类型和目的节点类型相关的缩放因子，此外，式(3)在计算键向量 $\boldsymbol{K}$ 和查询向量 $\boldsymbol{Q}$ 的相似度时引入与边类型相关转换矩阵 $\boldsymbol{W}$，即在相似度计算时，不同的边类型使用不同的 $\boldsymbol{W}$，这使注意力计算可以考虑源节点和目的节点间的连接关系。在漏洞检测场景下，$\boldsymbol{W}$ 作为 FDN 的一类网络参数，在训练过程中，因为 FCDE 和 FDDE 连接的节点语句包含更多与候选漏洞语句相关的信息，梯度下降算法将调整 FCDE 和 FDDE 的权重，以使 FDN 网络损失更小，漏洞检测性能更高。

对于每一个注意力头，当得到目标节点 $t$ 与邻居 $s$ 注意力分数后，使用式(4)组合每个注意力头上的注意力分数，并经过 Softmax 函数处理，将注意力分数转换为概率值。

$$\text{Attention}_{\text{HGT}}(s,e,t)=\underset{\forall s\in N(t)}{\text{Softmax}}\left(\Big\|_{i\in[1,h]}\text{ATT\_head}^i(s,e,t)\right) \tag{4}$$

在 FDG 上进行异构注意力计算的示意如图 9 所示。图 9 中，中间节点是候选漏洞语句节点，其他节点对应良性语句。对于中间节点而言，每个邻居节点对其产生不同程度的影响，异构注意力计算旨在获取其每个邻接节点相对于中间节点的重要性分数。因为 FCDE 和 FDDE 分别连接候选漏洞语句的控制依赖节点和数据依赖节点，这 2 种语义信息相比于其他节点的语义信息更重要，所以 FDN 会自动增加 FCDE 和 FDDE 的连接强度，以适应漏洞检测场景。

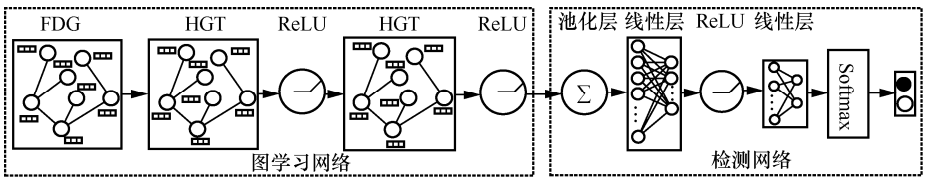


图 8 FDN 网络结构

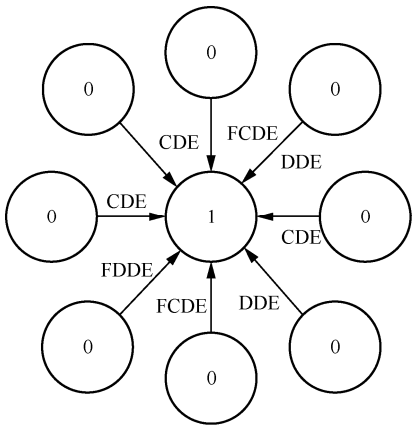


图 9　异构注意力计算示意

2) 邻居信息传递

首先，在第 $i$ 个注意力头的计算消息头

$$\mathrm{MSG\_head}(s,e,t)^i = M\_\mathrm{Linear}^i_{\tau(s)}\left(\boldsymbol{H}^{(l-1)}[s]\right)\boldsymbol{W}^{\mathrm{MSG}}_{\phi(e)} \tag{5}$$

其中，$\boldsymbol{W}$ 为计算消息头场景下与边类型相关的转换矩阵，$M\_\mathrm{Linear}_i$ 为第 $i$ 个注意力头和节点类型相关的线性映射。然后组合 $h$ 个消息头，即

$$\mathrm{Message}_{\mathrm{HGT}}(s,e,t) = \Big\Vert_{i\in[1,h]} \mathrm{MSG\_head}^i(s,e,t) \tag{6}$$

3) 目标节点聚合

首先，通过将目标节点 $t$ 的邻居与其对应的注意力分数加权求和，获得聚合消息向量为

$$\tilde{\boldsymbol{H}}^{(l)}[t] = \bigoplus_{\forall s\in N(t)}\left(\mathrm{Attention}_{\mathrm{HGT}}(s,e,t)\mathrm{Message}_{\mathrm{HGT}}(s,e,t)\right) \tag{7}$$

然后，将聚合消息向量输入激活函数，通过针对特定节点类型的线性映射将聚合消息向量映射到目标节点类型的特征空间，与上一层目标节点的表示向量相加，得到更新后的目标节点向量为

$$\boldsymbol{H}^{(l)}[t] = A\_\mathrm{Linear} - A_{\tau(t)}\left(\sigma(\tilde{\boldsymbol{H}}^{(l)}[t])\right) + \boldsymbol{H}^{(l-1)}[t] \tag{8}$$

其中，$A\_\mathrm{Linear}$ 为与节点类型相关的线性映射，$\sigma$ 为激活函数。

### 4.3　漏洞检测

检测网络旨在进行全局池化和全局表示向量降维，并进行漏洞检测。首先，将经过信息交换的 FDG 输入检测网络，经过全局均值池化过程处理后，得到 FDG 的全局表示向量 $\boldsymbol{H}_{\mathrm{fdg}}$

$$\boldsymbol{H}_{\mathrm{fdg}} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{9}$$

其中，$N$ 表示 FDG 的节点个数，$x$ 表示单个节点的表示向量。全局表示向量 $\boldsymbol{H}_{\mathrm{fdg}}$ 不仅反映了此时全部节点的状态，而且蕴含数据依赖和控制依赖关系，整体反映了 FDG 的语义信息。

为提取全局表示向量的有效特征，剔除与漏洞信息无关的冗余特征，采用线性层对全局表示向量进行降维操作，得到 FDG 的低维表示向量。在 FDN 中，该线性层的输入维度为节点维度，输出维度为原节点维度的 $\frac{1}{2}$。

此时，FDG 的低维表示向量蕴含 FDG 所包含的全部信息。于是采用线性层进行漏洞检测。该线性层的输入是 FDG 的低维表示向量，线性层内部对 FDG 的低维表示向量进行加权计算后，得到线性层输出维度的 2 个神经元激活值，它们分别代表 FDG 有无漏洞的置信度，即源代码漏洞检测结果。最后，使用 Softmax 函数将 2 个神经元的激活值转换为概率值，得到漏洞检测结果的概率表示。

## 5　实验与结果

### 5.1　数据集

为验证本文所提方法的有效性，选取软件保障参考数据集（SARD, software assurance reference dataset）和来源于真实软件项目的 Devign 数据集[13]、Reveal 数据集[7]进行验证实验。

SARD 由美国国家标准与技术研究院提出，SARD 中的代码一部分是生产环境中源程序的半合成代码，一部分是用于学术研究的人工合成代码。数据集中源代码有 3 种类型，分别是有漏洞的源代码、无漏洞的源代码以及修补过漏洞的源代码。

SARD 包含 118 种 CWE 漏洞，本文选择数据量较多的 7 种漏洞类型进行验证。SARD 中源代码并未以函数片形式存在，因此本文使用 Ecplise CDT 来逐个解析源代码文件的结构，获取源代码文件中的全部声明，然后遍历每个声明判断其是否为函数声明。如果是，则获取它的函数定义，另存为函数片。同时，判断函数声明中是否包含“good”或者“bad”关键字，如果包含，则将对应的标签值 0 或 1 写入函数片文件名，1 表示函数片存在漏洞，0 表示无漏洞。预处理后的 SARD 数据集信息如表 1 所示。

Devign 数据集由 Zhou 等[13]提出。该数据集从 Linux Kernel、Qemu、Wireshark 和 FFmpeg 项目中

表 1 预处理后的 SARD 信息

| 类型 | 名称 | 函数片/个 | FDG/个 | 有漏洞的 FDG/个 | 良性 FDG/个 |
|---|---|---|---|---|---|
| CWE20 | 不合适的输入验证 | 3 452 | 4 015 | 846 | 3 169 |
| CWE78 | 命令注入 | 1 7000 | 18 420 | 4 200 | 14 220 |
| CWE129 | 数组索引验证不当 | 11 208 | 10 019 | 2 977 | 7 042 |
| CWE190 | 整数上溢 | 25 913 | 28 943 | 6 925 | 22 018 |
| CWE400 | 资源耗尽 | 9 990 | 10 023 | 2 744 | 7 279 |
| CWE787 | 越界写入 | 14 797 | 14 980 | 4 210 | 10 770 |
| CWE789 | 分配失控 | 7 300 | 8 167 | 1 241 | 6 926 |

提取与安全问题相关的代码提交，通过比对提交前后函数片是否修改来确定是否存在漏洞。同时，由安全研究人员人工确认函数片的安全性，以保证数据集的质量[13]。该数据集能够反映现实软件项目代码的复杂性，可以较好地评估漏洞检测方法的性能。Devign 数据集中存在漏洞的函数片为 12 458 个，无漏洞的函数片为 14 855 个，每个函数片已经包含标签值。

Reveal 数据集由 Chakraborty 等[7]提出。该数据集通过跟踪 Linux Debian Kernel 和 Chromium 这 2 个开源项目的历史漏洞生成。这 2 个软件项目有长期的更新历史，分别代表了操作系统和浏览器这 2 个重要的软件领域，同时它们具有大量公开可用的漏洞报告，适合源代码漏洞检测方法的验证[7]。Reveal 数据集中存在漏洞的函数片为 2 238 个，无漏洞的函数片为 20 487 个，每个函数片已经包含标签值。经过 FDG 生成过程，Devign 数据集和 Reveal 数据集的信息如表 2 所示。

表 2 预处理后的数据集信息

| 数据集名称 | 函数片/个 | FDG/个 | 有漏洞的 FDG/个 | 良性 FDG/个 |
|---|---|---|---|---|
| Devign | 27 313 | 28 760 | 13 754 | 15 006 |
| Reveal | 22 725 | 20 109 | 2 753 | 17 356 |

### 5.2 实验环境配置

实验使用的 CPU 为 Intel(R) Xeon(R) Gold 5215，内存大小为 48 GB，显卡型号为 Tesla-V100。本文使用 Pytorch 1.11.0 和 Pytorch-geometric 2.0.4 实现 FBLD 方法，训练集与测试集的比例为 7:3，训练轮次为 100 轮，训练学习率设置为 0.001，训练过程中学习率随时间指数级衰减。图学习网络部分的 HGT 的注意力头数设置为 2。

### 5.3 评价标准

实验采用的评价指标包括准确率、精确率、召回率和 F1 分数。每个指标的计算方法为

准确率（A, accuracy）。准确分类的样本占总样本数量的比例，准确率计算方法为

$$A=\frac{\mathrm{TP+TN}}{\mathrm{TP+FP+TN+FN}} \tag{10}$$

精确率（P, precision）。在所有被判断为存在漏洞的样本中，判断正确的样本比例，精确率计算方法为

$$P=\frac{\mathrm{TP}}{\mathrm{TP+FP}} \tag{11}$$

召回率（R, recall）。被成功检测出的漏洞样本占所有漏洞样本的比例，召回率计算方法为

$$R=\frac{\mathrm{TP}}{\mathrm{FN+TP}} \tag{12}$$

F1 分数（F1, F1-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 计算方法为

$$\mathrm{F1}=\frac{2PR}{P+R} \tag{13}$$

在上述指标计算式中，TP（true positive）为漏洞样本检测为漏洞样本的样例数，TN（true negative）为良性样本检测为良性样本的样例数，FP（false positive）为漏洞样本检测为良性样本的样例数，FN（false negative）为良性样本检测为漏洞样本的样例数。

### 5.4 漏洞检测实验结果与分析

为验证本文提出的 FBLD 的源代码漏洞检测性能，在 SARD 数据集、Devign 数据集和 Reveal 数据集上分别采用 FBLD 和 7 种源代码漏洞检测先进方法（SySeVR 方法[3]、Reveal 方法[7]、Russell 方法[9]、VulDeePecker 方法[10]、μVulDeePecker 方法[11]、VulDeeLocator 方法[12]和 Devign 方法[13]）进行漏洞检测。8 种方法在 SRAD 上的漏洞检测评价指标如表 3～表 6 所示，在 Devign 数据集和 Reveal 数据集上的漏洞检测评价指标分别如图

**表 3　SARD 数据集上各方法准确率对比**

| 方法 | CWE20 | CWE78 | CWE129 | CWE190 | CWE400 | CWE787 | CWE789 |
|---|---|---|---|---|---|---|---|
| Russell | 72.45% | 92.71% | 85.97% | 87.49% | 88.58% | 74.42% | 89.24% |
| VulDeePecker | 75.19% | 95.71% | 83.31% | 86.97% | 91.26% | 75.62% | 89.85% |
| μVulDeePecker | 75.93% | 95.95% | 89.41% | 87.52% | 93.71% | 82.38% | 89.59% |
| SySeVR | 78.58% | 97.01% | 90.25% | 93.00% | 94.59% | 85.69% | 91.90% |
| VulDeeLocator | 76.68% | 97.06% | 91.37% | 93.39% | 95.28% | 84.65% | 90.94% |
| Devign | 77.93% | 96.66% | 90.19% | 95.53% | 96.47% | 85.09% | 91.97% |
| Reveal | 78.93% | 97.28% | 93.49% | 97.32% | **97.99**% | 85.58% | 92.46% |
| **FBLD** | **81.42**% | **98.37**% | **96.24**% | **98.17**% | 97.93% | **87.59**% | **94.72**% |

**表 4　SARD 数据集上各方法精确率对比**

| 方法 | CWE20 | CWE78 | CWE129 | CWE190 | CWE400 | CWE787 | CWE789 |
|---|---|---|---|---|---|---|---|
| Russell | 38.18% | 87.01% | 79.88% | 85.12% | 77.78% | 53.44% | 67.43% |
| VulDeePecker | 42.86% | **95.96**% | 75.61% | 82.53% | 83.72% | 55.56% | 68.93% |
| μVulDeePecker | 44.98% | 88.31% | 84.61% | 75.21% | 85.57% | **71.73**% | 66.27% |
| SySeVR | 49.36% | 92.94% | 83.33% | 90.16% | 88.64% | 66.75% | 75.37% |
| VulDeeLocator | 46.35% | 92.44% | 85.56% | 85.45% | 88.95% | 66.07% | 69.11% |
| Devign | 48.18% | 91.83% | 83.10% | 94.19% | 92.59% | 65.82% | 75.55% |
| Reveal | 50.00% | 93.02% | 84.77% | 94.89% | **93.35**% | 66.62% | 76.60% |
| **FBLD** | **54.55**% | 94.31% | **90.63**% | **97.00**% | 93.33% | 69.23% | **83.92**% |

**表 5　SARD 数据集上各方法召回率对比**

| 方法 | CWE20 | CWE78 | CWE129 | CWE190 | CWE400 | CWE787 | CWE789 |
|---|---|---|---|---|---|---|---|
| Russell | 49.65% | 79.76% | 70.54% | 57.83% | 83.14% | 51.02% | 56.41% |
| VulDeePecker | 53.19% | 84.76% | 67.18% | 57.76% | 85.51% | 54.66% | 60.44% |
| μVulDeePecker | 62.31% | 94.79% | 78.67% | 71.36% | 92.67% | 61.59% | 64.14% |
| SySeVR | 63.83% | 94.04% | 83.98% | 79.42% | 92.64% | 95.12% | 69.30% |
| VulDeeLocator | 65.84% | 94.88% | 85.39% | 87.25% | 94.50% | 93.28% | 73.01% |
| Devign | 62.65% | 93.67% | 84.08% | 86.64% | 95.04% | 94.75% | 69.70% |
| Reveal | 65.01% | 95.23% | 95.20% | 93.86% | **99.98**% | 94.90% | 72.52% |
| **FBLD** | **70.92**% | **98.80**% | **97.41**% | **95.27**% | 99.76% | **98.40**% | **80.74**% |

**表 6　SARD 数据集上各方法 F1 分数对比**

| 方法 | CWE20 | CWE78 | CWE129 | CWE190 | CWE400 | CWE787 | CWE789 |
|---|---|---|---|---|---|---|---|
| Russell | 43.17% | 83.23% | 74.92% | 68.87% | 80.37% | 52.20% | 61.43% |
| VulDeePecker | 47.47% | 90.01% | 71.15% | 67.96% | 84.61% | 55.11% | 64.40% |
| μVulDeePecker | 52.25% | 91.43% | 81.53% | 73.24% | 88.98% | 66.27% | 65.19% |
| SySeVR | 55.67% | 93.49% | 83.66% | 84.45% | 90.59% | 78.45% | 72.21% |
| VulDeeLocator | 54.40% | 93.64% | 85.47% | 86.34% | 91.64% | 77.35% | 71.00% |
| Devign | 54.47% | 92.74% | 83.59% | 90.26% | 93.79% | 77.68% | 72.51% |
| Reveal | 56.53% | 94.12% | 89.68% | 94.37% | **96.55**% | 78.28% | 74.50% |
| **FBLD** | **61.67**% | **96.51**% | **93.89**% | **96.13**% | 96.44% | **81.27**% | **82.30**% |

10～图 13 所示。

实验结果表明，FBLD 在 3 个数据集上的关键评价指标大多优于其他方法，其中，准确率比最优方法提高 0.87%～7.39%，精确率提高 2.22%～9.56%，召回率提高 1.50%～22.32%，F1 分数提高 1.86%～16.69%。在 SARD 上的检测实

验主要针对 7 种漏洞类型（CWE20、CWE78、CWE129 、 CWE190 、 CWE400 、 CWE787 、CWE789），针对 CWE20、CWE129、CWE190、CWE789 漏洞类型，FBLD 的 4 个评价指标（准确率、精确率、召回率、F1 分数）均优于其他方法。其原因如下。1) FDG 显式维护候选漏洞语句的控制依赖链和数据依赖链，区分候选漏洞语句和其他语句的节点类型，这使漏洞检测网络 FDN 更容易捕获与候选漏洞语句相关的语义信息。2) FDN 能够学习并感知 FDG 中丰富的语义信息，并自动调整不同边类型和节点类型对于漏洞检测的重要性，提高了 FBLD 的漏洞检测性能。

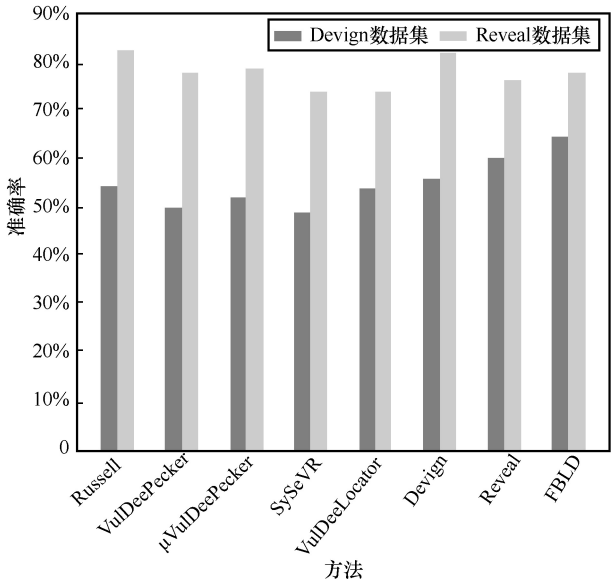


图 10 各方法在 Devign 和 Reveal 数据集上的准确率对比

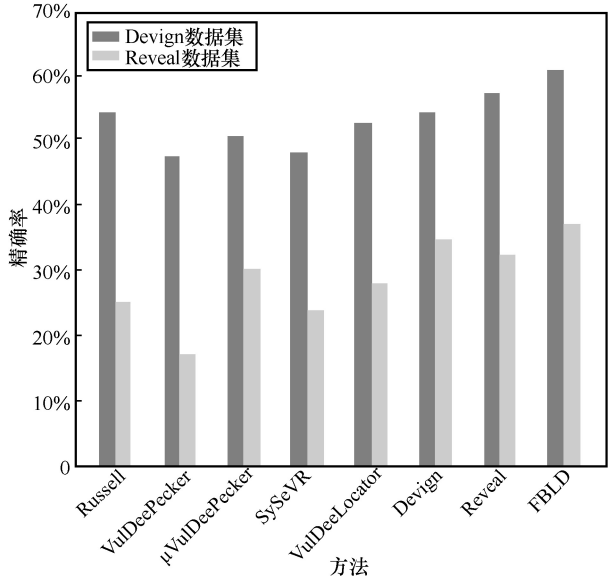


图 11 各方法在 Devign 和 Reveal 数据集上的精确率对比

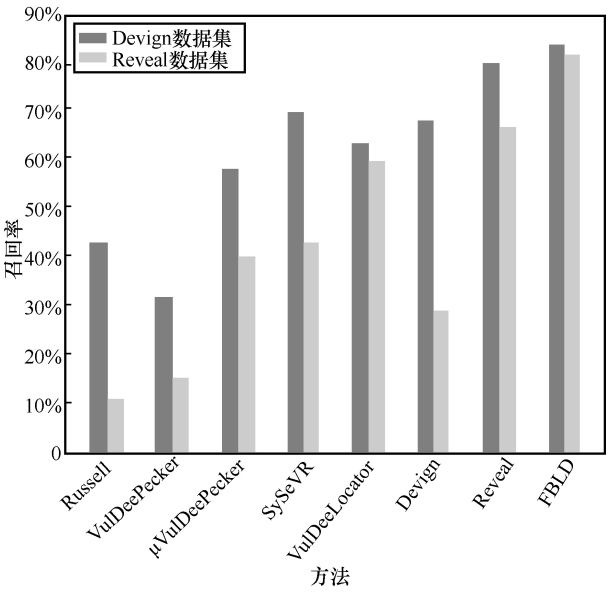


图 12 各方法在 Devign 和 Reveal 数据集上的召回率对比

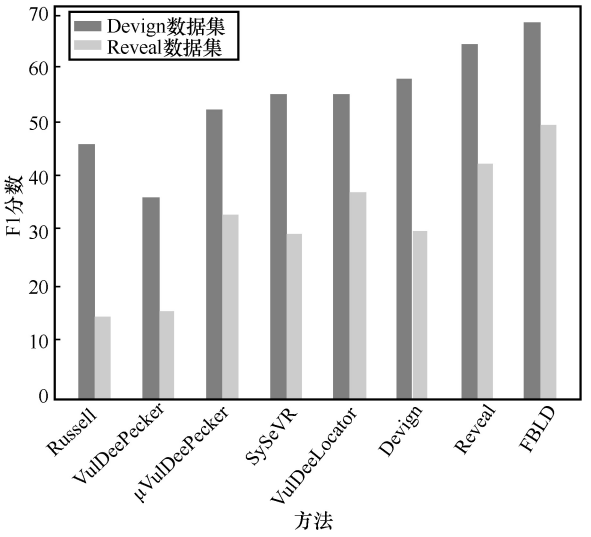


图 13 各方法在 Devign 和 Reveal 数据集上的 F1 分数对比

以 CWE787 越界写入漏洞为例，假设存在一个数组缓冲区，判断越界写入需要获得缓冲区的大小、待写入数据的大小，同时需要获取写入语句的控制信息，用于判断程序执行流是否会到达写入语句。在 FDG 生成过程中，内存写入函数符合敏感函数调用类型的漏洞语法特征，因此该函数被视为候选漏洞语句。内存写入函数的控制依赖链会被提取到 FDG 表示中，并且它的数据依赖链也会被纳入 FDG，这些变量可能是缓冲区指针，也可能是写入数据的大小。将上述关键信息生成的 FDG 输入 FDN 中进行推理与学习，FDN 就能对是否存在越界写入漏洞进行准确判断。从表 3～表 6 可知，针对 CWE400 漏洞类型，Reveal 方法的检测准确率为 97.99%，精

确率为93.35%，召回率为99.98%，F1 分数为96.55%；FBLD 的检测准确率为 97.93%，精确率为 93.33%，召回率为 99.76%，F1 分数为 96.44%。FBLD 的 4 个指标略低于 Reveal，均排名第二。其原因是 CWE400 为资源耗尽型漏洞，对此类漏洞的检查与判断需要获取资源数量、分配策略、回收策略等多种信息。FBLD 以控制依赖和数据依赖为主的信息获取方式未完整地获取这类信息，从而导致部分样本的误报。而 Reveal 使用 CPG 表示源代码，CPG 包含源代码的 AST、CFG、DFG 和 DDG 等多种表示图，尽管存在一些冗余信息，但较完整地包含了大部分语义信息。FBLD 为了显式维护候选漏洞语句的控制依赖链和数据依赖链，损失了部分信息，这使 Reveal 在 CWE400 上略优于 FBLD。

从表 4 可知，针对 CWE78 漏洞类型，VulDeePecker 的检测精确率最高，为 95.96%，FBLD 的精确率为 94.31%，略低于 VulDeePecker，排名第二。分析其原因如下，VulDeePecker 仅提取敏感 API 一种漏洞语法特征，并根据敏感 API 的数据依赖获取与其相关的语义切片。CWE78 是命令注入型漏洞，这种漏洞与 API 调用紧密相关（如 execve 等函数调用）。FBLD 提取包含敏感 API 在内的多种特征，多种特征的结合可能会使神经网络在部分情况下产生误判，导致 FBLD 对此类漏洞的检测精确率略低于 VulDeePecker。

不同于 SARD 的合成代码，Devign 和 Reveal 数据集来源于真实的开源软件项目，在对比检测实验中，7 种方法的评价指标均低于在 SARD 上的评价指标。由于 Reveal 数据集存在数据不平衡问题，导致 7 种方法的准确率相对于其他指标偏高，但这种不平衡反映了源代码漏洞检测的真实环境。因为在大多数情况下，存在漏洞的样本数量远小于良性样本的数量。在 Devign 和 Reveal 数据集上，FBLD 仍然表现良好，其重点指标优于其他方法。其中，FBLD 对漏洞样本检测的召回率最高提升 22.32%，表明 FBLD 可以发现更多存在漏洞的样本；FBLD 的 F1 分数最高提升 16.69%，表明 FBLD 提高了源代码漏洞检测的综合性能。分析其原因如下。

1) FBLD 提取源代码函数片的全部候选漏洞语句后，对每个候选漏洞语句生成对应的 FDG，因而每个函数片对应若干个 FDG，当函数片对应的某个 FDG 被源代码漏洞检测网络确认存在漏洞，则判定函数片存在漏洞。FBLD 依据 4 种漏洞语法特征，可以覆盖绝大部分 CWE 漏洞类型，与仅使用一种漏洞语法特征的 μVulDeePecker 和 VulDeePecker 相比，FBLD 方法在 Devign 数据集上召回率分别提高 163.12%和 44.03%。此外，FBLD 采用的 FDN 可以充分学习并感知候选漏洞语句的控制依赖信息和数据依赖信息，因而检测关键指标优于其他 7 种方法。

2) Reveal 虽然使用 CPG 表示源代码，但是未重点分析与候选漏洞语句密切相关的数据流和控制流，存在较多冗余信息，因此性能低于 FBLD。Devign 使用 AST、CFG、DFG 以及自然代码序列表示源代码，在一定程度上可以反映源代码特征，但是未在代码表示图中显式保存与漏洞信息直接相关的数据和控制信息，从而使神经网络较难学习代码表示图的信息。

3) VulDeeLocator 和 SySeVR 使用 4 种漏洞语法特征对源代码进行切片，最大程度获取了源代码的原始信息，因此在基于标记序列的 5 种方法（Russell 方法、VulDeePecker、SySeVR、μVulDeePecker 和 VulDeeLocator）中表现最佳，但是这 2 种方法未显式维护候选漏洞语句的数据依赖和数据依赖关系，增加了神经网络学习控制依赖和数据依赖信息的难度，因此漏洞检测性能整体略低于 FBLD、Devign、Reveal 等基于图结构的漏洞检测方法。

4) 与 VulDeeLocator 和 SySeVR 相比，μVulDeePecker 和 VulDeePecker 因仅使用敏感 API 一种漏洞语法特征进行函数切片，忽略了大量数组、指针和表达式相关的漏洞信息，因此性能低于 SySeVR 和 VulDeePecker。此外，μVulDeePecker 使用由控制依赖和数据依赖引起的语义信息，而 VulDeePecker 仅使用由数据依赖引起的语义信息，从而其漏洞检测性能低于 μVulDeePecker。与 VulDeePecker 使用双向长短期记忆神经网络处理语句的长距离上下文关系相比，Russell 方法使用 RNN 构建漏洞检测网络，RNN 处理长距离信息关联时依赖极易出现梯度消失现象，在一定程度上影响了漏洞检测性能。

上述实验结果表明，与其他 7 种方法相比，本文提出的 FBLD 方法在源代码漏洞检测方面的综合性能更好。

### 5.5　算法时间复杂性验证

本文在 Reveal 数据集上验证 FDG 生成算法的时间复杂性，Reveal 数据集中全部函数片生成的 FDG 共 20 109 个，表 7 记录了 Reveal 数据集生成的 FDG 的节点数量、边数量和消耗时间三类信息。数据表明，生成 FDG 的最大时间消耗为 78.96 s，

消耗时间的中位数为 2.53 s，证明 FDG 生成算法的时间复杂度可控，具备实际可行性。

**表 7 FDG 生成算法时间复杂性验证结果**

| 名称 | 最小值 | 中位数 | 75 分位数 | 最大值 |
|---|---|---|---|---|
| 节点数量/个 | 4 | 13 | 21 | 306 |
| 边数量/条 | 6 | 65 | 132 | 6 307 |
| 消耗时间/s | 0.001 | 2.53 | 14.19 | 78.96 |

本文在 Reveal 数据集上通过训练时间和检测时间对比了不同方法的漏洞检测模型时间复杂度，对比结果如表 8 所示，表 8 中各方法的训练轮数均为 100。实验结果表明，本文所提方法的训练时间和平均检测时间分别为 87 min 和 4.89 s，分析其原因为 FDN 包含 2 个 HGT 层和多个线性层，图神经网络的训练与计算消耗了较多时间。尽管本文的漏洞检测模型训练时间较长，但对于新样本的平均检测时间可控，且训练过程仅需一次，故本文方法具备一定的实际意义。

**表 8 漏洞检测模型时间消耗对比**

| 方法 | 训练时间/min | 平均检测时间/s |
|---|---|---|
| Russel | 18 | 0.95 |
| VulDeePecker | 24 | 1.24 |
| μVuDeePecker | 29 | 1.78 |
| SySeVR | 36 | 2.25 |
| VulDeeLocator | 32 | 2.12 |
| Devign | 49 | 3.40 |
| Reveal | 56 | 3.51 |
| **FBLD** | 87 | 4.89 |

## 6 结束语

针对现有源代码漏洞检测方法无法全面获取漏洞语句的相关信息，导致漏洞检测误报率高等问题，本文提出一种基于特征依赖图的源代码漏洞检测方法。采用一种新颖的源代码表示结构 FDG 分析候选漏洞语句的控制依赖链和数据依赖链，并在 FDG 中显式维护候选漏洞语句的控制依赖链和数据依赖链，更容易捕获候选漏洞语句的语义信息。本文提出的面向 FDG 的漏洞检测网络 FDN，可以有效推理并感知 FDG 包含的语义信息，同时可以动态调整漏洞语句相关的边和节点的重要性。在 3 个数据集上的实验结果表明，本文方法的源代码漏洞检测性能优于其他现有方法。

在未来研究中，将尝试设计漏洞语法特征更新机制以检测未知漏洞，同时，尝试更高效的代码表示结构和漏洞检测神经网络，进一步提高准确率并降低误报率，提升源代码漏洞检测方法的性能。

## 参考文献：


[1] LIN G J, WEN S, HAN Q L, et al. Software vulnerability detection using deep neural networks: a survey[J]. Proceedings of the IEEE, 2020, 108(10): 1825-1848.

[2] MIAO Y T, CHEN C, PAN L, et al. Machine learning-based cyber attacks targeting on controlled information[J]. ACM Computing Surveys, 2022, 54(7): 1-36.

[3] 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.

[4] ZHANG J, PAN L, HAN Q L, et al. Deep learning based attack detection for cyber-physical system cybersecurity: a survey[J]. IEEE/CAA Journal of Automatica Sinica, 2022, 9(3): 377-391.

[5] QIU J Y, ZHANG J, LUO W, et al. A survey of android malware detection with deep neural models[J]. ACM Computing Surveys, 2021, 53(6): 1-36.

[6] WANG H T, YE G X, TANG Z Y, et al. Combining graph-based learning with automated data collection for code vulnerability detection[J]. IEEE Transactions on Information Forensics and Security, 2021, 16: 1943-1958.

[7] CHAKRABORTY S, KRISHNA R, DING Y, et al. Deep learning based vulnerability detection: are we there yet? [J]. IEEE Transactions on Software Engineering, 2022, 48(9): 3280-3296.

[8] YAMAGUCHI F, GOLDE N, ARP D, et al. Modeling and discovering vulnerabilities with code property graphs[C]//Proceedings of 2014 IEEE Symposium on Security and Privacy. Piscataway: IEEE Press, 2014: 590-604.

[9] RUSSELL R, KIM L, HAMILTON L, et al. Automated vulnerability detection in source code using deep representation learning[C]//Proceedings of 2018 17th IEEE International Conference on Machine Learning and Applications. Piscataway: IEEE Press, 2018: 757-762.

[10] LI Z, ZOU D Q, XU S H, et al. VulDeePecker: a deep learning-based system for vulnerability detection[J]. arXiv Preprint, arXiv: 1801.01681, 2018.

[11] 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.

[12] LI Z, ZOU D Q, XU S H, et al. VulDeeLocator: a deep learning-based fine-grained vulnerability detector[J]. IEEE Transactions on Dependable and Secure Computing, 2022, 19(4): 2821-2837.

[13] ZHOU Y, LIU S, SIOW J, et al. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in neural information processing sys-

tems, 2019, 32(1): 10197-10207.

[14] LI Y J, TARLOW D, BROCKSCHMIDT M, et al. Gated graph sequence neural networks[J]. arXiv Preprint, arXiv: 1511.05493, 2015.

[15] ALLAMANIS M, BROCKSCHMIDT M, KHADEMI M. Learning to represent programs with graphs[J]. arXiv Preprint, arXiv: 1711.00740, 2017.

[16] WU Z H, PAN S R, CHEN F W, et al. A comprehensive survey on graph neural networks[J]. IEEE Transactions on Neural Networks and Learning Systems, 2021, 32(1): 4-24.

[17] HERREMANS D, CHUAN C H. Modeling musical context with Word2Vec[J]. arXiv Preprint, arXiv: 1706.09088, 2017.

[18] HU Z N, DONG Y X, WANG K S, et al. Heterogeneous graph transformer[C]//Proceedings of The Web Conference 2020. New York: ACM Press, 2020: 2704-2710.

[作者简介]

**杨宏宇**（1969– ），男，吉林长春人，博士，中国民航大学教授，主要研究方向为网络信息安全。

**杨海云**（1997– ），男，陕西宝鸡人，中国民航大学硕士生，主要研究方向为网络信息安全。

**张良**（1987– ），男，天津人，博士，美国亚利桑那大学研究员，主要研究方向为强化学习和基于深度学习的信号处理。

**成翔**（1988– ），男，新疆乌鲁木齐人，博士，扬州大学实验师，主要研究方向为网络与系统安全、网络安全态势感知、APT 攻击检测。