华中科技大学

图神经网络实验报告

专　　业：　计算机科学与技术

班　　级：　CS2201

学　　号：　U202215348

姓　　名：　李烨丞

电　　话：　13097395920

邮　　箱：　2674152530@qq.com

独创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名： 

日期：2025 年 11 月 26 日

成　　绩	
教师签名	

目　录

1 概述

软件漏洞是网络安全链中最薄弱的一环。随着软件规模日益庞大和开源代码的普及，漏洞的数量和影响范围呈指数级增长。这些漏洞可能造成数据泄露、服务中断、经济损失，甚至威胁国家安全。因此，在软件上线前或攻击发生前，自动、准确地识别出代码中的潜在漏洞，具有极其重要的现实意义。

在深度学习兴起之前，主流的自动化漏洞检测方法主要包括静态分析、动态分析与符号执行等。静态分析在不运行代码的情况下，通过分析源代码或二进制代码的语法、结构、语义来发现漏洞，其覆盖面广，可以检查所有可能的执行路径，但是报告的很多“漏洞”在实际中不可利用，需要大量人工审计，并且依赖于预定义的、手写的漏洞规则库，难以发现新的、复杂的或特定领域的漏洞模式；动态分析通过实际运行程序并输入测试用例来监控程序的运行时行为，以发现异常，误报率低，缺点是代码难以覆盖所有可能的输入和执行路径，资源消耗大；符号执行通过将程序执行路径转化为数学约束表达式，并利用求解器来判断路径的可达性与输出，理论上非常强大，能深入探索复杂逻辑，但对于稍复杂的程序，路径数量会呈指数级增长，导致无法完成分析。这些方法自动化和泛化能力有限。

随着深度学习在自然语言处理领域的成功，研究人员开始尝试将代码视为一种“语言”，将深度学习应用于漏洞检测。如 VulDeePecker 将代码切片转化为符号序列，使用 BiLSTM 来学习序列中的漏洞模式[3]；还有一些研究使用 CNN 来捕捉代码中的局部模式[2]。这些方法的局限性在于将代码扁平化为序列，完全破坏了其固有的语法结构和逻辑结构，无法理解“变量 A 的值来源于变量 B”这类数据流信息，也无法理解“函数 C 只有在条件 D 为真时才会执行”这类控制流信息，而这些对于识别复杂漏洞至关重要。代码的本质是图，而不是序列。

为了克服序列模型的缺点，研究者开始利用代码的丰富结构信息。将抽象语法树 AST、控制流图 CFG 和数据流图 DFG 合并成一个统一的、具有多种边类型的异构图。CPG 能够同时表达代码的语法、控制流和数据流语义，为全面理解代码逻辑提供了强大的基础。

2 背景知识

GNN 是一类直接在图结构上操作的神经网络。它通过消息传递机制，让图中的节点从其邻居节点聚合信息，从而学习到包含局部图结构信息的节点嵌入。这使得 GNN 天然适合处理像 CPG 这样的图结构数据。

GNN 能够同时利用代码的文本内容和复杂的程序结构，自动从原始代码图中学习漏洞特征，减少了对人工特征工程的依赖，并能够捕捉到那些分散在代码不同部分、但通过数据或控制流关联起来的复杂漏洞模式，在漏洞检测中十分有优势。

相比于其他模型，GNN 在处理代码上的核心优势有以下几点：

（1）GNN 对节点的排列顺序不敏感。无论代码的文本行如何排列，只要其内在的 AST、CFG 结构不变，GNN 学习到的图表示就是相似的。

（2）GNN 显式地利用了代码中各种预定义的关系。它知道变量 A 和变量 B 之间存在“数据依赖”关系，而语句 C 和语句 D 之间存在“控制流”关系。这种关系意识是序列模型完全不具备的。

（3）许多复杂的代码漏洞的模式并非集中在连续的代码行中，而是散布在代码的不同部分，通过控制和数据流连接在一起。GNN 的消息传递机制能够沿着这些路径传播信息，从而捕捉到这种“长距离依赖”的漏洞模式。

（4）像 GraphSAGE 这样的 GNN 变体具备强大的归纳能力。这意味着它们可以在一个图上训练，然后泛化到从未见过的、全新的图上进行预测[4]。

3 问题定义

本任务的目标是构建一个二分类模型，该模型能够接收一个函数的源代码作为输入，并输出一个二进制标签，用以判断该函数是否包含漏洞。

输入：一个函数 F。

输出：一个预测标签 $\hat{y} \in \{0, 1\}$，其中 $\hat{y} = 1$ 表示 F 是易受攻击的，$\hat{y} = 1$ 表示 F 是安全的。

将此任务形式化为一个基于图的监督学习问题。数据集中每个样本是一个二元组 (c_i, y_i)。其中 c_i 表示第 i 个函数，y_i 是对应的真实标签。

每个函数 c_i 被转换为一个图 $G_i = (V, E, X)$ 作为模型的实际输入。其中 V 是图中节点的集合。每个节点代表代码中的一个语法元素；E 是图中边的集合，包含多种预定义的类型，用于编码代码语义；X 是节点的特征矩阵。每个节点 $v \in V$ 都有一个特征向量 x_v，该向量通常由其代表的代码文本和其节点类型共同编码而成。

我们的目标是学习一个映射函数 $f_\theta : G_i \mapsto \hat{y}_i$，该函数以一个图 G_i 作为输入，并预测其标签 $\hat{y}_i$。这里的 θ 代表模型的可学习参数。

通过优化一个损失函数（如二元交叉熵）来训练模型参数 θ，使得模型的预测 $\hat{y}_i$ 尽可能接近真实标签 y_i。

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_{BCE}(y_i, \hat{y}_i) + \lambda \cdot \Omega(\theta)$$

其中 N 是训练样本数量，$\mathcal{L}_{BCE}$ 是交叉熵损失，$\Omega(\theta)$ 是正则化项，λ 是超参数。

4 模型概览

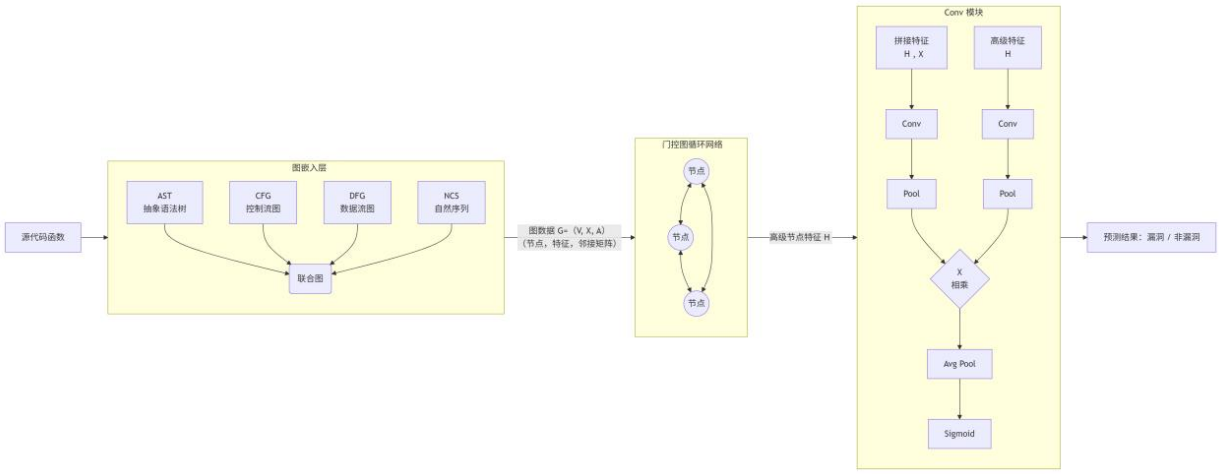

图 1 模型结构图

Devign 模型的核心思想是：将代码的多种语义表示融合成一个联合图，利用 GNN 学习图中节点的丰富特征，最后通过一个创新的卷积模块聚合全图信息进行分类[1]。整个框架可分为三个核心组成部分。

（1）图嵌入层

将原始的源代码函数 c_i 转换为一个结构化的图数据 $g_i(V, X, A)$，作为模型的输入；输出一个联合的、多边类型的代码属性图。

论文定义了四种类型的边，将不同的代码语义整合到一张图中，形成异构图。

AST 边：直接从 AST 中提取，表示语法上的父子关系。

CFG 边：从控制流图中提取，表示程序执行的可能路径。

DFG 边：从数据流图中提取，表示变量之间的依赖关系。

NCS 边：为了保留代码的原始序列信息，将 AST 中的叶子节点按照它们在源代码中出现的顺序连接起来。

构建节点时，使用一个在大型代码库上预训练的 Word2Vec 模型，将节点的“代码”文本映射为一个固定维度的向量，并将节点的“类型”通过标签编码转换为整数，然后也嵌入为一个向量。最后，将这两个向量拼接起来，形成节点的初始特征 。

（2）门控图循环层

通过图神经网络的消息传递机制，让每个节点聚合其邻居节点的信息，学习到包含丰富上下文的节点表示。

输入代码属性图 $g_i(V, X, A)$，输出包含高级语义的节点表示矩阵 $H^{(T)}$。

将节点的初始特征 x_j 作为其第一层隐藏状态 $h_j{}^{(1)}$ 的初始值。使模型进行 T 次迭代。在每一步 t，对于每种边类型 p，模型使用一个独立的权重矩阵 W_p 来计算沿该类型边传递的消息。对于节点 v_j，其从类型 p 的邻居收到的消息 $a_{j,p}{}^{(t-1)} = A_p{}^T(W_p[h_1{}^{(t-1)T}, \ldots, h_m{}^{(t-1)T}] + b)$ 是其所有该类型邻居节点上一时刻状态的加权和，再将一个节点从所有不同类型边收到的消息聚合起来 $a_j{}^{(t-1)} = \sum_{p=1}^{k} a_{j,p}{}^{t-1}$，最后使用门控循环单元来更新节点的状态 $h_j{}^{(t)} = GRU(h_j{}^{(t-1)}, a_j{}^{(t-1)})$。经过 T 步迭代后，得到每个节点的最终表示 $H^{(T)} = \{h_1{}^{(T)}, \ldots, h_m{}^{(T)}\}$。此时的节点表示已经融合了多维度代码语义信息。

（3）Conv 模块

将学习到的所有节点表示 $H^{(T)}$ 聚合起来，生成一个能够代表整个图的固定维度的向量，并最终进行分类。

维护两条路径。一条将最终的节点表示 $H^{(T)}$ 与初始节点特征 X 拼接起来，记为 $[\mathrm{H}^{(\mathrm{T})}, \mathrm{X}]$，这条路径包含了经过 GNN 提炼后的高级特征和原始特征；一条仅使用最终的节点表示 $\mathrm{H}^{(\mathrm{T})}$，这条路径专注于学习到的高级特征。对每一条路径，分别应用一系列一维卷积层和最大池化层。$\mathrm{Z}^{(1)} = \sigma([\mathrm{H}^{(\mathrm{T})}, \mathrm{X}]), \ldots, \mathrm{Z}^{(\mathrm{l})} = \sigma(\mathrm{Z}^{(\mathrm{l}-1)})$，$\mathrm{Y}^{(1)} = \sigma(\mathrm{H}^{(\mathrm{T})}), \ldots, \mathrm{Y}^{(\mathrm{l})} = \sigma(\mathrm{Y}^{(\mathrm{l}-1)})$，其中 $\sigma(\cdot)$ 表示一个“卷积+ReLU 激活+池化”的复合操作。这些卷积层的作用是从节点序列中提取与图分类任务相关的、更粗粒度的特征模式。将两条路径最终输出的特征 $\mathrm{Z}^{(\mathrm{l})}$ 和 $\mathrm{Y}^{(1)}$ 分别通过一个多层感知机，将两个 MLP 的输出进行逐元素相乘，对相乘后的结果进行全局平均池化，将其压缩为一个单一的标量，再通过一个 Sigmoid 函数，将该标量转换为一个介于 0 和 1 之间的概率，作为最终的预测输出 $\hat{y}_i$。

5 模型设计

在实验所给的代码中，论文中的图嵌入层与门控图循环层模块已被实现，关键的在于 Conv 模块。

在论文中提到，图分类的标准方法是全局聚合所有这些生成的节点嵌入，如使用线性加权求和来将所有嵌入进行扁平化加总，公式为 $\tilde{y}_i = \mathrm{Sigmoid}(\sum \mathrm{MLP}([H_i^{(T)}, x_i]))$。

为了对比论文中提到的 Conv 模块的训练效果，可以在代码中实现通过构建不同的模型，进行对照试验，来体现出 Conv 模块的性能。在具体实现中，我一共实现了四种 Readout 如表 1 所示，将在下一节具体介绍。

表 1 实现的 Readout 种类

类型	特点
ReadoutInvariantPool	标准方法，使用了置换不变的池化函数进行聚合。
ReadoutConv1D	标准方法，使用了不对称的池化函数进行聚合。
ReadoutPaperDirect	使用论文提出的框架。
ReadoutPaperWeighted	在论文提出的框架上修改了如何总结两条路径。

6 实验设计

由于之前没有做过神经网络相关项目，所以在实现 Conv 模块之前，我先尝试使用标准图读出操作来编写 Readout 模块。其中标准图读出操作通常先使用一个共享的神经网络对每个节点特征进行变换，然后将变换后的节点特征聚合为图表示，最后将池化后的图表示输入到一个分类器中进行分类。在变换节点特征和聚合节点特征时选择的方法需要满足结点之间置换不变的特性，所以我首先选择了平均池化和最大池化，最后再使用一个 MLP 进行分类。

在图嵌入层中，代码设定了一个图的节点数量为 max_nodes，会对数据进行填充与截断，一个节点的节点特征数为 feature_dim，这些特征被写入初始特征矩阵 x，所以 x 的形状为 [graph_num * max_nodes, feature_dim]，其中每一行代表一个节点的特征，每 max_nodes 行代表一个图的所有节点，一共有 graph_num 行。初始的 x 通过图卷积层后转换得到 h，然后再通过 h 与 x 预测标签值。这里的问题在于，一个图的节点数量可能不到 max_nodes 个，而一个节点的节点特征数也可能不到 feature_dim 个，所以在 x 中会有许多的填充 0。本人验证过初始特征 x，发现 x 中的所有节点的特征数都是 feature_dim，所以在这里只需要注意填充的全零行，也就是填充的不与任何节点相邻的节点。

这样的 x 经过图卷积层得到 h，由于填充的节点不与任何节点相邻，所以 h 是没有问题的。但是在接下来的读出操作中，我们把 h 与 x 连接后，如果之间使用连接后的矩阵经过 Readout，那么填充的节点也会被用于预测，而原来不存在于图中的节点被用来进行训练就会导致训练的结果有很大的问题。

为了解决这个问题，我们可以检测 x 的每一行，如果一行为全 0 值，那么这一行就是被填充的节点，使用掩码 mask 记录每一行，得到一个形状为 [graph_num, max_nodes] 的矩阵，代表一个图的一个节点是否为被填充的节点。

这样的代码的训练结果如图 2 所示，可以看到模型几乎没有学习到有用的特征，预测的结果全为 1。

```
Confusion matrix:
[[116   0]
 [114   0]]
TP: 0, FP: 0, TN: 116, FN: 114
Accuracy: 0.5043478260869565
Precision: 0.0
Recall: 0.0
F-measure: 0.0
Precision-Recall AUC: 0.5785294654937743
AUC: 0.605754688445251
MCC: 0.0
Error: 49.56521739130435
```

图 2 ReadoutInvariantPool 预测结果

于是我再次阅读了论文，论文中提出在数据处理部分根据 AST 的自然顺序来对节点进行固定排序，所以在论文提出的 Conv 模块中并没有使用置换不变的函数，那么在我的尝试代码中也可以使用不对称的函数。我重新使用了基于 1D 卷积的非置换不变 Readout 模型，该模型在保持节点级别 MLP 变换的基础上，使用 1D 卷积层来捕捉节点序列中的顺序模式，充分利用 AST 预定义排序带来的结构信息。模型首先通过共享 MLP 对每个节点的拼接特征进行变换，然后将处理后的节点特征序列通过多层 1D 卷积提取局部模式，最后通过全局最大池化和 MLP 分类器进行图级别分类。与上个版本的关键区别在于使用 1D 卷积替代了对称池化操作，这使得模型能够学习节点顺序中蕴含的代码结构信息。

这样的代码训练结果如图 3 所示，这次的训练结果不再是全部预测相同的结果，正确率也有一定的提升，说明使用不对称的函数确实是优化的一个方向。

```
Confusion matrix:
[[88 28]
 [76 38]]
TP: 38, FP: 28, TN: 88, FN: 76
Accuracy: 0.5478260869565217
Precision: 0.5757575757575758
Recall: 0.3333333333333333
F-measure: 0.4222222222222222
Precision-Recall AUC: 0.5372581337687763
AUC: 0.5855641258318208
MCC: 0.10163850370561245
Error: 45.21739130434783
```

图 3 ReadoutConv1D 预测结果

之后我使用论文提出的框架进行了代码编写，实现了基于双路径 1D 卷积的 Readout 模块，该模块遵循论文中 Conv 模块的设计，包含两个独立的卷积路径：路径 1 处理最终节点特征与初始节点特征的拼接，路径 2 仅处理最终节点特征。每个路径都经过多层 1D 卷积和池化操作，然后分别通过 MLP 分类器得到预测结果，最后将两个路径的输出进行逐元素相乘作为最终预测，但是训练的结果如图 4 所示，模型并没有学习到有用的特征，预测的结果全为 0。

我认为这里的问题在于直接相乘 output1 与 output2。当 output 中的值为负数时，经过 Sigmoid 函数后会变为小于 0.5 的值，更倾向于预测 0；当 output 中的值为正数时，经过 Sigmoid 函数后会变为大于 0.5 的值，更倾向预测 1。而相乘两个负值会得到一个正值，这样会导致模型归纳两个预测 0 的路径后得出预测 1 的结论，这样显然是不符合逻辑的。（也许是我对论文的理解有误）

```
Confusion matrix:
[[  0 116]
 [  0 114]]
TP: 114, FP: 116, TN: 0, FN: 0
Accuracy: 0.4956521739130435
Precision: 0.4956521739130435
Recall: 1.0
F-measure: 0.6627906976744187
Precision-Recall AUC: 0.5483929806251191
AUC: 0.5795145190562613
MCC: 0.0
Error: 50.43478260869565
```

图 4　ReadoutPaperDirect 预测结果

为了解决逐元素乘法导致的逻辑矛盾问题，我对模型进行了改进。现在我们假设 output1 与 output2 结果都是有分量的，但是分量不相等，使用公式 $output = Sigmoid(\rho \times output1 + (1-\rho) \times output2)$ 作为输出结果，其中 $0 \leq \rho \leq 1$ 代表权重。这里使用不同的 ρ 值进行测试，在 $\rho = 0.75$ 时的训练结果较佳。预测结果如图 5 所示。对于这个想法，接下来的优化方向可以引入自

适应权重调整策略，在每次训练时，我们可以尝试求出 output1 的 loss 与 output2 的 loss，通过两者 loss 的大小来动态调整两种的权重，实现这个机制。

```
Confusion matrix:
[[77 39]
 [61 53]]
TP: 53, FP: 39, TN: 77, FN: 61
Accuracy: 0.5652173913043478
Precision: 0.5760869565217391
Recall: 0.4649122807017544
F-measure: 0.5145631067961165
Precision-Recall AUC: 0.5590973594422932
AUC: 0.5732380520266183
MCC: 0.1313544162097107
Error: 43.47826086956522
```

图 5　ReadoutPaperWeighted 预测结果

综合来看，模型在实验中的准确率较低，只比随机预测的准确率高一点，完全没有达到论文中提出的模型的准确率，这可能是由多方面因素共同导致的。首先，在模型架构理解上可能存在偏差，虽然我尽力复现了论文中的双路径卷积设计，但在具体的网络参数、层间连接方式等方面可能与原文存在差异；其次，论文中提到的复合图结构包含了四种不同类型的边，而我可能未能完全复现这种复杂的多关系图结构；此外，在训练策略上，包括学习率调度、正则化方法、批次大小等超参数设置也可能与论文存在差异。这些因素都可能显著影响最终性能。

本人实验仓库：https://github.com/lmokeL/devign_lab。

7 总结与展望

通过这次实验，我接触了图神经网络的一个实际应用——代码漏洞检测。我之前对 GNN 的了解主要停留在理论层面，这次实践让我明白了它具体是如何工作的。我们把代码转换成图结构，图中的节点代表代码元素，边代表它们之间的关系；然后 GNN 模型通过分析这张图来学习识别可能存在安全漏洞的函数模式。

这个过程让我认识到，代码不仅仅是文本序列，其内部逻辑和关系用图来表示非常直观。虽然理解和实现整个模型有一定难度，但看到模型能够从复杂的代码关系中学习并做出判断，让我对 GNN 处理复杂结构化数据的能力有了更深的理解。

这次实验将课堂上学到的 GNN 知识和代码安全这个实际问题联系了起来，让我觉得所学的理论非常有用，也激发了我继续探索人工智能应用的兴趣。

8 课程感想

本人在之前学习 GNN 时，多停留在公式和概念层面，因为各种神经网络的公式看起来确实让人眼花缭乱，让一个没有系统学习过相关内容的人很难有勇气去实践。本次实验我亲手将 GNN 应用于一个极具挑战性的现实问题——代码安全，这让我真正体会到 GNN 在处理关系型数据上的强大能力，也是我第一次完成的基于神经网络的小项目。通过调整各种超参数与结构，看到消息传递机制在具体的代码图上运作，不断优化模型的性能，并最终解决实际问题，这种体验令人振奋。

同时这次实验也改变了我对代码的看待方式。代码不再仅仅是文本，而是一个充满丰富语义的图结构。这种视角的转变，对于理解程序的深层逻辑和潜在缺陷至关重要。

参考文献

[1] Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. San Diego, CA, USA: Network and Distributed System Security Symposium.

[2] Russell, R. L., Kim, L., Hamilton, L. H., Lazovich, T., Harer, J. A., Ozdemir, O., Ellingwood, P. M., & McConley, M. W. (2018). Automated Vulnerability Detection in Source Code Using Deep Representation Learning. Orlando, FL, USA: IEEE International Conference on Machine Learning and Applications.

[3] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., & Zhong, Y. (2018). VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. San Diego, CA, USA: Network and Distributed System Security Symposium.

[4] Hailton, W. L., Ying, R., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. Long Beach, CA, USA: Conference on Neural Information Processing Systems.