小 型 微 型 计 算 机 系 统
Journal of Chinese Computer Systems
DOI:10.20009/j.cnki.21-1106/TP.2025-0161
2026 年 4 月 第 4 期
Vol.47 No.4 2026

基于联合特征表示的跨语言代码相似性检测方法

丁思源[1,2],李 斌[1,2],王 浩[1,2],周清雷[1,2,3]

[1](郑州大学 计算机与人工智能学院,郑州 450001)
[2](嵩山实验室,郑州 450001)
[3](国家超级计算郑州中心,郑州 450001)
E-mail:18211882344@163.com

摘 要: 随着跨平台软件的日益普及,许多功能相同的程序需要在不同编程语言中实现,这促使相似性检测技术必须从单一语言扩展到跨语言领域,以便能够应用于漏洞检测等多个方面.然而,跨语言代码相似性检测的主要挑战在于如何提取不同编程语言的语义特征,不同语言较大的语法差异降低了代码之间的相似性.为了应对这个困难,本文提出了一种基于联合特征表示的跨语言代码相似性检测方法.首先,提取了不同语言的抽象语法树,并对不同语言的 AST 节点进行了统一,旨在缩小不同编程语言间的特征差异,以实现跨语言的代码相似性检测.此外,集成基于序列和基于图的神经网络,将不同语言代码的抽象语法树路径序列与扩展代码属性图(Extended Control Plane Gateway)的特征相结合形成联合特征,从而全面地理解不同语言代码的语义和结构信息.最后,使用来自两个开源代码编程竞赛网站的数据集,将本文方法与其他方法进行实验对比.结果显示本文提出的方法在多个评估指标上均优于其他工具和方法.此外,为了进一步验证本文方法在真实场景下的有效性和实用性,从合成数据集 SARD 中挑选了多种不同漏洞类型的跨语言漏洞数据集进行了实验.实验结果表明,本文方法表现出了较好的有效性.

关 键 词: 跨语言;代码相似性检测;深度学习;中间表示
中图分类号: TP311 **文献标识码:** A **文 章 编 号:** 1000-1220(2026)04-0819-10

Cross-language Code Similarity Detection Method Based on Joint Feature Representation

DING Siyuan[1,2],LI Bin[1,2],WANG Hao[1,2],ZHOU Qinglei[1,2,3]

[1](School of Computer and Artificial Intelligence,Zhengzhou University,Zhengzhou 450001,China)
[2](Songshan Laboratory,Zhengzhou 450001,China)
[3](National Supercomputing Center in Zhengzhou,Zhengzhou 450001,China)

Abstract: With the rise of cross-platform software,many similar programs need to be implemented in different programming languages,necessitating the extension of code similarity detection techniques from single-language to cross-language applications,especially for vulnerability detection. The main challenge lies in extracting semantic features from diverse programming languages,as significant syntactic differences reduce perceived similarity. This paper proposes a cross-language code similarity detection method based on joint feature representation. We extract and unify abstract syntax trees from various languages to minimize feature differences,facilitating effective detection. Additionally,we integrate sequence-based and graph-based neural networks,combining AST path sequences with features from the Extended Control Plane Gateway to form joint features,thus enhancing the understanding of semantic and structural information. Our experiments with datasets from two open-source programming competition websites show that our method outperforms existing tools across multiple evaluation metrics. To further validate its effectiveness,we also test various cross-language vulnerability datasets from the SARD synthetic dataset,demonstrating promising results.

Keywords: cross-language;code similarity detection;deep learning;intermediate representation

0 引 言

随着大型在线代码库和跨平台软件的兴起,许多功能相同的程序需要用不同的编程语言实现,以适应不同的平台.像 GitHub 这样的基于网络的开源托管服务的出现,显著改变了代码的访问方式,使得软件开发过程能够受益于访问这些类似的大型开源软件代码库[1].

先前的研究表明,在大型软件系统中,程序代码中有 20-30% 的部分是相似的.当这些重复的代码片段存在安全漏洞时,这个漏洞可能会迅速蔓延到所有包含该代码的系统和应用程序中,造成广泛的安全威胁[2].例如,在 2021 年 12 月,Apache Log4j 被发现存在一个编号为 CVE-2021-44228 的远程代码执行漏洞.该漏洞波及了苹果、亚马逊、谷歌、百度以及腾讯等众多公司,这是因为它们在软件开发过程中,直接或间接

收稿日期:2025-04-16 收修改稿日期:2025-05-20 基金项目:嵩山实验室项目(241110210200)资助;先进密码技术与系统安全四川省重点实验室开放课题项目(SKLACSS-202408)资助. 作者简介:丁思源,男,2001 年生,硕士研究生,研究方向为网络安全和代码相似性检测;李 斌(通信作者),男,1986 年生,博士,讲师,研究方向为信息安全和高性能计算;王 浩,男,1998 年生,硕士研究生,研究方向为网络安全和漏洞检测、模糊测试;周清雷,男,1962 年生,博士,教授,CCF 杰出会员,研究方向为拟态防御和信息安全.

地使用了含有此漏洞的开源 Log4j2 项目. 针对代码复用可能带来的问题,采用跨语言的代码相似性检测技术可以作为一个有效的解决方案. 然而,准确且高效地捕捉代码语义本身就是一项具有挑战性的任务. 此外,不同编程语言各自拥有独特的语法规则和编码特性,这使得准确提取跨语言代码的语义信息变得更加复杂和困难[3].

在代码语义信息的提取方面,传统的方法通过编译器将不同语言的源代码转换成中间语言形式[4],通过比较这些中间语言的相似度来识别相似代码. 然而,这类方法受限于编译器和中间语言的通用性,因此此类方法只能针对特定的编程语言进行相似性检测. 为了突破这一局限,Mike 等人将代码视作普通的文本数据,并通过提取代码的特征信息来进行代码相似性检测[5]. Li 等人和 Harer 等人将代码中的标记(tokens)作为基本单位,使用标记生成嵌入向量,并将其输入到神经网络中进行代码特征学习[6]. 然而,这种基于标记或文本的方法仅捕获了代码的词法层面信息,对于代码更深层次的结构和语义特征缺乏足够的捕捉能力,无法全面反映代码的结构和语义特性. 鉴于此,Azcona 等人[7]和 Mou 等人[8]提出了基于抽象语法树(Abstract Syntax Tree,AST)的代码学习方法,可以通过对抽象语法树的遍历或结构中学习特征,并具有一定的语义效果. Zhang 等人将代码转换成抽象语法树,随后利用深度学习技术从 AST 中提取出有价值的特征[9]. 但是由于不同编程语言之间存在显著的语法差异,它们的抽象语法树也相应地展现出较大的不同,这在一定程度上限制了仅依赖语法树进行跨语言相似性检测的准确性. 另外,还有一类基于函数语义的检测方法,函数语义不仅描述了函数所执行的具体任务或功能,还隐含了函数处理数据的方式和逻辑结构. 该类方法通过深入分析函数的语义特征,可以更准确地把握函数之间的内在联系和差异[10]. 但是,基于图的程序表示方法难以学习代码的全局和顺序语义信息.

为了解决跨语言代码相似性检测在语义提取方面的问题,本文提出了一种基于联合特征表示的跨语言代码相似性检测方法. 针对跨语言语义学习,使用一个统一词汇表对不同语言的 AST 节点进行统一,进而缩减不同语言之间的差异. 在语义提取方面,采用 Transformer 模型来处理扁平化的 AST 路径序列特征,以此捕获代码的全局逻辑结构特征. 同时,通过对 AST 增加 5 种额外类型的边来构造扩展代码属性图(Extended Code Property Graph,EX-CPG),并利用门控图神经网络(Gated Graph Neural Network,GGNN)来提取增强图结构中的结构化特征. 随后,集成两个网络生成的两种特征表示,形成联合特征,从而全面获取代码的结构和语义特征. 最后,使用融合后的联合特征进行相似性检测. 通过实验对比,验证了本文方法在跨语言代码相似性检测方面的有效性和稳定性. 综上所述,本文的主要贡献如下:

1)提出了一种基于联合特征表示的跨语言代码相似性检测方法. 通过集成两种特征表示来获取不同语言代码的语义信息,提高跨语言代码相似性检测的效果.

2)使用统一词汇将不同语言 AST 节点进行统一,降低不同编程语言间的差异. 并将代码中的元素(如关键字、标识符、操作符等)通过词汇映射转换为统一的数值向量.

3)在来自两个知名的开放源代码编程竞赛网站的合成数据集上进行了对比实验,实验结果表明,本文提出的方法在召回率、精确率、以及 F1 分数等多个关键性能指标上优于其他 4 个基准方法.

4)在合成数据集 SARD 中,选取了几种不同类型的真实漏洞,并将本文所提出的方法应用于这些漏洞的检测任务上. 实验结果显示,本文方法展现出了较好的应用效果.

1 相关工作

代码相似性检测的核心在于评估代码之间的相似程度,以确定是否存在文本级或语义级的相似性. 然而,高效且准确地捕获代码语义是一项艰巨的任务. 更进一步的挑战在于,不同编程语言各具特色的语法规则和编码特性,使得跨语言代码语义的精确提取尤为复杂. 本节先根据代码的表示方式的不同,分别介绍了目前主流的几类代码相似性检测的方法,而后介绍了跨语言的代码相似性检测方法.

1.1 代码相似性检测

针对代码相似性检测,可以根据代码的不同表示方式划分为:基于标记(token)的方法、基于抽象语法树的方法和基于语义的分析方法. 代码构建于一系列有意义的元素之上,这些元素是代码的基本语法单元,并且具有特定的语法结构和语义要求. 基于标记的方法是将源代码拆解为一系列连续的标记,这些标记涵盖了源代码中的元素(关键字、变量名、符号、数字等). 通过这种方式,可以更加深入理解代码的结构,更准确地捕捉代码的词法细节以及部分语法结构. Wang 等人提出的 CCAligner 代码相似度检测工具,采用滑动窗口机制,通过提取连续代码片段作为 token 比对单元,并将窗口内序列哈希化后匹配相似度[11]. 然而,一个显著的局限在于,这些标记主要聚焦于代码的词法层面,未能充分揭示代码深层的结构特征与语义内涵.

代码的抽象语法树是一种包含代码结构和语法信息的表示形式. 与 token 相比,AST 提供了更为丰富和深入的代码信息. 通过 AST 的结构,可以精确的获取每个节点的类型、节点之间的关系以及节点的位置信息. 在实际应用中,为了降低处理 AST 图形结构的复杂度,有时需要将其扁平化为路径序列. Azcona 等人[12]通过遍历抽象语法树的树形结构,深入学习了代码的结构特征,在此过程中展现出了 AST 相当程度的语义分析能力.

函数语义体现了函数所执行的具体功能及其特性,而数据流和控制流则详细描绘了数据在程序中的动态路径,包括其移动、处理及存储的各个环节,同时也反映了代码的控制结构. 为了对函数进行相似性分析,一种常用的方法是基于语义将代码表示为程序依赖图(Program Dependence Graph). 此外,生成函数签名也是一种有效的函数相似性分析手段. Ben 等人[13]利用中间表示构建了名为 XFG 的图结构,随后采用神经网络深入学习并解析了 XFG 图的语义特征. 但获取 IR 需要代码编译,这使得无法处理一些不完整的代码片段.

1.2 跨语言的代码相似性检测

对于跨语言相似性检测,需要采用特定的方法消除不同编程语言之间的差异. 一类方法是基于代码信息的检测方法,这些方法通过提取代码多种类型的信息特征(如 token、抽象

语法树等),使得不同语言的代码统一表示,进而进行相似性比较.并且为了进一步消除不同编程语言间中间表示的差异,会采用特定的策略,例如提取语言无关的特征、使用统一的词汇表等.此外,检测过程还会提取其他特征,如 API 使用情况和输入输出信息等,以全面表示代码的语义信息. COSAL[14] 结合了静态分析与动态分析,通过计算 token 序列的余弦相似度、AST 的树编辑距离以及执行记录的输入输出信息余弦相似度,实现了对代码相似性的全面评估.而 BIGPT 则通过在大规模代码数据集上训练自编码器模型,学习程序特征的潜在表示,进而实现了跨语言代码相似性的快速检索[15].张锋等人提出一种适用于跨语言代码相似性检测的代码表示方法,该方法通过图注意力网络提取程序流程图的特征作为代码的表示,并且对代码进行逐行的交叉匹配,提高相似性检测的准确率[16].

还有一类方法是基于代码向量化的检测方法.这类方法是使用 CodeBERT[17]、InferCode[18] 等预训练模型,将多种编程语言代码转换至统一代码空间进行向量表示,进而捕捉丰富的代码信息.通过对这些预训练模型进行微调,并针对性地设计下游任务,这类方法能够有效地进行跨语言代码相似性比较. C4 使用 CodeBERT 作为预训练模型,将代码片段编码成向量表示.并且通过对比学习技术对这些向量表示进行微调,以区分相似和非相似代码对,从而最小化相似代码对之间的向量距离[19]. InferCode 将 AST 视为一个文档,利用 TBCNN[20] 对 AST 进行遍历,将 AST 的子树结构转换为向量表示.这些向量表示包含了代码的语法信息,使得 InferCode 能够有效地比较不同编程语言中的代码片段.

2　基于联合特征表示的跨语言代码相似性检测方法

2.1　总体结构

如图 1 所示是本文所提出方法的总体架构.本文方法主要由两个部分组成:1)接收来自多种编程语言的源代码作为输入,通过将代码转化为抽象语法树序列和图来实现代码语义的表示;2)运用深度学习技术来提取待代码对的特征,并利用这些特征以判断代码对之间的相似性.

首先,从源代码中移除所有注释和字符串字面量,这一步旨在精简信息,确保分析过程不受无关内容的干扰.接下来,利用 Tree-sitter、Joern 等工具将处理后的源代码转化为 AST 和 EX-CPG.为了支持不同语言代码语义的学习,采用统一词汇表,在抽象语法树和代码属性图上分别进行序列嵌入和图嵌入,从而实现不同编程语言中间表示的统一.之后,将序列嵌入向量和图嵌入向量分别输入到两个不同的子网络中,这两个网络分别用于捕获代码的序列特征和语义特征,并通过一个全连接层将这两个网络产生的特征向量融合为一个统一的联合特征向量.最终,将待检测的两个代码的联合特征向量作为相似性检测任务的输入,通过预测其相似的概率并与设定的阈值进行比较,从而得出检测结果.

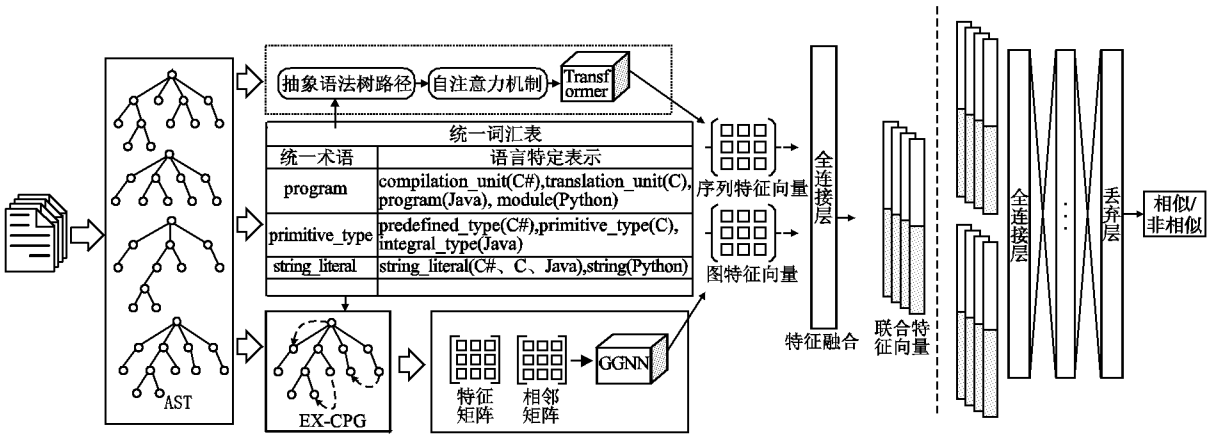

图 1　基于联合特征表示的跨语言代码相似性检测方法

Fig. 1　Cross-language code similarity detection method based on joint feature representation

2.2　统一抽象语法树

由于不同编程语言所遵循的编码规范和特性并不相同,它们在文本表示上会展现出一定的差异性.本文选择使用 tree-sitter 来对不同语言的源代码进行语法分析,使用此工具进行语法分析的主原因是:它足够通用,能支持多达 50 余种编程语言,并且它提供了便捷且易于使用的 Python 应用程序接口.而且它足够稳定,即便语法错误,也能够解析程序的语法.

不同的编程语言有不同的语言特性和编码规则,所以不同语言提取出的抽象语法树会有一定的差异.为了体现不同语言的差异,使用如图 2 所示的函数作为例子,此函数分别使用 C#、C + +、Python 这 3 种编程语言实现,图 3 显示了如上

```
public int function(int x,int y) {
    x=start();
    y=1;
    if(a>b)x=x+y;
    else x=x*y
}
C#
public int function(int x,int y){
    x=start();
    y=1;
    if(a>b)x=x+y;
    else x=x*y
}
C++
def function(x,y):
    x=start()
    y=1
    if x>y:x=x+y
    else:x=x*y
Python
```

图 2　3 种编程语言实现的函数示例

Fig. 2　Function implementation example in three programming languages

3 种不同编程语言程序所生成的抽象语法树结构.如图 3 所示,不同语言的 AST 节点名称存在很多相同和不同之处.不

同之处在于他们中的很多节点名称不同,而相同之处在于很多即使名称不同的节点名称也具有相同的语义. 比如在 C#和 C + + 的 AST 中的节点 parameter_list、Python 中的 parameters,虽然节点名称不同,但实际上都是对函数的参数列表的声明.

为了有效降低因不同编程语言在抽象语法树节点命名上存在的差异而带来的影响,本文对跨语言的 AST 节点名称进行标准化统一,将不同语言 AST 中虽名称各异但实质意义相同的节点类型赋予统一且具普遍意义的名称. 例如,无论在哪种编程语言中,作为整个程序结构起点的根节点,其名称均被统一替换为 program,这一命名直观地反映了该节点作为程序整体代表的角色. 同样地,对于那些在不同语言中用于表示一段代码逻辑单元的代码块节点,统一采用 block 这一简洁而明确的名称.

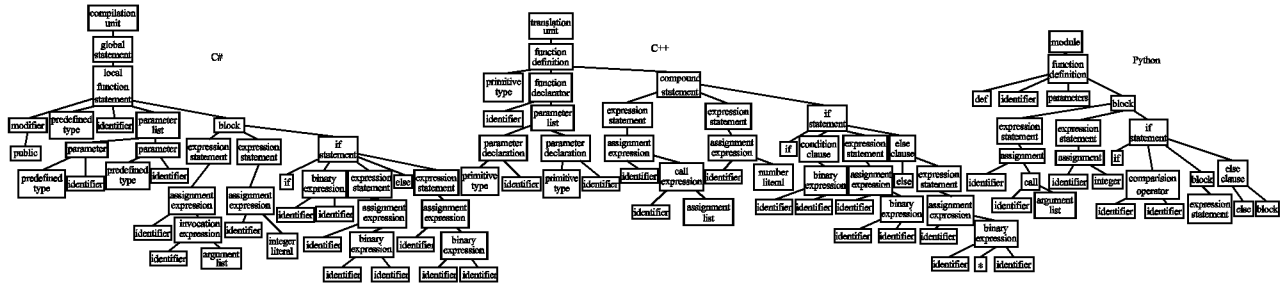

图 3 不同编程语言 AST 节点对比图

Fig. 3 Comparison of AST nodes in different programming languages

统一词汇将不同语言 AST 中相同类型的节点进行统一命名,从而减小了不同语言直接的差异. 具体步骤如图 4 所示,首先将都代表根节点的 3 个节点名称通过统一字典转化为统一名称,而后使用独热编码来将节点索引转换为数值特征. 通过这一机制生成的嵌入向量,能够更加有效地捕捉和学习各种编程语言的特性,从而解决不同语言直接语法差异的问题.

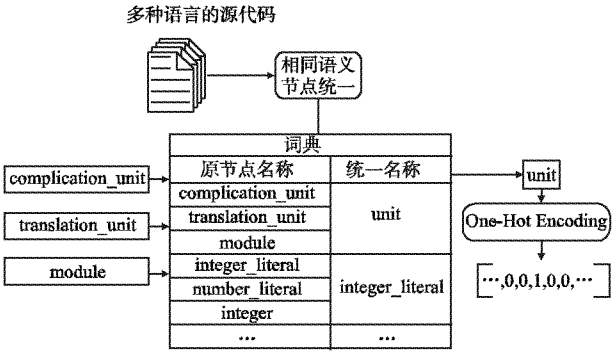

图 4 统一 AST 节点命名及特征转换图

Fig. 4 Unified naming and feature transformation of ASTNodes graph

2.3 扩展代码属性图

标准的抽象语法树原本仅包含用于表示节点间父子关系的子边. 为了更全面地捕获代码的结构信息,本文在原始 AST 之上增添了五种新的边类型,实现了对原始 AST 的扩展. 本文将这种扩展后的 AST 视为一种扩展的代码属性图,并依据 2.2 节所述的方法,为其生成统一的图嵌入向量. 将生成的 EX-CPG 输入到下游的神经网络中,以进一步捕获代码的特征. 附加边的具体描述如下:

a) 数据流和控制流:将从程序控制数据图中提取的数据和控制路径集成到 AST.

b) GuardedBy:采用 GuardedBy 边来连接变量的每个 AST 标记与其对应的封闭保护表达式.

c) Jump:利用 Jump 边来建立变量与控制依赖关系之间的连接.

d) NextToken:用于将每个语法标记 token 与其后续标记相连. 这一设计旨在捕捉语句中操作码与操作数的顺序信息.

举例来说,图 5 展示的是依据图 2 中的源代码所构建的扩展代码属性图. 图中的节点代表程序的构成,而边则表示节点间对应的关系. 控制流表示程序中不同节点之间的控制关系,数据流表示了变量 x 和 y 在不同节点之间的流动. 对于图中的 if 语句,将对 x 赋值的 GuardedBy 边添加到 $x > y$ 对应的 AST 节点. 使用 Jump 边连接 if 和 else 节点,记录了不同控制流的关系. 使用 NextToken 边将每个语法标记节点连接到其后继节点.

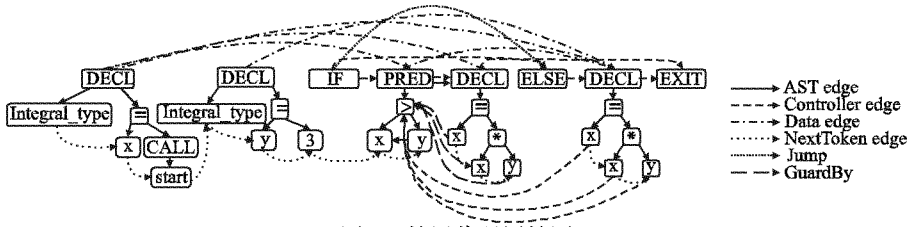

图 5 扩展代码属性图

Fig. 5 Augmented code property graph

2.4 特征融合

为了解决跨语言代码相似性检测在语义提取方面的问题,本文采用 2.3 节的方法生成扩展代码属性图,并使用统一词汇表对抽象语法树的路径序列和扩展代码属性图进行序列

嵌入和图嵌入,将相似的语法结构和语义关系标准化. 接下来,分别将序列嵌入向量和图嵌入向量分别输入到 Transformer 和门控图神经网络中进行训练,以捕获代码的特征表示. 随后,通过一个全连接层将这两个网络输出的特征向量融合为一个统一的联合特征向量. 最后,将待检测的两段代码的联合特征向量作为相似性检测任务的输入,预测出相似的概率并与预设阈值进行比较,从而获得最终的相似性检测结果.

2.4.1　序列嵌入

为了学习代码的语义及句法的特征,本文使用 Transformer 对代码的 AST 序列特征进行提取. 并且为了能够捕捉序列中各个元素之间的依赖关系,本方法使用了自注意力结构. 具体而言,首先对抽象语法树进行先序遍历,从中生成路径序列,并通过统一词汇表将每个节点映射到嵌入向量. 随后,将这些嵌入向量输入至仅含编码器的 Transformer 模型,每个编码器单元包含双向多头自注意力层及前馈神经网络层.

Transformer 是一种基于注意力机制的深度学习模型,主要由编码器和解码器组成. 其核心的自注意力机制允许模型在处理输入序列时关注不同位置的信息,从而有效减轻长距离依赖问题. 自注意力由查询(Query)、键(Key)和值(Value)3 部分构成,其计算公式为:

$$Attention(Q,K,V)=SoftMax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

其中,$Q\in\mathbb{R}^{l\times d}$、$K\in\mathbb{R}^{l\times d}$和 $V\in\mathbb{R}^{l\times d}$为嵌入路径序列向量,$l$ 代表路径序列的嵌入维度大小,d 表示输入路径的实际长度. 为了获取输入序列中更为丰富的语义内容,本文采用了多头机制来实现自注意力功能. 多头机制将查询、键和值分割为 h 个头,每个头的自注意力计算如公式所示:

$$head_i=Attention(Q_i,K_i,V_i)=SoftMax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2}$$

随后,将所有头的注意力向量进行拼接,从而得到多头自注意力层的最终计算结果:

$$MultiHead(Q,K,V)=Concat(head_1,head_2,\cdots,head_h)W^O \tag{3}$$

为了更有效地捕捉输入序列中的特征,在每个编码器单元加入一个前馈神经网络层以进一步处理多头注意力的输出. 前馈神经网络由两个线性变换和一个非线性激活函数组成. 其计算过程如公式所示:

$$FFH(X)=ReLU(XW_1+b_1)W_2+b_2 \tag{4}$$

通过上述自注意力机制和前馈神经网络,捕获了代码嵌入向量的内部关系. 最终的节点表示 $T=[t_1^{tr},t_2^{tr},\cdots,t_{|d|}^{tr}]$ 包含了输入序列中所有节点的信息,即为生成的序列特征向量.

2.4.2　图嵌入

虽然基于序列的 AST 网络能够从路径序列中学习代码的全局结构以及句法特征,但由于路径序列本质上是一种扁平化的表达方式,因此它不可避免地会忽略掉代码原本所具有的一些语义结构信息. 在相似的代码中,虽然不同语言的代码结构不相同,但代码逻辑通常相同. 如图 4 中,不同语言 AST 结构的 if_statement 部分之间的结构具有相似之处. 因此,本文使用门控图神经网络提取代码的局部特征,将 2.3 节所述的 EX-CPG 作为输入,以捕捉代码的语义特征. 针对 EX-CPG 中新增的五种边类型,采用合并邻接矩阵的方法进行处理.

GGNN 是一种专为图结构数据设计的神经网络模型,能够高效捕捉节点间的依赖关系和层次结构. GGNN 通过在 EX-CPG 中聚合和学习每个节点的邻居节点的特征,递归地传递和聚合节点信息,从而可以捕捉到代码的局部语义信息. 具体而言,GGNN 首先将每个节点初始化为特征向量,并利用一个迭代的信息传播机制不断更新节点状态. 具体流程如下:

节点初始化:首先,对 EX-CPG 中的每个节点进行初始化,使用之前生成的统一图嵌入表示作为节点的初始特征向量,如公式(5)所示:

$$h_i^{(0)}=v_i \tag{5}$$

领域聚合:在每个时间步中,节点将其当前嵌入向量作为消息发送给所有相邻节点. 每个节点聚合其邻居节点发送的消息,以更新自身的表示. 即公式(6):

$$m_i^{(t)}=\sum_{j\in N(i)}Message(h_i^{(t-1)},h_j^{(t-1)}) \tag{6}$$

状态更新:随后,使用门控机制更新节点状态. 即公式(7):

$$h_i^{(t)}=Update(h_i^{(t-1)},m_i^{(t)}) \tag{7}$$

其中,*Update* 函数结合节点 i 在时间步 $t-1$ 的状态 $h_i^{(t-1)}$ 和聚合的消息 $m_i^{(t)}$ 来生成新的节点,状态 $h_i^{(t)}$,$h_i^{(t)}$ 是节点 i 在时间步 t 的新状态. 对图中的所有节点进行 T 轮迭代,经过 T 轮迭代后,每个节点的状态 $h_i^{(t)}$ 就是该节点的最终表示. 最终输出表示为:$G=[g_1^{tr},g_2^{tr},\cdots,g_{|d|}^{tr}]$,其中 d 表示图中节点的数量.

2.4.3　联合特征向量

Transformer 对 AST 序列的特征进行抽取,从而得到代码的全局结构和逻辑特征,凭借强大的自注意力机制,能够捕捉到代码中长距离依赖关系和整体逻辑流. 同时,通过使用 GGNN 提取扩展属性图的特征来捕获代码的局部结构和语义特征,通过图节点间的消息传递和状态更新,能够深入获取代码图结构中的细节信息和节点间的依赖关系. 在得到了如上两个模型所生成的特征向量之后,需要集成这两个不同的特征以进行下游的相似性检测任务. 为此,本文提出了一种集成方案,旨在将序列化的全局信息与结构化的局部信息进行综合整合. 具体流程如下:

特征表示:通过上游工作,得到代码的序列特征和图特征,如公式(8)和公式(9)所示:

$$T=[t_1^{tr},t_2^{tr},\cdots,t_{|d|}^{tr}] \tag{8}$$

$$G=[g_1^{tr},g_2^{tr},\cdots,g_{|d|}^{tr}] \tag{9}$$

其中,$T\in\mathbb{R}^{d\times m}$表示在 AST 序列中学习到的特征,$G\in\mathbb{R}^{d\times m}$表示从 EX-CPG 图中所获得的特征,$d$ 是 AST 中节点的数量,m 表示每个节点特征向量的维度.

特征配对:将 AST 中每个节点 i 对应的序列特征和图特征进行配对,式中的 P_i 表示第 i 对特征向量的配对:

$$P_i=(t_i^{tr},g_i^{tr}) \tag{10}$$

特征向量连接:在对每个节点的特征向量进行两两配对之后,将配对的特征向量进行连接,以生成新的特征向量,公式如下:

$$h_{combined}^i=Concat\,(t_i^{tr},g_i^{tr}) \tag{11}$$

其中，$h^i_{combined}$为新生成的节点 i 的特征向量，*Concat* 表示将两个向量首尾相接.

最后，将所有节点的特征向量组合在一起：

$$H_{combined}=[h^1_{combined},h^2_{combined},\cdots,h^d_{combined}] \tag{12}$$

通过上述过程，得到的$H_{combined}\in\mathbb{R}^{2\times d\times m}$即为代码的联合特征向量，其综合了代码的全局结构特征和局部结构及语义特征.

2.5 相似性判断

通过融合上述两种模型生成的不同类型特征向量，可以得到待比较的两段代码的联合特征向量 X_1 和 X_2，该向量能够全面体现代码的全局与局部特征. 利用这些特征向量进行相似性比较，以判断这两段代码之间的相似程度.

在相似性比较过程中，首先将 X_1 和 X_2 进行拼接，得到一个新的特征向量 X：

$$X=[X_1,X_2] \tag{13}$$

接下来，将拼接后的特征向量 X 被输入到全连接神经网络中，该网络结构由输入层、多个隐藏层和输出层组成. 隐藏层中的多个神经元通过激活函数进行非线性变换，以增强模型的表达能力. 此外，为了防止过拟合，在隐藏层中加入了丢弃层. 经过 *Sigmoid* 激活函数处理后，输出值 P 表示两段代码相似的概率，计算公式如下：

$$P=Sigmoid(W\times X+b) \tag{14}$$

其中，X 是拼接后的特征向量，W 是权重矩阵，b 是偏置项. 经过 *Sigmoid* 函数的规范化，输出的相似概率 P 的值域为[0, 1]. 选择一个适当的阈值 T，用于判断相似性：如果 $P\geqslant T$，则认为两段代码是相似的，反之若 $P<T$，则认为它们非相似.

在训练过程中，使用交叉熵作为损失函数，并采用 Adam 优化器来最小化该损失. 损失的计算公式如下：

$$Loss=-(y\times\log(p)+(1-y)\times log(1-p)) \tag{15}$$

其中p是计算出的相似概率，y是样本标签，若两段代码相似则 y 为 1，反之 y 为 0. 如图6是将融合后的联合特征向量设置为 100 时，对应的全连接神经网络结构. 在对两段代码的特征向量进行拼接后输入到神经网络中，并经过 3 个隐藏层的降维计算后，最后在输出层使用激活函数 *Sigmoid* 计算相似概率，设置阈值为 0.5，进行相似性预测.

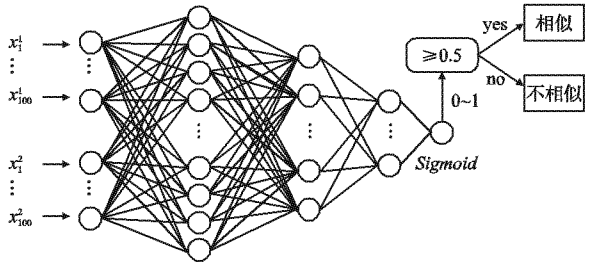

图 6 全连接神经网络结构及相似性预测

Fig. 6 Fully connected neural network structure for similarity prediction

3 实验设计

本节旨在通过设计实验，验证本文所提出的跨语言代码相似性检测方法的有效性. 本节所做的对比实验，均是基于从两个著名的开源编程竞赛网站：AtCoder 和 CodeJamData 收集而来的数据集. 首先，为了验证本方法相比其他的有效性，实验 1 将本方法与多个基准方法在跨语言数据集上进行实验对比. 第二，为了验证本方法在特定语言对下的有效性，实验 2 将本文提出的方法与其他几种方法在不同语言对进行了比较实验，从而更全面地评估了该方法在不同语言组合下的表现. 此外，在实验 3 中，通过评估本文方法与其他方法在处理不同规模代码时的检测效果，进一步说明了本方法在应对不同规模代码时的性能和稳定性. 第三，利用消融实验验证了特征融合和统一词汇的有效性.

最后，为了验证本文方法在真实场景下的效果，在合成数据集 SARD 中选取了多种跨语言漏洞，将本文的方法应用于跨语言漏洞的检测任务中.

3.1 数据集

由于缺乏公开可用的跨语言代码相似性检测的数据集，为了实验和评估本文方法，本实验采用来自两个知名的开放源代码编程竞赛网站：AtCoder 和 CodeJamData 的源代码数据. 为了构建实验所需的数据集，本文对从这两个网站收集到的原始数据进行了预处理工作，包括去除注释、进行必要的标注等，最终整合成了一个综合数据集. 该数据集共囊括了 55367 个源文件，这些文件被归入了 1115 个相似的类别中，平均每个类别包含约 50 个相似的不同语言代码. 在划分数据集时，将这些类别以 8∶1∶1 的比例拆分为训练集、验证集和测试集. 通过在同一函数类别中随机选取程序对以生成相似的样本对，每个源文件都随机匹配一个对应的相似文件，因此该数据集具有 55367 个相似的样本对. 同样地，在构造非相似的样本对时，为每个源文件都从不同类别的程序中随机选择源文件来进行配对，因此该数据集拥有 55367 个非相似样本对. 具体划分如表 1 所示，其中每一列表示以此语言进行配对的数量.

表 1 相似与非相似样本对的数量分布

Table 1 Distribution of similar and dissimilar sample pairs

	C++	C#	Java	Python
相似样本	11888	13268	15479	14732
其他样本	11888	13268	15479	14732

3.2 实验环境

本文方法的所有模型均在 PyTorch 上实现，CUDA 版本为 12.1. 实验均在一台配备了 32 核 2.10GHz Intel(R) Xeon(R) Silver 4216 CPU 和 Tesla V100-PCIE-32GB GPU 的服务器上进行. 对所有方法均使用相同的训练集进行训练，以确保对比实验的公平性. 本文方法中的 GGNN 模型被设计为包含 8 个时间步长，图的邻接矩阵的大小统一为[400,400]. Transformer 模型架构包含了 4 个编码层，每层均配备 8 个注意力头，每个编码层的前馈层维度为 512. 在编码器的实现中，dropout 率被设置为 0.1，路径序列的长度统一为 700. 模型的迭代轮次设为 20，批量大小设为 32，并采用 Adam[21] 作为训练优化器.

3.3 实验指标

为了验证本文提出方法的有效性，本文使用精确率(*Precision*)、召回率(*Recall*)和 *F*1 分数(*F*1-*score*)作为评估指标. 其中，精确率是指在所有被模型预测为相似的代码对中，实际

为相似的代码对所占的比例. 召回率是指在所有实际为相似的代码对中,模型成功识别出的比例. 而 F1 分数是一个综合考虑精确率和召回率的性能指标,用于评估模型的整体表现. 精确率、召回率和 F1 值的计算方式如公式(16)~公式(18)所示:

$$Precision = \frac{TP}{TP + FP} \tag{16}$$

$$Recall = \frac{TP}{TP + FN} \tag{17}$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{18}$$

其中,TP 表示正确检测到的相似样本数,FP 表示错误将非相似的样本检测为相似的数量,FN 表示错误将相似的样本检测为非相似的数量.

4　实验结果

4.1　跨语言代码相似性检测对比实验

为了验证本文方法对跨语言代码相似性检测的有效性,将本文方法与其他几种基准方法进行了实验对比,分别为基于静态分析和动态分析的结合的 COSAL[22],基于预训练模型的 C4,基于抽象语法树和 API 文档度量的 CLCDSA[23],基于不同代码版本的 CLCMiner[24]、基于交叉匹配的张等[16]的方案和基于扩展抽象语法树的 CT3AST[25],实验结果如图 7 所示.

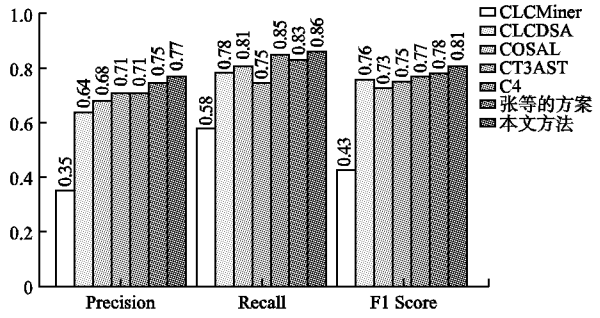

图 7　跨语言代码相似性检测方法的效果比较

Fig. 7　Comparison cross-language code similarity detection methods

实验结果表明,CLCMiner 的表现最差,这是因为 CLCMiner 通过分析代码的修订历史检测跨语言相似代码,仅关注了单一标记信息,忽略了其他相关因素,且因不同语言代码在词法和语法上的差异,仅凭相似标记难以识别. CLCDSA 通过从一组预定义的特征中选择对跨语言代码相似性检测最有效的特征来进行跨语言代码相似性检测. 但因为 CLCDSA 依赖于对特定的语法特征进行相似性检测,这会导致无法捕捉到所有相关的相似性,从而影响检测的全面性和准确性,且未处理代码冗余,干扰属性计数的结果. COSAL 虽然通过动静态结合的方法,取得了较好的效果,但动态分析会增加计算开销,并且相比于 CLCDSA 方法,COSAL 的提升效果并不大. C4 利用预训练模型 CodeBERT 将不同语言的代码转换为高维向量表示,采用对比式学习使得功能相同的代码对在表示空间中更接近,而功能不同的代码对则更远. 但 C4 只能对小于 512token 的代码进行处理,代码量较大的程序可能无法直接应用,且 C4 未充分考虑语法差异和跨语言特征多样性,导致效果逊于本方法. 张等的方案将代码转换为程序流程图作为中间表示,可以有效屏蔽不同编程语言之间的语法差异,提升跨语言检测的通用性. 该方法通过图注意力网络,结合动态学习节点间权重的信息,捕捉循环、分支等关键结构语义的重要性,优于传统 AST 方法,但交叉匹配方法需逐行计算所有节点间的相似度,时间复杂度是 $O(N^2)$,模型训练需 100 轮迭代,实际应用效率会受限. CT3AST 在传统 AST 基础上加入了 8 种逻辑关系边,通过增强 AST 的语义表示能力,结合 GNN 的特征学习,显著提升了检测精度和鲁棒性,但仅结合 AST 的边及其增强边的信息,仅能得到代码的局部特征,因此具有一定局限性. 本文所提出的方法利用代码的抽象语法树和扩展代码属性图作为中间表示,并融合了它们的特征,还利用统一词汇表将不同语言的 AST 中相同类型的节点进行统一命名,极大的减少了不同语言之间的差异,使得本文提出的方法在精确度、召回率和 F1 分数方面均优于上面 5 种基线方法.

4.2　特定语言组合对性能影响的对比实验

本实验旨在评估本文提出的方法与其他几个基准方法在不同编程语言组合下的性能表现. 为此,通过对包含 55367 个源文件的数据集中的文件进行随机匹配,以形成不同语言之间的样本. 根据代码类别的不同对数据集进行了相应的标注. 根据语言组合的不同划分为 C++与 C#、C++与 Java、C++与 Python、C#与 Java、C#与 Python 以及 Java 与 Python 6 种不同组合. 对于每种组合,都确保了样本数量的充足,并且每队样本的正负样本比例都是 5:5. 在实验中,使用精确度、召回率和 F1 分数这 3 个关键指标来评估不同方法在不同语言组合下的准确性、效率和鲁棒性. 实验结果如表 2 所示,本文所提出的方法在所有评估指标上均表现出色,不仅在精度、召回率和 F1 分数方面均高于其他 3 个对比方法,而且还展现出良好的稳定性和一致性.

表 2　不同语言组合的结果

Table 2　Results on different language combinations

语言组合	CLCMiner			CT3AST			C4			本文方法		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
C++&C#	0.56	0.37	0.45	0.79	0.87	0.83	0.77	0.85	0.81	0.82	0.84	0.83
C++&Java	0.57	0.32	0.41	0.69	0.82	0.75	0.71	0.82	0.76	0.82	0.88	0.85
C++&Python	0.54	0.37	0.44	0.76	0.84	0.8	0.77	0.86	0.81	0.84	0.85	0.84
C#&Java	0.53	0.34	0.41	0.8	0.81	0.8	0.78	0.85	0.81	0.83	0.87	0.85
C#&Python	0.55	0.36	0.44	0.75	0.83	0.79	0.71	0.81	0.76	0.85	0.89	0.87
Java&Python	0.59	0.36	0.45	0.81	0.82	0.81	0.79	0.83	0.81	0.86	0.83	0.84

具体而言,在精度方面,本文方法的平均精度为 0.83,远高于 CLCMiner 的 0.55、CT3AST 的 0.75 和 C4 的 0.77. 这表

明本文方法在识别不同组合的相似代码时,都具有较好的准确性. 在召回率方面,本文方法的平均召回率为 0.86, 同样高于其他方法,意味着本文方法能够识别出更多的真正相似代码对. 在 F1 分数方面,本文方法的平均 F1 分数为 0.84,也明显优于 CLCMiner 的 0.44、CT3AST 的 0.79 和 C4 的 0.79. 本文方法中不同语言组合的精度、召回率和 F1-Score 的标准差分别为 0.0061、0.025 和 0.0121,这些较小的标准差表明本文方法在不同语言组合下所得的结果偏差较小,即方法在不同语言组合下的性能表现相对一致,没用显著的波动或偏差. 这一结果表明,本文方法具有较高的稳定性和适应性,能够在不同的编程语言组合中保持稳定的性能表现.

4.3 不同规模代码行数对方法性能影响的对比实验

代码规模的增加会导致程序变得复杂,会对相似性检测的效果造成影响. 为了评估本文提出的方法与其他几个基准方法在不同代码规模下的性能表现,将数据集中的代码根据文件的行数规模划分为 5 个组,每组包含数据集中行数的 20% 范围. 具体而言,每个组包含的行数分别为:1 ~ 10、10 ~ 20、20 ~ 40、40 ~ 90、90 及以上. 表 3 展示了本方法与另外 4 个方法在不同行数下 F1 值的变化.

表 3 不同代码行数下的 F1 值对比

Table 3 Comparison of F1 values under different code lines

代码行数	CLCDSA	COSAL	C4	张等的方法	本文方法
1 ~ 10	0.54	0.56	0.57	0.51	0.58
10 ~ 20	0.61	0.65	0.63	0.63	0.68
20 ~ 40	0.72	0.79	0.79	0.8	0.83
40 ~ 90	0.72	0.67	0.77	0.78	0.81
>90	0.69	0.65	0.75	0.71	0.75

实验结果显示,随着代码行数的增加,所有方法的效果都有所提升. 这一现象的原因在于,较长的代码文件往往蕴含更丰富的上下文信息和更复杂的逻辑结构,使得模型能够学习到更为精细的模式和结构,从而提高了对相似代码的辨识能力. 然而,当代码行数超过 40 行之后,代码行数增加会导致 AST 节点数量的增加,节点类型的多样化,使得模型在检测过程中丢失了重要的相似特征,从而降低了检测的效果. 综上所述,本文方法在不同代码规模下展现出了良好的稳定性,特别是在处理 20 ~ 40 行代码时表现最优,这证明了本文方法在不同代码规模下具有较强的适应性和稳定性.

4.4 消融实验

4.4.1 特征融合的有效性实验

为了验证特征融合在跨语言代码相似性检测中的有效性,本实验将本文所提出的模型与其变体进行了对比实验,实验结果如表 4 所示.

表 4 不同模型架构的检测结果

Table 4 Detection results of different model architectures

	Transformer	GGNN	本文方法
Precision	0.72	0.75	0.77
Recall	0.78	0.82	0.86
F1	0.74	0.78	0.81

从实验数据可以看出,特征融合后的模型在 Precision、Recall 以及 F1 上均优于仅使用 Transformer 或 GGNN 的模型变体. 虽然 Transformer 网络能够从先序遍历的 AST 路径序列中学习代码的全局结构和句法特征,但路径序列本质上是一种扁平化的表达方式. 这种扁平化处理方式会忽略代码原本所具有的层次结构和一些重要的语义关系,例如函数之间的调用关系、变量作用域等. 而 GGNN 通过 EX-CPG 通过处理图结构数据,能够更好地捕获代码的语义结构信息,但 GGNN 难以高效地捕捉到全局的句法特征. 通过融合这两种模型,可以弥补各自在捕捉句法特征和语法特征方面的不足. 通过 AST 序列,Transformer 网络能够提取出全局的句法特征,而 GGNN 则能够从代码属性图中学习到局部的语义结构特征. 两者的结合能够发挥各自的优势,使得跨语言代码相似性检测有更高的准确性.

综上所述,通过将 Transformer 所捕捉的序列特征与 GGNN 所捕捉的图特征相结合,模型能够充分利用代码的不同维度信息,对代码结构和语义信息的捕捉能力,从而提高了跨语言代码相似性检测的准确性.

4.4.2 统一词汇的有效性

为了减小不同编程语言之间的差异,本文对不同语言的抽象语法树进行统一词汇处理. 为了验证其有效性,本实验采用了两种模型进行对比:一种是基于原始 AST 的模型,它直接使用每种语言特有的 AST 表示;另一种是本文提出的模型,AST 表示经过统一词汇处理. 两种模型都采用了相同的特征提取和相似性计算方法,实验结果如图 8 所示.

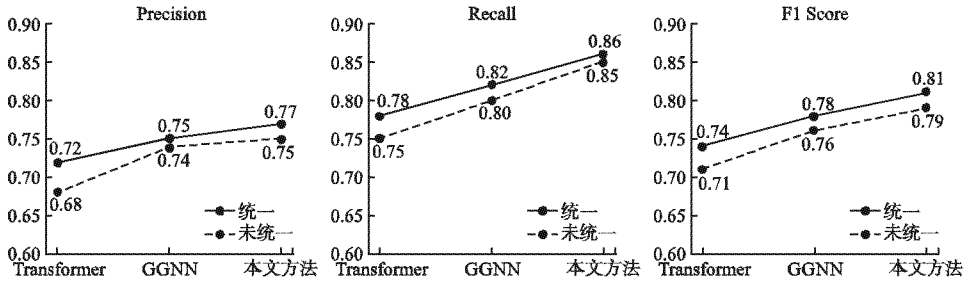

图 8 统一词汇的消融实验结果

Fig. 8 Ablation experiment results of the unified vocabulary

实验结果表明,对数据进行统一词汇之后的模型在精确率、召回率和 F1 分数上均优于基于原始 AST 的模型. 这表明,采用统一词汇处理能够显著缩小不同编程语言在特征表达上的差异,进而提升了代码相似性检测的效果. 此外,统一

词汇对 Transformer 网络而言所带来的性能提升是最为显著的.这主要是因为 Transformer 网络是提取 AST 遍历序列的特征.而这些输入到网络的初始嵌入向量,是通过词汇映射来确定的.因此,当词汇得到统一时,对 Transformer 网络的影响最为直接,性能提升也是最大的.而 GGNN 网络中,邻接矩阵的构建是基于扩展抽象语法树的结构特性,统一词汇仅在生成特征矩阵时起到作用.这一环节涉及将节点映射到相应的特征向量,因此统一词汇对 GGNN 网络的影响很小.

4.5　跨语言漏洞检测实验验证

基于上述几个实验的结果,可以得出本文提出的方法在跨语言相似性检测方面表现优异.因此,为了验证本方法在实际场景中的表现,在合成数据集 SARD 中选取了由 C、Java、Python 这 3 种语言实现的包含多种漏洞类型的 27688 个漏洞样本,这些漏洞样本都进行了函数级切片.将数据集中每种漏洞类型按照 8:2 的比例划分出检测数据和对照数据,以保证对照数据中均含有所有类型的漏洞样本,具体划分如表 5 所示.具体而言,使用本文所提出的方法将检测数据与对照数据进行相似性检测,并在预测出的相似概率最高的漏洞类别进行匹配,在数据集上的实验结果如表 6 所示.

表 5　跨语言漏洞样本的分布与划分

Table 5　Distribution and division of cross-language vulnerability samples

CWE-ID	漏洞类型	检测数据	对照数据
CWE-122	堆溢出	5221	1305
CWE-078	命令注入	8215	2053
CWE-190	整形溢出	8716	2178

表 6　不同模型架构的检测结果

Table 6　Detection results of different model architectures

漏洞类型	检测正确的数量	实际数量
堆溢出	4334	5221
命令注入	6819	8215
整形溢出	6015	8716

实验结果表明,本文提出的方法应用在漏洞检测方面也具有较高的准确性,但针对不同漏洞类型的检测效果存在明显差异.

具体而言,堆溢出和命令注入的检测表现较为出色,这主要归因于这两类漏洞在不同编程语言间的代码模式具有较高的相似性,使得其代码的抽象语法树在多种语言中的语法结构有较高相似性.相比之下,整型溢出的检测效果相对较差,实际存在的漏洞数量为 8716 个,而检测到的仅为 6015 个,准确率约为 0.69.整型溢出检测效果不佳的原因主要在于其代码模式不够显著,难以在代码的抽象语法树中找到明显且可识别的结构,从而增加了相似性检测的难度.此外,不同编程语言对整数类型的支持和实现方式存在明显差异,例如 Java 和 Python 支持大整数,而 C 语言则对整数范围有所限制,这也进一步影响了整型溢出漏洞的检测效果.

5　结　论

跨语言的代码相似性检测会受到不同编程语言之间差异的影响,使得跨语言相似性比较的准确率较低.为了解决跨语言代码学习的问题,本文提出了一种基于统一抽象语法树的集成学习方法,利用不同语言抽象语法树之间结构和节点语义具有相似性的特点,作为相似性比较的特征.在代码表示方面,将代码提取为抽象语法树,通过 Transformer 学习抽象语法树路径序列的特征作为代码的表示,得到代码的全局结构和逻辑特征.同时,为了学习不同语言抽象语法树之间局部结构的信息,在原始 AST 之上增添了 5 种新的边以构造扩展属性图,并通过 GGNN 网络学习扩展属性图的特征来捕获代码的局部结构和语义特征.最后,将两个子网络的特征融合构成综合特征作为代码特征进行相似性比较.此外,本文将不同语言的抽象语法树的通语义节点进行统一,减少了不同语言之间的差异.通过与其他传统工具和深度学习模型进行对比实验,证明了本文方法的有效性,能够有效的提高跨语言代码相似性检测的能力.并且通过将本文方法应用于跨语言漏洞检测中,进一步验证了本文方法在实际应用中的有效性.

然而,本文所提出的方法仍存在诸多不足,未来的研究工作将围绕以下几个方面展开:1)对图节点的嵌入方法进行优化,以减少语义信息的丢失;2)使用新型神经网络架构,以提升跨语言代码相似性检测的能力.

References:

[1] Ding Z,Li H,Zheng M,et al. Research and application of software reuse identification method based on code similarity analysis[C]// 4th International Conference on Electronic Information Engineering and Computer Communication(EIECC),IEEE,2024:1048-1052.

[2] Tambon F,Moradi Dakhel A,Nikanjam A,et al. Bugs in large language models generated code:an empirical study[J]. Empirical Software Engineering,2025,30(3):1-48.

[3] NVD. CVE-2021-44228 [EB/OL]. https://nvd. nist. gov/vuln/detail/CVE-2021-44228,2023-04-27.

[4] Al-Omari F,Keivanloo I,Roy C K,et al. Detecting clones across microsoft. net programming languages[C]//19th Working Conference on Reverse Engineering,IEEE,2012:405-414.

[5] Cosma G,Joy M. An approach to source-code plagiarism detection and investigation using latent semantic analysis[J]. IEEE Transactions on Computers,2011,61(3):379-394.

[6] Li L,Feng H,Zhuang W,et al. Cclearner:a deep learning-based clone detection approach[C]//IEEE International Conference on Software Maintenance and Evolution(ICSME),2017:249-260.

[7] Azcona D,Arora P,Hsiao I H,et al. user2code2vec:embeddings for profiling students based on distributional representations of source code[C]//Proceedings of the 9th International Conference on Learning Analytics & Knowledge,2019:86-95.

[8] Mou L,Li G,Zhang L,et al. Convolutional neural networks over tree structures for programming language processing[C]//Proceedings of the AAAI Conference on Artificial Intelligence,2016:1287-1293.

[9] Zhao J,Xia K,Fu Y,et al. An AST-based code plagiarism detection algorithm[C]//10th International Conference on Broadband and Wireless Computing,Communication and applications(BWCCA),IEEE,2015:178-182.

[10] Wang J,Zhang C,Chen L,et al. Improving ML-based binary function similarity detection by assessing and deprioritizing control flow

graph features[C]//33rd USENIX Security Symposium(USENIX Security 24),2024:4265-4282.

[11] Wang P,Svajlenko J,Wu Y,et al. CCAligner:a token based large-gap clone detector[C]//Proceedings of the 40th International Conference on Software Engineering,2018:1066-1077.

[12] Kang W,Son B,Heo K,Tracer:signature-based static analysis for detecting recurring vulnerabilities [C]//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security,2022:1695-1708.

[13] Ben-Nun T,Jakobovits A S,Hoefler T. Neural code comprehension:a learnable representation of code semantics[J]. Advances in Neural Information Processing Systems,2018:3589-3601,doi:10.48550/arXiv.1806.07336.

[14] Mathew G,Stolee K T. Cross-language code search using static and dynamic analyses[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,2021:205-217.

[15] Chen B,Abedjan Z. Interactive cross-language code retrieval with auto-encoders[C]//36th IEEE/ACM International Conference on Automated Software Engineering(ASE),2021:167-178.

[16] ZHANG F,WEI Y L,QIN Y C. Cross language code plagiarism detection based on program flow chart and graph attention network [J]. Journal of Chinese Computers Systems,2025,46(1):249-256.

[17] Feng Z,Guo D,Tang D,et al. Codebert:a pre-trained model for programming and natural languages [J]. arXiv preprint arXiv:2002.08155,2020:1536-1547.

[18] Bui N D Q,Yu Y,Jiang L. Infercode:self-supervised learning of code representations by predicting subtrees[C]//IEEE/ACM 43rd International Conference on Software Engineering(ICSE),2021:1186-1197.

[19] Tao C,Zhan Q,Hu X,et al. C4:contrastive cross-language code clone detection[C]//Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension,2022:413-424.

[20] Mou L,Li G,Jin Z,et al. TBCNN:a tree-based convolutional neural network for programming language processing[J]. arXiv preprint arXiv:1409.5718,2014:1287-1293.

[21] Jiang L,Su Z. Automatic mining of functionally equivalent code fragments via random testing[C]//Proceedings of the 18th International Symposium on Software Testing and Analysis,2009:81-92.

[22] Mathew G,Stolee K T. Cross-language code search using static and dynamic analyses[C]//Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,2021:205-217.

[23] Nafi K W,Kar T S,Roy B,et al. Clcdsa:cross language code clone detection using syntactical features and api documentation[C]//34th IEEE/ACM International Conference on Automated Software Engineering(ASE),2019:1026-1037.

[24] Cheng X,Peng Z,Jiang L,et al. Clcminer:detecting cross-language clones without intermediates[J]. IEICE Transactions on Information and Systems,2017,100(2):273-284.

[25] Swilam Z,Hamdy A,Pester A. Advanced cross-language clone detection using modified AST and graph neural network[C]//International Conference on Computer and Applications(ICCA),IEEE,2024:1-6.

附中文参考文献:

[16] 张 峰,韦友良,秦玉成. 基于程序流程图和图注意力网络的跨语言代码抄袭检测方法[J]. 小型微型计算机系统,2025,46(1):249-256.